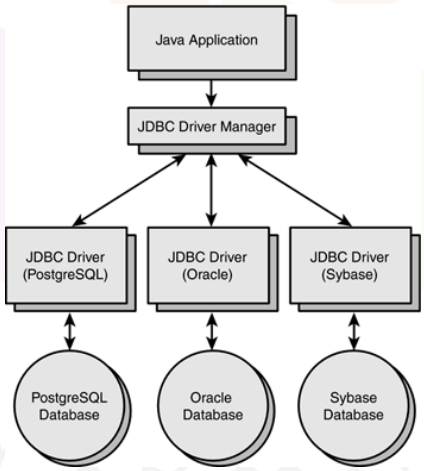
JDBC
[Java DataBase Connectivity]

Introduction

- Almost all the web applications need to work with the data stored in the databases.
- JDBC is Java specification that allows the Java programs to access the databases.
- The classes and interfaces related to JDBC are available in the *java.sql* package.

JDBC Architecture



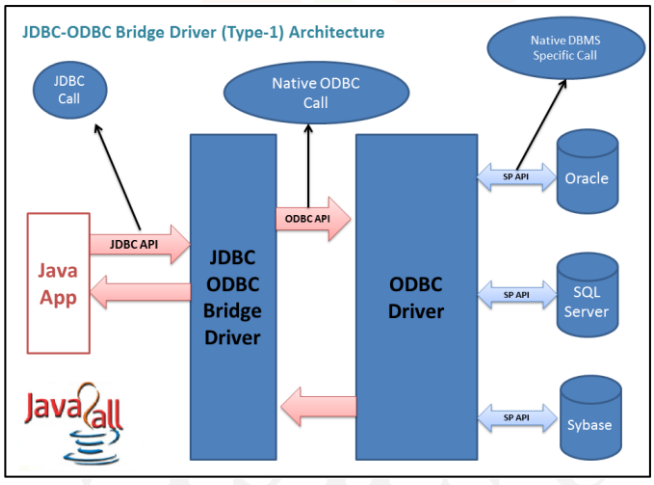
JDBC Driver

- Using JDBC, a java application can access all types of databases.
- A Java class that provides the implementation for the JDBC interface is known as a JDBC driver.
- The JDBC driver hides the details of the underlying database.
- Database vendors like Oracle, IBM, Microsoft provides the JDBC drivers free of cost.

JDBC Driver Categories

- Type 1 driver: JDBC-ODBC bridge
- Type 2 driver: Native-API driver (Partly Java driver)
- Type 3 driver: Network-Protocol driver (Pure Java driver for database middleware)
- Type 4 driver: Native-Protocol driver (Pure Java driver directly connected to database)

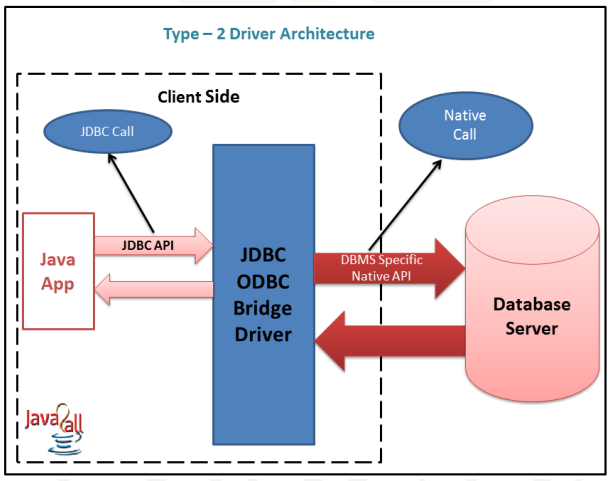
Type 1 Driver



Type 1 Driver (cont…)

- Advantages:
  - Connect to any database which provides ODBC driver.
  - Easiest driver to use in the java applications.
- Disadvantages:
  - ODBC driver needs to be installed on the client machine.
  - Platform dependent (due to ODBC).
  - Not suitable for applets as ODBC driver needs to be installed on the client machine.

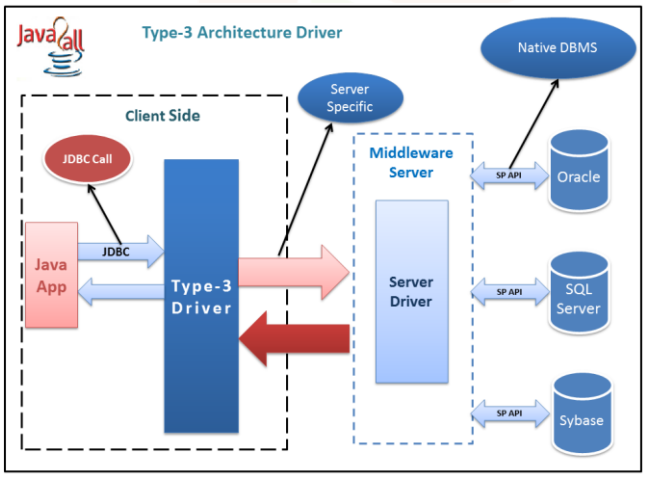
Type 2 Driver



Type 2 Driver (cont…)

- Advantage:
  - As there is no ODBC driver, type 2 driver is faster than type 1 driver.
- Disadvantages:
  - Driver need to be installed on the client machine.
  - Not suitable for web applications.
  - All databases does not provide client side libraries.

Type 3 Driver

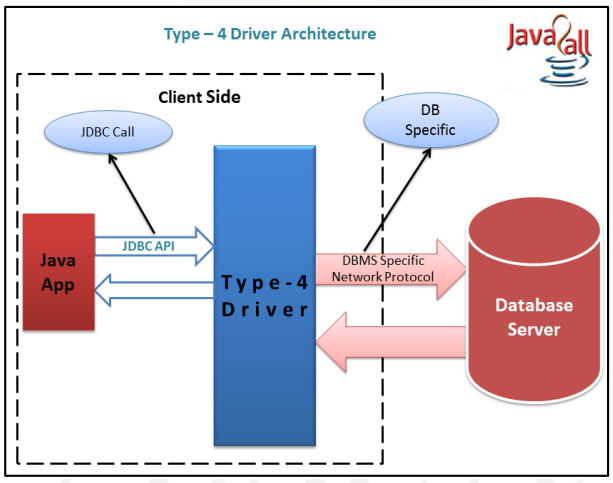


Type 3 Driver (cont…)

- Advantages:
  - There is no need to install database specific library files on the client as the middleware is database independent.
  - Suitable for enterprise web applications.
  - Middleware supports other services like connection pooling, logging, load balancing etc.

- Disadvantages:
  - An extra layer(middleware) might affect performance.
  - Developing database specific coding at the middleware might increase the complexity.

Type 4 Driver



Vishnu Institute of technology – Website: www.vishnu.edu.in

Type 4 Driver (cont…)

- Advantages:
  - It is 100% pure Java driver (platform independent).
  - No extra layers are needed.
  - Debugging the application is easier.

- Disadvantage:
  - Database specific driver needed at the client side.

java.sql Package

Class or Interface	Description
java.sql.DriverManager	Class responsible for locating and loading a driver and establishing connection with the database
java.sql.Connection	Interface responsible for creating a Statement object
java.sql.Statement	Interface for executing static SQL statements on a table
java.sql.PreparedStatement	Interface for executing parameterized SQL statements on a table
java.sql.CallableStatement	Interface for executing a stored procedure available in the database
java.sql.ResultSet	Interface for navigating the data returned by the execution of a SQL statement

Basic Steps in Working with Databases

1. Loading a driver
2. Connecting to a database
3. Executing a SQL statement

Loading a Driver

- To load a driver, first the driver needs to be downloaded from the vendor's website or locate it in the DBMS's directory.
- Usually the driver will be available in a *.jar* file.
- For Oracle 10g DBMS, filename is ojdbc14.jar and is located in the following path <<Drive name:>>\oraclexe\app\oracle\product\10.2.0\server\jdbc\lib

Loading a Driver (cont…)

- After downloading or locating the *.jar* file, place it in Tomcat's *lib* directory and restart the server or add the location of the *.jar* file to the CLASSPATH environment variable.

- To use the JDBC driver, an instance of the driver class has to be created and registered with the *DriverManager* class available in the *java.sql* package.

Loading a Driver (cont…)

- The *DriverManager* class along with the JDBC driver translates the JDBC call to appropriate database call.

- One way to register the driver with the *DriverManager* is to use the static method *forName* available on the class *Class* along with the driver class as an argument.

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

Loading a Driver (cont…)

- Second way to register the driver is by using static method *registerDriver* available on the *DriverManger* class.

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

- The second way of registering the driver is recommended by some of the vendors.

Making a Connection

- For establishing a connection to the database, use *getConnection* static method available in *DriverManager* class. This method returns a *Connection* object.

- Following are the overloaded versions of *getConnection* method:

```
public static Connection getConnection(String URL, String user, String password)
public static Connection getConnection(String URL)
public static Connection getConnection(String URL, Properties p)
```

Making a Connection (cont…)

- Establishing connection to Oracle database using a type-4 driver:

```
Connection con =
DriverManager.getConnection("jdbc:oracle:thin:@lo
calhost:1521:xe",“username",“password");
```


Executing a SQL Statement

- After establishing a connection to the database, SQL statements can be executed by using the methods provided by the *Connection* interface.

Executing a SQL Statement (cont…)

- Following are the methods provided by the *Connection* interface:

1. Statement createStatement()
2. Statement createStatement(int resultSetType, int resultSetConcurrency)
3. Statement createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability)

1. PreparedStatement prepareStatement(String query)
2. PreparedStatement prepareStatement(String query, int resultSetType, int resultSetConcurrency)
3. PreparedStatement prepareStatement(String query, int resultSetType, int resultSetConcurrency, int resultSetHoldability)

1. CallableStatement prepareCall(String call)
2. CallableStatement prepareCall(String call, int resultSetType, int resultSetConcurrency)
3. CallableStatement prepareCall(String call, int resultSetType, int resultSetConcurrency, int resultSetHoldability)

Statement Interface

- Following methods are provided by *Statement* interface to execute SQL statements:

executeUpdate(String query) – Used to execute DDL, DML and DCL commands. Return value is the number of rows affected.

executeQuery(String query) – Used to execute SELECT command. Returns a *ResultSet* object containing the result returned by the database.

execute(String query) – Used to execute any type of SQL command if the type of command is not known prior to the execution of the query. Return value is a boolean. Returns *true* if the return value is a *ResultSet* object. Returns *false* if the return value is update count.

PreparedStatement Interface

- The *PreparedStatement* interface is used to execute pre-compiled SQL statements. Useful when the same query is executed with different parameters.
- Queries executed using the *PreparedStatement* interface are known as parameterized queries.

PreparedStatement Interface (cont…)

Example

```
PreparedStatement  ps = con.prepareStatement(“insert
   into users values(?,?)”);
ps.setString(1, “user1”);
ps.setString(2, “user2”);
ps.executeUpdate();
```


CallableStatement Interface

- The *CallableStatement* interface is used to execute stored procedures available in the DBMS.
- The *prepareCall* method accepts a String parameter which represents the procedure to be called and returns a *CallableStatement* back.

CallableStatement Interface (cont…)

Example:

```
String  stcall = "{call  changePassword(?,?)}";
CallableStatement  cstmt = con.prepareCall(stcall);
cstmt.setString(1, value);
cstmt.setString(2, value);
cstmt.executeUpdate();
```


ResultSet Interface

- The *ResultSet* interface object represents the tabular data returned by the executeXXX methods available on the *Statement*, *PreparedStatement* and *CallableStatement* methods.
- A pointer which allows to retrieve the data from the result set is called a *cursor.*
- To move the cursor the *ResultSet* interface provides *next* method which moves the cursor to the next row. This method returns *true* if there is a row or *false* otherwise.

ResultSetMetaData Interface

- The *ResultSetMetaData* interface is used to get the meta data of the data present in the *ResultSet* object.

```
ResultSet  rs = stmt.executeQuery("select * from users");
ResultSetMetaData  rsmd = rs.getMetaData();
```

ResultSetMetaData Interface (cont…)

- Following methods are available in the *ResultSetMetaData* interface:

int getColumnCount() – Returns the number of columns in the result

String getColumnName(int) – Returns the name of column in the result set

int getColumnType(int) – Returns the type of the specified column

String getColumnTypeName(int) – Returns the name of data type of the column as a String

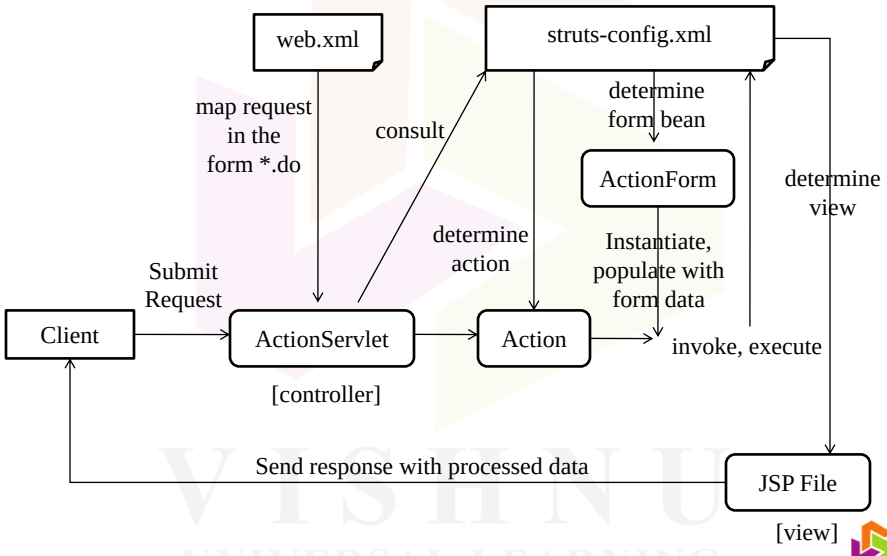
int isNullable(int) – Returns a constant indicating whether the specified column can have NULL value or not

Introduction to Struts Framework

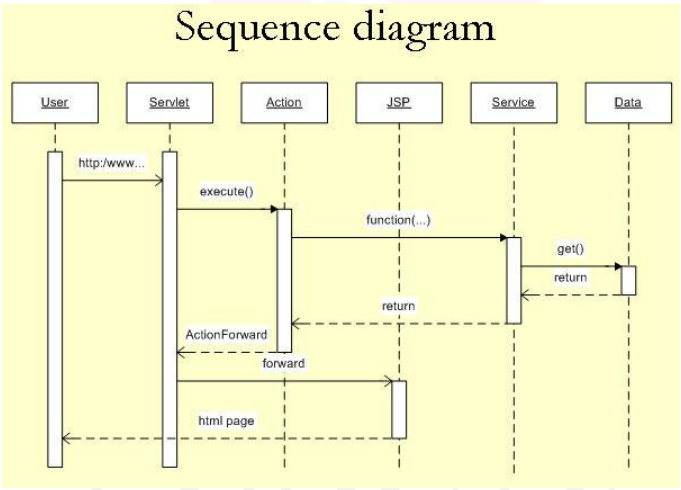
- Struts is an application framework for developing Java EE web applications which follows Model-View-Controller (MVC) architecture.

- Struts allows developers to create *flexible*, *extensible* and *maintainable* large web applications.

Struts Architecture



Struts Sequence Diagram



web.xml in Struts

```
<web-app>
    <servlet>
        <servlet-name>action</servlet-name>
        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
        <init-param>
            <param-name>config</param-name>
            <param-value>/WEB-INF/structs-config.xml</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
</web-app>
```


struts-config.xml in Struts

```
<action-mappings>
        <action name="LoginForm" path="/login" scope="request" type="
            com.myapp.struts.LoginAction" validate="false">
            <forward name="success" path="/success.jsp" />
            <forward name="failure" path="/login.jsp" />
        </action>
</action-mappings>


<form-beans>
        <form-bean name="LoginForm" type="com.myapp.struts.LoginForm" />
</form-beans>
```

LoginAction Class

```
public class LoginAction extends org.apache.struts.action.Action
{
        private final static String SUCCESS = "success";
        private final static String FAILURE = "failure";
        public ActionForward execute(ActionMapping mapping, ActionForm form,
                HttpServletRequest request, HttpServletResponse response) throws
                Exception
        {
                LoginForm lf = (LoginForm) form;
                String name = lf.getName();
                String pass = lf.getPass();
                if(valid)
                        return mapping.findForward(SUCCESS);
                else
                        { lf.setError();
                          return mapping.findForward(FAILURE); }
        }
}
```