Java Data Base Connectivity

Pieter van den Hombergh

Fontys Hogeschool voor Techniek en Logistiek

March 25, 2018

Java Data Base Connectivity

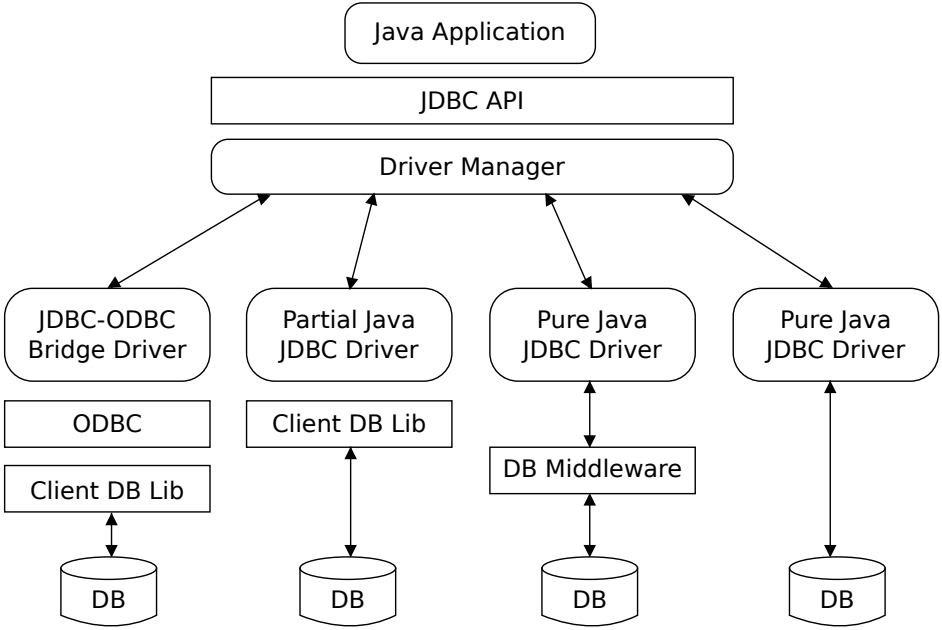
JDBC JDBC is a Java database connectivity technology (Java Standard Edition platform) from Oracle Corporation.

- This technology is an API for the Java programming language that defines how a client may access a database.
- It provides methods for querying and updating data in a database.
- JDBC is oriented towards relational databases and is (database) vendor neutral.
- **source: wikipedia**

JDBC History

- Part of Java since 1997 (Sun Java version JDK 1.1).
- Part of the Java Community Process (JCP) from version 3.1, Java 1.4
- Current version JDBC 4.2 with Java 8 (profile >=compact 2)
- Most of the jdbc classes and interfaces are in the `java.sql` package.

JDBC architecture and driver variants



Database and driver

- From the java API you can see that the java.sql mainly contains interfaces.
- For each database you typically need a separate implementation.
- But the way you talk to the database (Java methods, not sql) is the same for all databases.
- Built in into the JDK is JavaDB a.k.a. Derby DB, an in memory database.
- For postgresql `https://jdbc.postgresql.org/download.html`
  Postgresql jdbc download
  Pick version 42.0.0 for the latest version. The latest stable prostgreSQL version is 9.6.

The 7 steps using JDBC

To effectively use JDBC in your application you need to perform 7 steps in your program.

1. (Load the driver, no longer needed for Java version >= Java6)
2. Define the Connection URL
3. Establish the Connection
4. Create a Statement object
5. Execute a query
6. Process the results
7. Close the connection

Often you do steps 1 through 4 once and reuse the connection after that. Since opening a connection is expensive[1], postpone step 7 until you are done with the connection.

[1] in particular when using a remote host over SSL/TLS

Select and load driver and create connection

- Because the `Driver` is an interface, you need to load the appropriate class from the appropriate library (jar file).
- First make sure you have the driver installed. Advice: install it under `/usr/share/java` under Linux or OSX or similar under windows `C:\usr\share\java`
- Make it known to netbeans by creating a **Library**
- Name the library like this: `‘‘Postgresql JDBC4.1 Driver’’`, so
  1. you learn how to do this.
  2. we all use the **same name**, which is essential when working in groups.

DEMO

Load the driver into you application

- Set up the driver and load the driver.
- Add the library to the project.
- See postgresql jdbc tutorial.

Connection setup

```
28      public static void main(String[] args) throws
29              ClassNotFoundException, IOException {
30          String sql = "SELECT * FROM president ";
31          System.out.println("========== alternative usage ==");
32
33          try (Connection con = new ConnectionFactory()
34                  .getConnection()) {
35              ResultSetPrinter.getDBProperties(con);
36              doQuery(con, sql, out);
37
38          } catch (SQLException ex) {
39              logger.log(Level.SEVERE, null, ex);
```

Note that the connection is a resource (the first) in a try-with-resources block.

Connection acquisition and use

```
38      try ( Connection conn = DriverManager
39              .getConnection( url, props );
40              Statement st = conn.createStatement();
41              ResultSet rs = st.executeQuery( sql ); ) {
42        while ( rs.next() ) {
43          ResultSetMetaData meta = rs.getMetaData();
44          // note count starts for 1. Database ;-))
45          for ( int col = 1; col <= meta
46                  .getColumnCount(); col++ ) {
47            String colName = meta.getColumnName( col );
48            String colType = meta.getColumnTypeName( col );
49            String colClass = meta.getColumnClassName( col );
50            Object data = rs.getObject( col );
51            out.printf( "%s (%s) = %s\n", colName,
52                        colClass,
53                        Objects.toString( data, "NULL" ) );
54          }
55        }
56      } catch ( SQLException ex ) {
57        logger.log( Level.SEVERE, null, ex );
58      }
```

Note the use of try-with-resources, so you can skip closing connection, statement and result set.

Passing the connection, alternative usage

- Creating and deleting connections can be costly.
- In a web app you typically get one per request-response and use it throughout the processing

```
77  private static void doQuery( Connection con, String query,
78          PrintStream out ) throws SQLException {
79    try ( Statement st = con.createStatement();
80            ResultSet rs = st.executeQuery( query ) ) {
81      while ( rs.next() ) {
82        ResultSetMetaData meta = rs.getMetaData();
83        // note count starts for 1. Database ;-))
84        for ( int cols = 1; cols <= meta
85                .getColumnCount(); cols++ ) {
86          String colName = meta.getColumnName( cols );
87          String colType = meta.getColumnTypeName( cols );
88          Object data = rs.getObject( cols );
89          out.printf( "%s (%s) = %s\n", colName,
90                      colType,
91                      Objects.toString( data, "NULL" ) );
92        }
93      }
94    }
95  }
```

- Note that the connection stays open in this usage

No hardcoded secrets

It is not a good idea to put security relevant information in your source code.

- some of these things are configuration data.
  - These should be separate from the source code
  - You want them changeable on deployment. Think staging server or use alternate database.
  - Passwords should NOT be well known.

Reading information from a file

- The java approach is to put this information into a properties file or in a xml file.
- Here we show the properties approach.
- Reading the file depends on the location of the properties file.
  - in a (local) file system. - handle as file.
  - from a server (url). Get stream from url.
  - For the deployment artifact (jar, war or ear file), get stream from jar...

Read from jar

When reading from a jar, you should know the PATH or 'directory' structure.

```
String url = "jdbc:postgresql:"
    + "//localhost/presidentDB";
Properties props = new Properties();
InputStream inputStream = PgJDBCDemo.class.getClassLoader()
    .getResourceAsStream("resources/conn.properties");
System.out.println("inputStream = " + inputStream);
props.load(inputStream);
props.forEach((a, b) -> System.out.println(a + " = " + b));
```

DEMO: Have a look in the project.

Best solution

A combination of two:

- A default properties file, which is part of the deployment artifact.
- A 'real' configuration file, which is managed by the system administrator and adapted to the real values at deployment time.
- This last file can then be properly managed (security) wise.
- The real file should get preference, when available, so your code should check for its existence first and only default when the file is not available.

Meta: data on the data

- On the database it selves
  - `connection.getMetaData().getDatabaseProductName()`
    `connection.getMetaData().getDatabaseProductVersion()`
- On the tables
  - Consult the schema **information_schema**
  - `select table_name from information_schema.tables where table_schema='public'`
- On result sets:
  - `resultSet.getMetaData().getColumnCount()`.
    Remember that column numbers start with 1, not 0
  - You can get information on column type, width, etc

JDBC Statements

- A statement is a resource (and `AutoClosable`).
- It can be reused with different query text, but there always is only one result set with any statement.
- Typically, you use a statement per query text.
- Statements should not be shared between `Threads`
- Close the statement and result set after use. (Easy if you use try-with-resources, then it is done *automagically*.

Using statements.

The purpose of a statement is to be executed.

- First you need to declare what needs to be done.
- For that you use ordinary SQL, like in DBS1.

```
doQuery(con, sql, out);
```

- If you use fixed (as in hardcoded) query parameters this is fine.
- That is what you do when e.g. reading from the data dictionary.

Prepared Statements

Use case:

- When using a statements over and over again.
- When your queries take parameter values for column values.

Use prepared statements.

- The query string is a template, that is filled in before using it.

Advantage:

- Easier to use then building query strings over and over again.
- Faster on most drivers and databases.
- Is much more robust against SQL injection.

SQL injection

Something prepared statements help to prevent.

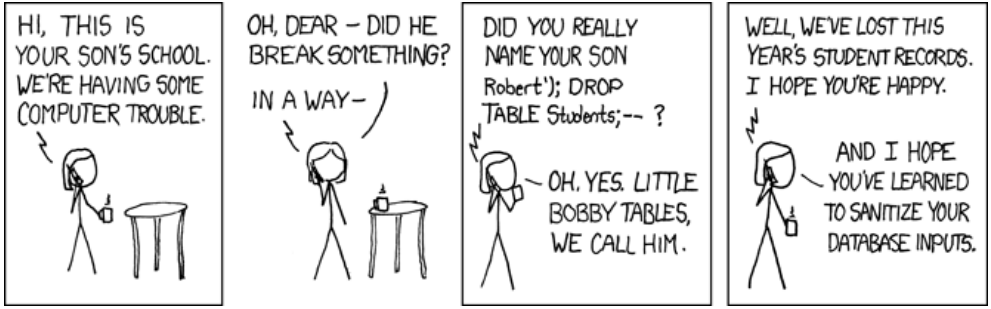


Figure: Her daughter is named "*Help I'm trapped in a driver's license factory.*" source:`https://xkdc.com/327`

By the way: did you know that PHP also supports prepared statements for databases that have them, like PostgreSQL.

Demo

Drop the presidents table:

```
'; drop table president cascade;commit;
```

Prepared statement example

Let's anticipate the next election result, assuming Hillary wins.

goal and test

```
28      String pName = "RODHAM-CLINTON HD";
29      String pParty = "RODHAM-CLINTON HD";
30
31      President hillary;
32      hillary = new President( pName, 1947,
33                               pParty, 5 );
34      String isSheIn
35              = "select * from president where name=?";
```

Let's get Hillary elected

Insert a prezz

```
75  static int insertNextPresident( Connection con,
76          President p ) throws SQLException {
77
78    String template
79            = "insert into president "
80            + "(name,birth_year,party,state_id_born)\n"
81            + "values(?,?,?,?)";
82    PreparedStatement inStmt = con.prepareStatement(
83            template );
84    inStmt.setString( 1, p.name );
85    inStmt.setInt( 2, p.birthyear );
86    inStmt.setString( 3, p.party );
87    inStmt.setInt( 4, p.state_id_born );
88    return inStmt.executeUpdate();
89  }
```

See how pres values are set into the statement by using the column numbers.

Let's test if we can get her in

and out again...

Just kidding

```
36    try ( Connection con = createDemoConnection() ) {
37      con.setAutoCommit( false );
38      insertNextPresident( con, hillary );
39      try ( PreparedStatement pst = con.prepareStatement(
40              isSheIn ); ) {
41        pst.setString( 1, pName );
42        ResultSet rs = pst.executeQuery();
43        new ResultSetPrinter( rs ).printTable( System.out );
44        con.rollback(); // just testing...
45      }
46    }
```

- Note that the connection's auto-commit is turned off in line 37. This allows us to roll back our changes in line 44.
- Also note that you can roll back the effects on the tables, but the sequence used for the pres id value is NOT rolled back.
  - This is by (database) design[2]. A sequence should stay unique. Once issued, it is not revocable.

[2] all database supporting sequences show this behavior

Question

For any remaining questions, have a look at the tutorial.
You might also want to study the lesson example. See the Java2 repository.