Introduction to JDBC

**1. Overview**

In this article, we're going to take a look at JDBC (Java Database Connectivity) which is an API for connecting and executing queries on a database.

JDBC can work with any database as long as proper drivers are provided.

**2. JDBC Drivers**

A JDBC driver is a JDBC API implementation used for connecting to a particular type of database. There are several types of JDBC drivers:

* Type 1 – contains a mapping to another data access API; an example of this is the JDBC-ODBC driver
* Type 2 – is an implementation that uses client-side libraries of the target database; also called a native-API driver
* Type 3 – uses middleware to convert JDBC calls into database-specific calls; also known as a network protocol driver
* Type 4 – connect directly to a database by converting JDBC calls into database-specific calls; known as database protocol drivers or thin drivers,

The most commonly used type is type 4, as it has the advantage of being **platform-independent**. Connecting directly to a database server provides better performance compared to other types. The downside of this type of driver is that it's database-specific – given each database has its own specific protocol.

**3. Connecting to a Database**

To connect to a database, we simply have to initialize the driver and open a database connection.

**3.1. Registering the Driver**

For our example, we will use a type 4 database protocol driver.

Since we're using a MySQL database, we need the *mysql-connector-java* dependency:

<**dependency**>

<**groupId**>mysql</**groupId**>

<**artifactId**>mysql-connector-java</**artifactId**>

<**version**>6.0.6</**version**>

</**dependency**>Copy

Next, let's register the driver using the *Class.forName()* method, which dynamically loads the driver class:

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

In older versions of JDBC, before obtaining a connection, we first had to initialize the JDBC driver by calling the *Class.forName* method. As of JDBC 4.0, **all drivers that are found in the classpath are automatically loaded**. Therefore, we won't need this *Class.forName*part in modern environments.

**3.2. Creating the Connection**

To open a connection, we can use the *getConnection()* method of *DriverManager* class. This method requires a connection URL *String*parameter:

**try** (**Connection** con = DriverManager

.getConnection("jdbc:mysql://localhost:3306/myDb", "user1", "pass")) {

// use con here

}Copy

**Since the *Connection*is an *AutoCloseable*resource, we should use it inside a *try-with-resources*block**.

The syntax of the connection URL depends on the type of database used. Let's take a look at a few examples:

jdbc:mysql://localhost:3306/myDb?user=user1&password=passCopy

jdbc:postgresql://localhost/myDbCopy

jdbc:hsqldb:mem:myDbCopy

To connect to the specified *myDb* database, we will have to create the database and a user, and add grant necessary access:

**CREATE** DATABASE myDb;

**CREATE** **USER** 'user1' IDENTIFIED **BY** 'pass';

**GRANT** **ALL** **on** myDb.\* **TO** 'user1';Copy

**4. Executing SQL Statements**

The send SQL instructions to the database, we can use instances of type *Statement*, *PreparedStatement,* or *CallableStatement,*which we can obtain using the *Connection* object.

**4.1. *Statement***

The *Statement* interface contains the essential functions for executing SQL commands.

First, let's create a *Statement* object:

**try** (**Statement** stmt = con.createStatement()) {

// use stmt here

}Copy

Again, we should work with *Statement*s inside a *try-with-resources*block for automatic resource management.

Anyway, executing SQL instructions can be done through the use of three methods:

* *executeQuery()* for SELECT instructions
* *executeUpdate()* for updating the data or the database structure
* *execute()* can be used for both cases above when the result is unknown

Let's use the *execute()* method to add a *students* table to our database:

**String** tableSql = "CREATE TABLE IF NOT EXISTS employees"

+ "(emp\_id int PRIMARY KEY AUTO\_INCREMENT, name varchar(30),"

+ "position varchar(30), salary double)";

stmt.execute(tableSql);Copy

**When using the *execute()* method to update the data, then the *stmt.getUpdateCount()* method returns the number of rows affected.**

If the result is 0 then either no rows were affected, or it was a database structure update command.

If the value is -1, then the command was a SELECT query; we can then obtain the result using *stmt.getResultSet()*.

Next, let's add a record to our table using the *executeUpdate()* method:

**String** insertSql = "INSERT INTO employees(name, position, salary)"

+ " VALUES('john', 'developer', 2000)";

stmt.executeUpdate(insertSql);Copy

The method returns the number of affected rows for a command that updates rows or 0 for a command that updates the database structure.

We can retrieve the records from the table using the *executeQuery()* method which returns an object of type *ResultSet*:

**String** selectSql = "SELECT \* FROM employees";

**try** (**ResultSet** resultSet = stmt.executeQuery(selectSql)) {

// use resultSet here

}Copy

We should make sure to close the *ResultSet*instances after use. Otherwise, we may keep the underlying cursor open for a much longer period than expected. To do that, it's recommended to use a *try-with-resources*block, as in our example above.

**4.2. *PreparedStatement***

*PreparedStatement* objects contain precompiled SQL sequences. They can have one or more parameters denoted by a question mark.

Let's create a *PreparedStatement* which updates records in the *employees* table based on given parameters:

**String** updatePositionSql = "UPDATE employees SET position=? WHERE emp\_id=?";

**try** (**PreparedStatement** pstmt = con.prepareStatement(updatePositionSql)) {

// use pstmt here

}Copy

To add parameters to the *PreparedStatement*, we can use simple setters – *setX()* – where X is the type of the parameter, and the method arguments are the order and value of the parameter:

pstmt.setString(1, "lead developer");

pstmt.setInt(2, 1);Copy

The statement is executed with one of the same three methods described before: *executeQuery(), executeUpdate(), execute()* without the SQL *String* parameter:

**int** rowsAffected = pstmt.executeUpdate();Copy

**4.3. *CallableStatement***

The *CallableStatement* interface allows calling stored procedures.

To create a *CallableStatement* object, we can use the *prepareCall()* method of *Connection*:

**String** preparedSql = "{call insertEmployee(?,?,?,?)}";

**try** (**CallableStatement** cstmt = con.prepareCall(preparedSql)) {

// use cstmt here

}Copy

Setting input parameter values for the stored procedure is done like in the *PreparedStatement* interface, using *setX()* methods:

cstmt.setString(2, "ana");

cstmt.setString(3, "tester");

cstmt.setDouble(4, 2000);Copy

If the stored procedure has output parameters, we need to add them using the *registerOutParameter()* method:

cstmt.registerOutParameter(1, Types.INTEGER);Copy

Then let's execute the statement and retrieve the returned value using a corresponding *getX()* method:

cstmt.execute();

**int** new\_id = cstmt.getInt(1);Copy

For example to work, we need to create the stored procedure in our MySql database:

delimiter //

**CREATE** **PROCEDURE** insertEmployee(**OUT** emp\_id **int**,

**IN** emp\_name **varchar**(30), **IN** position **varchar**(30), **IN** salary **double**)

**BEGIN**

**INSERT** **INTO** employees(name, position,salary) **VALUES** (emp\_name,position,salary);

**SET** emp\_id = LAST\_INSERT\_ID();

**END** //

delimiter ;Copy

The *insertEmployee* procedure above will insert a new record into the *employees* table using the given parameters and return the id of the new record in the *emp\_id* out parameter.

To be able to run a stored procedure from Java, the connection user needs to have access to the stored procedure's metadata. This can be achieved by granting rights to the user on all stored procedures in all databases:

**GRANT** **ALL** **ON** mysql.proc **TO** 'user1';Copy

Alternatively, we can open the connection with the property *noAccessToProcedureBodies* set to *true*:

con = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/myDb?noAccessToProcedureBodies=true",

"user1", "pass");Copy

This will inform the JDBC API that the user does not have the rights to read the procedure metadata so that it will create all parameters as INOUT *String* parameters.

**5. Parsing Query Results**

After executing a query, the result is represented by a *ResultSet* object, which has a structure similar to a table, with lines and columns.

**5.1. *ResultSet*Interface**

The *ResultSet* uses the *next()* method to move to the next line.

Let's first create an *Employee* class to store our retrieved records:

**public** **class** **Employee** {

**private** **int** id;

**private** String name;

**private** String position;

**private** **double** salary;

// standard constructor, getters, setters

}Copy

Next, let's traverse the *ResultSet* and create an *Employee* object for each record:

**String** selectSql = "SELECT \* FROM employees";

**try** (**ResultSet** resultSet = stmt.executeQuery(selectSql)) {

List<Employee> employees = **new** **ArrayList**<>();

**while** (resultSet.next()) {

**Employee** emp = **new** **Employee**();

emp.setId(resultSet.getInt("emp\_id"));

emp.setName(resultSet.getString("name"));

emp.setPosition(resultSet.getString("position"));

emp.setSalary(resultSet.getDouble("salary"));

employees.add(emp);

}

}Copy

Retrieving the value for each table cell can be done using methods of type *getX(*) where X represents the type of the cell data.

The *getX()* methods can be used with an *int* parameter representing the order of the cell, or a *String* parameter representing the name of the column. The latter option is preferable in case we change the order of the columns in the query.

**5.2. Updatable *ResultSet***

Implicitly, a *ResultSet* object can only be traversed forward and cannot be modified.

If we want to use the *ResultSet* to update data and traverse it in both directions, we need to create the *Statement* object with additional parameters:

stmt = con.createStatement(

ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_UPDATABLE

);Copy

To navigate this type of *ResultSet*, we can use one of the methods:

* *first(), last(), beforeFirst(), beforeLast()* – to move to the first or last line of a *ResultSet* or to the line before these
* *next(), previous()* – to navigate forward and backward in the *ResultSet*
* *getRow() –* to obtain the current row number
* *moveToInsertRow(), moveToCurrentRow()* – to move to a new empty row to insert and back to the current one if on a new row
* *absolute(int row) –* to move to the specified row
* *relative(int nrRows)* – to move the cursor the given number of rows

Updating the *ResultSet* can be done using methods with the format *updateX()* where X is the type of cell data. These methods only update the *ResultSet* object and not the database tables.

To persist the *ResultSet* changes to the database, we must further use one of the methods:

* *updateRow()* – to persist the changes to the current row to the database
* *insertRow(), deleteRow()* – to add a new row or delete the current one from the database
* *refreshRow()* – to refresh the *ResultSet* with any changes in the database
* *cancelRowUpdates()* – to cancel changes made to the current row

Let's take a look at an example of using some of these methods by updating data in the *employee's* table:

**try** (**Statement** updatableStmt = con.createStatement(

ResultSet.TYPE\_SCROLL\_INSENSITIVE, ResultSet.CONCUR\_UPDATABLE)) {

**try** (**ResultSet** updatableResultSet = updatableStmt.executeQuery(selectSql)) {

updatableResultSet.moveToInsertRow();

updatableResultSet.updateString("name", "mark");

updatableResultSet.updateString("position", "analyst");

updatableResultSet.updateDouble("salary", 2000);

updatableResultSet.insertRow();

}

}Copy

**6. Parsing Metadata**

The JDBC API allows looking up information about the database, called metadata.

**6.1. *DatabaseMetadata***

The *DatabaseMetadata* interface can be used to obtain general information about the database such as the tables, stored procedures, or SQL dialect.

Let's have a quick look at how we can retrieve information on the database tables:

**DatabaseMetaData** dbmd = con.getMetaData();

**ResultSet** tablesResultSet = dbmd.getTables(null, null, "%", null);

**while** (tablesResultSet.next()) {

LOG.info(tablesResultSet.getString("TABLE\_NAME"));

}Copy

**6.2. *ResultSetMetadata***

This interface can be used to find information about a certain *ResultSet*, such as the number and name of its columns:

**ResultSetMetaData** rsmd = rs.getMetaData();

**int** nrColumns = rsmd.getColumnCount();

IntStream.range(1, nrColumns).forEach(i -> {

**try** {

LOG.info(rsmd.getColumnName(i));

} **catch** (SQLException e) {

e.printStackTrace();

}

});Copy

**7. Handling Transactions**

By default, each SQL statement is committed right after it is completed. However, it's also possible to **control transactions programmatically**.

This may be necessary in cases when we want to preserve data consistency, for example when we only want to commit a transaction if a previous one has completed successfully.

First, we need to set the *autoCommit* property of *Connection* to *false*, then use the*commit()* and *rollback()* methods to control the transaction.

Let's add a second update statement for the *salary* column after the employee *position* column update and wrap them both in a transaction. This way, the salary will be updated only if the position was successfully updated:

**String** updatePositionSql = "UPDATE employees SET position=? WHERE emp\_id=?";

**PreparedStatement** pstmt = con.prepareStatement(updatePositionSql);

pstmt.setString(1, "lead developer");

pstmt.setInt(2, 1);

**String** updateSalarySql = "UPDATE employees SET salary=? WHERE emp\_id=?";

**PreparedStatement** pstmt2 = con.prepareStatement(updateSalarySql);

pstmt.setDouble(1, 3000);

pstmt.setInt(2, 1);

**boolean** autoCommit = con.getAutoCommit();

**try** {

con.setAutoCommit(false);

pstmt.executeUpdate();

pstmt2.executeUpdate();

con.commit();

} **catch** (SQLException exc) {

con.rollback();

} **finally** {

con.setAutoCommit(autoCommit);

}Copy

For the sake of brevity, we omit the *try-with-resources*blocks here.

**8. Closing the Resources**

When we're no longer using it,**we need to close the connection to release database resources**.

We can do this using the *close()* API:

con.close();Copy

However, if we're using the resource in a *try-with-resources* block, we don't need to call the *close()* method explicitly, as the *try-with-resources*block does that for us automatically.

**The same is true for the *Statement*s, *PreparedStatement*s, *CallableStatement*s, and *ResultSet*s.**

**9. Conclusion**

In this tutorial, we had a look at the basics of working with the JDBC API.