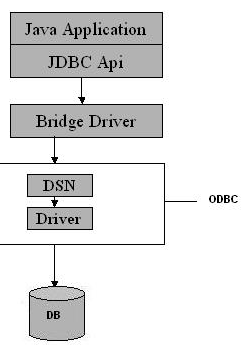
# JDBC Connectivity

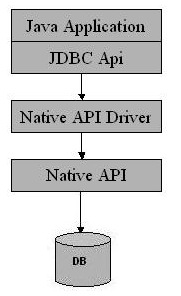
### Types of Driver

There are 4 types of drivers:

* Type I – also called as “JDBC - ODBC” driver.  
  JAVA API talks to Native API and then it talks to the Database.

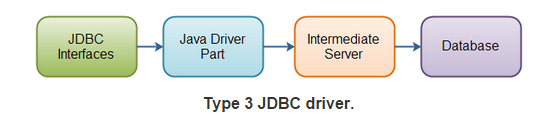


* Type II – also called as “Thick / Native” OR Partial Java driver.  
  There is a Native API which will talks to the Database.



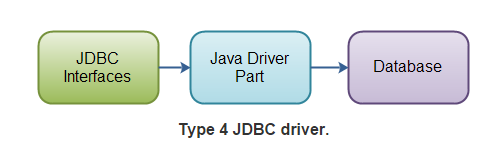
* Type III – also called as “All Java/Net Protocol” driver.  
  It is the driver which is based on 3-tier architecture. Client talks to the server using Net Protocol  
  driver and then server talks to the Database using any of the other driver (I / II / IV). It would be use where lots of concurrency to access to a database, lots of traffic to access to a database.

It requires another server application to install and maintain. Traversing the recordset may take longer, since the data comes through the backend server.



* Type IV – also called as “Thin / Pure” driver.  
  It is Pure JAVA API which directly talks to the database. Most often we use Type IV driver. Therefore, type IV is faster and most popular driver 4 types of JDBC Driver

With type 4 drivers, the user needs a different driver for each database.



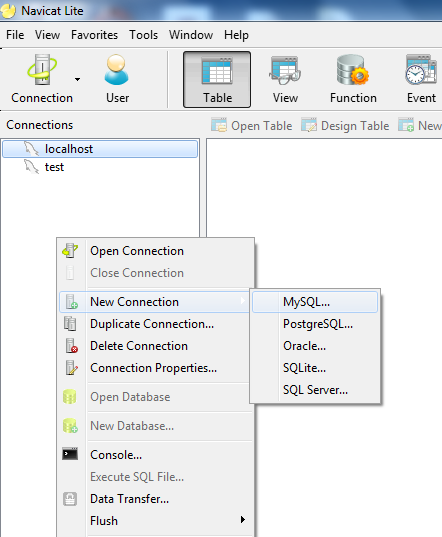
### Install & Configure Database

Install any database like mysql, hsqldb, etc

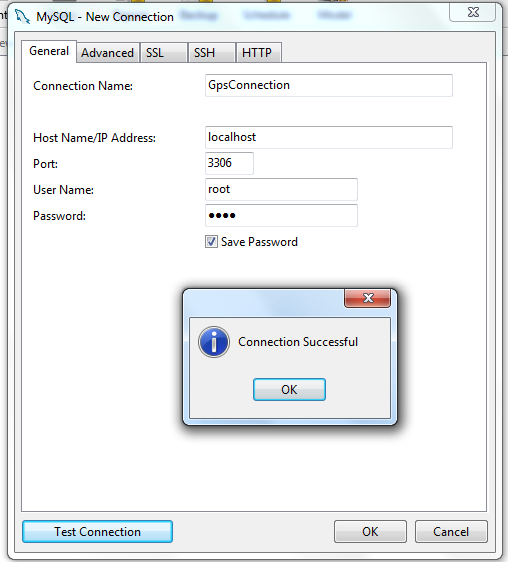
1. For HSQLDB, download it from - <http://sourceforge.net/projects/hsqldb/files/hsqldb/hsqldb_2_3/>
2. Inside Eclipse-> Window -> Show View -> Other -> Data Source Explorer
3. Right click on “Database Connection” -> Select “HSQLDB” as connection profile -> Select driver using New driver definition -> Specify Jar file “hsqldb.jar” inside hsqldb-2.3.2\hsqldb\lib
4. Provide database location, password & then Test Connection -> Ping succeeded
5. Next -> Finish
6. Add driver into your classpath
7. Install MySQL 5.5 Database (default port: 3306, Username: root & Password: root)

Add External Jar.

1. Install Client App like navicat100\_lite\_en to connect to database
2. Open Navicat -> Create new Database Connection using MySQL



1. “GpsConnection” & Test Connection -> finally OK



1. Create new database “gpsdb” & set mysql-connector-java java to classpath

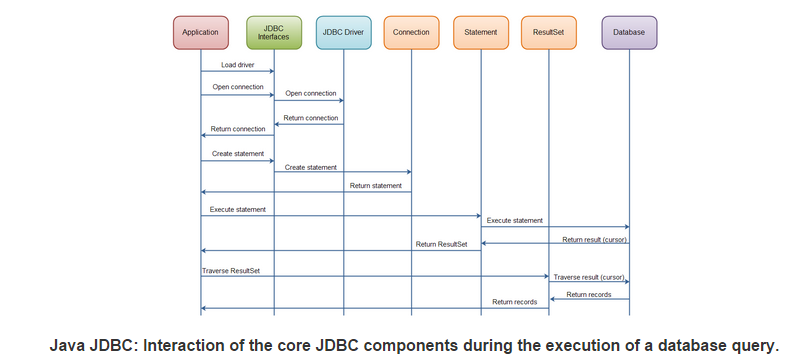
### Steps to Create a Database Connection

Steps to Create a JDBC Connection are:

1. Loading the driver using class.forName  
   Class.forName(driver).newInstance();
2. Establishing JDBC Connection either using Driver Manager / Data Source  
   Connection con = DriverManager.getConnection(url+db, user, pass);  
    OR  
   Connection con = new DataSource().getConnection();
3. Prepare SQL Statement / Prepared Statement  
   String SQL = “SELECT \* FROM  employee”;

Statement st = con.createStatement();

1. Execute the query  
   ResultSet res = st.executeQuery( SQL);
2. Process the Result  
   **while  (res.next()) {   
              String employeeName  = res.getInt( " employee\_name " );   
              System.out.println( employeeName  );   
   }**
3. Close the Connection  
   con.close();



### Create few table using SQL Scrapbook

Create **table** Employee (

EmpID **varchar**(200),

EmpName **varchar**(200),

EmpSalary **float**,

EmpAge **int**

);

**insert** **into** Employee **values** ('GPS0001','Amit',222.2,34);

**insert** **into** Employee **values** ('GPS0002','Amit1',223.2,35);

**insert** **into** Employee **values** ('GPS0003','Amit2',224.2,36);

commit;

### Database URL’s

|  |  |  |
| --- | --- | --- |
| **RDBMS** | **JDBC driver name** | **URL format** |
| MySQL | com.mysql.jdbc.Driver | **jdbc:mysql://**hostname/ databaseName |
| ORACLE | oracle.jdbc.driver.OracleDriver | **jdbc:oracle:thin:@**hostname:port Number:databaseName |
| DB2 | COM.ibm.db2.jdbc.net.DB2Driver | **jdbc:db2:**hostname:port Number/databaseName |
| Sybase | com.sybase.jdbc.SybDriver | **jdbc:sybase:Tds:**hostname: port Number/databaseName |

### Read data from Database using Statement

**Example** - com.src.java.jdbc – ReadData.java

Read data from Database using PreparedStatement

**Example** - com.src.java.jdbc – ReadDataByPS.java

**Question: Difference between Statement and PreparedStatement?**  
A Statement is usually parsed and executed each time it is submitted to the database. A PreparedStatement may be parsed once and executed repeatedly with different parameters.  
Most relational databases handle a JDBC / SQL query in four steps:    
    1. Parse the incoming SQL query  
  2. Compile the SQL query   
 3. Plan/optimize the data acquisition path   
 4. Execute the optimized query / acquire and return data     
A Statement will always proceed through the four steps above for each SQL query sent to the database. A PreparedStatement pre-executes steps (1) - (3) in the execution process above. Thus, when creating a PreparedStatement some pre-optimization is performed immediately. The effect is to lessen the load on the database engine at execution time

Read data from Database using DataSource  
**Example** - com.src.java.jdbc – ReadDataByPSusingDataSource.java

**Question: What is difference between DataSource and DriverManager of JDBC?**Connection object got from Datasource.getConnection() ,when closed will be returned back to connection pool..but will not be closed permanently whereas Connection object retrieved from DriverManager.getConnection() when closed will be returned permanently.  
One difference between both is that Datasource has properties with it, which represents the Source of data it represents.  
Other is, DataSource supports the JNDI Service, so we can give any logical name to that datasource eg. MSSQLDataSource to MS SQL server, MySQLDataSource to MySQL.   
One more is, DataSource can be separately created and managed and can be accessed from any of the Application through lookup facility of JNDI. DataSource is the standard interface in JDBC, which need to be implemented by database Driver, can be implemented to maintain the pool of connections.

**Question: Do we need to close ResultSet before closing connection?**

Yes but depend upon scenario. Most Java programmers close a connection with database directly without closing the ResultSet. This is okay with standalone Java programs, where the Java compiler automatically closes the Resultset for you, but when we deploy this code in a servlet, and then we are not guaranteed this behavior. So to be safe, remember to close the ResultSet explicitly before closing the Connection.

### ResultSet Type

When you create a ResultSet there are three attributes you can set. These are-

1. Type
2. Concurrency
3. Holdability

You set these already when you create the Statement or PreparedStatement, like this:

Statement statement = connection.createStatement(ResultSet.TYPE\_FORWARD\_ONLY,

ResultSet.CONCUR\_READ\_ONLY, ResultSet.CLOSE\_CURSORS\_OVER\_COMMIT);

PreparedStatement statement = connection.prepareStatement(sql,

ResultSet.TYPE\_FORWARD\_ONLY, ResultSet.CONCUR\_READ\_ONLY, ResultSet.CLOSE\_CURSORS\_OVER\_COMMIT

);

At the time of writing there are three ResultSet types:

1. ResultSet.TYPE\_FORWARD\_ONLY
2. ResultSet.TYPE\_SCROLL\_INSENSITIVE
3. ResultSet.TYPE\_SCROLL\_SENSITIVE

The default type is TYPE\_FORWARD\_ONLY

TYPE\_FORWARD\_ONLY means that the ResultSet can only be navigated forward. That is, you can only move from row 1, to row 2, to row 3 etc. You cannot move backwards in the ResultSet.

TYPE\_SCROLL\_INSENSITIVE means that the ResultSet can be navigated (scrolled) both forward and backwards. You can also jump to a position relative to the current position, or jump to an absolute position. The ResultSet is insensitive to changes in the underlying data source while the ResultSet is open. That is, if a record in theResultSet is changed in the database by another thread or process, it will not be reflected in already openedResulsSet's of this type.

TYPE\_SCROLL\_SENSITIVE means that the ResultSet can be navigated (scrolled) both forward and backwards. You can also jump to a position relative to the current position, or jump to an absolute position. The ResultSet is sensitive to changes in the underlying data source while the ResultSet is open. That is, if a record in the ResultSet is changed in the database by another thread or process, it will be reflected in already opened ResulsSet's of this type.

### ResultSet Concurrency

A ResultSet can have one of two concurrency levels:

1. ResultSet.CONCUR\_READ\_ONLY
2. ResultSet.CONCUR\_UPDATABLE

CONCUR\_READ\_ONLY means that the ResultSet can only be read.

CONCUR\_UPDATABLE means that the ResultSet can be both read and updated.

### ResultSet Holdability

The ResultSet holdability determines if a ResultSet is closed when the commit() method of the underlyingconnection is called.

There are two types of holdability:

1. ResultSet.CLOSE\_CURSORS\_OVER\_COMMIT
2. ResultSet.HOLD\_CURSORS\_OVER\_COMMIT

The CLOSE\_CURSORS\_OVER\_COMMIT holdability means that all ResultSet instances are closed when connection.commit() method is called on the connection that created the ResultSet.

The HOLD\_CURSORS\_OVER\_COMMIT holdability means that the ResultSet is kept open when the connection.commit() method is called on the connection that created the ResultSet.

The HOLD\_CURSORS\_OVER\_COMMIT holdability might be useful if you use the ResultSet to update values in the database. Thus, you can open a ResultSet, update rows in it, call connection.commit() and still keep the sameResultSet open for future transactions on the same rows.

Insert data into Database

**Example** - com.src.java.jdbc – InsertData.java

Update data into Database

**Example** - com.src.java.jdbc – UpdateData.java

Delete data from Database

**Example** - com.src.java.jdbc – DeleteData.java

### Batch Updates

A batch update is a batch of updates grouped together, and sent to the database in one "batch", rather than sending the updates one by one.

Sending a batch of updates to the database in one goes, is faster than sending them one by one, waiting for each one to finish. There is less network traffic involved in sending one batch of updates (only 1 round trip), and the database might be able to execute some of the updates in parallel. The speed up compared to executing the updates one by one, can be quite big.

You can batch SQL inserts, updates and deletes. It does not make sense to batch select statements.

**Example of Statement Batch Updates**

Statement statement = null;

try{

statement = connection.createStatement();  
 statement.addBatch("update people set firstname='John' where id=123");  
 statement.addBatch("update people set firstname='Eric' where id=456");  
 statement.addBatch("update people set firstname='May' where id=789");

int[] recordsAffected = statement.executeBatch();  
} finally {  
 if(statement != null) statement.close();  
}

**Example of PreparedStatement Batch Updates**

String sql = "update people set firstname=? , lastname=? where id=?";

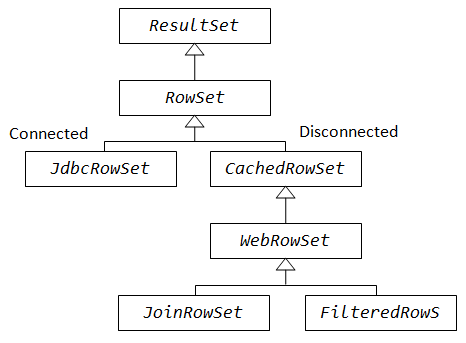
PreparedStatement preparedStatement = null;  
try{  
 preparedStatement = connection.prepareStatement(sql);

preparedStatement.setString(1, "Gary");  
 preparedStatement.setString(2, "Larson");  
 preparedStatement.setLong (3, 123);  
 preparedStatement.addBatch();

preparedStatement.setString(1, "Stan");  
 preparedStatement.setString(2, "Lee");  
 preparedStatement.setLong (3, 456);  
 preparedStatement.addBatch();

int[] affectedRecords = preparedStatement.executeBatch();  
}finally {  
 if(preparedStatement != null) {  
 preparedStatement.close();  
 }  
}

### Rowset



**RowSet** interface is basically an extension of JDBC ResultSet and is a part of thejavax.sql package. RowSet object holds tabular data in a way that makes it more flexible and easier to use than a result set. A RowSet object is considered either **connected** or **disconnected**.

Connected RowSet are those which establish a connection with the database and retain it until the application terminates.

Disconnected RowSet on the other hand establishes a connection, executes a query such as retrieving data from the database and closes the connection.

According to the implementation there are five different categories of RowSet – **JdbcRowSet, CachedRowSet, WebRowSet, FilteredRowSet,**and **JoinRowSet**.

Out of these five only **JdbcRowSet** is connected RowSet, rest of them are disconnected row sets.

**JdbcRowSet**

It basically acts as a wrapper around the *ResultSet* object with some additional functionality. The primary advantage of using JdbcRowSet is that it enables the*ResultSet* object to be used as a JavaBeans component.

**Example** - com.src.java.jdbc – JdbcRowSetDemo.java

**CachedRowSet**

It is a disconnected *RowSet* that acts as a container for database records and caches them in  memory. In a sense we may think of it as a subset of actual records maintained in memory. This reminds us somewhat of the concept of 'VIEW' we create from the actual table through SQL. Being disconnected, *CachedRowSet* is more lightweight. On the downside, connection has to be established every time to reflect any changes, performance may be slower than *JdbcRowSet* but on the upside it is lightweight and we can get a better efficiency when working with large amount of data.

**WebRowSet**

It extends *CachedRowSet* capabilities but is very special in the sense that in addition to providing all the features of *CachedRowSet*, it can read and write XML document. This is particularly seen in an enterprise application scenario or in web service communication. Records fetched from the database can be shipped across tiers over internet protocol in an enterprise application without having to maintain an active connection.

**Example** - com.src.java.jdbc – WebRowSetDemo.java

**FilteredRowSet**

It is an extension of *WebRowSet*. So it has all the capabilities of *WebRowSet* as well as *CachedRowSet*. With this *RowSet* we can apply filtering criteria to fetch selected rows from the data source so that we can work with the relevant data. This *RowSet* object paves the way for a disconnected yet live filtering mechanism without firing a query into the database every time. We can limit what we want to visualize from the record list without the involvement of a database connection. It is something like using the WHERE clause without writing an SQL.

**JoinRowSet**

It also extends *WebRowSet*. In addition to the capabilities of *WebRowSet* and *CachedRowSet*, we can perform a SQL JOIN operation without connecting to a data source. This*RowSet* enables us to create SQL JOIN between *RowSet* objects. Related records from different *RowSets* can be combined to form this*RowSet* object.  
  
**Question: What is difference between ResultSet and RowSet?**

* A ResultSet maintains a connection to a database and because of that it can’t be serialized and also we can’t pass the Resultset object from one class to other class across the network.
* RowSet is a disconnected, serializable version of a JDBC ResultSet and also the RowSet extends the ResultSet interface so it has all the methods of ResultSet. The RowSet can be serialized because it doesn’t have a connection to any database and also it can be sent from one class to another across the network.

**Question: What is difference between Connected and Disconnected RowSet?**

* Disconnected database connected to the database only when they want to read or write, all the times they are disconnected from the database, on the other hand, Connected database keep JDBC connection alive all time.
* Only JdbcRowSet from JDBC 4.1 API is connected, rest like CachedRowSet, WebRowSet are disconnected RowSet implementation.
* Disconnected RowSet are Serializable and that's why suitable for sending over the network

### CallableStatement

The interface used to execute SQL stored procedures. The JDBC API provides a stored procedure SQL escape syntax that allows stored procedures to be called in a standard way for all RDBMSs. This escape syntax has one form that includes a result parameter and one that does not.

To call stored procedures, you invoke methods in the CallableStatement class. The basic steps are:

1. Invoke the Connection.prepareCall method to create a CallableStatement object.
2. Invoke the CallableStatement.setXXX methods to pass values to the input (IN) parameters.
3. Invoke the CallableStatement.registerOutParameter method to indicate which parameters are output-only (OUT) parameters, or input and output (INOUT) parameters.
4. Invoke one of the following methods to call the stored procedure:

**CallableStatement.executeUpdate**

Invoke this method if the stored procedure does not return result sets.

**CallableStatement.executeQuery**

Invoke this method if the stored procedure returns one result set.

**CallableStatement.execute**

Invoke this method if the stored procedure returns multiple result sets.

1. If the stored procedure returns result sets, retrieve the result sets.
2. Invoke the CallableStatement.getXXX methods to retrieve values from the OUT parameters or INOUT parameters.
3. Invoke the CallableStatement.close method to close the CallableStatement object when you have finished using that object.

Eaxmple- Using CallableStatement methods for a stored procedure call with parameter markers

int ifcaret;

int ifcareas;

int xsbytes;

String errbuff;

Connection con;

CallableStatement cstmt;

ResultSet rs;

...

cstmt = con.prepareCall("CALL DSN8.DSN8ED2(?,?,?,?,?)"); **1**

// Create a CallableStatement object

cstmt.setString (1, "DISPLAY THREAD(\*)"); **2**

// Set input parameter (DB2 command)

cstmt.registerOutParameter (2, Types.INTEGER); **3**

// Register output parameters

cstmt.registerOutParameter (3, Types.INTEGER);

cstmt.registerOutParameter (4, Types.INTEGER);

cstmt.registerOutParameter (5, Types.VARCHAR);

cstmt.executeUpdate(); // Call the stored procedure **4**

ifcaret = cstmt.getInt(2); // Get the output parameter values **6**

ifcareas = cstmt.getInt(3);

xsbytes = cstmt.getInt(4);

errbuff = cstmt.getString(5);  
cstmt.close();

Example- Retrieving known result sets from a stored procedure

CallableStatement cstmt;

ResultSet rs;

int i;

String s;

...

cstmt.execute(); // Call the stored procedure **1**

rs = cstmt.getResultSet(); // Get the first result set **2**

while (rs.next()) { // Position the cursor **3**

i = rs.getInt(1); // Retrieve current result set value

System.out.println("Value from first result set = " + i);

// Print the value

}

cstmt.getMoreResults(); // Point to the second result set **4a**

// and close the first result set

rs = cstmt.getResultSet(); // Get the second result set **4b**

while (rs.next()) { // Position the cursor **4c**

s = rs.getString(1); // Retrieve current result set value

System.out.println("Value from second result set = " + s);

// Print the value

}

rs.close(); // Close the result set  
cstmt.close(); // Close the statement

# Connection Pooling using JNDI Services

### Inside Tomcat using Servlet.xml

For Example-  
<Context path="/pE" docBase="c:/temp/pE" debug="0" reloadable="true" >

<Resource name="jdbc/myDataSource" auth="Container" type="javax.sql.DataSource" />

<ResourceParams name="jdbc/myDataSource">

<parameter>

<name>factory</name>

<value>org.apache.commons.dbcp.BasicDataSourceFactory</value>

</parameter>

<parameter>

<name>dataSourceName</name>

<value>myDataSource</value>

</parameter>

<parameter>

<name>maxActive</name>

<value>25</value>

</parameter>

<parameter>

<name>maxIdle</name>

<value>10</value>

</parameter>

<parameter>

<name>maxWait</name>

<value>10000</value>

</parameter>

<parameter>

<name>url</name>

<value>jdbc:Microsoft:sqlserver://localhost:1433;DatabaseName=pE</value>

</parameter>

<parameter>

<name>username</name>

<value>sa</value>

</parameter>

<parameter>

<name>password</name>

<value>sql</value>

</parameter>

<parameter>

<name>driverClassName</name>

<value>com.microsoft.jdbc.sqlserver.SQLServerDriver</value>

</parameter>

</ResourceParams>

</Context>

Make an entry inside web.xml:  
<resource-ref>

<res-ref-name>jdbc/myDataSource</res-ref-name>

<res-type>javax.sql.DataSource</res-type>

<res-auth>Container</res-auth>

</resource-ref>

How we get Connection using JNDI inside Servlet?  
import java.io.\*;

import java.sql.\*;

import javax.naming.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

import javax.sql.\*;

public class validateLogin extends HttpServlet

{

private Connection conn = null;

private ResultSet rs = null;

private PreparedStatement psmt = null;

private String login,password,sql,UserName;;

private boolean userExist=false;

private DataSource ds=null;

public void init(ServletConfig config) throws ServletException

{

try

{

Context initCtx = new InitialContext();

ds = (DataSource) initCtx.lookup("java:comp/env/jdbc/myDataSource");

}

catch (Exception e)

{

throw new ServletException(e.getMessage());

}

}

private Connection getConnection() throws SQLException

{

return ds.getConnection();

}

public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException

{

login = request.getParameter("txtLogin");

password = request.getParameter("txtPassword");

try

{

conn = getConnection();

sql = "select \* from jobSeekerDetails where UserName=? and Password=?";

psmt = conn.prepareStatement(sql);

psmt.setString(1,login);

psmt.setString(2,password);

rs = psmt.executeQuery();

if(rs.next())

{

userExist = true;

HttpSession session = request.getSession(true);

session.setAttribute("userLogin",login);

response.sendRedirect("../");

return;

}

else

{

response.sendRedirect("../errorPage.jsp");

}

}

catch(Exception e)

{

System.out.println(e.getMessage() + " validateLogin.jsp Line No : " + 51);

}

finally

{

try

{

rs.close();

conn.close();

}

catch(Exception e){}

}

}

}

### Connection Pooling using Context.xml

Tomcat JDBC connection pool can be configured with META-INF/context.xml.

<Context>

<Resource name="jdbc/EmployeeDB"

auth="Container"

type="javax.sql.DataSource"

username="dbusername"

password="dbpassword"

driverClassName="org.hsql.jdbcDriver"

url="jdbc:HypersonicSQL:database"

maxActive="8"

maxIdle="4"/>

</Context>

To use the DataSource in your Java code:  
Context initCtx = new InitialContext();

DataSource ds = (DataSource) initCtx.lookup("java:comp/env/jdbc/EmployeeDB");

Connection conn = ds.getConnection();

// ... use this connection to access the database ...

conn.close();

In contrast to Tomcat documentation, there's no need to declare resource-ref in web.xml.