Introduction to JDBC API(Day- 1)

JDBC (Java Database Connectivity) API is a Java-based API that allows Java programs to interact with relational databases like MySQL, Oracle, PostgreSQL, SQL Server, etc. It provides a standard interface for connecting to databases, executing SQL queries, and processing the results.

JDBC consists of two main layers:
1. **JDBC API** – Interfaces and classes in the java.sql and javax.sql packages.

2. **JDBC Driver** – The implementation provided by database vendors to communicate with the actual database.

Types of JDBC Drivers

Type 1: JDBC-ODBC Bridge driver
Type 2: Native-API driver
Type 3: Network Protocol driver
Type 4: Thin driver (Pure Java driver - most commonly used):

Thin driver:

- It is a pure Java driver used on the client side, without an Oracle client installation.
- It can be used with both applets and applications.
- It is platform-independent and does not require any additional Oracle software on the client side.
- The JDBC Thin driver communicates with the server using SQL*Net to access the Oracle Database.
- The JDBC Thin driver allows a direct connection to the database by providing an implementation of SQL*Net on top of Java sockets.
- The driver supports the TCP/IP protocol and requires a TNS listener on the TCP/IP sockets on the database server.

Common JDBC Operations
---
* Connecting to a database
* Creating a table
* Inserting, updating, deleting records
* Retrieving data
* Using transactions
* Calling stored procedures

Note:
java.sql.Connection:
---
java.sql.Connection interface is the root of JDBC API.
- In Java, java.sql.Connection is an interface provided by the Java Standard API that represents a connection to a specific database.
- In Java, java.sql.Connection is an interface in the java.sql package used to establish a connection between a Java application and a database using JDBC (Java Database Connectivity).

- It is part of the java.sql package and plays a crucial role in database interaction.
- The Connection object is used to establish a connection with a database, perform database operations (such as querying, updating, and modifying data), and manage transactions.

Structure of java.sql.Connection:

```
public interface Connection extends Wrapper, AutoCloseable
```

Here,
Wrapper: Helps in unwrapping vendor-specific implementations.
AutoCloseable: Lets you use try-with-resources to auto-close the connection.
- It represents a session with a specific database.
- You cannot create an object of Connection directly (since it's an interface).

- You get its object using a JDBC driver via:

```
Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
```

Key Functions of java.sql.Connection:

---

* Establishing a Database Connection:
The Connection interface is implemented by database-specific classes, which are used to create a connection to the database. Typically, you obtain a Connection object through the DriverManager or a DataSource.

* Executing SQL Statements
Once a Connection object is created, you can use it to create statements (Statement, PreparedStatement, or CallableStatement) to execute SQL queries or updates.

* Managing Transactions
The Connection interface provides methods to manage database transactions, including committing and rolling back transactions. By default, each SQL statement is executed in its own transaction, but you can disable auto-commit mode and manually control the transaction boundaries.

* Closing the Connection
After performing database operations, it is essential to close the connection to release database resources.

Instead of repeatedly creating and closing Connection objects, connection pooling is commonly used to optimize performance. A connection pool maintains a pool of reusable connections, which can be fetched and returned as needed.

Important method of java.sql.Connection:

```
public static Connection getConnection(String url, String user, String password) throws SQLException
```

url: The connection URL (defines the database, the server location, and sometimes other connection parameters).
user: The username to authenticate the connection.
password: The password for the provided username.

Syntax:

```
Connection c = DriverManager.getConnection
      ("url","username","password");
```

Once the connection is established we need to perform some actions on the database, to do it JDBC statements come into picture.

JDBC Statements:

In Java, JDBC Statements are objects used to send SQL queries to the database and retrieve results. They are part of the java.sql package and play a central role in interacting with databases through JDBC.

Types of JDBC Statements

There are three main types of JDBC statement objects:

1. Statement
2. PreparedStatement
3. CallableStatement

1. Statement:

---

* It is an interface available in java.sql.
* Used to execute simple SQL queries (like SELECT, CREATE, INSERT, UPDATE, DELETE) without parameters.
* Best for static SQL.

e.g:

```
Statement s = c.createStatement();
      ResultSet rSet= s.executeQuery("select * from emp");
```

Q. What is createStatement()?

Answer: createStatement() is a method of the Connection interface in JDBC.
- It is used to create a Statement object, which allows you to send SQL queries (like SELECT, INSERT, UPDATE, DELETE) to the database.
- Internally it holds Anonymous Local Inner class as implementation class of Statement.

e.g:

```
Connection c = DriverManager.getConnection
      ("jdbc:oracle:thin:@localhost:1521:XE","system","tiger");
Statement s = c.createStatement();
```

Important methods from Statement:

i. public abstract java.sql.ResultSet executeQuery(java.lang.String) throws java.sql.SQLException;

- it executes an SQL SELECT query.
- This method will return ResultSet object containing the data returned by the query.
- Here the parameter will be a SELECT query.

Example:

```
Connection c = DriverManager.getConnection
      ("jdbc:oracle:thin:@localhost:1521:XE","system","tiger");
Statement s = c.createStatement();
```

```
ResultSet rSet = s.executeQuery("select * from Employee");
```

ii. public abstract int executeUpdate(java.lang.String) throws java.sql.SQLException;

- It is used to execute NON SELECT queries(CREATE,INSERT,UPDATE,DELETE)
- How many number of records we are inserting,updating.. are returned hence the return type is int.

```
import java.sql.*;
public class JdbcExample {
   public static void main(String[] args) {
      try {
         Connection conn = DriverManager.getConnection
                  ("jdbc:oracle:thin:@localhost:1521:XE", "username", "password");

         Statement stmt = conn.createStatement();
         ResultSet rs = stmt.executeQuery("SELECT * FROM employees");

         while (rs.next()) {
            System.out.println("ID: " + rs.getInt("id") +
                     ", Name: " + rs.getString("name"));
         }

         conn.close();
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }
```