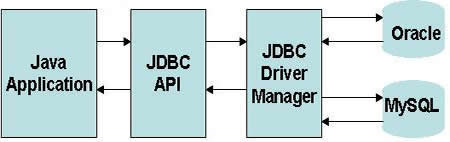
**JDBC Architecture Diagram**  
  
  
This tutorial will guide you to develop JDBC applications using different type of drivers using JDBC API. The JDBC API and JDBC Driver both are different from each other. The JDBC driver is different for different database, like MySQL driver is different than Oracle and it is also different than MS-SQL server.  
  
**JDBC API**- It contains the common classes and interfaces to work in database, like - **Connection, PreparedStatement, Statement, ResultSet, ResultSetMetaData, DatabaseMetaData** etc.

c

package com.jdbc;

public class JdbcDrivers {

public static void main(String[] args) {

/\*

\* 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)

driver-1

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.

driver-2

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.

driver-3

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.

driver-4

we are using driver-4

Advantage:

Better performance than all other drivers.

No software is required at client side or server side.

Disadvantage:

Drivers depends on the Database.

\*/

}

}

package com.jdbc;

public class JdbcSteps {

public static void main(String[] args) {

/\*There are 5 steps to connect any java application with the database in java using JDBC. They are as follows:

Register the driver class

Creating connection

Creating statement or prepared statement

Executing queries execute or executeupdate(non -select queries ex:update,delete,insert all related DML operations)

Closing connection

1) Register the driver class

The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax of forName() method

public static void forName(String className)throws ClassNotFoundException

Example to register the OracleDriver class

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

2) Create the connection object

The getConnection() method of DriverManager class is used to establish connection with the database.

Syntax of getConnection() method

1) public static Connection getConnection(String url)throws SQLException

2) public static Connection getConnection(String url,String name,String password)

throws SQLException

Example to establish connection with the Oracle database

Connection con=DriverManager.getConnection(

"jdbc:oracle:thin:@localhost:1521:xe","system","password");

3) Create the Statement object

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Syntax of createStatement() method

public Statement createStatement()throws SQLException

Example to create the statement object

Statement stmt=con.createStatement();

4) Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax of executeQuery() method

public ResultSet executeQuery(String sql)throws SQLException

Example to execute query

ResultSet rs=stmt.executeQuery("select \* from emp");

while(rs.next()){

System.out.println(rs.getInt(1)+" "+rs.getString(2));

}

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax of close() method

public void close()throws SQLException

Example to close connection

con.close();

\*

\*/

}

}

package com.jdbc;

public class Connectionmethods {

public static void main(String[] args) {

/\*

\* Connection interface:

A Connection is the session between java application and database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(),rollback() etc.

By default, connection commits the changes after executing queries.

Commonly used methods of Connection interface:

1) public Statement createStatement(): creates a statement object that can be used to execute SQL queries.

2) public Statement createStatement(int resultSetType,int resultSetConcurrency): Creates a Statement object that will generate ResultSet objects with the given type and concurrency.

3) public void setAutoCommit(boolean status): is used to set the commit status.By default it is true.

4) public void commit(): saves the changes made since the previous commit/rollback permanent.

5) public void rollback(): Drops all changes made since the previous commit/rollback.

6) public void close(): closes the connection and Releases a JDBC resources immediately.

\*/

}

}

package com.pioneercoders.statementexamples;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.SQLException;

public class JdbcPreparedStatementExample {

public static void main(String[] args) throws ClassNotFoundException, SQLException {

/\*

\* Register the driver class

Creating connection

Creating statement/PreparedStatement

Executing queries

Closing connection

\*/

Class.forName("com.mysql.jdbc.Driver");//Driver for mysql

//The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

// localhost is mysql server IP address if it is running in our local machine we have to use

//Localhost if it is running server machine or client machine we have to use

//IP address ex:192.168.100.88

//3306 is mysql port number

//med\_hosp\_db\_2 is database schema name

//root to connect with database we use username

//root to connect with database we use password

Connection con= DriverManager.getConnection("jdbc:mysql://localhost:3306/med\_hosp\_db\_2", "root", "root");

//by using DriverManager we are getting database connection

//after getting Connection to perform crud(create,read,update,delete operations on database we will use statements or preparedstatements

//by writing statements the changes directly applied on database

//how to insert data into table using prepared statement

String sql="insert into user\_mst\_tbl(id,user\_id,user\_pwd,role\_id) values (?,?,?,?)";

PreparedStatement pstmt= con.prepareStatement(sql);

//we are passing values based on datatype

pstmt.setInt(1, 72);//index position

pstmt.setString(2, "krish");

pstmt.setString(3, "abc");

pstmt.setInt(4, 6);

//tyeps of queries

// 1.select queries

//2.Non Select queries(insert,update,delete)

//for select queries we have to use executeQuery() returntype resultset

//for non select queries we have to use executeupdate() return type int

//non select we are applying changes on db after changes applied on db we have to know the changes applied are not that's why they designed the method executeUpdate() method like that and its return type is int

int inserted= pstmt.executeUpdate();

System.out.println(inserted);

}

}

package com.pioneercoders.statementexamples;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

public class StatementExample {

/\*

\* To perform crud operations on database we will statements and prepared statements in this we are using statement how to insert values in db

\*

\* First create the table in database

\*

\* CREATE TABLE `jdbc`.`user\_mst\_tbl` (

`ID` bigint(20) NOT NULL AUTO\_INCREMENT,

`USER\_ID` varchar(255) DEFAULT NULL,

`USER\_PWD` varchar(255) DEFAULT NULL,

`ROLE\_ID` bigint(20) DEFAULT NULL,

PRIMARY KEY (`ID`));

\*

\* for any jdbc application these steps are common

\*

\* Register the driver class of particular vendor (i'm using mysql driver )

Creating connection (to establish connection with database

Creating statement/PreparedStatement to perform crud operations (select,insert,delete,update)

Executing queries for select use execute() method for non-select (insert,update,delete)queries use executeUpdate() method

Closing connection

\*

\*

\*/

public static void main(String[] args) throws ClassNotFoundException, SQLException {

Class.forName("com.mysql.jdbc.Driver");//Driver for mysql

Connection con= DriverManager.getConnection("jdbc:mysql://localhost:3306/jdbc", "root", "root");

//by using statement i'm inserting data into database

Statement st = con.createStatement();

String sql= "insert into user\_mst\_tbl(id,user\_id,user\_pwd,role\_id) values(1,'sr12','sr',3)";

// i'm inserting one row into the database executeUpdate() return type is int if data is inserted in db it will return value 'one'

int inserted = st.executeUpdate(sql);

if(inserted>0){

System.out.println("Successfully Inserted"+" "+inserted);

}

con.close();

}

}

package com.pioneercoders.resultsetexamples;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

public class ResultSetExample {

/\*

\* If we want to select the data from database ,jdbc provides ResultSet

\* Interface to select data from database and return the data in resultset

\* format

\*

\* The SELECT statement is the standard way to select rows from a database

\* and view them in a result set. The java.sql.ResultSet interface

\* represents the result set of a database query.

\*

\* ResultSet object maintains a cursor that points to the current row in the

\* result set. The term "result set" refers to the row and column data

\* contained in a ResultSet object.

\*/

public static void main(String[] args) throws SQLException,

ClassNotFoundException {

Class.forName("com.mysql.jdbc.Driver");

Connection con = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/jdbc", "root", "root");

String selectquery = "select \* from employee\_mst\_tbl";

PreparedStatement pstmt = con.prepareStatement(selectquery);

ResultSet rs = pstmt.executeQuery();

while (rs.next()) {// it will checks one by one

/\*

\* mention the datatypes matches with the database in sql in Java

\* varchar String number(in sql) /int(in mysql) int

\*

\* in resultset we can mention index numbers or column names

\* rs.getInt(1) or rs.getInt(id)

\*/

System.out.println("Details are " + " " + rs.getInt(1));// it will returns first column value

//

//

System.out.println("Detials are " + " " + rs.getString(2) + " "

+ rs.getString(3) + " " + rs.getString(4) + " "

+ rs.getInt(5));

/\* it will give the result as follows

\*

\* Details are 20

Detials are gopi 20-02-1993 gopi@gmail.com 8754126

Details are 22

Detials are rajeshh 10-06-1993 2345@gmail.com 8542167

Details are 30

Detials are anand 12-01-1993 appaji@gmail.com 7451426

Details are 40

Detials are rajesh 20-06-1993 2345@gamil.com 777478512

\*/

}

}

}

package com.pioneercoders.resultsetmetadataex;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.ResultSetMetaData;

import java.sql.SQLException;

public class ResultSetMetaDataEx {

public static void main(String[] args) throws ClassNotFoundException, SQLException {

//by using resultset we are getting only data but we

//dont know tables information like columns and no of columns to know table full details

//we have to use ResultSetMetaData metadata means information about resultset like column details

//in this i want to know user\_mst\_tbl

//i want to know number of columns, column name of particular index,column name datatype we have to use the

//following methods

/\*

\* Method Description

public int getColumnCount()throws SQLException it returns the total number of columns in the ResultSet object.

public String getColumnName(int index)throws SQLException it returns the column name of the specified column index.

public String getColumnTypeName(int index)throws SQLException it returns the column type name for the specified index.

\*/

Class.forName("com.mysql.jdbc.Driver");

Connection con= DriverManager.getConnection("jdbc:mysql://localhost:3306/med\_hosp\_db\_2", "root", "root");

PreparedStatement pstmt= con.prepareStatement("select \* from user\_mst\_tbl");

//first get the ResultSet data

ResultSet rs= pstmt.executeQuery();

while(rs.next()){//it iterates row one by one by moving cursor forward

}

//but i want to the know datatypes and columns of the user table

//to get metadata of the user table we have to use ResultsetMetaData

//to get metadata we have to call getMetaData()

ResultSetMetaData rd= rs.getMetaData();

//i want to know no of columns

System.out.println("No of Columns "+" "+rd.getColumnCount());

//i want to first column name

System.out.println("First Column Name"+" "+rd.getColumnName(1));

//i want to know the data type of firstcolumn

System.out.println("First Column name DataType"+" "+rd.getColumnTypeName(1));

}

}

package com.pioneercoders.databasemetadataex;

import java.sql.Connection;

import java.sql.DatabaseMetaData;

import java.sql.DriverManager;

import java.sql.SQLException;

public class DataBaseMetaDataEx {

public static void main(String[] args) throws ClassNotFoundException, SQLException {

//to communicate with database we have to know

//its driver and port and Ip address and username and password

//but if we want to know additional information like database vendor

//database version and vendor details

//we have to use DataBaseMetaData

Class.forName("com.mysql.jdbc.Driver");

Connection con= DriverManager.getConnection("jdbc:mysql://localhost:3306/med\_hosp\_db\_2", "root", "root");

DatabaseMetaData dbmd=con.getMetaData();

System.out.println("Driver Name: "+dbmd.getDriverName());

System.out.println("Driver Version: "+dbmd.getDriverVersion());

System.out.println("UserName: "+dbmd.getUserName());

System.out.println("Database Product Name: "+dbmd.getDatabaseProductName());

System.out.println("Database Product Version: "+dbmd.getDatabaseProductVersion());

con.close();

}

}

package com.pioneercoders.rowsetex;

import java.sql.SQLException;

import javax.sql.rowset.JdbcRowSet;

import javax.sql.rowset.RowSetProvider;

public class RowSetEx {

public static void main(String[] args) throws ClassNotFoundException, SQLException {

/\*

\*

\* JDBC RowSet

The instance of RowSet is the java bean component because it has properties and java bean notification mechanism. It is introduced since JDK 5.

It is the wrapper of ResultSet. It holds tabular data like ResultSet but it is easy and flexible to use.

The implementation classes of RowSet interface are as follows:

JdbcRowSet

CachedRowSet

WebRowSet

JoinRowSet

FilteredRowSet

\*/

//by using rowset only we are setting database properties

//because internally rowset uses properties file to set the data

//it is easy to switch and flexible to use

//if we set database details to rowset we can switch to another database easily

//let see how to set db details to rowset

//load the driver

Class.forName("com.mysql.jdbc.Driver");

//Creating and Executing RowSet

JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();

rowSet.setUrl("jdbc:mysql://localhost:3306/med\_hosp\_db\_2");

rowSet.setUsername("root");

rowSet.setPassword("root");

rowSet.setCommand("select \* from user\_mst\_tbl");

rowSet.execute();

while (rowSet.next()) {

// Generating cursor Moved event

System.out.println("Id: " + rowSet.getInt(1));

System.out.println("UserName: " + rowSet.getString(2));

System.out.println("Userpwd " + rowSet.getString(3));

System.out.println("Roleid " + rowSet.getString(4));

}

}

}

package com.jdbc;

public class DriverManager {

public static void main(String[] args) {

/\*DriverManager class:

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().

Commonly used methods of DriverManager class:

1) public static void registerDriver(Driver driver): is used to register the given driver with DriverManager.

2) public static void deregisterDriver(Driver driver): is used to deregister the given driver (drop the driver from the list) with DriverManager.

3) public static Connection getConnection(String url): is used to establish the connection with the specified url.

4) public static Connection getConnection(String url,String userName,String password): is used to establish the connection with the specified url, username and password.

\*/

}

}

package com.pioneercoders.batchprocessingex;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

public class BatchProcessingEx {

public static void main(String[] args) throws SQLException, ClassNotFoundException {

//if we want to insert into db we want to write statement or prepared statement

//When multiple inserts are to be made to the table in a database, the trivial way is to execute a query per record.

//However, this involves acquiring and releasing connection every time a record is inserted, which hampers application performance.

//We overcome, this(acquiring and releasing connection every-time) by making use of Batch operations in JDBC.

//We set the parameters in the java.sql.PreparedStatement and it to the batch using PreparedStatement.addBatch() method.

//Then when the batch size reaches a desired threshold, we execute the batch using PreparedStatement.executeBatch() method.

//another way to improve the performance is ConnectionPooling

//first of all set the autocommit false because one insert doesnt effect on another query

//However, when executing Queries in Batch, it is sometimes important to maintain the atomicity of the database. This can be a problem if one the commands in the batch throws some error since the commands after the exception will not be executed leaving the database in an inconsistent state. So we set the auto-commit to false and if all the records are executed successfully, we commit the transaction.

//This maintains the integrity of the the database.

Class.forName("com.mysql.jdbc.Driver");

Connection con= DriverManager.getConnection("jdbc:mysql://localhost:3306/med\_hosp\_db\_2", "root", "root");

con.setAutoCommit(false);

Statement st= con.createStatement();

st.addBatch("insert into user\_mst\_tbl(id,user\_id,user\_pwd,role\_id)values(1023,'vamsi','bcd',6)");

st.addBatch("update user\_mst\_tbl set user\_id='vamsi' where user\_pwd='bcd'");

st.addBatch("delete from user\_mst\_tbl where user\_pwd='bc'");

//st.addBatch("select \* from role\_mst\_tbl");

//we cannot apply batch for select statements st.addBatch("select \* from user\_mst\_tbl");because it internally executes executeupdate method only

//Exception in thread "main" java.sql.BatchUpdateException: Can not issue SELECT via executeUpdate().

//at com.mysql.jdbc.Statement.executeBatch(Statement.java:961)

//at com.jdbc.Batch.dataBatch(Batch.java:16)

//at com.jdbc.TestBatch.main(TestBatch.java:8)

int k[]=st.executeBatch();

System.out.println(k.length);

con.commit();

}

}

**JDBC Connection Pooling**

JDBC connection pooling is conceptually similar to any other form of object pooling. Database connections are often expensive to create because of the overhead of establishing a network connection and initializing a database connection session in the back end database. In turn, connection session initialization often requires time consuming processing to perform user authentication, establish transactional contexts and establish other aspects of the session that are required for subsequent database usage.

Additionally, the database's ongoing management of all of its connection sessions can impose a major limiting factor on the scalability of your application. Valuable database resources such as locks, memory, cursors, transaction logs, statement handles and temporary tables all tend to increase based on the number of concurrent connection sessions.

All in all, JDBC database connections are both expensive to initially create and then maintain over time. Therefore, as we shall see, they are an ideal resource to pool.

If your application runs within a J2EE environment and acquires JDBC connections from an appserver defined datasource then your application is probably already using connection pooling. This fact also illustrates an important characteristic of a best practices pooling implementation -- your application is not even aware it's using it! Your J2EE application simply acquires JDBC connections from the datasource, does some work on the connection then closes the connection. Your application's use of connection pooling is transparent. The characteristics of the connection pool can be tweaked and tuned by your appserver's administrator without the application ever needing to know.

If your application is not J2EE based then you may need to investigate using a standalone connection pool manager. Connection pool implementations are available from JDBC driver vendors and a number of other sources.

**JDBC Connection Scope**

How should your application manage the life cycle of JDBC connections? Asked another way, this question really asks - what is the scope of the JDBC connection object within your application? Let's consider a servlet that performs JDBC access. One possibility is to define the connection with servlet scope as follows.

|  |
| --- |
| import java.sql.\*;  public class JDBCServlet extends HttpServlet {      private Connection connection;      public void init(ServletConfig c) throws ServletException {       //Open the connection here     }      public void destroy() {      //Close the connection here     }      public void doGet (HttpServletRequest req, HttpServletResponse res) throws ServletException {        //Use the connection here       Statement stmt = connection.createStatement();       ..<do JDBC work>..   } } |

Using this approach the servlet creates a JDBC connection when it is loaded and destroys it when it is unloaded. The doGet() method has immediate access to the connection since it has servlet scope. However the database connection is kept open for the entire lifetime of the servlet and that the database will have to retain an open connection for every user that is connected to your application. If your application supports a large number of concurrent users its scalability will be severely limited!

**Method Scope Connections**

To avoid the long life time of the JDBC connection in the above example we can change the connection to have method scope as follows.

|  |
| --- |
| public class JDBCServlet extends HttpServlet {    private Connection getConnection() throws SQLException {     ..<create a JDBC connection>..   }    public void doGet (HttpServletRequest req, HttpServletResponse res) throws ServletException {      try {       Connection connection = getConnection();       ..<do JDBC work>..       connection.close();     }     catch (SQLException sqlException) {       sqlException.printStackTrace();     }   } } |

This approach represents a significant improvement over our first example because now the connection's life time is reduced to the time it takes to execute doGet(). The number of connections to the back end database at any instant is reduced to the number of users who are concurrently executing doGet(). However this example will create and destroy a lot more connections than the first example and this could easily become a performance problem.

In order to retain the advantages of a method scoped connection but reduce the performance hit of creating and destroying a large number of connections we now utilize connection pooling to arrive at our finished example that illustrates the best practices of connecting pool usage.

|  |
| --- |
| import java.sql.\*; import javax.sql.\*;  public class JDBCServlet extends HttpServlet {    private DataSource datasource;    public void init(ServletConfig config) throws ServletException {     try {       // Look up the JNDI data source only once at init time       Context envCtx = (Context) new InitialContext().lookup("java:comp/env");       datasource = (DataSource) envCtx.lookup("jdbc/MyDataSource");     }     catch (NamingException e) {       e.printStackTrace();     }   }    private Connection getConnection() throws SQLException {     return datasource.getConnection();   }    public void doGet (HttpServletRequest req, HttpServletResponse res) throws ServletException {     Connection connection=null;     try {       connection = getConnection();       ..<do JDBC work>..     }      catch (SQLException sqlException) {       sqlException.printStackTrace();     }     finally {       if (connection != null)          try {connection.close();} catch (SQLException e) {}       }     }   } } |

This approach uses the connection only for the minimum time the servlet requires it and also avoids creating and destroying a large number of physical database connections. The connection best practices that we have used are:

* A JNDI datasource is used as a factory for connections. The JNDI datasource is instantiated only once in init() since JNDI lookup can also be slow. JNDI should be configured so that the bound datasource implements connecting pooling. Connections issued from the pooling datasource will be returned to the pool when closed.
* We have moved the connection.close() into a finally block to ensure that the connection is closed even if an exception occurs during the doGet() JDBC processing. This practice is essential when using a connection pool. If a connection is not closed it will never be returned to the connection pool and become available for reuse. A finally block can also guarantee the closure of resources attached to JDBC statements and result sets when unexpected exceptions occur. Just call close() on these objects also.

**A transaction is a logical group of work that contains one or more SQL statements. Either all, or none of the statements need to be performed in order to preserve data integrity"**  
  
A complete task which is a combination of the multiple smaller tasks.  
For a major task to be completed, smaller tasks need to be successfully completed.  
If any one task fails then all the previous tasks are reverted back to the original state, means that database server guarantee to follow the **ACID** (**Automicity, Consistency, Isolation and Durability**) properties. A transaction is an atomic unit.

### ACID Properties -

* **Atomicity :**
  + Implies indivisibility
  + Any indivisible operation (one which will either complete in totally, or not at all) is said to be atomic
* **Consistency :**
  + A transaction must transition persistent data from one consistent state to another
  + In the event of a failure occurs during processing, data must be restored to the state it was in prior to the transaction
* **Isolation :**
  + Transactions should not affect each other
  + A transaction in progress, not yet committed or rolled back, must be isolated from other transactions
* **Durability :**
  + Once a transaction commits successfully, the state changes committed by that transaction must be durable < persistent, despite any failures that occur afterwards

### JDBC Transaction Management :

The **Connection** interface is used to manage transactions in Java applications. There is three methods **setAutoCommit(), commit() and rollback()** used to implement transaction in JDBC. By default true is set in setAutoCommit() method, when any SQL statement submit to the database server, database server commit it. We can set false in **setAutoCommit()** method to commit or rollback transaction either commit or rollback explicitly respectively by calling commit(0 and rollback() method.   
  
**Example - I am using Account table. The amount must be available at least Rs.- 1000.00. After withdrawing amount from account, it will check the amount in account, If it is less than 1000 the transaction must be rollback.**