Connecting to a database with JDBC

What is JDBC

Java DataBase Connectivity (**JDBC**) is a Java API that was designed to access data stored in relational databases.

By using JDBC, you can establish a connection to a database, execute SQL statements, and handle the results received from the database in answer to your query.

If JDBC API provides interfaces for connecting Java application to relational databases, **JDBC drivers** implement those interfaces for the concrete database management system (DBMS).

JDBC driver emits the connection to the database and implements the protocol for transferring queries and their results between your application and the database.

Such an approach allows us to use JDBC API in the same way for different DBMS by using different JDBC drivers.

Establishing a connection

To establish a connection between your application and the database, you need an object instantiated by a class that implements the DataSource interface.

As we've mentioned, JDBC drivers provide an implementation for JDBC API interfaces, so the DataSource interface is implemented by a driver vendor and represents a particular DBMS.

Before connection is established, we need to specify a database URL.

The **database URL** is a string that contains information about the database, including the name of the database, path to the database, configuration properties, etc.

Each DBMS has its own rules and syntax for a connection URL. Most DBMS require a database name, hostname, port number, and user credentials specified in the URL.

Finally, we need to call the DataSource.getConnection method. It returns a Connection object that represents a connection with the database. Once the connection is established you can interact with the database.

Let's dive into the practice and connect our application to the **SQLite** DBMS. At first, download and install SQLite. It is a very compact DBMS, which is not used

for production purposes very often, but it is well suited for learning how things work.

The next step is to add SQLite JDBC Driver to the application. For that, we can use Gradle and add the following dependency to the **build.gradle** file:

```
dependencies {
    implementation 'org.xerial:sqlite-jdbc:3.30.1'
}
```

Now, let's create a class where we can test our JDBC connection. To start with, we create a string that contains the SQLite database URL:

```
public class CoolJDBC {
    public static void main(String[] args) {
        String url = "jdbc:sqlite:path-to-database";
    }
}
```

path-to-database can represent either an absolute or a relative path to the database.

For example, if you install SQLite to the C:\sqlite folder and execute sqlite3 fruits.db command your URL can be "jdbc:sqlite:C:/sqlite/fruits.db". At the same time, if a database file is located inside the project folder, your URL can be "jdbc:sqlite:fruits.db".

Now we are ready to establish a connection between the Java application and SQLite. For that, we will create an SQLiteDataSource object that is DataSource interface implementation provided by the SQLite JDBC driver.

Then, we need to set a data source connection URL by calling SQLiteDataSource.setUrl method. Finally, we will declare the Connection object and assign to it the return value of the DataSource.getConnection method inside the try-with-resources statement.

The try-with-resources statement is a try statement that declares a resource — an object that must be closed after the program finished its use. We should close a JDBC connection to release any other database resources the connection may be holding on.

You can also close the connection explicitly by calling the Connection.close method.

```
public class CoolJDBC {
    public static void main(String[] args) {
```

```
        String url = "jdbc:sqlite:path-to-database";

        SQLiteDataSource dataSource = new SQLiteDataSource();
        dataSource.setUrl(url);

        try (Connection con = dataSource.getConnection()) {
            if (con.isValid(5)) {
                System.out.println("Connection is valid.");
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

As you can notice, inside the try-catch block we call the isValid method of the Connection object with "5" as a parameter. isValid method checks if the connection has not been closed and is still valid while waiting for the specified number of seconds to validate the connection. If the URL is correct, our application console output will contain a "Connection is valid." statement.

In the example above, we've set a data source URL and then called the DataSource.getConnection method without any parameters. However, it can take the username and password as parameters. These variables are used for authentication purposes, and it is very likely that in the real-life scenario you should specify them.

JDBC Statements

What is a Statement

To perform actions on a database, we need to use SQL statements. An interface java.sql.Statement represents such statements in the JDBC API.

At first, we need to establish a connection with the database in order to execute statements from our application.

Then we should create a Statement object using a Connection object.

More precisely, we need to call the createStatement() method of the Connection that creates a Statement.

Statement execution

Once the Statement object is created, we can execute SQL statements by calling its execution methods.

The most generic method is execute(String sql). It performs a given SQL statement and returns true if there is a return data, otherwise, the method returns false.

For example, for the SELECT statement it returns true and for the INSERT statement false.

However, the Statement interface has other more specific execution methods. One of them is executeUpdate(String sql). Unlike
the execute the executeUpdate method returns the number of rows affected by the SQL statement.

Use executeUpdate method for INSERT, DELETE and UPDATE statements or for statements that return nothing, such as CREATE or DROP.

Let's create an SQLite database westeros.db and then create a table of the Greater Houses of the Seven Kingdoms using the executeUpdate method.

```
public class Westeros {
    public static void main(String[] args) {
        String url = "jdbc:sqlite:C:/sqlite/westeros.db";

        SQLiteDataSource dataSource = new SQLiteDataSource();
        dataSource.setUrl(url);

        try (Connection con = dataSource.getConnection()) {
            // Statement creation
            try (Statement statement = con.createStatement()) {
                // Statement execution
                statement.executeUpdate("CREATE TABLE IF NOT EXISTS HOUSES("
+
                        "id INTEGER PRIMARY KEY," +
                        "name TEXT NOT NULL," +
                        "words TEXT NOT NULL)");
            } catch (SQLException e) {
                e.printStackTrace();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Since JDBC spec requires Statement to be closed when no longer reachable, we have used the try-with-resources statement for creating Statement objects.

Once we execute the program above, we will create a table HOUSES that stores an id of the house, its name, and words. Now, let's add several houses to the table. For that, we will use executeUpdate again and add the following code:

```
int i = statement.executeUpdate("INSERT INTO HOUSES VALUES " +
        "(1, 'Targaryen of King''s Landing', 'Fire and Blood')," +
        "(2, 'Stark of Winterfell', 'Summer is Coming')," +
        "(3, 'Lannister of Casterly Rock', 'Hear Me Roar!')");
```

As you can guess, the value of i will be equal to 3, since we have inserted 3 houses to the database.

Note, executeUpdate method requires to wrap *text* values into a single quote character ('). If the value contains this character, you have to replace it with double single quotes ('') to be parsed correctly.

Since the real words of the Stark of Winterfell house are "Winter is coming", we have to update it. For that, we will execute the SQL UPDATE statement using executeUpdate method:

```
int u = statement.executeUpdate("UPDATE HOUSES " +
        "SET words = 'Winter is coming' " +
        "WHERE id = 2");
```

Since we've updated only one record the value of u will be equal to 1.
Now, when you've created several records with Great Houses of Westeros, we would definitely need to retrieve it back from the database. For that, we need to execute the SQL SELECT statement. The appropriate Statement method for the execution of SELECT statements is executeQuery(String sql). This method is similar to the already discussed methods, however, it returns a ResultSet object. The ResultSet object represents a table that contains records from the database result set.

Processing ResultSet

For processing ResultSet, we can use its next() method. Each call of the next() moves a pointer to the record forward one position, starting from the first record. For retrieving column values we will use ResultSet getter methods of the appropriate type. For example, for the column with a TEXT type and INTEGER type, we can use getString and getInt methods respectively. ResultSet getters can accept two types of arguments: column index (starting from 1) and column label.

It is possible to use getString getter for retrieving columns values with any type. However, in that case, the value will be converted to the java.lang.String type.

Let's look at the example, where we retrieve and print all records from the HOUSES table one by one. For that we need to add the following code:

```
try (ResultSet greatHouses = statement.executeQuery("SELECT * FROM HOUSES")) {
    while (greatHouses.next()) {
        // Retrieve column values
        int id = greatHouses.getInt("id");
        String name = greatHouses.getString("name");
        String words = greatHouses.getString("words");

        System.out.printf("House %d%n", id);
        System.out.printf("\tName: %s%n", name);
        System.out.printf("\tWords: %s%n", words);
    }
}
```

Since JDBC spec required ResultSets be closed when no longer reachable, we have used the try-with-resources statement for creating ResultSet objects.

Note that we have called the next() method inside the while loop. Since the next() method returns a boolean value (true if there are more records in the ResultSet), calling this method inside a while loop is a convenient way to process the ResultSet.