JDBC

JDBC is an acronym for Java Database Connectivity. It’s an advancement for ODBC ( Open Database Connectivity ). JDBC is an standard API specification developed in order to move data from frontend to backend. This API consists of classes and interfaces written in Java. It basically acts as an interface (not the one we use in Java) or channel between your Java program and databases i.e it establishes a link between the two so that a programmer could send data from Java code and store it in the database for future use.

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

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

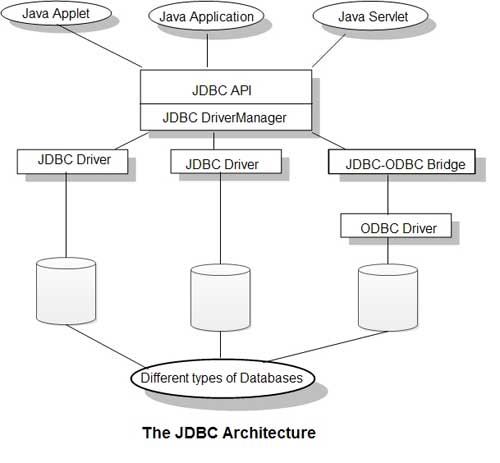
**JDBC ARCHITECTURE**

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.



**COMMON JDBC COMPONENTS**

The JDBC API provides the following interfaces and classes :

1. **DriverManager**: This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
2. **Driver**: This interface handles the communications with the database server. We rarely interact directly with Driver objects. Instead, we use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
3. **Connection:** This interface has all the methods for contacting a database. All the communication with the database is through the connection object only.
4. **Statement**: The objects created from this interface are used to submit SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
5. **ResultSet**: These objects hold data retrieved from a database after executing an SQL query using Statement objects. It acts as an iterator to allow us to move through its data.
6. **SQLException**: This class handles any errors that occur in the database application.

JDBC DRIVERS

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver

2. Native-API driver (partially java driver)

3. Network Protocol driver (fully java driver)

4. Thin driver (fully java driver)

Drivers are pluggable items so that existing drivers can be replaced by new drivers of the database without rewriting the application

1. **JDBC-ODBC bridge driver**

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.

1. **Native-API driver**

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

1. **Network Protocol driver**

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in Java. Type 3 database requests are passed through the network to the middle-tier server The middle-tier then translates the request to the database

1. **Thin driver**

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language. uses java networking libraries to communicate directly with the database server

**Creating JDBC Application**

There are following six steps involved in building a JDBC application −

* Import the packages
* Register the JDBC drive
* Open a connection
* Execute a query
* Extract data from result set
* Clean up the environment

**Import JDBC Packages**

The ‘import’ statements tell the Java compiler where to find the classes we reference in our code and are placed at the very beginning of the source code. To use the standard JDBC package, which allows us to select, insert, update, and delete data in SQL tables, we must add the following imports to our source code –

import java.sql.\* ; // for standard JDBC programs

import java.math.\* ; // for BigDecimal and BigInteger support

**Register JDBC Driver**

We must register the driver in our program before we use it. Registering the driver is the process by which the driver's class file is loaded into the memory, so that it can be utilized as an implementation of the JDBC interfaces. We need to do this registration only once in our program, which can be done in two ways :

**Approach I - Class.forName()**

The most common approach to register a driver is to use Java's ‘Class.forName()’ method, to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it makes the driver registration configurable and portable.

try {

Class.forName("oracle.jdbc.driver.OracleDriver");

}

catch(ClassNotFoundException ex) {

System.out.println("Error: unable to load driver class!");

System.exit(1);

}

**Approach II - DriverManager.registerDriver()**

The second approach we can use to register a driver, is to use the static ‘DriverManager.registerDriver()’ method. We should use the registerDriver() method if we are using a non-JDK compliant JVM, such as the one provided by Microsoft.

try {

Driver myDriver = new oracle.jdbc.driver.OracleDriver();

DriverManager.registerDriver( myDriver );

}

catch(ClassNotFoundException ex) {

System.out.println("Error: unable to load driver class!");

System.exit(1);

}

**ESTABLISH CONNECTION**

After loading the driver, establish connections using :

Connection con = DriverManager.getConnection(url,user,password)

user – username from which your sql command prompt can be accessed.password – password from which your sql command prompt can be accessed.

con: is a reference to Connection interface.

url : Uniform Resource Locator. It can be created as follows:

String url = “ jdbc:oracle:thin:@localhost:1521:xe”

Where oracle is the database used, thin is the driver used , @localhost is the IP Address where database is stored, 1521 is the port number and xe is the service provider. All 3 parameters above are of String type and are to be declared by programmer before calling the function. Use of this can be referred from final code.

**Create a statement**

Once a connection is established you can interact with the database. The JDBCStatement, CallableStatement, and PreparedStatement interfaces define the methods that enable you to send SQL commands and receive data from your database.

Use of JDBC Statement is as follows:

Statement st = con.createStatement();

Here, con is a reference to Connection interface used in previous step .

**Execute the query**

Now comes the most important part i.e executing the query. Query here is an SQL Query . Now we know we can have multiple types of queries. Some of them are as follows:

* Query for updating / inserting table in a database.
* Query for retrieving data .

The executeQuery() method of Statement interface is used to execute queries of retrieving values from the database. This method returns the object of ResultSet that can be used to get all the records of a table.The executeUpdate(sql query) method ofStatement interface is used to execute queries of updating/inserting .

Example:

int m = st.executeUpdate(sql);

if (m==1)

System.out.println("inserted successfully : "+sql);

else

System.out.println("insertion failed");

Here sql is sql query of the type String

**Close the connections**

So finally we have sent the data to the specified location and now we are at the verge of completion of our task .By closing connection, objects of Statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Example :

con.close();

**Implementation**

importjava.sql.\*;

importjava.util.\*;

class Main

{

public static void main(String a[])

{

//Creating the connection

String url = "jdbc:oracle:thin:@localhost:1521:xe";

String user = "system";

String pass = "12345";

//Entering the data

Scanner k = new Scanner(System.in);

System.out.println("enter name");

String name = k.next();

System.out.println("enter roll no");

int roll = k.nextInt();

System.out.println("enter class");

String cls = k.next();

//Inserting data using SQL query

String sql = "insert into student1 values('"+name+"',"+roll+",'"+cls+"')";

Connection con=null;

try

{

DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

//Reference to connection interface

con = DriverManager.getConnection(url,user,pass);

Statement st = con.createStatement();

int m = st.executeUpdate(sql);

if (m == 1)

System.out.println("inserted successfully : "+sql);

else

System.out.println("insertion failed");

con.close();

}

catch(Exception ex)

{

System.err.println(ex);

}

}

}

**Dynamic Queries**

Dynamic SQL or Dynamic Query is a programming technique that enables one to build SQL

statements dynamically at runtime. One can create more general purpose, flexible applications by

using dynamic SQL because the full text of a SQL statement may be unknown at compilation.

For example: dynamic SQL can help us create a procedure that operates on a table whose name

is not known until runtime. These applications allow users to input or choose query search or

sorting criteria at runtime. It is basically like assembling SQL query segments together based on

input entered.

Example Pseudocode :

select\_clause = ‘SELECT’

from\_clause = ‘FROM’

where\_clause = ‘WHERE’

orderby\_clause = ‘ORDER BY’

if [query on person] then

select\_clause += ‘p.name, p.dob’

from\_clause += ‘person p’

orderby\_clause += 'p.name’

if [query on address] then

select\_clause += ‘a.address\_text’

from\_clause += ‘, address a’

where\_clause += ‘p.address\_id = a.id AND a.id =:p1’

else

where\_clause += ‘p.id =:p1’

end if

end if

sql stmt = select\_clause + free\_clause + where\_clause + orderby\_clause + ‘ ;’

This stmt which is derived from user inputs is later used to execute queries to obtain the desired

result set at run time.