

Theory: JDBC Transactions

🕒 21 minutes 0 / 5 problems solved

Skip this topic

Start practicing

228 users solved this topic. Latest completion was about 2 hours ago.

§1. What is a Transaction?

So far, we have learned how to execute SQL statements using JDBC API. As you might remember, once you call an `execute`, `executeQuery`, or `executeUpdate` method of the `Statement` or `PreparedStatement` object, the SQL statement query will be executed. But it is not always what we need. There are many situations where the business logic of your application requires not to execute a query unless another one completes.

For example, let's say you order a car online. For that, an ordering application should perform the following steps: withdraw money from your bank account, put money into the seller account, and place a purchase order. Imagine that each step executes independently, and system error occurs after the first two steps, which means that money will be withdrawn from your account and send to the seller, but you will never receive your car.

To avoid such situations, we can treat all three steps as a single unit that can succeed or fail completely. In that case, if a system error or any other failure occurs at any stage of the ordering, your money won't be transferred to the seller, and the seller doesn't place a purchase order. Such a unit of several SQL statements is called a *transaction*.

§2. Commit and Auto-commit mode

To make all changes made by a transaction permanent in the database, we need to commit the transaction. However, when you create the JDBC connection, it is in *auto-commit mode*, which means that each statement is treated as a separate transaction. That is why database data is changed immediately after calling a statement execution method. As we discussed earlier, it is not always the behavior that we desire.

To disable auto-commit mode, we need to explicitly disable it for the current connection by calling the `setAutoCommit(false)` method of the `Connection` object. Once the auto-commit mode is disabled, calling execution methods of statement objects will not affect the database until you explicitly *commit* a transaction by calling the `commit` method of the `Connection` object. Let's look at the example and create an SQLite database called `store.db` that holds `order` and `invoice` tables:

```
1 CREATE TABLE "invoice"
2 (
3     id            INTEGER PRIMARY KEY,
4     shipping_address TEXT    NOT NULL,
5     total_cost    INTEGER NOT NULL
6 );
7
8 CREATE TABLE "order"
9 (
10     id            INTEGER PRIMARY KEY,
11     invoice_id    INTEGER NOT NULL,
12     product_name TEXT    NOT NULL,
13     FOREIGN KEY (invoice_id) REFERENCES invoice (id)
14 );
```

Now we can create a `Connection` object called `con` and execute `con.setAutoCommit(false)` method to disable auto-committing so that we will create both an invoice and an order in the case of successful execution or none of them in the case of failure:

Current topic:

[JDBC Transactions](#) ...

Topic depends on:

✗ [JDBC Prepared Statements](#) ...

Table of contents:

[↑ JDBC Transactions](#)

[§1. What is a Transaction?](#)

[§2. Commit and Auto-commit mode](#)

[§3. Rollback](#)

[§4. Savepoint](#)

[§5. Conclusion](#)

[Feedback & Comments](#)

```

1 public class StoreDB {
2
3     public static void main(String[] args) {
4         String url = "jdbc:sqlite:path-to-store.db";
5
6         SQLiteDataSource dataSource = new SQLiteDataSource();
7         dataSource.setUrl(url);
8
9         String insertInvoiceSQL = "INSERT INTO \"invoice\" " +
10             "(id, shipping_address, total_cost) VALUES (?, ?, ?)";
11
12         String insertOrderSQL = "INSERT INTO \"order\" " +
13             "(id, invoice_id, product_name) VALUES (?, ?, ?)";
14
15         try (Connection con = dataSource.getConnection()) {
16
17             // Disable auto-commit mode
18
19             con.setAutoCommit(false);
20
21             try (PreparedStatement insertInvoice = con.prepareStatement(insertInvoiceSQL);
22                 PreparedStatement insertOrder = con.prepareStatement(insertOrderSQL)) {
23
24                 // Insert an invoice
25
26                 int invoiceId = 1;
27
28                 insertInvoice.setInt(1, invoiceId);
29
30                 insertInvoice.setString(2, "Dearborn, Michigan");
31
32                 insertInvoice.setInt(3, 100500);
33
34                 insertInvoice.executeUpdate();
35
36                 // Insert an order
37
38                 int orderId = 1;
39
40                 insertOrder.setInt(1, orderId);
41
42                 insertOrder.setInt(2, invoiceId);
43
44                 insertOrder.setString(3, "Ford Model A");
45
46                 insertOrder.executeUpdate();
47
48                 con.commit();
49
50             }
51
52         } catch (SQLException e) {
53
54             e.printStackTrace();
55
56         }
57     }
58 }

```

```
4  
2     }  
4  
3     }
```

Whenever the commit is made, all transaction statements are executed and stay permanently in the database. It means that after calling the `con.commit` method, an invoice and the corresponding order are created.

If the `setAutoCommit(true)` method is called during a transaction, the transaction will be committed.

§3. Rollback

Imagine that an error happens during statement execution, and `SQLException` is thrown. In the code above, we catch this exception and print an error stack trace. But we don't explicitly make sure that the transaction ends. As a result, we may commit invalid changes later if we continue using the same connection. To prevent this, we can rollback all transaction changes to undo all the changes made in the current transaction.

In JDBC, you can rollback a transaction by invoking the `Connection` object's `rollback` method. It is recommended to call the `rollback` method in the catch block if your code throws an `SQLException` during a transaction. Let's rewrite our catch statement to rollback the transaction:

```
1     catch (SQLException e) {  
2         if (con != null) {  
3             try {  
4                 System.err.print("Transaction is being rolled back");  
5                 con.rollback();  
6             } catch (SQLException excep) {  
7                 excep.printStackTrace();  
8             }  
9         }  
1        }  
0    }
```

Not explicitly ending the transaction may also lead to the transaction living longer than it is necessary.

§4. Savepoint

As we have mentioned earlier, there are two ways to end the transaction: using the `commit` method that saves all transaction changes or using the `rollback` method that discards them. In most cases, this is exactly what we need. However, sometimes we have to save some changes and discard others. For that, we have to create a *savepoint* in the code. All changes introduced up to the savepoint will be saved and can be committed later.

To create a savepoint in JDBC, we can call the `setSavepoint` method of the `Connection` object that returns a `Savepoint` object. We can roll back the transaction to the savepoint so that all changes made before the savepoint will be preserved, and all changes after the savepoint will be discarded. To rollback the transaction to the specified savepoint, we can use the `rollback(Savepoint savepoint)` method that accepts a `Savepoint` object. To make it more concrete, let's consider the following example:

```

1 String insertInvoiceSQL = "INSERT INTO \"invoice\" " +
2   "(id, shipping_address, total_cost) VALUES (?, ?, ?)";
3
4 String selectAddressSQL = "SELECT shipping_address " +
5   "FROM \"invoice\" WHERE id = ?";
6
7 try (Connection con = dataSource.getConnection()) {
8
9   // Disable auto-commit mode
10
11   con.setAutoCommit(false);
12
13
14   try (PreparedStatement insertInvoice = con.prepareStatement(insertInvoiceSQL))
15   {
16
17
18       // Create a savepoint
19
20       Savepoint savepoint = con.setSavepoint();
21
22
23       // Insert an invoice
24
25       int invoiceId = 1;
26
27       insertInvoice.setInt(1, invoiceId);
28
29       insertInvoice.setString(2, "Dearborn, Michigan");
30
31       insertInvoice.setInt(3, 100500);
32
33       insertInvoice.executeUpdate();
34
35
36       PreparedStatement selectAddress = con.prepareStatement(selectAddressSQL);
37
38       selectAddress.setInt(1, invoiceId);
39
40       ResultSet resultSet = selectAddress.executeQuery();
41
42
43       if (resultSet.getString(1).equals("Dearborn, Michigan")) {
44
45           con.rollback(savepoint);
46
47       }
48
49       con.commit();
50
51   }
52
53   } catch (SQLException e) {
54
55       e.printStackTrace();
56
57   }

```

In the code above, we create a `Savepoint` object called `savepoint` and rollback the transaction changes to the savepoint if the invoice with `id` equals 1 has shipping address equals `"Dearborn, Michigan"`.

It is possible to release a savepoint, or in other words, to remove it from the current transaction. One way to release a savepoint is by using the `Connection.releaseSavepoint` method that takes a `Savepoint` object as a parameter. Any savepoint that has been created in a transaction is automatically released and becomes invalid when the transaction is committed or when the entire transaction is rolled back.

Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints created after the savepoint that was rolled back. If a savepoint has been released, attempting to reference it in a rollback operation throws an `SQLException`.

§5. Conclusion

In this topic, we discussed what a transaction is and why we need it. Since JDBC enables auto-commit mode by default, we discussed why it could be detrimental from the business logic perspective. To disable the auto-commit mode, we can use the `Connection.setAutoCommit` method. If we disable the auto-commit mode, we have to explicitly end a transaction by committing all transaction changes using the `Connection.commit` method or by rolling back the transaction using the `Connection.rollback` method. We also examined transaction savepoints as an additional mechanism to control your application.

 Report a typo

27 users liked this theory. 2 didn't like it. What about you?



Start practicing

This content was created 2 months ago and updated 10 days ago. [Share your feedback below in comments to help us improve it!](#)

[Comments \(3\)](#)

[Hints \(0\)](#)

[Useful links \(0\)](#)

[Show discussion](#)