Database Access

Database Programming using JDBC:

JDBC is an international standard for programming access to SQL databases. It was developed by *JavaSoft*, a subsidiary of ***Sun Microsystems***.

Relational Database Management System supports SQL. As we know that Java is platform independent, JDBC makes it possible to write a single database application that can run on different platforms and interact with different Database Management Systems.

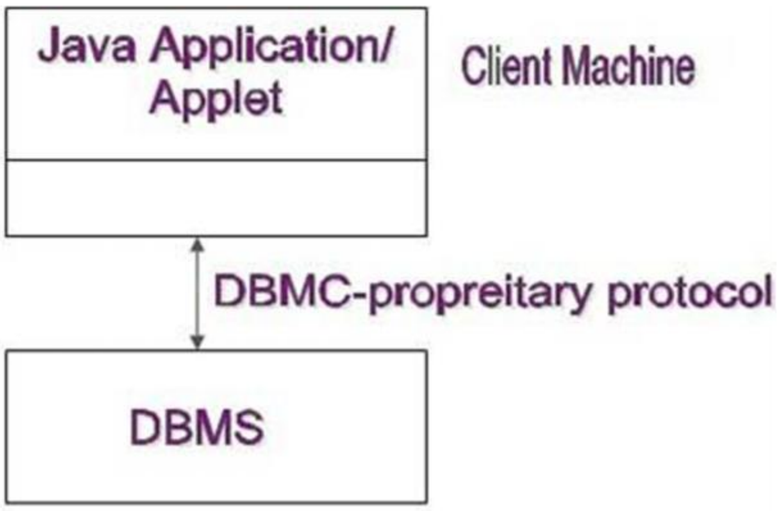
Java Database Connectivity is similar to Open Database Connectivity (ODBC) which is used for accessing and managing database, but the difference is that JDBC is designed specifically for Java programs, whereas ODBC is not dependent upon any language.

JDBC Driver Model

JDBC supports two-tier and three-tier model:

I. Two-tier Model

In this model the Java applets and application are directly connected with any type of database. The client directly communicates with database server through JDBC driver.

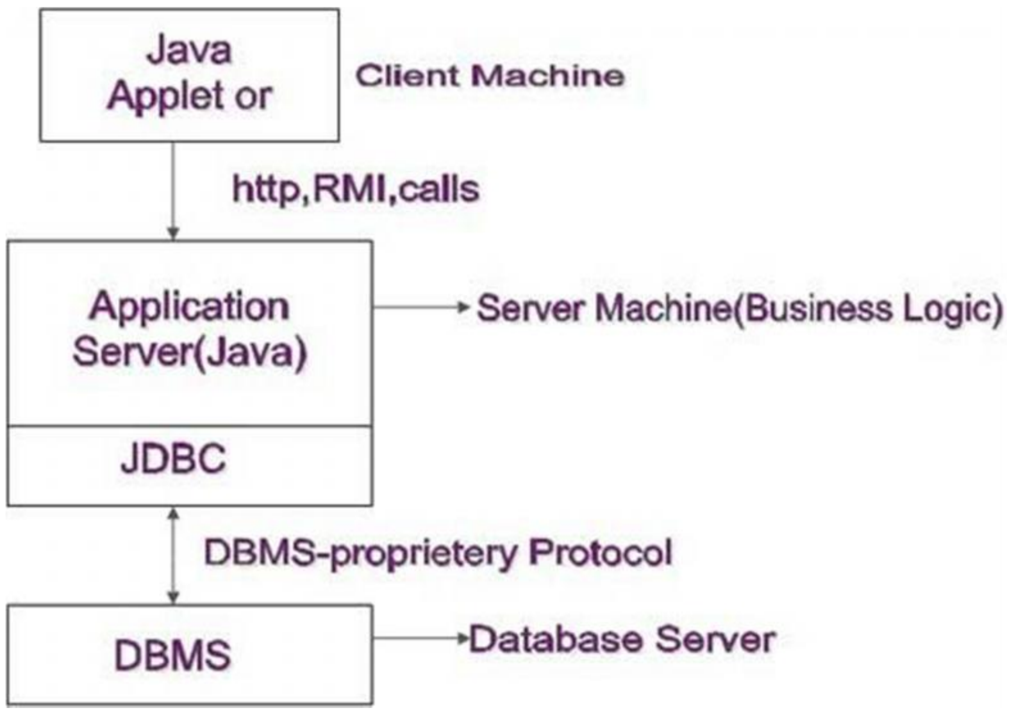


The JDBC ManagerDriver API sends it to the various SQL statements.

As another layer, the Manager should communicate with various third party drivers that actually connect to the database and return information from the query or perform action specified by the query.

II. **Three-tier Model**

In this model, client connects with database server through a middle-tier server, which is used for various purposes. Middle-tier server performs various functions.



It extracts the SQL command from the client and sends these commands to the database server. Also, it extracts the result from the database server and submits the same to the client.

Exploring JDBC Architecture

JDBC is open specification given by Sun Microsystems having rules and guidelines to develop JDBC driver. JDBC driver is a bridge software between Java application and database software. It converts Java call to database call and vice versa. Java application talks with database server using JDBC driver.

- JDBC call converts database call.
- JDBC driver connects to database server and sends command to the database **server.**
- Database executes the statement.
- Sends the output back to JDBC driver.
- JDBC driver sends back to java application.

Each JDBC driver is specific to one database software.

We can get the JDBC driver from three parties:

1. Sun Microsystems
2. Database vendor
3. Third party vendor

It is recommended to use database vendor- supplied JDBC drivers.

Non-Java applications use ODBC drivers to interact with database software.

There are four different methodologies or mechanisms to develop JDBC drivers based on the rules and guidelines of JDBC specification. These are:

i. *Type-1(JDBC-ODBC Bridge Driver)*

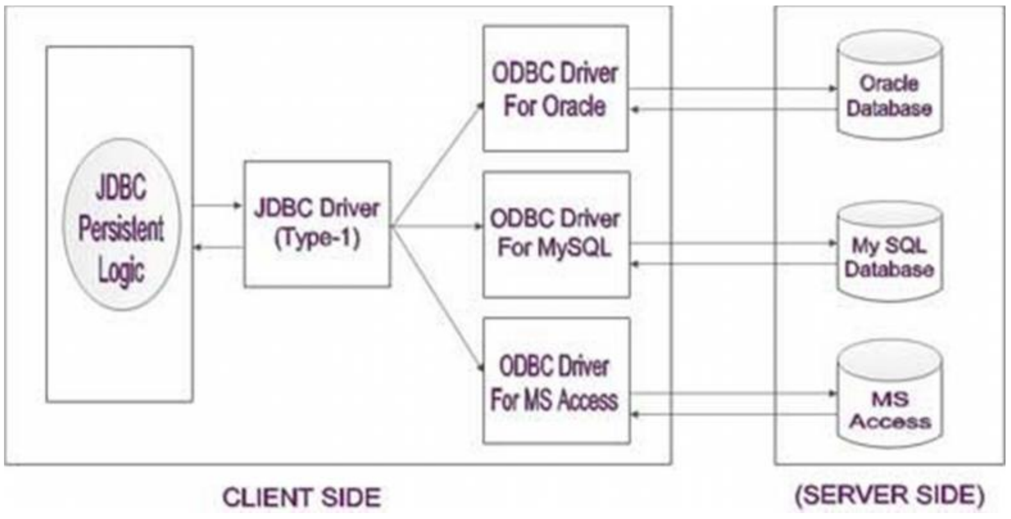
ii. *Type-n Native API/ partly Java Driver)*

iii. *Type-3(Net-Protocol/All Java Driver)*

iv. *Type-4(Native- Protocol/All Java Driver)*

i. Type-l(JDBC-ODBC Bridge Driver)

The type-1 mechanism-based drivers are given drivers to take the support of ODBC drivers while introducing to database software. In this process, the type 1 driver takes support of the native code to communicate with ODBC drivers.



JDBC Type-1 driver is not specific to any database software because it does not interact with database software directly. It interacts with database software by using the database software-specific ODBC driver.

Type-1 driver is supplied only by Sun Microsystems. It has a built-in JDK software. JDK software supplies a basic and built-in service called DriverManager which serves to manage a set of JDBC drivers and to establish the connection with database software by using JDBC driver.

Type-1 driver class name is: *sun.jdbc.odbc.JdbcOdbcDriver*

Every JDBC driver must be registered with DriverManager service, as this creates JDBC class object in DriverManager service.

Advantages of JDBC-ODBC Bridge Driver

• Serves as a single driver that can be used to interact with different data stores.

• Allows you to communicate with all the databases supported by the ODBC driver.

• Represents a vendor-independent driver and is available with JDK

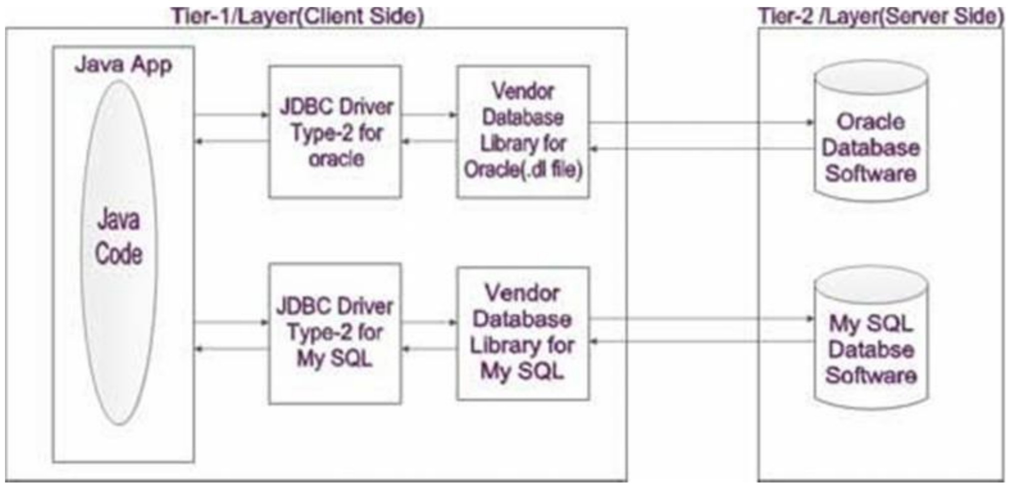
Disadvantages of JDBC-ODBC Bridge Driver

• Decreases the execution speed due to more number of transactions. (Include JDBC ODBC DB Native call)

• Depends on the ODBC driver due to which Java application indirectly becomes dependent on ODBC drivers.

ii. Type-2(JAVA to Native API)

Type-2 driver converts JDBC calls in a client machine. It uses native code to communicate with vendor database library.


Tier-1/Layer(Client Side)
Tier-2 /Layer(Server Side)
Java App
Java Code
JDBC Driver Type-2 for oracle
Vendor Database Library for Oracle(.dl file)
Oracle Database Software
JDBC Driver Type-2 for My SQL
Vendor Database Library for My SQL
My SQL Databse Software

Type-2 JDBC driver takes the support of vendor database software. In the figure, the java application is programmed using JDBC API, making JDBC calls. These JDBC calls are then converted into database specific native calls and the request is then dispatched to the database specific native libraries.

Type-2 drivers are suitable to use with server-side application. It is not recommended to use type-2 drivers with client-side application since native libraries for the client platform should be installed on the client machines.

Studying Javax.sql.* package:

1. **Loading the Driver**
   To begin with, you first need load the driver or register it before using it in the program . Registration is to be done once in your program. You can register a driver in one of two ways mentioned below :

- **Class.forName() :** Here we load the driver's class file into memory at the runtime. No need of using new or creation of object .The following example uses Class.forName() to load the Oracle driver –

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

- **DriverManager.registerDriver():** DriverManager is a Java inbuilt class with a static member register. Here we call the constructor of the driver class at compile time . The following example uses DriverManager.registerDriver()to register the Oracle driver

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver())
```

2. **Create the connections**
   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.

3. **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 .

4. 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

5.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.

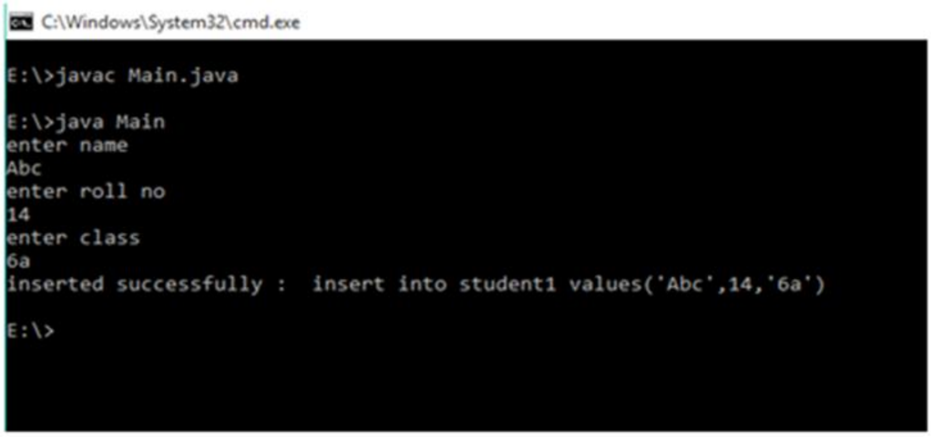
Example :

```
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);
}
```

}

}



```
C:\Windows\System32\cmd.exe

E:\>javac Main.java

E:\>java Main
enter name
Abc
enter roll no
14
enter class
6a
inserted successfully :  insert into student1 values('Abc',14,'6a')

E:\>
```

Accessing a Database from a JSP Page:

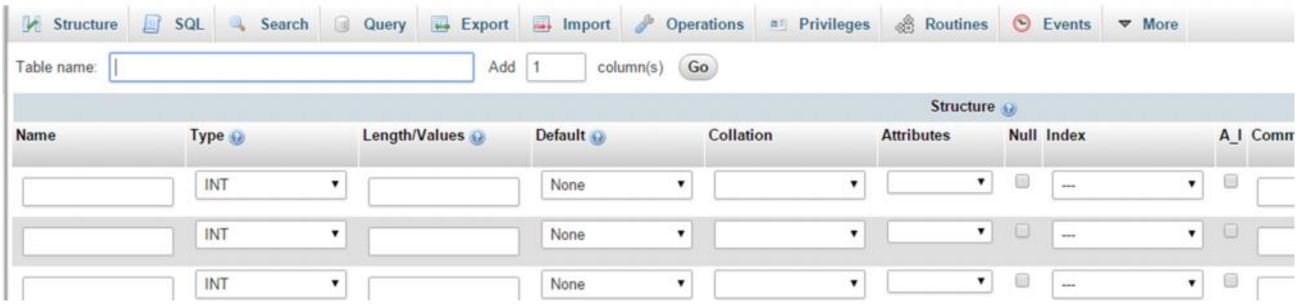
The database is used for storing various types of data which are huge and has storing capacity in gigabytes. JSP can connect with such databases to create and manage the records.

In this tutorial, we will learn about how to create a table in the database, and how to create records in these tables through JSP.

Create Table

In MYSQL database, we can create a table in the database with any MYSQL client.

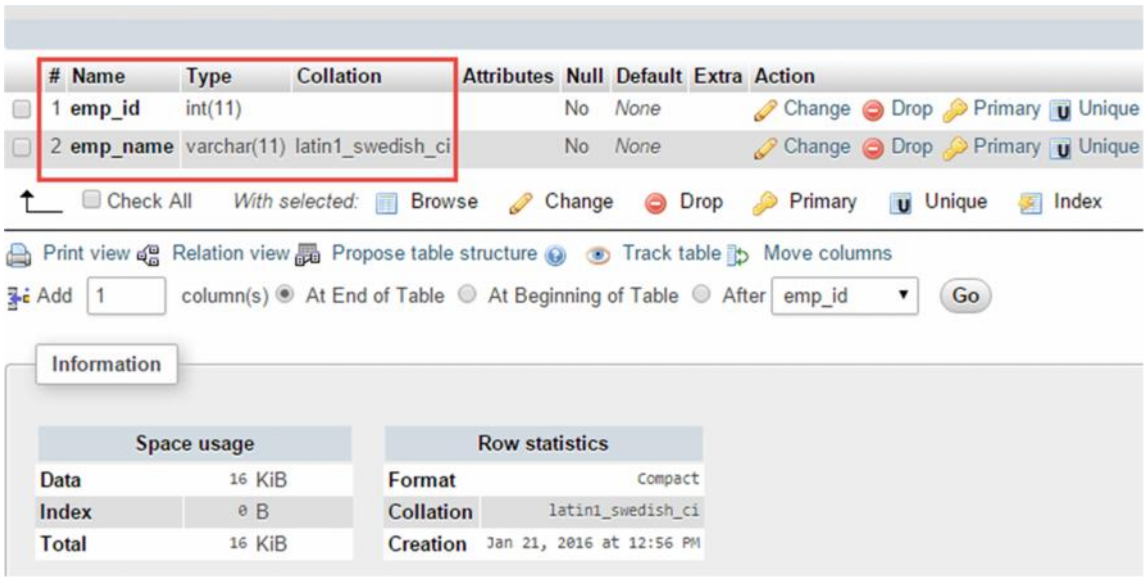
Here we are using PHPMyadminclient, and there we have an option "new" to create a new table using below screenshot.



In this, we have to provide table name as guru_test, and we will create two fields'emp_id and emp_name.

Emp_idis havingdatatype as int

Emp_nameis havingdatatype as varchar



#	Name	Type	Collation	Attributes	Null	Default	Extra	Action
1	**emp_id**	int(11)			No	*None*		Change Drop Primary Unique
2	**emp_name**	varchar(11)	latin1_swedish_ci		No	*None*		Change Drop Primary Unique

Space usage	
Data	16 KiB
Index	0 B
Total	16 KiB

Row statistics	
Format	Compact
Collation	latin1_swedish_ci
Creation	Jan 21, 2016 at 12:56 PM

Application – Specific Database Actions Deploying JAVA Beans in a JSP Page:

A JavaBean is a specially constructed Java class written in the Java and coded according to the JavaBeans API specifications.

Following are the unique characteristics that distinguish a JavaBean from other Java classes −

- It provides a default, no-argument constructor.
- It should be serializable and that which can implement the **Serializable** interface.
- It may have a number of properties which can be read or written.
- It may have a number of "**getter**" and "**setter**" methods for the properties.

JavaBeans Properties

A JavaBean property is a named attribute that can be accessed by the user of the object. The attribute can be of any Java data type, including the classes that you define.

A JavaBean property may be **read, write, read only**, or **write only**. JavaBean properties are accessed through two methods in the JavaBean's implementation class

S.No.	Method & Description
1	get**PropertyName**() For example, if property name is *firstName*, your method name would be **getFirstName()** to read that property. This method is called accessor.
2	set**PropertyName**() For example, if property name is *firstName*, your method name would be **setFirstName()** to write that property. This method is called mutator.

A read-only attribute will have only a **getPropertyName()** method, and a write-only attribute will have only a **setPropertyName()** method.

JavaBeans Example

Consider a student class with few properties −

```
package sample;

public class StudentsBean implements java.io.Serializable {
   private String firstName = null;
   private String lastName = null;
   private int age = 0;

   public StudentsBean() {
   }
   public String getFirstName(){
      return firstName;
   }
   public String getLastName(){
      return lastName;
   }
   public int getAge(){
      return age;
   }
   public void setFirstName(String firstName){
      this.firstName = firstName;
   }
   public void setLastName(String lastName){
      this.lastName = lastName;
   }
   public void setAge(Integer age){
      this.age = age;
   }
}
```

Accessing JavaBeans

The **useBean** action declares a JavaBean for use in a JSP. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the JSP. The full syntax for the useBean tag is as follows −

<jsp:useBean id = "bean's name" scope = "bean's scope" typeSpec/>

Here values for the scope attribute can be a **page, request, session** or **application based** on your requirement. The value of the **id** attribute may be any value as a long as it is a unique name among other **useBean declarations** in the same JSP.

Following example shows how to use the useBean action −

```
<html>
   <head>
      <title>useBean Example</title>
   </head>
   
   <body>
      <jsp:useBean id = "date" class = "java.util.Date" />
      <p>The date/time is <%= date %>
   </body>
</html>
```

You will receive the following result

Accessing JavaBeans Properties

Along with **<jsp:useBean...>** action, you can use the **<jsp:getProperty/>** action to access the get methods and the **<jsp:setProperty/>** action to access the set methods. Here is the full syntax −

```
<jsp:useBean id = "id" class = "bean's class" scope = "bean's scope">
   <jsp:setProperty name = "bean's id" property = "property name"
      value = "value"/>
   <jsp:getProperty name = "bean's id" property = "property name"/>
   ...........
</jsp:useBean>
```

The name attribute references the id of a JavaBean previously introduced to the JSP by the useBean action. The property attribute is the name of the **get** or the **set** methods that should be invoked.

Following example shows how to access the data using the above syntax −

```
<html>
  <head>
    <title>get and set properties Example</title>
  </head>

  <body>
    <jsp:useBean id = "students" class = "com.google.StudentsBean">
      <jsp:setProperty name = "students" property = "firstName" value = "jntu"/>
      <jsp:setProperty name = "students" property = "lastName" value = "vzm"/>
      <jsp:setProperty name = "students" property = "age" value = "10"/>
    </jsp:useBean>

    <p>Student First Name:
      <jsp:getProperty name = "students" property = "firstName"/>
    </p>

    <p>Student Last Name:
      <jsp:getProperty name = "students" property = "lastName"/>
    </p>

    <p>Student Age:
      <jsp:getProperty name = "students" property = "age"/>
    </p>

  </body>
</html>
```

Let us make the **StudentsBean.class** available in CLASSPATH. Access the above JSP. the following result will be displayed −

```
Student First Name: jntu

Student Last Name: vzm

Student Age: 10
```