**JDBC**

What is JDBC?

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

* Making a connection to a database.
* Creating SQL or MySQL statements.
* Executing SQL or MySQL queries in the database.
* Viewing & Modifying the resulting records.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as −

* Java Applications
* Java Applets
* Java Servlets
* Java ServerPages (JSPs)

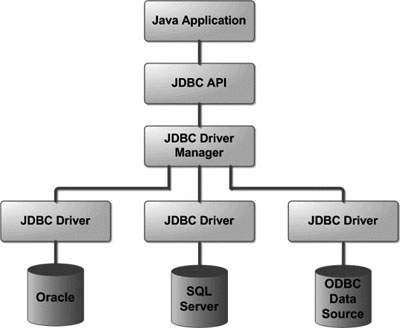
JDBC Architecture

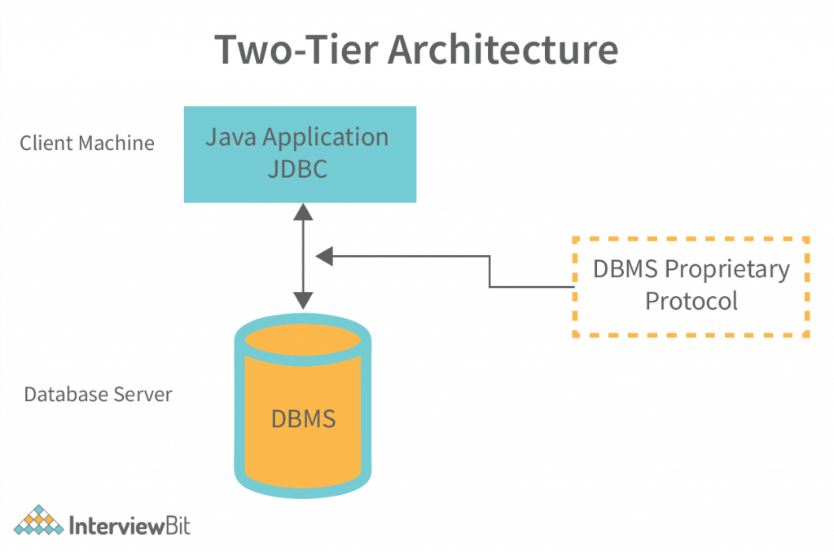
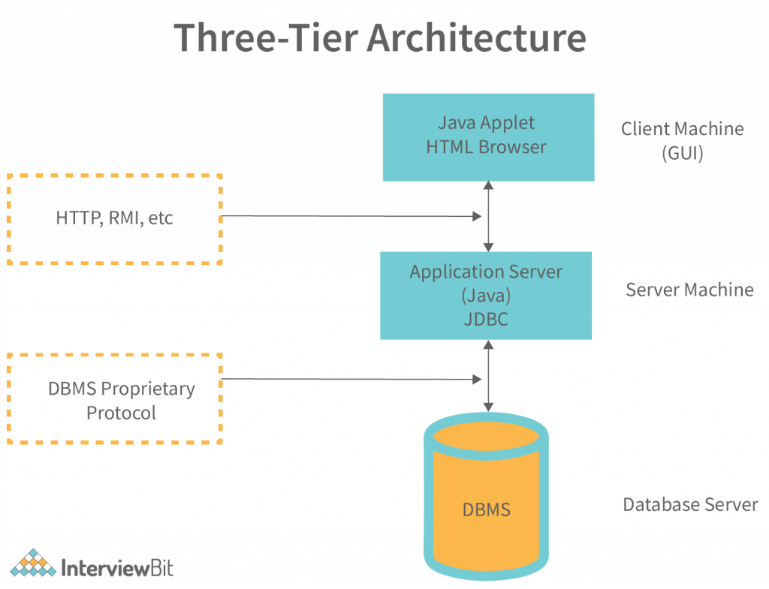
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers −

* **JDBC API** − This provides the application-to-JDBC Manager connection.
* **JDBC Driver API** − This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application –



The components of JDBC are listed below. These elements assist us in interacting with a database. The following are the JDBC components:

1. **JDBC Driver Manager:** In a JDBC application, the Driver Manager loads database-specific drivers. This driver manager makes a database connection. To handle the user request, it additionally makes a database-specific call to the database.
2. **Driver:** A driver is an interface that manages database server connectivity. Communication is handled using DriverManager objects.
3. **JDBC-ODBC Bridge Drivers:** They are used to link database drivers to the database. The JDBC method calls are translated into ODBC method calls by the bridge. To access the ODBC (Open Database Connectivity) characteristics, it uses the sun.jdbc.odbc package, which includes the native library.
4. **JDBC API:** Sun Microsystem has provided JDBC API, which allows you to write a Java program that talks with any database without modifying the code. The JDBC API is implemented by the JDBC Driver.
5. **JDBC Test Suite:** The JDBC Test Suite aids in the testing of JDBC Driver operations such as insertion, deletion, and updating. It aids in determining whether or not the JDBC Drivers will run the program. It ensures that the program will be run by JDBC Drivers with confidence and conformity.
6. **Database Server:** This is the database server that the JDBC client wants to communicate with, such as Oracle, MySQL, SQL Server, and so on.
7. **Statement:**To send SQL statements to the database, you use objects built using this interface. In addition to performing stored procedures, certainly derived interfaces accept parameters.
8. **RuleSet:** These objects retain data retrieved from a database when you use Statement objects to conduct a SQL query. It functions as an iterator, allowing you to cycle through the data it contains.
9. **SQL Exception:**This class is responsible for any errors that occur in a database application.

**JDBC - Statements, PreparedStatement and CallableStatement**

Once a connection is obtained we can interact with the database. The JDBC *Statement, CallableStatement,* and *PreparedStatement* interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

They also define methods that help bridge data type differences between Java and SQL data types used in a database.

|  |  |
| --- | --- |
| **Interfaces** | **Recommended Use** |
| Statement | Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters. |
| PreparedStatement | Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime. |
| CallableStatement | Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters. |

The Statement Objects

Creating Statement Object

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement( ) method, as in the following example −

Statement stmt = null;

try {

stmt = conn.createStatement( );

. . .

}

catch (SQLException e) {

. . .

}

finally {

. . .

}

Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.

* **boolean execute (String SQL)**: Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
* **int executeUpdate (String SQL)** − Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
* **ResultSet executeQuery (String SQL)** − Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Closing Statement Object

Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

Statement stmt = null;

try {

stmt = conn.createStatement( );

. . .

}

catch (SQLException e) {

. . .

}

finally {

stmt.close();

}

A **PreparedStatement** is a pre-compiled SQL statement. It is a subinterface of **Statement**. Prepared Statement objects have some useful additional features than Statement objects. Instead of hard coding queries, PreparedStatement object provides a feature to execute a parameterized query.

**Advantages of PreparedStatement**

* When PreparedStatement is created, the SQL query is passed as a parameter. This Prepared Statement contains a pre-compiled SQL query, so when the PreparedStatement is executed, DBMS can just run the query instead of first compiling it.
* We can use the same PreparedStatement and supply with different parameters at the time of execution.
* An important advantage of PreparedStatements is that they prevent SQL injection attacks.

Steps to use prepared Statement

1. Create Connection to Database.

Connection myCon = DriverManager.getConnection(path,username,password)

1. Prepare Statement

//Instead of hardcoding queries like,

select \* from students where age>10 and name ='Chhavi'

//Set parameter placeholders(use question mark for placeholders) like,

select \* from students where age> ? and name = ?

PreparedStatement myStmt;

myStmt = myCon.prepareStatement(select \* from students where age> ? and name = ?);

1. Set parameter values for type and position.

myStmt.setInt(1,10);

myStmt.setString(2,"Chhavi");

1. Execute the query.

ResultSet myRs= myStmt.executeQuery();

**Methods of PreparedStatement:**

* **setInt(int, int):**This method can be used to set integer value at the given parameter index.
* **setString(int, string):**This method can be used to set string value at the given parameter index.
* **setFloat(int, float):**This method can be used to set float value at the given parameter index.
* **setDouble(int, double):**This method can be used to set a double value at the given parameter index.
* **executeUpdate():**This method can be used to create, drop, insert, update, delete etc. It returns int type.
* **executeQuery():**It returns an instance of ResultSet when a select query is executed.

// Java program to execute a query using PreparedStatement

import java.sql.\*;

public class GFG {

    // Driver Code

    public static void main(String[] args) throws Exception

    {

        // Register Driver Class

        Class.forName("org.apache.derby.jdbc.ClientDriver");

        // Connection to your database

        Connection con = DriverManager.getConnection();

        // Query which needs parameters

        String query

            = "Select \* from students where age> ? and name = ?";

        // Prepare Statement

        PreparedStatement myStmt

            = con.prepareStatement(query);

        // Set Parameters

        myStmt.setInt(1, 20);

        myStmt.setString(2, 'Prateek');

        // Execute SQL query

        ResultSet myRs = myStmt.executeQuery();

        System.out.println('Age      Name');

        // Display function to show the Resultset

        while (myRs.next()) {

            String Name = rs.getString("name");

            int age = rs.getInt("age");

            System.out.println(Name + "     " + age);

        }

        // Close the connection

        con.close();

    }

}

**The CallableStatement Objects :** Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

**Creating CallableStatement Object**

Suppose, you need to execute the following Oracle stored procedure –

DELIMITER $$

DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$

CREATE PROCEDURE `EMP`.`getEmpName`

(IN EMP\_ID INT, OUT EMP\_FIRST VARCHAR(255))

BEGIN

SELECT first INTO EMP\_FIRST

FROM Employees

WHERE ID = EMP\_ID;

END $$

DELIMITER ;

Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three.

Here are the definitions of each –

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| IN | A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods. |
| OUT | A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the getXXX() methods. |
| INOUT | A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods. |