**Appunti sul JDBC (Java Database Connectivity)**

**3.0 - Java Database Connectivity (JDBC)**

**3.1 - Introduzione al JDBC**

L'**API JDBC** è un'API (Application Programming Interface)\*di Java che consente di *accedere ai dati tabulari*, in particolare ai dati archiviati in un database relazionale.

Il JDBC permette di **scrivere applicazioni Java** che gestiscono tre attività fondamentali:

1. **Connessione al database**: consente di stabilire una connessione con un database.
2. **Invio di query e istruzioni di aggiornamento**: consente di inviare comandi SQL al database.
3. **Recupero e elaborazione dei risultati**: consente di recuperare i risultati delle query e manipolarli all'interno dell'applicazione Java.

\*

API:

è un insieme di regole e specifiche che definisce come diversi software possono interagire tra di loro. In altre parole, un'API permette a un'applicazione di comunicare con altre applicazioni, servizi o componenti software.

Caratteristiche principali di un'API:

Interfaccia di comunicazione: Un'API stabilisce i metodi e i formati di dati che due software utilizzano per scambiarsi informazioni. Può essere un'API di libreria (per l'accesso a funzionalità di un'applicazione) o un'API di servizio web (per comunicare con un server remoto).

Astrazione: Le API nascondono la complessità del sistema sottostante, rendendo più facile l'accesso a funzionalità complesse senza dover conoscere nel dettaglio come sono implementate. Gli sviluppatori interagiscono con un'API piuttosto che con il codice sottostante.

Standardizzazione: Le API forniscono una documentazione e uno standard che definiscono come le richieste e le risposte devono essere strutturate. Ciò permette agli sviluppatori di capire come usare correttamente l'API, anche se non hanno accesso al suo codice interno.

Esempi di API:

API di sistema operativo: Come le API di Windows o Linux che permettono agli sviluppatori di interagire con le funzionalità di basso livello, come la gestione della memoria o la lettura da file.

API di librerie: Una libreria di programmazione (ad esempio, la libreria standard di Java) fornisce un'API che definisce come usare le sue funzioni. Ad esempio, per fare una connessione a un database in Java, puoi usare l'API JDBC.

API web (RESTful o SOAP): Le API web sono un tipo specifico di API che permettono a software di comunicare attraverso il web. Queste API sono spesso utilizzate per accedere a servizi online come social media, servizi bancari o piattaforme di e-commerce. Ad esempio, le API di Twitter ti permettono di postare tweet o leggere quelli esistenti.

\*

**3.2 - Come utilizzare il JDBC**

Per utilizzare JDBC, è necessario avere la libreria corretta per il tipo di database utilizzato. Ad esempio, se si utilizza MySQL, si deve usare la libreria **mysql-connector-j**, che fornisce il driver necessario per la connessione al database MySQL.

**3.2.1 - La parte pratica**

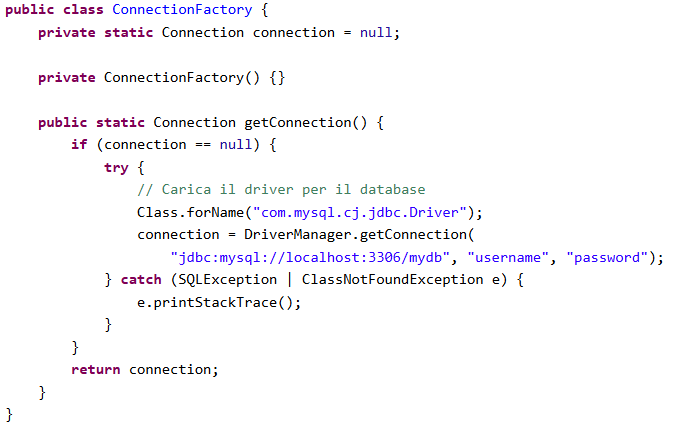
Prima di eseguire qualsiasi operazione sul database, è necessario stabilire una connessione. La connessione può essere gestita tramite una **ConnectionFactory**.

È importante che la connessione venga gestita come un **Singleton** per assicurarsi che venga aperta solo una connessione per applicazione e che venga riutilizzata ogni volta che è necessaria.

Le principali classi e interfacce utilizzate sono:

* **Connection**: rappresenta la connessione al database.
* **DriverManager**: gestisce i driver JDBC.
* **SQLException**: gestisce le eccezioni SQL.
* La connessione si stabilisce utilizzando il driver specifico per il database, tramite una stringa di connessione (URL).

Ecco un esempio di classe ConnectionFactory in Java:



**3.2.2 - Statement**

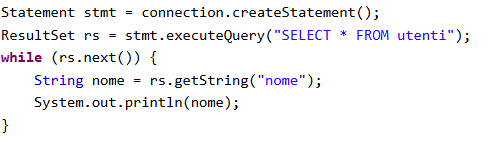
Una volta ottenuta la connessione, il prossimo passo è scrivere la query.

Si utilizza uno **Statement**, che è un'interfaccia che rappresenta una query SQL. Generalmente, le query contenute in uno Statement sono di tipo semplice e non parametriche.

I passi per eseguire un'operazione sul database sono i seguenti:

1. Stabilire una connessione.
2. Creare uno **Statement**.
3. Eseguire la query.
4. Recuperare e processare i risultati.
5. Chiudere la connessione.

Esempio di esecuzione di una query tramite Statement:



**3.2.3 - SQL Injection**

L'esecuzione di query con **Statement** è vulnerabile a **SQL Injection**, un attacco in cui un utente malintenzionato può inserire codice SQL dannoso, come DROP TABLE o DELETE, per compromettere i dati del database.

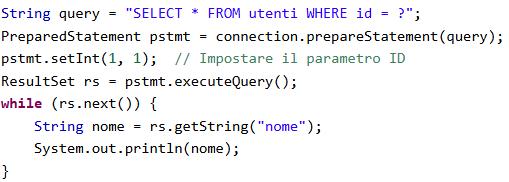
Ad esempio, un utente potrebbe inserire un ID maligno che chiama un comando SQL come il seguente:



**3.2.4 - Prepared Statement**

Per prevenire il rischio di **SQL Injection**, si utilizzano i **PreparedStatement**, che sono oggetti SQL precompilati con parametri sicuri. Con i **PreparedStatement**, i parametri vengono passati tramite metodi sicuri, impedendo che vengano iniettati comandi maligni.

Esempio di uso di **PreparedStatement**:

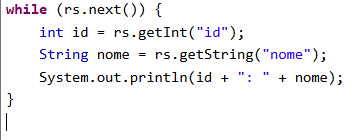


**3.2.5 - ResultSet**

Il **ResultSet** è un oggetto che rappresenta il risultato di una query SQL, contenente una tabella di dati e le righe che la compongono. Il metodo principale di **ResultSet** è **next()**, che restituisce true finché ci sono altre righe nella tabella.

Per recuperare i valori all'interno di un **ResultSet**, si utilizzano i metodi getter in base al tipo di dato e si può accedere sia tramite l'indice della colonna che tramite il nome della colonna.

Esempio di utilizzo di **ResultSet**:



ResultSet rs = null  
Connection connection = null;  
PreparedStatement preparedStatement = null;  
  
try {

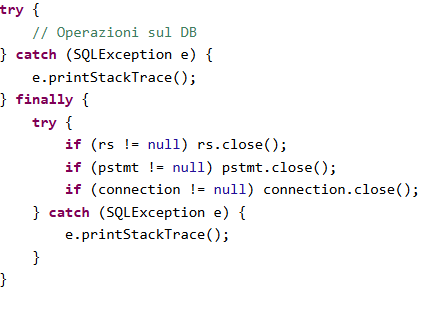
connection = getConnection();  
String sql = “SELECT id, nome, eta FROM utenti”;  
preparedStatement = connection.prepareStatement(sql);  
rs = preparedStatement.executeQuery();  
  
while (rs.next) {  
int id = rs.getInt(“id”);  
String nome = rs.getString(“nome”);  
int eta = rs.getInt(“eta”);  
}   
} catch (SQLException e) {  
e.printStackTrace();  
} finally {  
try {  
if (rs != null) rs.close();  
if (connection != null) connection.close();  
if(PreparedStatement != null) PreparedStatement.close();

} catch (SQLException e) {  
e.printStackTrace();  
}  
}  
}

**3.3 - Chiusura delle Risorse**

Dopo aver eseguito le operazioni sul database, è fondamentale chiudere correttamente le risorse, come **PreparedStatement** e **Connection**, per evitare perdite di memoria o problemi di connessione.

Esempio di chiusura delle risorse:



**Conclusioni**

Il **JDBC** è un componente fondamentale per l'interazione con i database relazionali in Java. Consente di eseguire operazioni SQL in modo semplice, ma richiede attenzione alla sicurezza (specialmente riguardo a SQL Injection) e alla gestione corretta delle risorse. Utilizzare **PreparedStatement** e chiudere correttamente le risorse sono pratiche essenziali per un'applicazione robusta e sicura.

**Domande su JDBC:**

1. **Cos'è JDBC e perché è stato introdotto?**
   * Quale problema risolve rispetto all'accesso tradizionale ai database?

JDBC è un API java di basso livello, utilizzata per consentire l’accesso ai dati presenti all’interno di un database relazionale, ha risolto svariati problemi, grazie JDBC è possibile accedere ai dati di un database potendo manipolare quei dati.

1. **Quali sono i passaggi fondamentali per connettersi a un database usando JDBC?**
   * Qual è il ruolo della classe DriverManager?
2. **Che differenza c'è tra Statement, PreparedStatement e CallableStatement?**
   * Perché si preferisce il PreparedStatement nella maggior parte dei casi?
3. **Come funziona il processo di esecuzione di una query in JDBC?**
   * Quali metodi vengono usati per query di lettura (SELECT) e query di modifica (INSERT, UPDATE, DELETE)?
4. **Cos'è il ResultSet e come lo usi?**
   * Come si naviga nei risultati di una query?
   * Qual è la differenza tra tipi di ResultSet come TYPE\_FORWARD\_ONLY e TYPE\_SCROLL\_INSENSITIVE?
5. **Come gestisci i dati in transazioni con JDBC?**
   * Come abiliti/disabiliti l’autocommit?
   * Quali metodi usi per confermare o annullare una transazione?
6. **Quali eccezioni puoi incontrare con JDBC e come gestirle?**
   * Qual è la differenza tra SQLException e SQLTimeoutException?
7. **Come si ottimizza la connessione al database?**
   * Cosa sono i **Connection Pool** e perché sono importanti?
8. **JDBC è adatto per progetti moderni?**
   * Qual è il suo ruolo attuale rispetto a framework come JPA o Hibernate?
9. **Quali sono le best practice per scrivere codice JDBC sicuro e manutenibile?**
   * Come eviti problemi come SQL Injection?
   * Come gestisci correttamente le risorse (connessioni, statement, result set)?
10. **Qual è la differenza tra JAR, WAR ed EAR?**