

# Unit –5

## Database Access

### 5.1 Database Programming using JDBC:

JDBC is a Java language API that provides access to databases which utilize the Structured Query Language (SQL). SQL DB applications can be entirely written in Java. The kinds of JDBC drivers. JDBC-ODBC bridge drivers are useful for small business applications. ODBC (Open Database Connectivity) is a technology developed by Microsoft to allow generic access to a DBMS on Windows platform.

#### JDBC Driver Types:

1. JDBC-ODBC bridge driver
2. Native-API driver
3. Network Protocol driver
4. Thin driver

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)

#### 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. This is now discouraged because of thin driver.

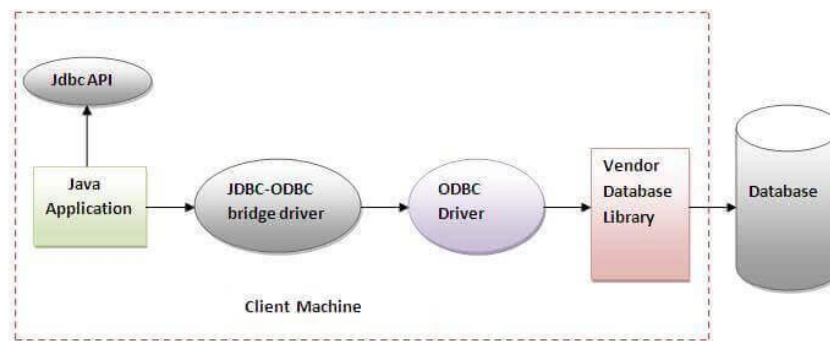


Figure- JDBC-ODBC Bridge Driver

### ***In Java 8, the JDBC-ODBC Bridge has been removed.***

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

#### **Advantages:**

- easy to use.
- can be easily connected to any database.

#### **Disadvantages:**

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

## **2) 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.

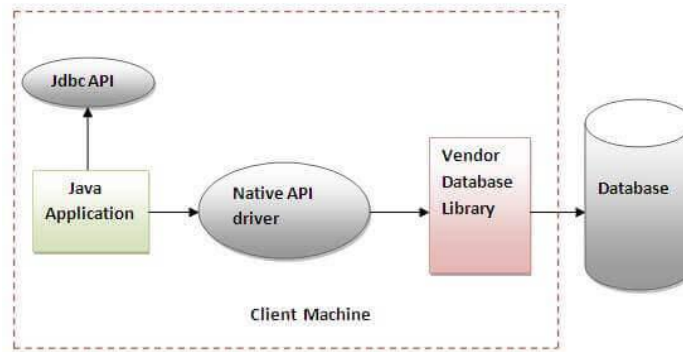


Figure- Native API Driver

### Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

### Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

## 3) 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.

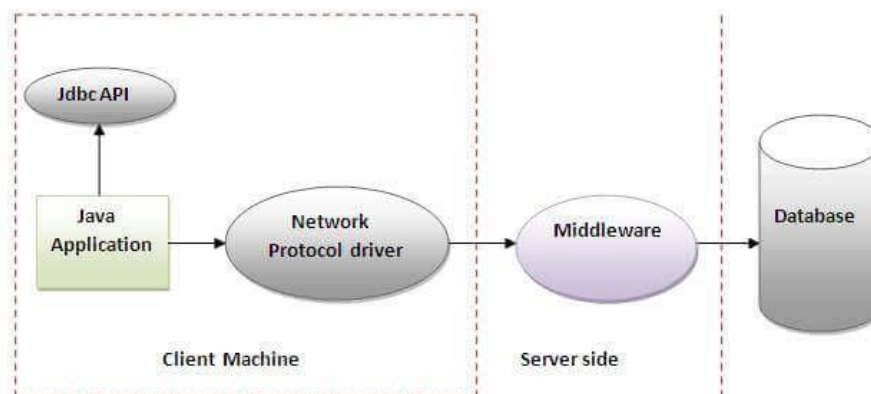


Figure- Network Protocol Driver

### Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

### Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

## 4) 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.

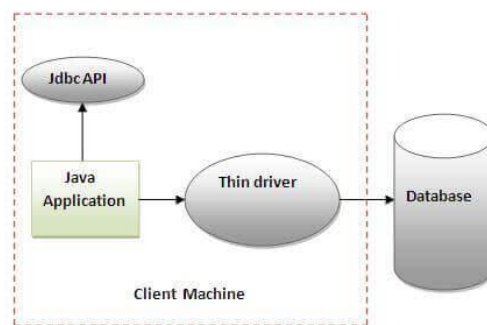


Figure- Thin Driver

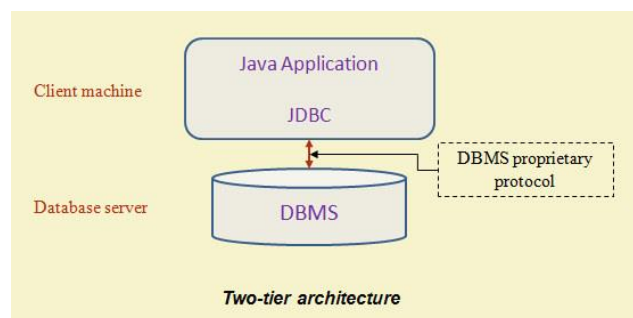
### Advantage:

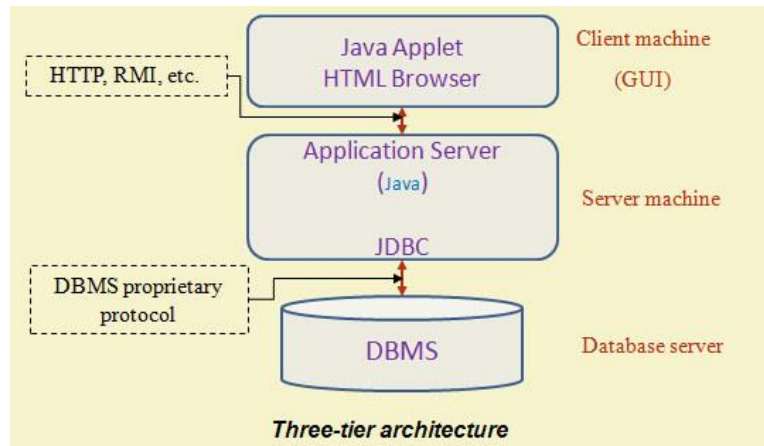
- Better performance than all other drivers.
- No software is required at client side or server side.

### Disadvantage:

- Drivers depend on the Database.

## JDBC Architecture





Driver selection is based on the architecture of the application (Two-tier vs Three-tier) and tradeoffs of speed, portability and reliability

## 5.2 Studying javax.sql package:

Javax.sql package provides the API for server-side data source access and processing from the Java programming language. This package supplements the java.sql package and, as of the version 1.4 release, is included in the Java™ 2 SDK, Standard Edition. It remains an essential part of the Java 2 SDK, Enterprise Edition (J2EETM).

### Interfaces in package javax.sql

<b>ConnectionEventListener</b>	An object that registers to be notified of events generated by a PooledConnection object.
<b>ConnectionPoolDataSource</b>	A factory for PooledConnection objects.
<b>DataSource</b>	A factory for connections to the physical data source that this DataSource object represents.
<b>PooledConnection</b>	An object that provides hooks for connection pool management.

<b>RowSet</b>	The interface that adds support to the JDBC API for the JavaBeans™ component model.
<b>RowSetInternal</b>	The interface that a RowSet object implements in order to present itself to a RowSetReader or RowSetWriter object.
<b>RowSetListener</b>	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.
<b>RowSetMetaData</b>	An object that contains information about the columns in a RowSet object.
<b>RowSetReader</b>	The facility that a disconnected RowSet object calls on to populate itself with rows of data.
<b>RowSetWriter</b>	An object that implements the RowSetWriter interface, called a writer.
<b>XAConnection</b>	An object that provides support for distributed transactions.
<b>XADataSource</b>	A factory for XAConnection objects that is used internally.

### Classes in package javax.sql

<b>ConnectionEvent</b>	An Event object that provides information about the source of a connection-related event.
<b>RowSetEvent</b>	An Event object generated when an event occurs to a RowSet object.

The javax.sql package provides for the following:

1. The DataSource interface as an alternative to the DriverManager for establishing a connection with a data source.
2. Connection pooling.
3. Distributed transactions.
4. Rowsets.

Applications use the DataSource and RowSet APIs directly, but the connection pooling and distributed transaction APIs are used internally by the middle-tier infrastructure.

### Using a DataSource Object to Make a Connection

The `javax.sql` package provides the preferred way to make a connection with a data source. The `DriverManager` class, the original mechanism, is still valid, and code using it will continue to run. However, the newer `DataSource` mechanism is preferred because it offers many advantages over the `DriverManager` mechanism.

These are the main advantages of using a `DataSource` object to make a connection:

5.2.1 Applications do not need to hard code a driver class.

5.2.2 Changes can be made to a data source's properties, which mean that it is not necessary to make changes in application code when something about the data source or driver changes.

5.2.3 Connection pooling and distributed transactions are available through a `DataSource` object that is implemented to work with the middle-tier infrastructure. Connections made through the `DriverManager` do not have connection pooling or distributed transaction capabilities.

Driver vendors provide `DataSource` implementations. A particular `DataSource` object represents a particular physical data source, and each connection the `DataSource` object creates is a connection to that physical data source.

A logical name for the data source is registered with a naming service that uses the Java Naming and Directory Interface™ (JNDI) API, usually by a system administrator or someone performing the duties of a system administrator. An application can retrieve the `DataSource` object it wants by doing a lookup on the logical name that has been registered for it. The application can then use the `DataSource` object to create a connection to the physical data source it represents.

A `DataSource` object can be implemented to work with the middle tier infrastructure so that the connections it produces will be pooled for reuse. An application that uses such a `DataSource` implementation will automatically get a connection that participates in connection pooling. A `DataSource` object can also be implemented to work with the middle tier infrastructure so that the connections it produces can be used for distributed transactions without any special coding.

### **Connection Pooling:**

Connections made via a `DataSource` object that is implemented to work with a middle tier connection pool manager will participate in connection pooling. This can improve performance dramatically because creating new connections is very expensive. Connection pooling allows a connection to be used and reused, thus cutting down substantially on the number of new connections that need to be created.

Connection pooling is totally transparent. It is done automatically in the middle tier of a J2EE configuration, so from an application's viewpoint, no change in code is required. An application simply uses the `DataSource.getConnection` method to get the pooled connection and uses it the same way it uses any

Connection object.

The classes and interfaces used for connection pooling are:

- ✓ `ConnectionPoolDataSource`
- ✓ `PooledConnection`
- ✓ `ConnectionEvent`
- ✓ `ConnectionEventListener`

The connection pool manager, a facility in the middle tier of a three-tier architecture, uses these classes and interfaces behind the scenes. When a `ConnectionPoolDataSource`

object is called on to create a `PooledConnection` object, the connection pool manager will register as a `ConnectionEventListener` object with the new `PooledConnection` object. When the connection is closed or there is an error, the connection pool manager gets a notification that includes a `ConnectionEvent` object.

### **Distributed Transactions:**

As with pooled connections, connections made via a `DataSource` object that is implemented to work with the middle tier infrastructure may participate in distributed transactions. This gives an application the ability to involve data sources on multiple servers in a single transaction.

The classes and interfaces used for distributed transactions are:

- ✓ `XADataSource`
- ✓ `XAConnection`

These interfaces are used by the transaction manager; an application does not use them directly.

The `XAConnection` interface is derived from the `PooledConnection` interface, so what applies to a pooled connection also applies to a connection that is part of a distributed transaction. A transaction manager in the middle tier handles everything transparently. The only change in application code is that an application cannot do anything that would interfere with the transaction manager's handling of the transaction. Specifically, an application cannot call the methods `Connection.commit` or `Connection.rollback`, and it cannot set the connection to be in auto-commit mode (that is, it cannot call `Connection.setAutoCommit(true)`).

An application does not need to do anything special to participate in a distributed transaction. It simply creates connections to the data sources it wants to use via the `DataSource.getConnection` method, just as it normally does. The transaction manager manages the transaction behind the scenes. The `XADataSource`



interface creates XAConnection objects, and each XAConnection object creates an XAResource object that the transaction manager uses to manage the connection.

Rowsets:

The RowSet interface works with various other classes and interfaces behind the scenes. These can be grouped into three categories.

1. Event Notification

✓ RowSetListener

A RowSet object is a JavaBeans<sup>TM</sup> component because it has properties and participates in the JavaBeans event notification mechanism. The RowSetListener interface is implemented by a component that wants to be notified about events that occur to a particular RowSet object. Such a component registers itself as a listener with a rowset via the RowSet.addRowSetListener method.

When the RowSet object changes one of its rows, changes all of its rows, or moves its cursor, it also notifies each listener that is registered with it. The listener reacts by carrying out its implementation of the notification method called on it.

✓ RowSetEvent

As part of its internal notification process, a RowSet object creates an instance of RowSetEvent and passes it to the listener. The listener can use this RowSetEvent object to find out which rowset had the event.

2. Metadata

✓ RowSetMetaData

This interface, derived from the ResultSetMetaData interface, provides information about the columns in a RowSet object. An application can use RowSetMetaData methods to find out how many columns the rowset contains and what kind of data each column can contain.

The RowSetMetaData interface provides methods for setting the information about columns, but an application would not normally use these methods. When an application calls the RowSet method execute, the RowSet object will contain a new set of rows, and its RowSetMetaData object will have been internally updated to contain information about the new columns.

3. The Reader/Writer Facility

A RowSet object that implements the RowSetInternal interface can call on the RowSetReader object associated with it to populate itself with data. It can also call on the RowSetWriter object associated with it to write any changes to its rows back to the data source from which it originally got the rows. A rowset that

remains connected to its data source does not need to use a reader and writer because it can simply operate on the data source directly.

#### ✓ RowSetInternal

By implementing the RowSetInternal interface, a RowSet object gets access to its internal state and is able to call on its reader and writer. A rowset keeps track of the values in its current rows and of the values that immediately preceded the current ones, referred to as the original values. A rowset also keeps track of (1) the parameters that have been set for its command and (2) the connection that was passed to it, if any. A rowset uses the RowSetInternal methods behind the scenes to get access to this information. An application does not normally invoke these methods directly.

#### ✓ RowSetReader

A disconnected RowSet object that has implemented the RowSetInternal interface can call on its reader (the RowSetReader object associated with it) to populate it with data. When an application calls the RowSet.execute method, that method calls on the rowset's reader to do much of the work. Implementations can vary widely, but generally a reader makes a connection to the data source, reads data from the data source and populates the rowset with it, and closes the connection. A reader may also update the RowSetMetaData object for its rowset. The rowset's internal state is also updated, either by the reader or directly by the method RowSet.execute.

#### ✓ RowSetWriter

A disconnected RowSet object that has implemented the RowSetInternal interface can call on its writer to write changes back to the underlying data source. Implementations may vary widely, but generally, a writer will do the following:

- Make a connection to the data source
- Check to see whether there is a conflict, that is, whether a value that has been changed in the rowset has also been changed in the data source
- Write the new values to the data source if there is no conflict
- Close the connection

The RowSet interface may be implemented in any number of ways, and anyone may write an implementation. Developers are encouraged to use their imaginations in coming up with new ways to use rowsets.

## 5.3 Steps required to Access DB

### Steps For Connectivity Between Java Program and Database

1. Import the Packages
2. Load the drivers using the forName() method
3. Register the drivers using DriverManager
4. Establish a connection using the Connection class object
5. Create a statement
6. Execute the query
7. Close the connections

Let us discuss these steps in brief before implementing by writing suitable code to illustrate connectivity steps for JDBC/

#### **Step 1: Import the Packages**

#### **Step 2: Loading the drivers**

In order to begin with, you first need to 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 as follows:

##### 2-A Class.forName()

Here we load the driver's class file into memory at the runtime. No need of using new or create objects. The following example uses Class.forName() to load the Oracle driver as shown below as follows:

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

##### 2-B 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 as shown below:

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

#### **Step 3: Establish a connection using the Connection class object**

After loading the driver, establish connections as shown below as follows:

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

- user: Username from which your SQL command prompt can be accessed.

- password: password from which the SQL command prompt can be accessed.
- con: It is a reference to the Connection interface.
- Url: Uniform Resource Locator which is created as shown below:

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 a 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 the programmer before calling the function. Use of this can be referred to from the final code.

#### **Step 4: 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();
```

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

#### **Step 5: Execute the query**

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

- The query for updating/inserting a table in a database.
- The query for retrieving data.

The executeQuery() method of the 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 of the Statement interface is used to execute queries of updating/inserting.

Pseudo Code:

```
int m = st.executeUpdate(sql);
if (m==1)
    System.out.println("inserted successfully : "+sql);
else
    System.out.println("insertion failed");
```

## 5.4 PreparedStatement [5 Marks]

JDBC API provides 3 different interfaces to execute different SQL Queries. They are:

- 1) Statement: Statement interface is used to execute normal SQL Queries.
- 2) PreparedStatement: It is used to execute dynamic or parametrized SQL Queries.
- 3) CallableStatement: It is used to execute the Stored Procedure.

In this , we will discuss the differences between Statement vs PreparedStatement vs CallableStatement in detail.

### 1) Statement:

In JDBC Statement is an Interface. By using Statement object we can send our SQL Query to Database. At the time of creating a Statement object, we are not required to provide any Query. Statement object can work only for static query.

Whenever we are using execute() method, every time Query will be compiled and executed. Because Query will be compiled every time, its performance is low. Best choice for Statement object, if you want to work with multiple queries.

### Example:

```
import java.sql.*;
import java.util.*;
class Example{
public static void main(String args[])throws Exception{
class.forName("com.mysql.jdbc.Driver");
System.out.println("Driver loaded");
Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/emp_record","root"," ");
System.out.println("Connection Established");
Statement st = con.createStatement();
```

```

ResultSet rs = st.executeQuery("select * from traning");
System.out.println("Rollno Student_Name Stream Percentage");
System.out.println("-----");
while(rs.next()){
System.out.println(rs.getInt(1)+ " " +rs.getString(2)+ " " +rs.getString(3)+ " "+rs.getFloat(4));
}
}
}

```

## 2) PreparedStatement:

PreparedStatement is an interface, which is available in java.mysql package. It extends the Statement interface.

Benefits of Prepared Statement:

It can be used to execute dynamic and parametrized SQL Query.

PreparedStatement is faster than Statement interface. Because in Statement Query will be compiled and execute every time, while in case of Prepared Statement Query won't be compiled every time just executed.

It can be used for both static and dynamic query.

In case of Prepared Statement no chance of SQL Injection attack. It is some kind of problem in database programming.

Suppose, I have an SQL Query. In this SQL Query, we have to use username and password. This query is checking username and password is valid or not. Because the end user provided input the query behavior is changing, it is not checking username and password is valid or not. If you change the behavior of the SQL query by adding special character in end user provided input this problem is known as SQL Injection attack.

### Example:

```

import java.sql.*;
import java.util.*;
class Example{
public static void main(String args[])throws Exception{

```

```

class.forName("com.mysql.jdbc.Driver");
Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/emp_record","root","");
String sqlquery = "insert into employee values(? ? ? ?)";
PreparedStatement pst = con.prepareStatement(sqlquery);
Scanner sc = new Scanner(System.in);
while(true){
System.out.println("Enter employee number: ")
int eno = sc.nextInt();
System.out.println("Enter employee name: ")
String ename = sc.next();
System.out.println("Enter employee salary: ")
double esal = sc.nextDouble();
System.out.println("Enter employee address: ")
String eaadr = sc.next();

pst.setInt(1,eno);
pst.setString(2,ename);
pst.setDouble(3, esal);
pst.setString(4, eaadr);
pst.executeUpdate();
System.out.println("Record inserted successfully ");
System.out.println("Do you want to insert more records[yes/no]");
String option = sc.next();
if(option.equalsIgnoreCase("no")){
break;
}
}
}
}

```

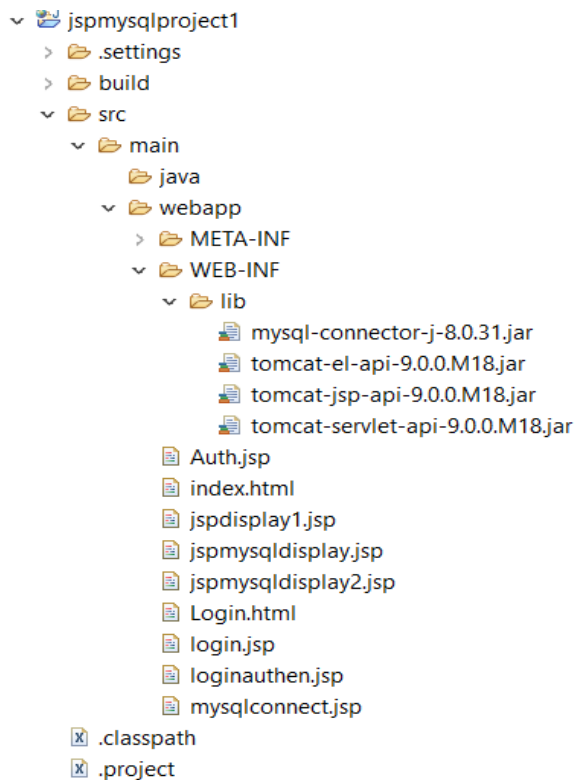
### 3) CallableStatement:

CallableStatement in JDBC is an interface present in a java.sql package and it is the child interface of Prepared Statement. Callable Statement is used to execute the Stored Procedure and functions. Similarly to method stored procedure has its own parameters. Stored Procedure has 3 types of parameters.

- 1) IN parameter: IN parameter is used to provide input values.
- 2) OUT parameter: OUT parameter is used to collect output values.
- 3) INOUT parameter: It is used to provide input and to collect output values.

The driver software vendor is responsible for providing the implementations for Callable Statement interface. If Stored Procedure has OUT parameter then to hold that output value we should register every OUT parameter by using registerOutParameter() method of CallableStatement. CallableStatement interface is better than Statement and PreparedStatement because it calls the stored procedure which is already compiled and stored in the database.

## 5.5 Accessing a Database from a JSP page





## Example Programs:

**Write a program to display the employee details in the table format that are there in the oracle from the emp table.[10 Marks 2020]**

**<!-- GetEmpDetails.jsp -->**

```
<%@ page import="java.sql.*"%>
<html>
<head>
<title>Employee Details</title>
</head>
<body>
    <h2 style='background-color: blue; color: white; text-align: center;'>
        Employee Details</h2>
    <%
        try {
            System.out.println("Getting Connection    ");
            Class.forName("com.mysql.jdbc.Driver");
            Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/gri
et", "root", "root");
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("select
username,password,age,address,email,phoneno from personal");
        }%>
        <table width=100% bgcolor=pink border=2 bordercolor=blue
cellspacing=0>
            <tr>
                <th>username</th>
                <th>password</th>
                <th>age</th>
                <th>address</th>
                <th>email</th>
                <th>phoneno</th>
```

```

</tr>
<%
while (rs.next()) {
%>
<tr>
    <td><%=rs.getString(1)%></td>
    <td><%=rs.getString(2)%></td>
    <td><%=rs.getString(3)%></td>
    <td><%=rs.getString(4)%></td>
    <td><%=rs.getString(5)%></td>
    <td><%=rs.getString(6)%></td>

</tr>
<%
}
out.println("</table>");
rs.close();
stmt.close();
} catch (Exception ex) {
out.println("<br> Error: " + ex);
}
%>

```

```

</body>
</html>

```

Output:

Employee Details					
username	password	age	address	email	phoneno
null	123	25	hyd	null	21234
null	123	25	hyd	null	21234
subbu1	1234	25	hyd	sr@gmail.com	21234
subbu1	1234	25	hyd	null	21234
subbu1	12345	25	hyd	asd@gmail.com	21234
tushar	9890	25	hyd	hydasd123@gmail.com	21234
tushar1	9890	25	hyd	hydasd123@gmail.com	21234
bhuvan	1234	25	hyd	bhuvan123@gmail.com	21234
swathi	1234	25	hyd	swathi@gmail.com	21234
sumanaa	1234	25	hyd	sumana123@gmail.com	21234
hanisha	123	25	hyd	hanisha123@gmail.com	21234
shiva	12345	25	hyd	abc@gmail.com	21234
chaitu	1234	25	hyd	asd@gmail.com	21234
vikas	1234	25	hyd	vikas123@gmail.com	21234
deekshitha	1234	25	hyd	deekshitha123@gmail.com	21234

## Example Program 2:(html+jsp)+mysql

**Display the Students details from the student table there in the database.[10 Marks]**

```
<!-- GetStuDetails.jsp -->
<%@ page import="java.sql.*"%>
<html>
<head>
<title>Student Details</title>
</head>
<body>
    <h2 style='background-color: pink; color: black; text-align: center'>
        Student Details</h2>
    <%
        try {

            System.out.println("Getting Connection      ");
            Class.forName("com.mysql.jdbc.Driver");
            Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/ana
nthula_movies", "root", "root");
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("select * from
student");
            %>
            <table width=75% bgcolor=cream border=2 align="center"
bordercolor=blue
            <u>cellspacing=0</u>
            <tr>
                <th>Student NO.</th>
                <th>Student NAME</th>
                <th>FEE</th>
            </tr>
            <%
                while (rs.next()) {
                    %>
```

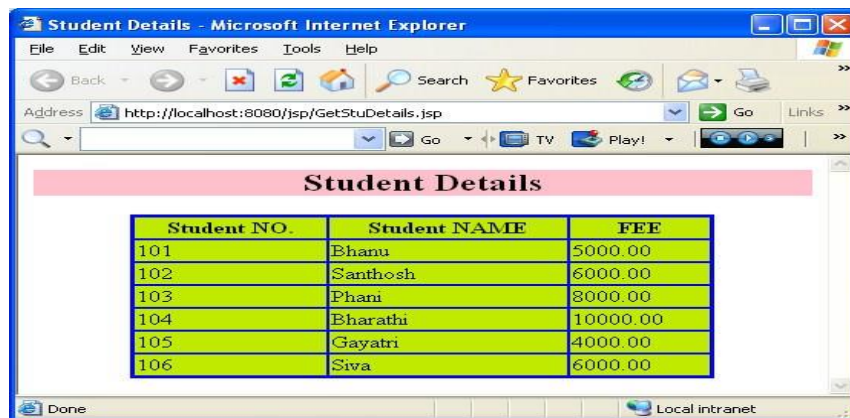
```

        <tr>
            <td><%=rs.getString(1)%></td>
            <td><%=rs.getString(2)%></td>
            <td><%=rs.getString(3)%></td>
        </tr>
    <%
    }
    out.println("</table>");
    rs.close();
    stmt.close();
    } catch (Exception ex) {
    out.println("<br> Error: " + ex);
    }
    %>

</body>
</html>

```

Output:



The screenshot shows a web browser window titled "Student Details - Microsoft Internet Explorer". The address bar shows "http://localhost:8080/jsp/GetStuDetails.jsp". The page content features a pink header bar with the text "Student Details". Below this is a table with three columns: "Student NO.", "Student NAME", and "FEE". The table contains six rows of data.

Student NO.	Student NAME	FEE
101	Bhanu	5000.00
102	Santhosh	6000.00
103	Phani	8000.00
104	Bharathi	10000.00
105	Gayatri	4000.00
106	Siva	6000.00

### Example Program3: html+jsp+mysql

**Write a html file to accept an student number and send that number to the jsp program, the jsp program receives the student number and send the corresponding student details back to Web Browser if that student is available.[10 Marks 2018]**

```
<!-- FindStu.html -->
```

```

<html>
<body bgcolor=brown text=white>
    <h2>Find Student Details</h2>
    <form method=GET action='FindStudent.jsp'>
        Enter Student Number :: <input type=text name=sno
size=15> <br>
        <br> <input type=submit value='Find Student
Record'>
    </form>
</body>
</html>

```

```

<!-- FindStudent.jsp -->
<%@ page import="java.sql.*"%>
<html>
<body>
    <h2>Student Details</h2>
    <%
    try {
        String sno = request.getParameter("sno");
        Class.forName("com.mysql.jdbc.Driver");
        Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/gri
et", "root", "root");
        PreparedStatement ps = con.prepareStatement("select
sno,sname,fee from student where sno=?");
        ps.setString(1, sno);
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            %>
            <h3>Record Found</h3>
            <br> Student Number:
            <b> <%=rs.getString(1)%>
            </b>
            <br> Student Name :
            <b> <%=rs.getString(2)%>
            </b>
            <br> Student Fee :

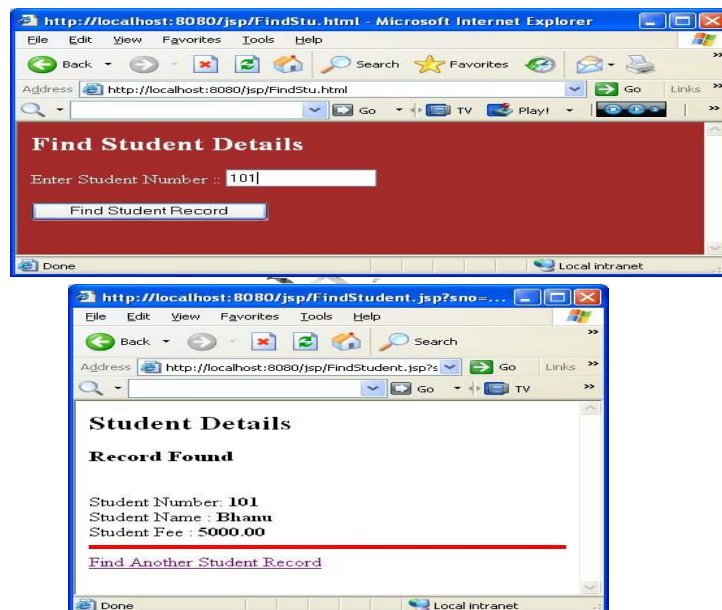
```

```

<b> <%=rs.getString(3)%>
</b>
<%
} else {
out.println("<h3> Record Not Found </h3>");
}
rs.close();
ps.close();
con.close();
} catch (Exception ex) {
out.println("Error: " + ex);
}
%>
<hr size=5 color=red>
<a href='FindStu.html'>Find Another Student Record</a>
</body>
</html>

```

Output:

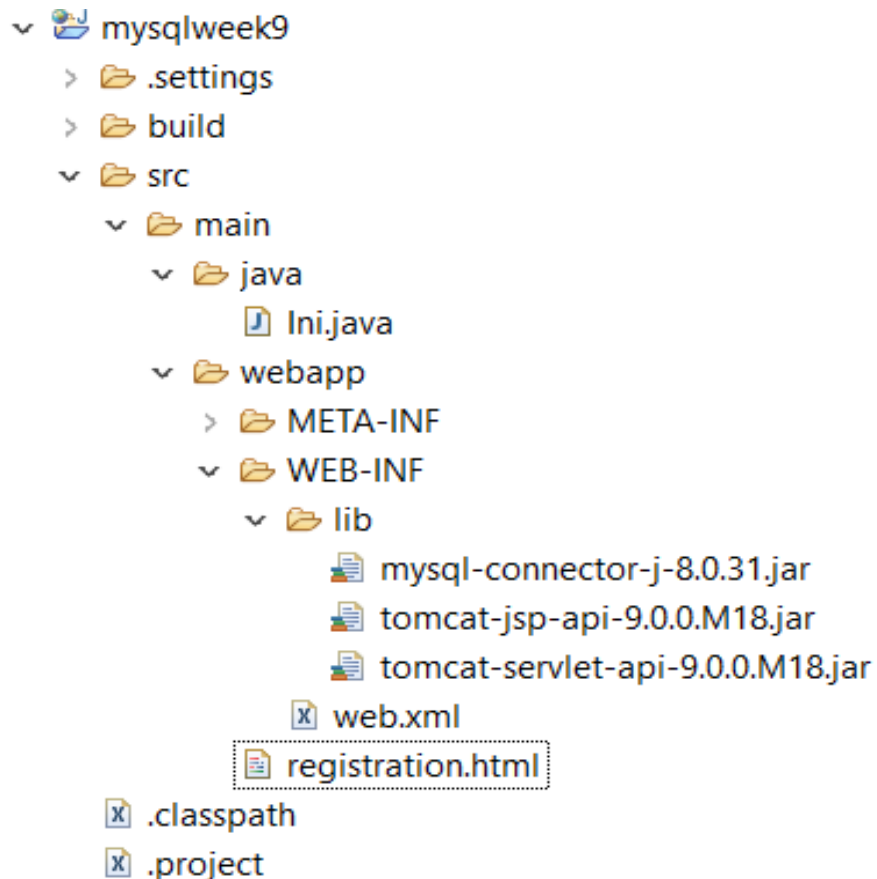




## Accessing a Database from a servlets page

Example 1: **html+servlets+mysql [10 Marks]**

### Program structure



## Registration.html

```
<html>
<head>
<title>Registration page</title>
</head>
<body bgcolor="#00FFf">
  <form METHOD="POST" ACTION="regis">
    <CENTER>
      <table>
        <center>
          <tr>
            <td>Username</td>
            <td><input type="text" name="usr"></td>
          </tr>
          <tr>
            <td>Password</td>
            <td><input type="password" name="pwd"></td>
          </tr>
          <tr>
            <td>Age</td>
            <td><input type="text" name="age"></td>
          </tr>
          <tr>
            <td>Address</td>
            <td><input type="text" name="add"></td>
          </tr>
          <tr>
            <td>email</td>
            <td><input type="text" name="mail"></td>
          </tr>
          <tr>
            <td>Phone</td>
            <td><input type="text" name="phone"></td>
          </tr>
          <tr>
            <td colspan=2 align=center><input type="submit"
              value="submit"></td>
          </tr>
        </center>
      </table>
    </form>
  </body>
```



## Ini.java

```
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;

/**
 * Servlet implementation class Ini
 */

public class Ini extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private String user1,pwd1,email1;
    public void service(ServletRequest req,ServletResponse res) throws
ServletException,IOException
    {
        user1=req.getParameter("usr");
        pwd1=req.getParameter("pwd");
        email1=req.getParameter("mail");
        res.setContentType("text/html");
        PrintWriter out=res.getWriter();
        try
        {
            Class.forName("com.mysql.jdbc.Driver");
            Connection
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/ananthula_movies","root","root");
            PreparedStatement st=con.prepareStatement("insert into personal
values(?,?,?,?);");
            st.setString(1,user1);
            st.setString(2,pwd1);
            st.setString(3,"25");
            st.setString(4,"hyd");
            st.setString(5,email1);
            st.setString(6,"21234");
            st.executeUpdate();
            con.close();
        }
        catch(SQLException s)
        { out.println("not found "+s);
```

```

    }
    catch(ClassNotFoundException c)
    {
        out.println("not found "+c);
    }
}
}

```

## Web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    <servlet>
        <servlet-name>init1</servlet-name>
        <servlet-class>Ini</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>init1</servlet-name>
        <url-pattern>/regis</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>init1</servlet-name>
        <servlet-class>Ini</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>init1</servlet-name>
        <url-pattern>/athent</url-pattern>
    </servlet-mapping>
</web-app>

```

## Introduction to struts framework [10 MARK 2017]

**Struts** is used to create a web applications based on servlet and JSP. Struts depend on the MVC (Model View Controller) framework. Struts application is a genuine web application. Struts are thoroughly useful in building J2EE (Java 2 Platform, Enterprise Edition) applications because struts takes advantage of J2EE design patterns. Struts follows these J2EE design patterns including MVC.

In struts, the composite view manages the layout of its sub-views and can implement a template, making persistent look and feel easier to achieve and customize across the entire application. A

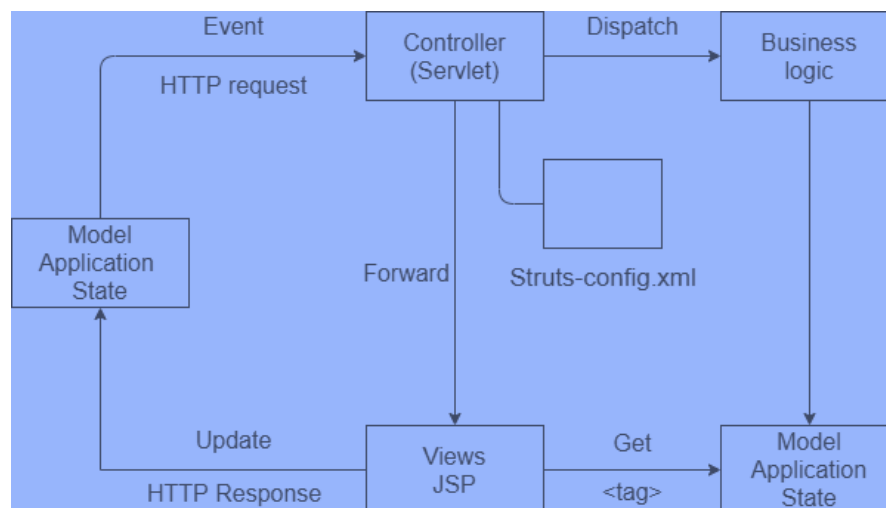
composite view is made up by using other reusable sub views such that a small change happens in a sub-view is automatically updated in every composite view.

Struts consists of a set of own custom tag libraries. Struts are based on MVC framework which is pattern oriented and includes JSP custom tag libraries. Struts also supports utility classes.

**Features of Struts:** Struts has the following features:

- Struts encourages good design practices and modeling because the framework is designed with “time-proven” design patterns.
- Struts is almost simple, so easy to learn and use.
- It supports many convenient features such as input validation and internationalization.
- It takes much of the complexity out as instead of building your own MVC framework, you can use struts.
- Struts is very well integrated with J2EE.
- Struts has large user community.
- It is flexible and extensible, it is easy for the existing web applications to adapt the struts framework.
- Struts provide good tag libraries.
- It allows capturing input form data into java bean objects called Action forms.
- It also hand over standard error handling both programmatically and declaratively.

**Working of Struts:**



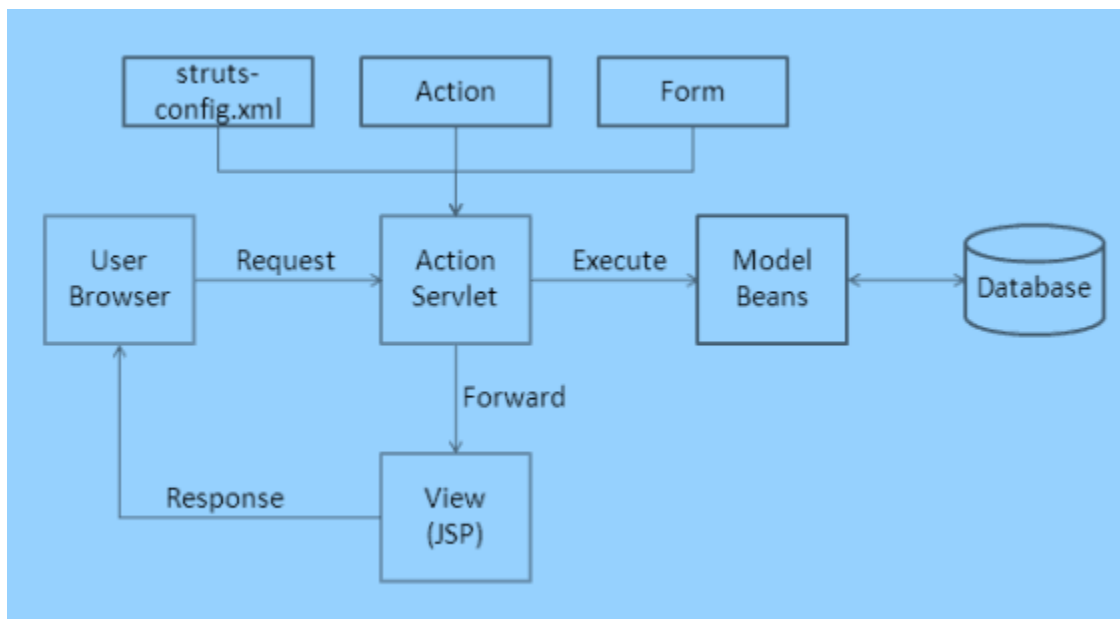
In the initialization phase, the controller rectify a configuration file and used it to deploy other control layer objects. Struts configuration is form by these objects combined together. The struts configuration defines among other things the action mappings for an application. Struts controller servlet considers the action mappings and routes the HTTP requests to other components in the framework. Request is first delivered to an action and then to JSP. The mapping helps the controller to change HTTP requests into application actions. The action objects can handle the request from and responds to the client (generally a web browser). Action objects have access to the applications controller servlet and also access to the servlet's methods. When delivering the control, an action objects can indirectly forward one or more share objects, including javabeans by establish them in the typical situation shared by java servlets.

## Struts MVC Architecture

The **model** contains the business logic and interact with the persistence storage to store, retrieve and manipulate data.

The **view** is responsible for displaying the results back to the user. In Struts the view layer is implemented using JSP.

The **controller** handles all the request from the user and selects the appropriate view to return. In Struts the controller's job is done by the ActionServlet.



The following events happen when the Client browser issues an HTTP request.

- The ActionServlet receives the request.
- The struts-config.xml file contains the details regarding the Actions, ActionForms, ActionMappings and ActionForwards.
- During the startup the ActionServlet reads the struts-config.xml file and creates a database of configuration objects. Later while processing the request the ActionServlet makes decision by referring to this object.

When the ActionServlet receives the request it does the following tasks.

- Bundles all the request values into a JavaBean class which extends Struts ActionForm class.
- Decides which action class to invoke to process the request.
- Validate the data entered by the user.
- The action class process the request with the help of the model component. The model interacts with the database and process the request.
- After completing the request processing the Action class returns an ActionForward to the controller.
- Based on the ActionForward the controller will invoke the appropriate view.
- The HTTP response is rendered back to the user by the view component.

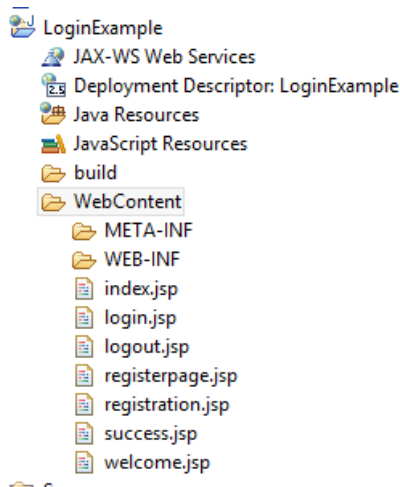
## Case study:

### JSP Database Access Example using JDBC and MySQL

In this example we will create a Login page to enter user name and password and one link for Registration page for new user registration. On submit, validate the user name/password against MySQL database. If the authentication is successful, forward to home page showing welcome message along with the user name. If the authentication fails, return back to the login page with appropriate error message. If there is exception/errors during authentication process return back to login page with appropriate error message.

#### Directory Structure Of Project

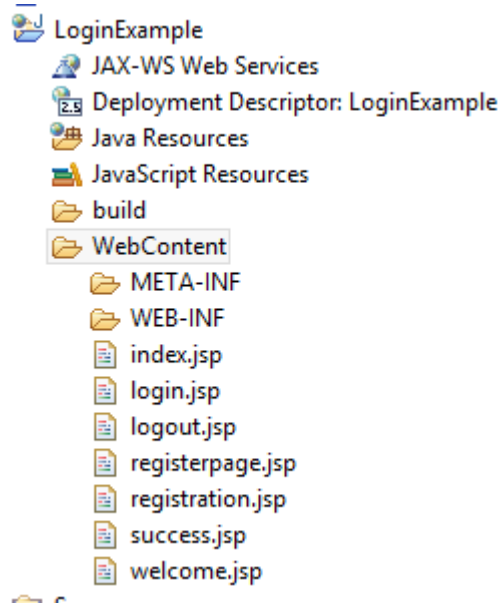
Directory Structure of project is shown below:



#### Step 1: Create a table Employee in mysql DB :

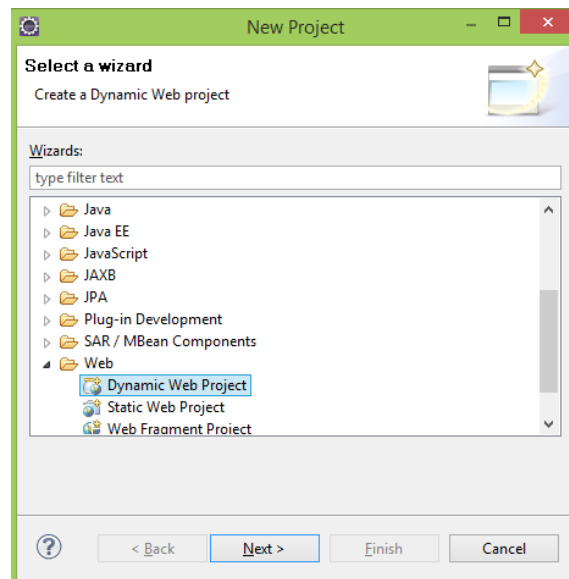
```
CREATE TABLE `EMPLOYEE` (  
  `id` int(10) unsigned NOT NULL auto_increment,  
  `FIRST_NAME` varchar(45) NOT NULL,  
  `LAST_NAME` varchar(45) NOT NULL,  
  `EMAIL` varchar(45) NOT NULL,  
  `USER_NAME` varchar(45) NOT NULL,  
  `PASSWORD` varchar(45) NOT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

## Project Directory Structure :

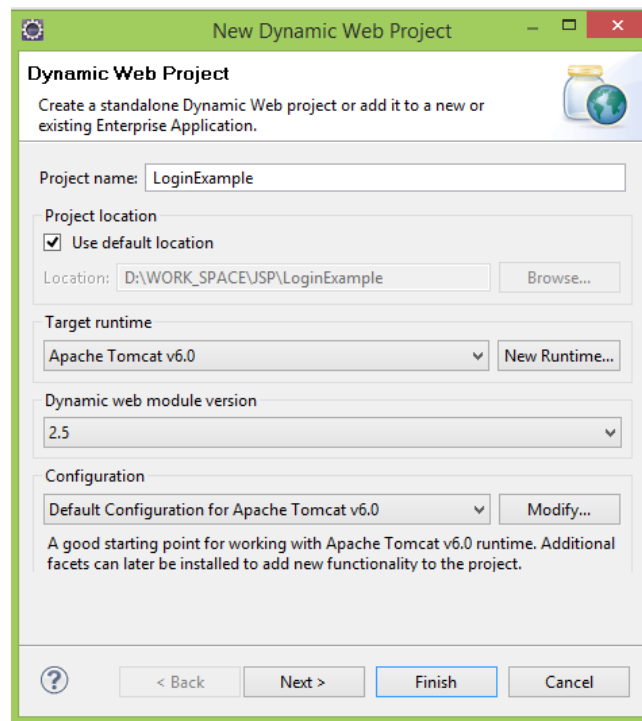


## Step 1: Create Dynamic Web project

Open Eclipse and go to **File -> New -> Project** and select **Dynamic Web Project** in the New Project wizard screen.



Provide the name of the project as **LoginExample** . Once this is done, select the target runtime environment as **Apache Tomcat v9.0** and click Next -> Next -> Finish.



### Step 3: Create JSP files

Create all JSP files inside WebContent Directory

#### index.jsp

```
<% @ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Login Example</title>
</head>
<body>
    <form method="post" action="login.jsp">
        <center>
            <table border="2" width="30%" cellpadding="3">
                <thead>
                    <tr>
                        <th colspan="2">Login Example</th>
```



```

        </tr>
    </thead>
    <tbody>
        <tr>
            <td>User Name</td>
            <td><input type="text" name="username" value="" /></td>
        </tr>
        <tr>
            <td>Password</td>
            <td><input type="password" name="password" value="" /></td>
        </tr>
        <tr>
            <td><input type="submit" value="Login" /></td>
            <td><input type="reset" value="Reset" /></td>
        </tr>
        <tr>
            <td colspan="2">New User <a href="registerpage.jsp">Register
                Here</a></td>
        </tr>
    </tbody>
</table>
</center>
</form>
</body>
</html>

```

## login.jsp

```

<% @page import="java.sql.ResultSet"%>
<% @page import="java.sql.Statement"%>
<% @page import="java.sql.Connection"%>
<% @page import="com.example.util.DBUtil"%>
<%
    String userName = request.getParameter("username");
    String password = request.getParameter("password");
    Connection con = DBUtil.getMySQLConnection();
    Statement st = con.createStatement();
    ResultSet rs;
    rs = st.executeQuery("select * from EMPLOYEE where USER_NAME='"
        + userName + "' and PASSWORD='" + password + "'");
    if (rs.next()) {
        session.setAttribute("username", userName);
        response.sendRedirect("success.jsp");
    }
    else
    {
        out.println("Invalid password <a href='index.jsp'>try again</a>");
    }
%>

```

## registerpage.jsp

```
<% @ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Registration Page</title>
</head>
<body>
    <form method="post" action="registration.jsp">
        <center>
            <table border="1" width="30%" cellpadding="5" bgcolor="pink">
                <thead>
                    <tr>
                        <th colspan="2">Registration Page</th>
                    </tr>
                </thead>
                <tbody>
                    <tr>
                        <td>First Name</td>
                        <td><input type="text" name="firstname" value="" /></td>
                    </tr>
                    <tr>
                        <td>Last Name</td>
                        <td><input type="text" name="lastname" value="" /></td>
                    </tr>
                    <tr>
                        <td>Email</td>
                        <td><input type="text" name="email" value="" /></td>
                    </tr>
                    <tr>
                        <td>User Name</td>
                        <td><input type="text" name="username" value="" /></td>
                    </tr>
                    <tr>
                        <td>Password</td>
                        <td><input type="password" name="password" value="" /></td>
                    </tr>
                    <tr>
                        <td><input type="submit" value="Submit" /></td>
                        <td><input type="reset" value="Reset" /></td>
                    </tr>
                </tbody>
            </table>
        </center>
    </form>
</body>
</html>
```

```

</tr>
<td colspan="2">Already registered?<a href="index.jsp">Login
Here</a></td>
</tr>
</tbody>
</table>
</center>
</form>
</body>
</html>

```

## registration.jsp

```

<% @page import="java.sql.Statement"%>
<% @page import="java.sql.Connection"%>
<% @page import="com.example.util.DBUtil"%>
<%
String userName = request.getParameter("username");
String password = request.getParameter("password");
String firstName = request.getParameter("firstname");
String lastName = request.getParameter("lastname");
String email = request.getParameter("email");
Connection con = DBUtil.getMySQLConnection();
Statement st = con.createStatement();
int i = st.executeUpdate
("insert into EMPLOYEE
(FIRST_NAME, LAST_NAME, EMAIL, USER_NAME, PASSWORD)
values ("
    + firstName
    + ", "
    + lastName
    + ", "
    + email
    + ", "
    + userName
    + ", "
    + password
    + ")");
if (i > 0) {
    response.sendRedirect("welcome.jsp");
} else {
    response.sendRedirect("index.jsp");
}
%>

```

### success.jsp

```
<%
    if ((session.getAttribute("username") == null)
        || (session.getAttribute("username") == "")) {
%>
You are not logged in<br/>
<a href="index.jsp">Please Login</a>
<%} else {
%>
Welcome <%=session.getAttribute("username")%>
<a href='logout.jsp'>Log out</a>
<%
    }
%>
```

### welcome.jsp

Registration is Successful.  
Please Login Here

### logout.jsp

```
<%
session.setAttribute("username", null);
session.invalidate();
response.sendRedirect("index.jsp");
%>
```

Output :

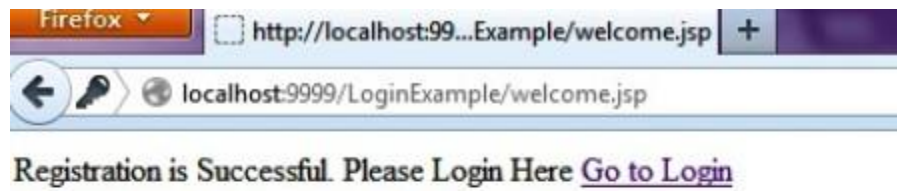
Screen 1 :

<b>Login Example</b>	
User Name	<input type="text"/>
Password	<input type="password"/>
<input type="button" value="Login"/>	<input type="button" value="Reset"/>
New User <a href="#">Register Here</a>	

Screen 2 :

<b>Registration Page</b>	
First Name	<input type="text"/>
Last Name	<input type="text"/>
Email	<input type="text"/>
User Name	<input type="text"/>
Password	<input type="password"/>
<input type="button" value="Submit"/>	<input type="button" value="Reset"/>
Already registered? <a href="#">Login Here</a>	

Screen 3 :



Screen 4 :

