Theory: Connecting to a database with JDBC

17 minutes 0 / 4 problems solved

§1. 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.

§2. 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 {
    compile group:'org.xerial', name:'sqlite-jdbc', version:'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

Current topic:

Connecting to a database with JDBC

Topic depends on:

- Introduction to databases
- Exception handling Stage 7
- Dependency management

Topic is required for:

- JDBC Statements

Table of contents:

↑ Connecting to a database with JDBC

§1. What is JDBC

§2. Establishing a connection

§3. Conclusion

Feedback & Comments

`"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 it to 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.

§3. Conclusion

In this topic, you have learned what JDBC and JDBC drivers are, why we need to specify the connection URL and how to establish a connection between your application and the database management system by using `Connection` and `DataSource` interfaces.

Report a typo

72 users liked this theory. **7** didn't like it. **What about you?**

Start practicing

Comments (9) Hints (2) Useful links (1) Show discussion