J2EE

JDBC

Connection pooling

Presented by Bartosz Sakowicz

JDBC

There are seven standard steps in querying databases:

1. **Load the JDBC driver.**
2. **Define the connection URL.**
3. **Establish the connection.**
4. **Create a statement object.**
5. **Execute a query or update.**
6. **Process the results.**
7. **Close the connection.**

Presented by Bartosz Sakowicz

DMCS TUL

Loading the driver

The **driver** is the piece of software that knows how to talk to the actual database server. To load the driver, all you need to do is to load the appropriate class; a static block in the class itself automatically makes a driver instance and registers it with the JDBC driver manager.

Example:

```
try {
        Class.forName("connect.microsoft.MicrosoftDriver");
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Class.forName("com.sybase.jdbc.SybDriver");
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch(ClassNotFoundException cnfe) {
        System.err.println("Error loading driver: " + cnfe);
}
```

Loading the driver(2)

JDBC driver translates calls written in the Java programming language into the specific format required by the server.

Most database vendors supply free JDBC drivers for their databases, but there are many third-party vendors of drivers for older databases.

An up-to-date list is avaliable at:

http://java.sun.com/products/jdbc/drivers.html.

Defining the connection URL

Once you have loaded the JDBC driver, you need to specify the location of the database server. URLs referring to databases use the **jdbc:** protocol and have the server host, port, and database name (or reference) embedded within the URL. The exact format will be defined in the documentation that comes with the particular driver.

Examples:

```
String host = "dbhost.yourcompany.com";
String dbName = "someName";
int port = 1234;
String oracleURL = "jdbc:oracle:thin:@" + host +
":" + port + ":" + dbName;
String sybaseURL = "jdbc:sybase:Tds:" + host +
":" + port + ":" + "?SERVICENAME=" + dbName;
```

Applets and JDBC

JDBC is most often used from servlets or regular desktop applications but is also sometimes employed from applets.

If you use JDBC from an applet, remember that browsers prevent applets from making network connections anywhere except to the server from which they were loaded. Consequently, to use JDBC from applets, either the database server needs to reside on the same machine as the HTTP server or you need to use a proxy server that reroutes database requests to the actual server.

Establishing the connection

To make the actual network connection, pass the URL, the database user-name, and the password to the getConnection method of the Driver-Manager class. Note that getConnection throws an **SQLException**, so you need to use a try/catch block:

```
String username = "myname";
String password = "secret";
Connection connection =
DriverManager.getConnection(oracleURL, username, password);
```

Establishing the ... (2)

An optional part of this step is to look up information about the database by using the getMetaData method of Connection. This method returns a DatabaseMetaData object which has methods to let you discover the name and version of the database itself (getDatabaseProductName, getDatabaseProductVersion) or of the JDBC driver (getDriverName, getDriverVersion). **Example**:

```
DatabaseMetaData dbMetaData = connection.getMetaData();
String productName =
dbMetaData.getDatabaseProductName();
System.out.println("Database: " + productName);
String productVersion =
dbMetaData.getDatabaseProductVersion();
System.out.println("Version: " + productVersion);
```

Creating a statement

A **Statement** object is used to send queries and commands to the database and is created from the Connection as follows:

Statement statement = connection.createStatement();

Executing a query

Once you have a Statement object, you can use it to send SQL queries by using the **executeQuery** method, which returns an object of type ResultSet. **Example:**

String query = "SELECT col1, col2, col3 FROM sometable";
ResultSet resultSet = statement.executeQuery(query);

To modify the database, use **executeUpdate** instead of executeQuery, and supply a string that uses UPDATE, INSERT, or DELETE. Other useful methods in the Statement class include **execute** (execute an arbitrary command) and **setQueryTimeout** (set a maximum delay to wait for results).

Processing the result

The simplest way to handle the results is to process them one row at a time, using the ResultSet's **next method** to move through the table a row at a time.

Within a row, ResultSet provides various **getXxx** methods that take a column index or column name as an argument and return the result as a variety of different Java types. For instance, use getInt if the value should be an integer, getString for a String, and so on for most other data types.

The first column in a ResultSet row has index 1, not 0 following SQL conventions.

Processing the result(2)

Example:

```
while(resultSet.next()) {
        System.out.println(results.getString(1) + " " +
        results.getString(2) + " " +
        results.getString(3));
}
```

In addition to the getXxx and next methods, other useful methods in the ResultSet class include **findColumn** (get the index of the named column), **wasNull** (was the last getXxx result SQL NULL?), and **getMetaData** (retrieve information about the ResultSet in a ResultSetMetaData object).

Closing the connection

To close the connection explicitly, you would do:

connection.close();

You should postpone this step if you expect to perform additional database operations, since the overhead of opening a connection is usually large.

JDBC example

```
try {
        Class.forName(driver);
        Connection connection =
        DriverManager.getConnection(url, username, password);
        Statement statement = connection.createStatement();
        String query = "SELECT * FROM fruits";
        ResultSet resultSet = statement.executeQuery(query);
        while(resultSet.next()) {
                System.out.print(" " + resultSet.getInt(1));
                System.out.print(" " + resultSet.getInt(2));
                System.out.print(" $" + resultSet.getFloat(3));
        }
} catch(ClassNotFoundException cnfe) {
        System.err.println("Error loading driver: " + cnfe);
} catch(SQLException sqle) {
        System.err.println("Error connecting: " + sqle);
}
```

Prepared statements

• If you are going to execute similar SQL statements multiple times, using **“prepared” statements** can be more efficient than executing a raw query each time.

• **The idea is to create a parameterized statement in a standard form that is sent to the database for compilation before actually being used.**

• You use a question mark to indicate the places where a value will be substituted into the statement.

• Each time you use the prepared statement, you replace some of the marked parameters, using a se*tXxx* call corresponding to the entry you want to set (using 1-based indexing) and the type of the parameter (e.g., setInt, setString, and so forth).

• You then use executeQuery (if you want a ResultSet back) or execute/executeUpdate as with normal statements.

Prepared ... - example

```
Connection connection =
        DriverManager.getConnection(url, user, password);
String template =
        "UPDATE employees SET salary = ? WHERE id = ?";
PreparedStatement statement =
        connection.prepareStatement(template);
float[] newSalaries = getNewSalaries();
int[] employeeIDs = getIDs();
for(int i=0; i<employeeIDs.length; i++) {
        statement.setFloat(1, newSalaries[i]);
        statement.setInt(2, employeeIDs[i]);
        statement.execute();
}
```

The performance advantages of prepared statements can vary , depending on how well the server supports precompiled queries and how efficiently the driver handles raw queries **(up to 50%)**.

executeUpdate

Whereas executeQuery returns a ResultSet object containing the results of the query sent to the DBMS, **the return value for executeUpdate is an int that indicates how many rows of a table were updated**.

Example:

```
updateSales.setInt(1, 50);
updateSales.setString(2, "Espresso");
int n = updateSales.executeUpdate();

// n equals number of updated rows
```

Transactions

A **transaction** is a set of one or more statements that are executed together as a unit, so either all of the statements are executed, or none of the statements is executed.

When a connection is created, it is in **auto-commit mode**. This means that each individual SQL statement is treated as a transaction and will be automatically committed right after it is executed. The way to allow two or more statements to be grouped into a transaction is to disable auto-commit mode.

Example (con is an active connection):

con.setAutoCommit(false);

Commiting a transaction

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
"UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE
?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
"UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE
COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

Rollbacking a transaction

Calling the method **rollback** aborts a transaction and returns any values that were modified to their previous values.

If you are trying to execute one or more statements in a transaction and get an **SQLException** , you should call the method rollback to abort the transaction and start the transaction all over again. That is the only way to be sure of what has been committed and what has not been committed. Catching an SQLException tells you that something is wrong, but it does not tell you what was or was not committed. Since you cannot count on the fact that nothing was committed, calling the method rollback is the only way to be sure.

Stored procedures

Stored procedures are supported by most DBMSs, but there is a fair amount of variation in their syntax and capabilities. **Example**:

```
String createProcedure =
"create procedure SHOW_SUPPLIERS " + "as " +
"select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME " +
"from SUPPLIERS, COFFEES " +
"where SUPPLIERS.SUP_ID = COFFEES.SUP_ID " +
"order by SUP_NAME";
Statement stmt = con.createStatement();
stmt.executeUpdate(createProcedure);
```

The procedure SHOW_SUPPLIERS will be compiled and stored in the database as a database object that can be called, similar to the way you would call a method.

Stored procedures(2)

JDBC allows you to call a database stored procedure from an application written in the Java programming language. The first step is to create a **CallableStatement** object. Than you should call proper execute method (depending on what is procedure created: SELECT, UPDATE and so on). **Example:**

```
CallableStatement cs =
        con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
// or executeUpdate() or execute()
```

Catching SQLException

```
} catch(SQLException ex) {
        System.out.println("\n--- SQLException caught ---\n");
        while (ex != null) {
        System.out.println("Message:" + ex.getMessage());
        // a string that describes the error
        System.out.println("SQLState: " + ex.getSQLState ());
        // a string identifying the error according to the X/Open
        // SQLState conventions
        System.out.println("ErrorCode: " + ex.getErrorCode ());
        // a number that is the driver vendor's error code number
        ex = ex.getNextException(); // there can be more than 1
        System.out.println(" ");
        }
}
```

SQLWarninig

•**SQLWarning** objects are a subclass of SQLException that deal with database access warnings.

•Warnings do not stop the execution of an application, as exceptions do; they simply alert the user that something did not happen as planned. For example, a warning might let you know that a privilege you attempted to revoke was not revoked. Or a warning might tell you that an error occurred during a requested disconnection.

•A warning can be reported on a Connection object, a Statement object (including PreparedStatement and CallableStatement objects), or a ResultSet object. Each of these classes has a **getWarnings** method, which you must invoke in order to see the first warning reported on the calling object.

SQLWarninig(2)

If getWarnings returns a warning, you can call the SQLWarning method **getNextWarning** on it to get any additional warnings.

Executing a statement automatically clears the warnings from a previous statement, so they do not build up. This means, however, that if you want to retrieve warnings reported on a statement, you must do so **before you execute another statement.**

Usage of warnings is analogous to usage of exceptions.

Cursor in scrollable ResultSet

One of the new features in the JDBC 2.0 API is the ability to move a result set's cursor backward as well as forward.

There are also methods that let you move the cursor to a particular row and check the position of the cursor.

Creating a scrollable ResultSet object example:

```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME,
PRICE FROM COFFEES");
```

Presented by Bartosz Sakowicz

DMCS TUL

createStatement arguments

First argument must be one of the following:

TYPE_FORWARD_ONLY creates a nonscrollable result set in which the cursor moves only forward. If you do not specify any constants for the type and updatability of a ResultSet object, you will automatically get one that is TYPE_FORWARD_ONLY

TYPE_SCROLL_INSENSITIVE - scrollable ResultSet, it does not reflect changes made while it is still open

TYPE_SCROLL_SENSITIVE - scrollable ResultSet, it reflect changes made while it is still open

Second argument must be: CONCUR_READ_ONLY (default) or CONCUR_UPDATABLE

previous() method

```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME,
PRICE FROM COFFEES");
srs.afterLast();
while (srs.previous()) {
        String name = srs.getString("COF_NAME");
        float price = srs.getFloat("PRICE");
        System.out.println(name + "     " + price);
 }
```

Other methods

The methods ***first , last , beforeFirst ,*** and ***afterLast*** move the cursor to the row indicated in their names.

The method ***absolute*** will move the cursor to the row number indicated in the argument passed to it. If the number is positive, the cursor moves the given number from the beginning, so calling absolute(1) puts the cursor on the first row. If the number is negative, the cursor moves the given number from the end, so calling absolute(-1) puts the cursor on the last row.

Other methods(2)

With the method ***relative*** , you can specify how many rows to move from the current row and also the direction in which to move. A positive number moves the cursor forward the given number of rows; a negative number moves the cursor backward the given number of rows.

The method ***getRow*** lets you check the number of the row where the cursor is positioned.

isFirst , isLast , isBeforeFirst , isAfterLast verify whether the cursor is at a particular position.

Making updates

• An update is the modification of a column value in the current row.
• To update ResultSet you need to create one that is updatable. In order to do this, you supply the ResultSet constant CONCUR_UPDATABLE to the createStatement method.
• Using the JDBC 1.0 API, the update would look like this:

stmt.executeUpdate("UPDATE COFFEES SET PRICE = 10.99" +
"WHERE COF_NAME = FRENCH_ROAST_DECAF");

• The following code fragment shows another way to accomplish the update using the JDBC 2.0 API:

uprs.last(); //position cursor
uprs.updateFloat("PRICE", 10.99); // update row

Making updates(2)

• Update operations in the JDBC 2.0 API affect column values in the row where the cursor is positioned.

• The ResultSet. updateXXX methods take two parameters: the column to update and the new value to put in that column. As with the ResultSet. getXXX methods, the parameter designating the column may be either the column name or the column number.

• There is a different updateXXX method for updating each datatype (updateString , updateBigDecimal , updateInt , and so on) just as there are different getXXX methods for retrieving different datatypes.

Making updates(3)

• To make the update take effect in the database and not just the result set, we must call the ResultSet method ***updateRow*** :

```
uprs.last();
uprs.updateFloat("PRICE", 10.99f);
uprs.updateRow();
```

• If you had moved the cursor to a different row before calling the method updateRow , the update would have been lost.

• If would like to change the update you can cancel the update by calling the method ***cancelRowUpdates*** .

• You have to invoke cancelRowUpdates before invoking the method updateRow ; once updateRow is called, calling the method cancelRowUpdates does nothing.

• Note that cancelRowUpdates cancels all of the updates in a row, so if there are many invocations of the updateXXX methods on the same row, you cannot cancel just one of them

Making updates(4)

Example:

uprs.last();
uprs.updateFloat("PRICE", 10.99);
uprs.cancelRowUpdates();
uprs.updateFloat("PRICE", 10.79);
uprs.updateRow();

Updates and related operations apply to the row where the cursor is positioned. Even if there are many calls to updateXXX methods, it takes only one call to the method updateRow to update the database with all of the changes made in the current row.

Inserting rows

When you have a ResultSet object with results from the table, you can build the new row and then **insert it into both the result set and the table in one step.**

You build a new row in what is called the **insert row,** a special row associated with every ResultSet object. This row is not part of the result set; it is a kind of separate buffer in which you compose a new row.

Inserting rows(2)

To insert row programatically you should perform following steps:

• Move the cursor to the insert row, which you do by invoking the method ***moveToInsertRow.***

• Set a value for each column in the row. You do this by calling the appropriate ***updateXXX*** method for each value.

• Call the method ***insertRow*** to insert the row you have just populated with values into the result set. This one method simultaneously inserts the row into both the ResultSet object and the database table from which the result set was selected.

Inserting rows(3)

Example:

```
uprs.moveToInsertRow();

uprs.updateString("COF_NAME", "Kona");

uprs.updateInt("SUP_ID", 150);

uprs.updateFloat("PRICE", 10.99);

uprs.updateInt("SALES", 0);

uprs.updateInt("TOTAL", 0);

uprs.insertRow();
```

After you have called the method insertRow , you can start building another row to be inserted, or you can move the cursor back to a result set row. You can use any of earlier mentioned methods or ***moveToCurrentRow.***

Deleting Rows

Move the cursor to the row you want to delete and then call the method **deleteRow**:

uprs.absolute(4);

uprs.deleteRow();

With some JDBC drivers, a deleted row is removed and is no longer visible in a result set. Some JDBC drivers use a blank row as a placeholder (a "hole") where the deleted row used to be. If there is a blank row in place of the deleted row, you can use the method *absolute* with the original row positions to move the cursor because the row numbers in the result set are not changed by the deletion.

In any case, you should remember that JDBC drivers handle deletions differently.

Making Batch Updates

A **batch update** is a set of multiple update statements that is submitted to the database for processing as a batch. Sending multiple update statements to the database together as a unit can, in some situations, be much more efficient than sending each update statement separately.

Multiple executeUpdate statements can be sent in the same transaction, but even though they are committed or rolled back as a unit, they are still processed individually.

With the JDBC 2.0 API, Statement, PreparedStatement, and CallableStatement objects have the ability to maintain a list of commands that can be submitted together as a batch. They are created with an associated list, which is initially empty. You can add SQL commands to this list with the method ***addBatch***, and you can empty the list with the method ***clearBatch***. You send all of the commands in the list to the database with the method ***executeBatch***.

Making Batch Updates(2)

Example:

```
con.setAutoCommit(false);

Statement stmt = con.createStatement();

stmt.addBatch("INSERT INTO COFFEES " +
"VALUES('Amaretto', 49, 9.99, 0, 0)");

stmt.addBatch("INSERT INTO COFFEES " +
"VALUES('Amaretto_decaf', 49, 10.99, 0, 0)");

int [] updateCounts = stmt.executeBatch();
```

Only commands that return an update count (commands such as INSERT INTO , UPDATE , DELETE , CREATE TABLE , DROP TABLE , ALTER TABLE , and so on) **can be executed as a batch** with the executeBatch method.

BatchUpdateException

If one of the commands cannot be executed for some other reason, the method executeBatch will throw a **BatchUpdateException** . In addition to the information that all exceptions have, this exception contains an array of the update counts for the commands that executed successfully before the exception was thrown. Because the update counts are in the same order as the commands that produced them, you can tell how many commands were successful and which commands they are.

BatchUpdateException is derived from SQLException.

BatchUpdateException(2)

Usage example:

```
try {
// make some updates
} catch(BatchUpdateException b) {
        System.err.println("SQLException: " + b.getMessage());
        System.err.println("SQLState: " + b.getSQLState());
        System.err.println("Message: " + b.getMessage());
        System.err.println("Vendor: " + b.getErrorCode());
        System.err.print("Update counts: ");
        int [] updateCounts = b.getUpdateCounts();
        for (int i = 0; i < updateCounts.length; i++) {
        System.err.print(updateCounts[i] + " "); } }
```

SQL3 datatypes

The datatypes commonly referred to as **SQL3 types** are the new datatypes being adopted in the next version of the ANSI/ISO SQL standard.

The new SQL3 datatypes give a relational database more flexibility in what can be used as a type for a table column.

SQL3 datatypes:

BLOB (Binary Large Object), which can store very large amounts of data as raw bytes.

CLOB (Character Large Object), which is capable of storing very large amounts of data in character format.

ARRAY makes it possible to use an array as a column value.

User-defined types (**UDTs**), structured types and distinct types, can now be stored as column values.

Presented by Bartosz Sakowicz

DMCS TUL

SQL3 datatypes(2)

SQL3 types ***getXXX*** methods (***setXXX*** and ***updateXXX*** by analogy):

BLOB getBlob

CLOB getClob

ARRAY getArray

Structured type getObject

ARRAY usage example

```
ResultSet rs = stmt.executeQuery( "SELECT SCORES FROM
STUDENTS   WHERE ID = 2238");

rs.next();

Array scores = rs.getArray("SCORES");
```

Connection pooling

Opening a connection to a database is a time-consuming process. For short queries, it can take much longer to open the connection than to perform the actual database retrieval. Consequently, it makes sense to reuse Connection objects in applications that connect repeatedly to the same database.

A connection pool class should be able to perform the following tasks:

1. Preallocate the connections.
2. Manage available connections.
3. Allocate new connections.
4. Wait for a connection to become available.
5. Close connections when required.

Presented by Bartosz Sakowicz

DMCS TUL

Prealocating the connections

Perform this task in the class constructor. Allocating more connections in advance speeds things up if there will be many concurrent requests later but causes an initial delay. As a result, a servlet that preallocates very many connections should build the connection pool from its init method, and you should be surethat the servlet is initialized prior to a “real” client request:

```
availableConnections = new Vector(initialConnections);

busyConnections = new Vector();

for(int i=0; i<initialConnections; i++) {

        availableConnections.addElement(makeNewConnection());

}
```

Managing available connections

If a connection is required and an idle connection is available, put it in the list of busy connections and then return it. The busy list is used to check limits on the total number of connections as well as when the pool is instructed to explicitly close all connections.

Connections can time out, so before returning the connection, confirm that it is still open. If not, discard the connection and repeat the process.

Discarding a connection opens up a slot that can be used by processes that needed a connection when the connection limit had been reached, so **use notifyAll** to tell all waiting threads to wake up and see if they can proceed (e.g., by allocating a new connection).

Managing available connections(2)

```
public synchronized Connection getConnection()throws
SQLException {
        if (!availableConnections.isEmpty()) {
                Connection existingConnection =
                (Connection)availableConnections.lastElement();
                int lastIndex = availableConnections.size() - 1;
                availableConnections.removeElementAt(lastIndex);
```

verte-->

Managing available connections(3)

```
        if (existingConnection.isClosed()) {
        notifyAll(); // Freed up a spot for anybody waiting.
        return(getConnection()); // Repeat process.
        } else {
        busyConnections.addElement(existingConnection);
        return(existingConnection);
        }
    }
}
```

Allocating new connections

If a connection is required, there is no idle connection available, and the connection limit has not been reached, then start a background thread to allocate a new connection. Then, wait for the first available connection, whether or not it is the newly allocated one.

```
if ((totalConnections() < maxConnections)
makeBackgroundConnection();

try {

wait(); // Give up lock and suspend self.

} catch(InterruptedException ie) {}

return(getConnection()); // Try again.
```

Waiting for a connection to become available

This situation occurs when there is no idle connection and you've reached the limit on the number of connections. The natural approach is to use the **wait** method, which gives up the thread synchronization lock and suspends the thread until **notify** or **notifyAll** is called. Since notifyAll could stem from several possible sources, threads that wake up still need to test to see if they can proceed. In this case, the simplest way to accomplish this task is to recursively repeat the process of trying to obtain a connection.

```
try {
wait();
} catch(InterruptedException ie) {}
return(getConnection());
```

Closing the connections

```
private void closeConnections(Vector connections) {
        try {
                for(int i=0; i<connections.size(); i++) {
                        Connection connection =
                        (Connection)connections.elementAt(i);
                        if (!connection.isClosed()) {
                                connection.close();
                        }
        }
        } catch(SQLException sqle) { }
}
```

ConnectionPoolServlet

```
public void init() {
// declare variables for database driver, username, ...
try {
connectionPool =
new ConnectionPool(driver, url, username, password,
initialConnections(),maxConnections() );
} catch(SQLException sqle) {
System.err.println("Error making pool: " + sqle);
getServletContext().log("Error making pool: " + sqle);
connectionPool = null;
}}
```

ConnectionPoolServlet(2)

```
    public void destroy() {
    connectionPool.closeAllConnections();
    }
    protected int initialConnections() {
            return(10);
    }
    protected int maxConnections() {
            return(50);
    }
}
```

Using the ServletContext to share connection pool

Group of servlets that all use the *books database* could share pools by having each servlet perform the following steps:

```
ServletContext context = getServletContext();
ConnectionPool bookPool =
(ConnectionPool)context.getAttribute("book-pool");
if (bookPool == null) {
        bookPool = new ConnectionPool(...);
        context.setAttribute("book-pool", bookPool);
}
```

Using the singleton class

A singleton class is a class for which only a single instance can be created, enforced through use of a private constructor.

The instance is retrieved through a static method that checks if there is already an object allocated, returning it if so and allocating and returning a new one if not.

Using the singleton class

Example:

```
public class BookPool extends ConnectionPool {
        private static BookPool pool = null;
        private BookPool(...) {
                super(...); // Call parent constructor        }
        public static synchronized BookPool getInstance() {
        if (pool == null) { pool = new BookPool(...); }
        return(pool);
        }
}
```