**Cos’è una Transazione?**

Una **transazione** è una sequenza di istruzioni SQL che devono essere eseguite come un'unità unica e indivisibile. Questo significa che tutte le operazioni all'interno della transazione devono avere successo perché i cambiamenti siano applicati al database; se una delle operazioni fallisce, nessun cambiamento viene applicato.

Le transazioni seguono generalmente questa struttura:

* **BEGIN TRANSACTION**: avvio della transazione.
* Esecuzione di una o più istruzioni SQL (inserimento, aggiornamento, eliminazione, ecc.).
* **COMMIT**: conferma delle operazioni se tutte hanno avuto successo, rendendo effettivi i cambiamenti.
* **ROLLBACK**: annullamento della transazione (se si verifica un errore), riportando il database allo stato precedente l'inizio della transazione.

In sostanza:

* Se tutte le istruzioni vanno a buon fine, la transazione viene **committata** (cioè confermata) e le modifiche diventano permanenti.
* Se si verifica un errore o se si decide di annullare la transazione, si esegue un **rollback**, riportando il database allo stato iniziale, come se la transazione non fosse mai iniziata.

**Proprietà ACID delle Transazioni**

Per garantire la coerenza e la sicurezza dei dati, le transazioni devono seguire le proprietà ACID, un acronimo che rappresenta:

1. **Atomicità** (*Atomicity*): Una transazione è **atomica**, cioè indivisibile. O viene completata interamente, o non viene eseguita affatto. Se una parte della transazione fallisce, tutte le modifiche effettuate vengono annullate.
2. **Coerenza** (*Consistency*): Una transazione porta il database da uno stato consistente a un altro stato consistente. Ciò significa che, se il database era integro prima della transazione, rimarrà integro anche dopo, rispettando tutte le regole e vincoli di integrità.
3. **Isolamento** (*Isolation*): Ogni transazione è **isolata** dalle altre e non vede i cambiamenti delle altre transazioni fino a quando queste non sono state committate. Questo evita che una transazione legga dati temporanei o parzialmente aggiornati da un'altra transazione in corso.
4. **Durabilità** (*Durability*): Una volta che una transazione è **committata**, i suoi effetti sono permanenti e persisteranno anche in caso di guasti o interruzioni del sistema. I dati vengono salvati in modo sicuro.

**Esempio di Transazione**

Immaginiamo una transazione che trasferisce denaro da un conto a un altro:

1. Inizio della transazione.
2. Rimozione dell'importo dal conto A.
3. Aggiunta dell'importo al conto B.
4. Commit: se entrambe le operazioni sono eseguite correttamente, la transazione viene confermata.
5. Rollback: se si verifica un problema (es. saldo insufficiente), la transazione viene annullata, e il denaro rimane nel conto A.

**Configurazione di JDBC per l'utilizzo**

Per configurare JDBC (Java Database Connectivity) e iniziare a utilizzarlo con un database come **MySQL**, segui questi passi:

1. **Scarica il driver JDBC per MySQL**:
   * Per connetterti al database MySQL, devi scaricare il **driver JDBC** specifico per MySQL. Per farlo, cerca su Internet "JDBC driver download MySQL" e scarica il file ZIP dalla pagina ufficiale.
   * All'interno di questo file ZIP, troverai un file .jar (Java Archive) che contiene tutte le classi necessarie per connetterti al database.
2. **Aggiungi il file .jar al progetto**:
   * Estrai il file .jar e copialo all'interno della cartella lib del tuo progetto in **Eclipse**.
   * **JAR** (Java ARchive) è un archivio compresso che contiene tutte le classi necessarie per la connessione al database, come i driver e le librerie.
3. **Configura il progetto per riconoscere il JAR**:
   * Una volta copiato il file .jar, devi dire a Eclipse dove trovarlo. Vai su:
     + **Right-click sul progetto** → **Build Path** → **Configure Build Path**.
     + Nella sezione **Libraries**, clicca su **Add JARs**.
     + Seleziona il file .jar che hai appena copiato nella cartella lib del tuo progetto. Ora il progetto è configurato per utilizzare JDBC.

**Cosa è JDBC?**

JDBC è una tecnologia che permette alle applicazioni Java di connettersi e interagire con i database. Utilizzando JDBC, puoi:

* Eseguire operazioni come **creare**, **leggere**, **aggiornare** e **eliminare** dati (le famose operazioni CRUD).
* Inviare comandi SQL al database per manipolare i dati.

JDBC funge da ponte tra la tua applicazione Java e il database, consentendoti di eseguire comandi SQL, che sono il linguaggio utilizzato dai database per manipolare i dati.

**Come connettersi al database:**

Per stabilire una connessione con un database utilizzando JDBC, occorre eseguire una serie di passaggi. Ecco le 5 istruzioni principali per configurare la connessione:

1. **Caricare il driver JDBC**: La prima cosa da fare è caricare il driver JDBC specifico per il tipo di database che stai utilizzando. Per MySQL, userai:

Class.forName("com.mysql.cj.jdbc.Driver");

Questo carica la classe del driver che permette la connessione al database MySQL. La classe è inclusa nel file .jar che hai aggiunto al progetto.

1. **Creare una connessione al database**: Utilizza il metodo getConnection() per stabilire la connessione. Il metodo accetta tre parametri:
   * **URL**: l'indirizzo di connessione al database.
   * **Utente**: il nome utente per l'autenticazione.
   * **Password**: la password per l'autenticazione. Ecco un esempio di connessione a un database MySQL:

Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/prova?useUnicode=true&useSSL=false", "root", "password");

​

Cosa contiene l’URL?

L'**URL** contiene:

* jdbc:mysql: specifica il protocollo di connessione (per MySQL).
* localhost: l'indirizzo del server dove risiede il database. Può essere "localhost" (se il database è sulla stessa macchina) o un indirizzo IP o nome di dominio.
* 3306: la porta predefinita su cui MySQL ascolta le connessioni.
* prova: il nome del database a cui ci si vuole connettere.
* Parametri come useUnicode=true sono opzionali ma utili per la gestione dei caratteri speciali.

1. **Creare una dichiarazione SQL**: Una volta ottenuta la connessione, puoi iniziare a eseguire comandi SQL. Di solito, crei un oggetto PreparedStatement per eseguire le query, poiché consente di eseguire query parametrizzate (e quindi sicure contro attacchi SQL Injection).

String sql = "SELECT \* **FROM** utenti **WHERE** id = ?";

PreparedStatement pstmt = conn.prepareStatement(sql);

pstmt.setInt(1, 1);  // Imposta il parametro della query (id = 1)

1. **Eseguire la query e gestire i risultati:** Esegui la query utilizzando il metodo executeQuery() per le SELECT o executeUpdate() per le operazioni di INSERT, UPDATE o DELETE. Ad esempio:

​ ResultSet rs = pstmt.executeQuery();

while (rs.next()) {

  System.out.println("ID: " + rs.getInt("id") + ", Nome: " + rs.getString("nome"));

}

1. **Chiudere la connessione**: È buona norma chiudere la connessione e altre risorse (come PreparedStatement e ResultSet) al termine dell'operazione per evitare perdite di memoria:

pstmt.close();

rs.close();

conn.close();

**Importante da ricordare:**

* **Class.forName(driver)**: Questa istruzione carica il driver JDBC che consente la connessione al database specifico. Ogni database ha un driver diverso.
* **getConnection(url, user, password)**: Questa istruzione stabilisce la connessione al database. L'URL di connessione specifica dove trovare il database e i dettagli necessari per la connessione.
* **PreparedStatement**: Utilizzato per preparare e eseguire comandi SQL in modo sicuro e parametrizzato, riducendo il rischio di SQL Injection.

In sintesi, i principali comandi e concetti per lavorare con **JDBC**:

1. **Connessione al Database:**
   * **getConnection(url, user, password)**: Stabilisce la connessione al database.
2. **Creazione e Esecuzione delle Query:**
   * **createStatement()**: Crea un oggetto Statement per eseguire query SQL senza parametri.
   * **prepareStatement(sql)**: Crea un oggetto PreparedStatement per eseguire query SQL con parametri.
   * **executeQuery(sql)**: Esegue una query che restituisce risultati (es. SELECT), restituendo un oggetto di tipo ResultSet.
   * **executeUpdate(sql)**: Esegue query di modifica (es. INSERT, UPDATE, DELETE), restituendo il numero di righe modificate.
   * **execute()**: Esegue una query di qualsiasi tipo e restituisce un booleano che indica se il risultato è un ResultSet.
3. **Gestione dei Risultati:**
   * **next()**: Sposta il cursore al prossimo record nel ResultSet.
   * **getInt(columnName)** e **getString(columnName)**: Ottiene il valore di una colonna nel ResultSet.
4. **Gestione delle Transazioni:**
   * **setAutoCommit(false)**: Disabilita l'auto-commit per gestire manualmente le transazioni.
   * **commit()**: Applica le modifiche fatte durante la transazione.
   * **rollback()**: Annulla le modifiche fatte durante la transazione.
5. **Chiusura delle Risorse:**
   * **close()**: Chiude Connection, Statement e ResultSet per liberare risorse.
6. **Gestione delle Eccezioni:**
   * **SQLException**: Gestisce gli errori durante l'esecuzione delle operazioni SQL.

**Operazioni con JDBC: INSERT, UPDATE, e DELETE**

* **INSERT, UPDATE, DELETE**: La sintassi per queste operazioni è molto simile tra loro. L'unica differenza principale è che **INSERT** e **UPDATE** non restituiscono risultati, mentre **SELECT** restituisce un insieme di dati. Questo implica che, se usiamo **INSERT, UPDATE** o **DELETE**, dobbiamo usare executeUpdate() per eseguire la query, mentre per **SELECT**, utilizziamo executeQuery().

**Gestione dei Risultati con ResultSet**

* Quando eseguiamo una query che restituisce dei dati (tipicamente una query SELECT), il risultato viene messo all'interno di un **ResultSet**. Pensalo come un "contenitore" che contiene i dati restituiti dalla query.
* Il **ResultSet** può essere visto come un "puntatore" che all'inizio è nullo, e si sposta tra le righe dei risultati tramite il metodo **next()**. Ogni volta che chiamiamo **next()**, il cursore si sposta alla riga successiva, e il metodo ritorna un valore booleano che indica se esistono altre righe da leggere:
  + **next()** restituisce **true** se ci sono altre righe.
  + **false** se non ci sono più righe.

**Esempio di Iterazione su ResultSet**

Per iterare attraverso tutte le righe restituite da una query, usiamo un ciclo **while**, che continua a iterare finché **next()** restituisce **true**.

while (resultSet.next()) {

   // Recupera i dati da ogni riga del ResultSet

  String nome = resultSet.getString("nome");

   int età = resultSet.getInt("età");

   // Fai qualcosa con i dati

}

​

* **getString()**, **getInt()**, e altri metodi simili servono per ottenere il valore delle colonne nella riga attuale del **ResultSet**.

**Sintesi**

* **INSERT**, **UPDATE**, e **DELETE** non restituiscono dati, quindi usiamo **executeUpdate()** per eseguire queste query.
* **SELECT** restituisce un **ResultSet**, e dobbiamo usare **next()** per spostare il cursore e leggere le righe restituite.
* L'uso di **next()** in un ciclo **while** ci consente di accedere a tutte le righe del risultato.

**Evoluzione della Tecnologia JDBC**

Un tempo, JDBC era il metodo principale per interagire con i database da Java. Oggi, viene utilizzato soprattutto dietro le quinte, integrato in tecnologie più avanzate come **JPA (Java Persistence API)** o framework come **Spring Data**, che semplificano ulteriormente l'interazione con i database.

**Dispensa JDBC**

JDBC è tecnologia Java che permette di connettersi e manipolare le tabelle di un DB.

I passi da seguire sono:

• Caricare un driver

• Aprire una connessione con il database

• Creare un oggetto Statement per interrogare il database

• Interagire con il database

• Gestire i risultati ottenuti

Un "**driver**" è un modulo software attraverso il quale un DBMS permette le connessioni. Un’applicazione che vuole usare un DB deve quindi avere a disposizione un apposito driver (fornito dal venditore del DB).

Per poter creare l'oggetto Statement, c'è bisogno dell'indirizzo del DB che in genere ha la seguente forma:

jdbc:<subprotocol>:<subname>

<subprotocol> si riferisce al driver

<subname> si riferisce alla sorgente dati

Queste stringhe variano in base al tipo di DB a cui ci connettiamo.

Quindi ogni client deve avere queste stringhe di connessione.

Esempio

HelloWorldVarie

3.7 Datasource

La piattaforma J2EE presenta uno strumento molto utile permette di disaccoppiare la sorgente dati e le classi JDBC utilizzate per la gestione delle query SQL, permettendo cosi di non memorizzare la username e la password sui dispositivi client. Tale servizio è affidato all’application server.

Il concetto è noto come **pool di connessioni** o **Datasource**. Il ciclo di vita di questo nuovo oggetto sarà affidato all’application server, che gestirà le risorse operando su tale pool, garantendo le connessioni al database a chi ne farà richiesta.

Oltre a garantire che le password saranno centralizzate nel server, quindi se dobbiamo modificare una utenza la modifichiamo sul server e non su tutti i client, ottimizza l’uso delle connessioni; stabilire una connessione con una base di dati è un'operazione molto costosa, dal punto di vista delle risorse richieste. E’ necessario caricare il driver del database, stabilire una comunicazione bidirezionale sicura fra l'applicazione che deve accedere ai dati e il database stesso, eseguire una procedura di autenticazione. Il datasource risolve questo grazie al meccanismo di pooling, ossia quando un client richiede una nuova connessione se ne esiste una già disponibile si usa quella altrimenti se ne crea una nuova, fino a un certo numero massimo.