UNIT – VI

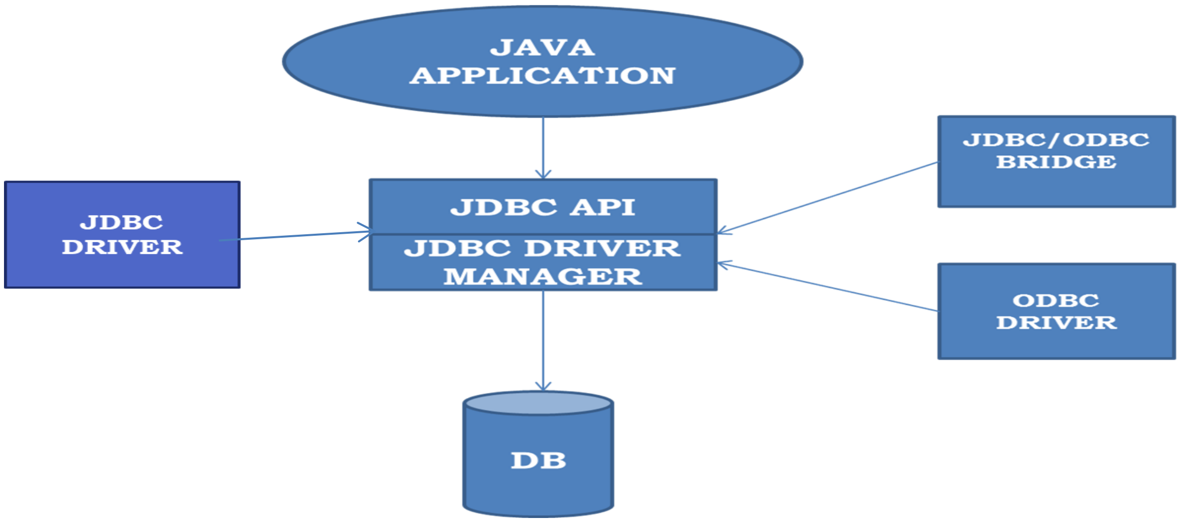
DATABASE ACCESS & AJAX

Introduction:

Every enterprise application is required to access a database either for retrieving the data to be processed (or) to store the processed data. Java offers a simple approach for database connectivity through JDBC using which a java application can be connected virtually to any database.

JDBC is not a s/w product. It is an API which defines interfaces and classes for writing database applications in java. In fact, JDBC is one of the service API's in the J2EE. The java.sql & javax.sql packages provide the necessary library support for the database aware java applications.

JDBC Architecture:



Java Application:

It can be a standalone java program (or) an applet (or) a servlet (or) an EJB, which uses the JDBC API to get connected and perform operations on the data present in database.

JDBC API:

It is a set of classes & interfaces used in a Java Program for database operations. The java.sql & javax.sql packages provide the necessary library support.

Driver Manager:

The primary purpose of the driver manager is to load specific drivers on behalf of user application.

JDBC Driver:

It is a s/w that translates the JDBC method calls into vendor-specific API calls.

ODBC Driver:

ODBC driver is dynamically loaded by the ODBC driver manager for making connection to target database.

Database:

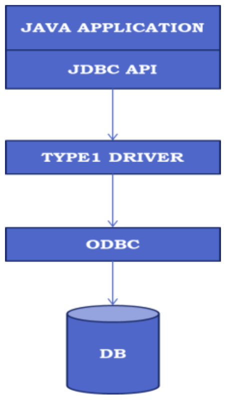
The real-time it is the database server.

The programmer needs a specific driver to connect to a specific database. Driver implementation come in four types:

1. Type 1: JDBC-ODBC Bridge Driver (Bridge)
2. Type 2: Native-API / Partly Java Driver (Native)
3. Type 3: All Java / Net - Protocol Driver (Middleware)
4. Type 4: All Java / Native – Protocol Driver (Pure)

1.Type 1: JDBC – ODBC Bridge Driver (Bridge):

Type 1 driver allows an application to access database through an intermediate ODBC driver. It provides a gateway to the ODBC API, since in purpose is to translate JDBC methods into the ODBC function calls. Here ODBC acts as a mediating layer between the JDBC driver and the vendors (Client libraries).



Advantages:

ODBC drivers are commonly available; hence it can work with huge number of ODBC drivers.

Disadvantages:

1. A ODBC binary code must be loaded on each client machine that uses this driver.

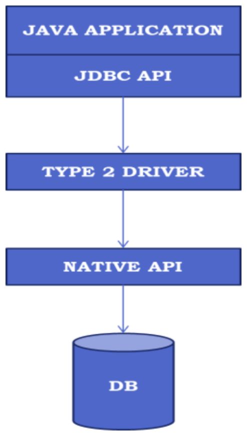
2. Translation overhead between JDBC & ODBC.

3. Does not support all features of java.

4. It works only under Microsoft windows and sun solaris OS.

2. Type 2: Native API / Partly Java Driver (Native):

Type 2 driver converts JDBC calls into data base specific calls for databases. The Type 2 driver communicates directly with database server & therefore requires some binary code to be present on client machine.



Advantages:

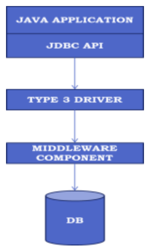
Type 2 driver offers significantly better performance than type 1 driver.

Disadvantages:

The vendor database library needs to be loaded on each client machine consequently. Type 2 drivers cannot be used for internet.

3. Type 3: All Java / Net – Protocol Driver (Middleware):

Type 3 driver translation JDBC calls into the middleware vendor's protocol which is subsequently translated to a DBMS protocol by middleware server. The Middleware provides connectivity to many different databases. The types of drivers are best suited for environment that need to provide connectivity to a variety of DBMS servers and heterogenous databases.



Advantages:

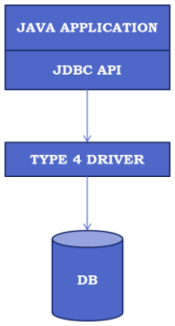
The Type 3 driver is server based. So, there is no need for any vendor database library to be present on client machines.

Disadvantages:

Type 3 drivers require database specific coding be done in the middle tier.

4. Type 4: All Java / Native – Protocol driver (Pure):

Type 4 driver talks directly to the database using Java sources. These type of drivers are completely implemented in java to achieve platform independence and eliminate deployment issues.



Advantages:

Performance is significantly better.

Disadvantages:

The user needs a different driver for each database.

Database Programming using JDBC:

In order to query a database using JDBC we need to follow the below steps:

1. Loading the JDBC driver
2. Defining a connection URL
3. Establishing a connection
4. Creating a statement object
5. Executing a query
6. Process the results
7. Close the Connection

1. Loading the JDBC driver:

To load a driver we specify the class name of the database driver in the class.forName() method. By doing so, we automatically create a driver instance and register it and the JDBC driver manager.

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

2. **Defining the connection URL:**

In JDBC, a connection URL specifies the server host port and database name with which we have to establish the connection

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

3. **Establishing the connection:**

With the connection URL, username and password, a network connection to the database can be established and once the connection is established, database queries can be performed entities the connection is closed.

Connection con=DriverManager.getConnectio(Url,username,password);

4. **Create a statement Object:**

Creating a statement object enables us to bend queries and commands to databases.

Statement st=con.Statement();

5. **Executing a query:**

Given a statement object we can send SQL statements to the database by using execute(), executeQuery(), executeUpdate() or executeBatch() methods.

st.executeQuery("select * from customer");

6. **Process the Results:**

When a database query is executed , a result set object is returned. The Result set represents a set of rows and columns that we can process by using various methods.

```
ResultSet rs=st.executeQuery("select * from customer");
String first_name,last_name;
boolean records=rs.next();
if(!records){
system.out.println("No records reference");
}
else{
do{
first_name=rs.getString("first_name");
last_name=rs.getString("last_name");
system.out.println("First name"+first_name+""+"Last Name"+last_name);
} while(rs.next());
}
```

7. **Close the Connection:**

When we are finished performing queries and processing results, we should close the connection releasing resources to the database.

```
con.close();
```

Accessing the Database from a JSP page:

```
<%@ Page import=" java.sql.* "%>
<html><head><title>Database Access </title></head>
<body bgcolor="pink">
<center><h1>Accessing Data from a Database </h1><br>
<% class.forName("oracle.jdbc.driver.oracleDriver");
String url="jdbc.oracle.thin@localhost:1521:XE";
Connection con=DriverManager.getConnection(url,"nvn","nvn");
Statement st=con.createStatement();
ResultSet rs=st.executeQuery("select * from customer");
%>
<table border="2"><tr><th>Last Name</th>
<th>First Name</th></tr>
<% while(rs.next()){%>
<tr><td><%= rs.getString("lastname"); %></td>
<td><%=rs.getString("firstname");%></td></tr>
<% } %></table></center></body></html>
```

Application – Specific Database Actions:

The various application specific database actions are:

1. Creating a table
2. Inserting data into a table
3. Retrieving the data to be processed
4. Updating the table data

Creating a table:

```
<%@ Page import="java.sql.*" %>
<html><head><title>Creating a table</title></head>
<body bgcolor="pink">
<center><h1>Creating a table</h1><br>
<%try {
Class.forName("jdbc.oracle.driver.OracleDriver");
String url="jdbc:oracle:thin:@localhost:1521:XE";
Connection con=DriverManager.getConnection(url,"nvn","nvn");
Statement st=con.createStatement();
String cmd="create table stafflist(ID(integer), name varchar2(30), Dept varchar2(30), Designation
varchar2(30);";
St.executeUpdate(cmd);
System.out.println("<b> Table created Successfully ");
}
Catch(Exception e){
Out.println("An error occurred");
}
%>
</center></body></html>
```

Inserting Data into a table:

```
<%@ Page import= "java.sql.*" %>
<html><head><title>Inserting Data </h1><br>
<% try {
Class.forName("jdbc.oracle.driver.OracleDriver");
String url="jdbc:oracle:thin:@localhost:1521:XE";
Connection con=DriverManager.getConnection(url,user,pwd);
Statement st=con.createStatement();
String cmd="Insert Into stafflist values(1,'abc','cse',"Asst.post");";
St.executeUpdate("cmd");
out.println("<b> Table data inserted successfully");
}
catch(Exception e){
out.println("An error occurred");
}%></center>
</body></html>
```

Studying javax.sql Package:

The javax.sql package provides the API for the serverside datasource access and processing from the java program.

Interfaces:

1. ConnectionEventListener:

An object that registers to be notified of events generated by a pooled connection object.

2. ConnectionPoolaDataSource:

A factory for pooled connection objects.

3. DataSource:

A factory for connections to the physical data Source that this DataSource represents.

4. PooledConnection:

An object that provides hooks for connection pool management.

5. RowSet:

The interface that adds support to the JDBC API for the JavaBean Component model.

6. Rowset Internal:

The interface that a Rowset object implements in order to present itself to a RowSetReader/RowSetWriter object.

7. RowsetListener:

An interface that must be implemented by a component that wants to be notified when a significant event happens in the life of a RowSet object.

8. RowSetMetaData:

An object that contains information about the columns in a RowSet object.

9. RowSet Reader:

The facility that a disconnected RowSet Object calls on to populate itself with rows of data.

10. RowSet Writer:

An object that implements the RowsetWriter interface called as Writer.

11. XAConnection:

An object that provides support for distributed transactions.

12. XADataSource:

A factory for XAConnection objects that are used internally.

Various Classes defined are:

1. ConnectionEvent:

An event object that provides information about the source of a Connection-related event.

2. RowSet Event:

An event object generated when an event occurs to a RowSet Object.

Javax.sql Package Description:

The javax.sql Package provides an API for Server side data Source access and processing from the java programming language. The package supplements the java.sql package and provides the following:

a. Data Source interface as an alternative to the Driver Manager for established a connection with a data Source.
b. Connection Pooling
c. Distributed transactions
d. RowSets

Applications use the Data Source and the Row Sets directly, out the connection pooling and distributed transactions are used internally by the middle-tier infrastructure.

a) Using the Data Source object for making a connection:

The javax.sql Package provides the preferred way to make a connection to the data source. It offers many advantages such as:

1. Applications do not need to hard code a driver class.
2. Changes can be made to data source properties which means that it is not necessary to make changes in application code when something about the data Source or driver changes.
3. Connection Pooling and distributed transaction are available through a DataSource object that is implements to work with the middle-tier infrastructure.

A Particular Data Source object represents a particular physical Data Source and each connection to Data Source object creates a connection to that physical Data Source.

A logical name for the data source is registered with a naming service that uses JNOI API. An Application can retrieve the Data Source object to create a connection to the physical Data Source it represents.

A Data Source Object can be implemented to work with the middle tier infrastructure so that the connections it provides will be pooled for reuse.

b) Connection Pooling:

Connections made via a Data Source object are implemented to work with a middle tier. Connection Pool Manager participate in connection pooling. Connection Pooling allows a connection to be used and reused thus cursing down substantially the need of making new connections.

Connection Pooling is totally transparent. It is done automatically in the middle tier of a J2EE configuration and so there is no need of change in code from application point of view. An Application simply uses the Data Source.getConnection() method to get the pooled connection and uses it the same way it uses any connection object. The classes are interfaces used for connection pooling are:

(1) Connection Pool DataSource

(2) Pooled Connection

(3) Connection Event

(4) Connection Event Listener

c) Distributed Transactions:

As with pooled connections, connections made via a Data Source objects are implemented to with the middle tier infrastructure may also participate in distributed transactions.

This gives an application the ablility to involve data sources on multiple servers in a single transaction.

d) RowSets:

The Rowset interface works with the various other classes & interfaces. These can be grouped into three categories:

i) Event Notification

ii) Meta data

iii) The Reader / Writer facilities.

Event Notifications:

(a) Rowset Listener:

A Rowset object is a JavaBean is a Java Bean Component and it has participates in the Java Bean event notification mechanism. The Rowset Listener interface is implemented by a component that wants to be notified about the events that occur to a particular Rowset object.

(b) Rowset Event:

A Rowset Object creates an instance of Rowset of Rowset Event and passes it to the listener. The listener can use this Rowset Event object to find which Rowset had the event.

1. <u>Meta Data:</u>

Data about Data (Information about Data)

1. <u>RowSetMetaData:</u>

This interface is derived from the Resultset MetaData and provides information about the columns in a Rowset object. It provides methods for setting information about columns.

2. The Reader / Writer facility:

A Rowset object that implements the Rowset Internal interface can call on the Rowset Reader object associated with it to populate itself with data.

It can also call on the Rowset writer object associated with it to write any changes to its rows back to the datasource from which it originally got the rows.