# Возможности JDBC — второй этап

В предыдущей статье мы только познакомились с JDBC и написали простое приложение, которое позволило нам соединиться с СУБД и получить данные с помощью SQL-запроса. Хоть программа и не очень сложная, но на мой взгляд, мы сделали весьма важный шаг — мы смогли соединиться с базой данных и сделать пусть и простой, но запрос. Все, что мы рассмотрим дальше — это уже более удобные и более профессиональные способы использования JDBC.

## Запросы на получение данных и запросы на обновление

SQL-запросы можно условно разделить на две группы:

1. Получение данных — к ним относится оператор SELECT
2. Изменение данных — к ним относятся операторы INSERT, UPDATE и DELETE

Для первой группы используется уже знакомый нам метод интерфейса **Statement** — **executeQuery()**. В принципе для начала этого метода вполне достаточно. Он покрывает очень большой процент запросов, которые разрабатываются для реальных систем. Позже мы познакомимся с дополнительными возможностями, но на данных момент советую запомнить — если надо получить данные из таблицы, то **executeQuery** в подавляющем большинстве случаев будет самым правильным выбором.
Для второй группа запросов (опять же в большинстве случаев) может использоваться другой метод

интерфейса **Statement** — **executeUpdate()**. Есл посмотреть документацию, то в отличии от **executeQuery()** (который возвращает **ResultSet**) этот метод возвращает целое число, которое говорит сколько строк в таблице было изменено при исполнении вашего запроса.
Например, вы можете оператором DELETE FROM JC_CONTACT удалить ВСЕ строки (посему будьте очень аккуратны). В этом случае метод **executeUpdate()** вернет количество удаленных строк. В некоторых ситуациях знание о количестве измененных строк бывает удобным для построения алгоритмов работы с данными.
В принципе с этим вопросов можно закончить — главное мы уже увидели. Для выборки данных — **executeQuery()**. Для изменения данных — **executeUpdate()**.

## Разновидности Statement

Самый простой интерфейс Statement мы уже видели. И хотя он вполне пригоден для работы, для сложных запросов он подходит не так хорошо. В некоторых источниках высказывается мнение, что использовать Statement вообще не надо — вместо него подходят более сложные и более функционально насыщенные интерфейсы.

1. PreparedStatement
2. CallableStatement

Возникает вполне резонный вопрос — а зачем эти интерфейсы нужны ? Давайте разбираться. Для начала мы рассмотрим интерфейс **PreparedStatement** и другие возможности JDBC. К интерфейсу **CallableStatement** обратимся позже — его использование во-первых не так часто встречается, и во-вторых — после всего рассмотренного, про него разговор уже можно делать достаточно коротким.

## PreparedStatement

Если честно перевести название, то можно получить что-то вроде «подготовленный оператор». Самым важным здесь является слово «подготовленный». В чем же заключается «подготовленность» ?
Прежде чем мы рассмотрим этот вопрос, предлагаю увидеть достаточно важный с точки зрения удобства момент, который возникает очень часто. Итак, в каком-либо приложении нам надо вставить данные о контакте в таблицу JC_CONTACT. Для этого нам надо подготовить запрос наподобие такого:

```
1 INSERT INTO JC_CONTACT (FIRST_NAME, LAST_NAME, PHONE, EMAIL) VALUES ('Peter','Belgy','+79112345678','peter@pisem.net');
```

На первый взгляд кажется все не так уж сложно и страшно. Надо написать код, который будет «собирать» нужную нам строку из параметров — имя, фамилия, адрес и телефон. Надо только не забыть, что все строковые данные надо «окружить» символом ординарная кавычка. Если мы это делаем в

отдельной функции, то получается что-то вроде такого

```
public String buildInsert(String firstName,, String lastName, String phone, String email) {
    String sql = "INSERT INTO JC_CONTACT (FIRST_NAME, LAST_NAME, PHONE, EMAIL) VALUES ('" + firstName + "','" + lastNam
    return sql;
}
```

Мы передаем в функцию в виде параметров имя, фамилию, телефон и адрес и из них составляем строку SQL-запроса. Кавычки немного портят картину, но пока не страшно.
Ок, а что делать с числами ? Их не надо “окружать” кавычками. Опаньки, в одном случае надо кавычки, в другом — не надо. Ситуация усложняется.
Теперь добавим еще одну проблему — а если внутри строки есть ординарная кавычка (и даже не одна) ? Надо предварительно искать такие кавычки и обрабатывать их. Мда-а-а. Как-то неуютно начинаем себя ощущать.
Если теперь прибавить обработку дат, то задача становится совсем «скучной» — надо делать огромное количество работы. С датами вообще неприятно — разные SQL-сервера принимают для дат разные форматы.
Итак, что мы видим ? Если нам надо использовать параметры внутри запроса, то в «ручном» режиме построение запроса становится очень неприятным делом. Причем не просто непрятным — я бы даже сказал «занудным». Надо учитывать огромное количество случаев и это ужасно скучная работа. В основном именно для таких случаев и был предложен интерфейс **PreparedStatement**.
Этот запрос позволяет вам сделать две вещи:

1. Заранее подготовить запрос с указанием мест, где будут подставляться параметры
2. Установить параметры определенного типа и выполнить после этого запрос с уже установленными параметрами

Конструкция для **PreparedStatement** для нашего варианта установки параметров будет выглядеть вот так:

```
// Переменные для примера
String firstName = "Dmitry";
String lastName = "Chekhov";
String phone = "+79871112233";
String email = "dmitry@pisem.net";

// Запрос с указанием мест для параметров в виде знака "?"
String sql = "INSERT INTO JC_CONTACT (FIRST_NAME, LAST_NAME, PHONE, EMAIL) VALUES (?, ?, ?,?)";

// Создание запроса. Переменная con - это объект типа Connection
PreparedStatement stmt = con.prepareStatement(sql);

// Установка параметров
stmt.setString(1, firstName);
stmt.setString(2, lastName);
```

```
stmt.setString(3, phone);
stmt.setString(4, email);

// Выполнение запроса
stmt.executeUpdate();
```

Как видите, все достаточно несложно.
Во-первых, при написании SQL-запроса на места, куда надо будет подставлять параметры, записываются знаки вопроса — “?”.
Во-вторых — запрос создается через вызов **con.prepareStatement()**
В-третьих — установка параметров идет через указание номера и значения. Обратите внимание, что номер параметров начинаются с 1, а не с 0, как вы возможно уже привыкли при работе с массивами и коллекциями. Интерфейс **PreparedStatement** содержит методы для установки строк — **setString**, для установки чисел — **setInt**, **setLong**, **setDouble**, для установки дат — **setDate**. И более сложных типов — это можно увидеть в документации.
В-четвертых — вызов **stmt.executeUpdate()** выполняется уже без указания строки запроса.
Крайне настоятельно рекомендую “подружиться” с **PreparedStatement** — это очень эффективный инструмент.

## Сгенерированные поля

Теперь я хочу обратить ваше внимание на еще один интересный момент. При вставке контакта через оператор **INSERT** мы не указываем поле **CONTACT_ID** (посмотрите внимательно на скрипт в конце раздела Установка СУБД PostgreSQL). База данных генерирует это значение автоматически, используя свои возможности.
Для PostgreSQL это достигается путем использования специального объекта — SEQUENCE (последовательность). Вызов **nextval** у этого объекта каждый раз генерирует следующее число. Это облегчает нам жизнь — мы не должны придумывать как же достичь уникальности ИД контакта — просто используем этот объект. Если в утилите pgAdmin выделить таблицу **JC_CONTACT**, то можно увидеть точный SQL-скрипт для ее генерации.

```
CREATE TABLE public.jc_contact
(
  contact_id integer NOT NULL DEFAULT nextval('jc_contact_contact_id_seq'::regclass),
  first_name character varying(50) NOT NULL,
  last_name character varying(50) NOT NULL,
  phone character varying(50) NOT NULL,
  email character varying(50) NOT NULL,
  CONSTRAINT jc_contact_pkey PRIMARY KEY (contact_id)
)
```

Обратим внимание на объявление поля **contact_id**.

```
contact_id integer NOT NULL DEFAULT nextval('jc_contact_contact_id_seq'::regclass),
```

Слово DEFAULT указывает, какое значение будет вставляться в это поле, если для него не будет указано значение. Это следующее число в последовательности **jc_contact_contact_id_seq**. Т.е. когда мы выполняем вставку без указание поля **CONTACT_ID**, то оно генерируется само. Смотрим еще раз внимательно на наш оператор вставки

```
INSERT INTO JC_CONTACT (FIRST_NAME, LAST_NAME, PHONE, EMAIL) VALUES ('Helga','Forte','+79118765432','helga@pisem.net');
```

Как видим, поля CONTACT_ID нет. Но когда мы выбираем данные — оно имеет значение. Мало того — попытка вставки значение NULL в то поле просто невозможно. И дублирование значений тоже невозможно. Все эти ограничение отслеживаются самой СУБД. В общем все достаточно удобно и хорошо. Генерится автоматически, проверяется на уникальность тоже автоматически. Чего еще можно желать.
Код на Java для вставки записи и последующего чтения теперь может быть такой — здесь я уже использовал интерфейс **PreparedStatement**:

```
package edu.javacourse.database;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class InsertDb {

    public static void main(String[] args) {
        InsertDb m = new InsertDb();
        try {
            Class.forName("org.postgresql.Driver");
            String url = "jdbc:postgresql://localhost:5432/contactdb";
            String login = "postgres";
            String password = "postgres";
            Connection con = DriverManager.getConnection(url, login, password);
            try {
                // Процедура вставки
                m.insert(con, "FirstName", "LastName", "phone", "email");
                // Процедура выборки
                m.select(con);
            } finally {
                con.close();
            }
```

```
28          } catch (Exception e) {
29              e.printStackTrace();
30          }
31      }
32
33      private void insert(Connection con, String firstName, String lastName, String phone, String email) throws SQLExcept
34          PreparedStatement stmt = con.prepareStatement("INSERT INTO JC_CONTACT (FIRST_NAME, LAST_NAME, PHONE, EMAIL) VAI
35          stmt.setString(1, firstName);
36          stmt.setString(2, lastName);
37          stmt.setString(3, phone);
38          stmt.setString(4, email);
39          stmt.executeUpdate();
40          stmt.close();
41      }
42
43      private void select(Connection con) throws SQLException {
44          Statement stmt = con.createStatement();
45          ResultSet rs = stmt.executeQuery("SELECT * FROM JC_CONTACT");
46          while (rs.next()) {
47              String str = rs.getString("contact_id") + ":" + rs.getString(2);
48              System.out.println("Contact:" + str);
49          }
50          rs.close();
51          stmt.close();
52      }
53  }
```

На первый взгляд может показаться, что все замечательно — мы вставили запись и можем даже ее увидеть в следующем операторе SELECT. Там же мы найдем значение поля **CONTACT_ID**. Но в данном случае мы просто знаем, что только мы вставляли запись и мы можем определить, какая запись вставилась и какое значение поля **CONTACT_ID** сгенерировалось. Но если операцию вставки делают одновременно десятки, сотни или даже тысячи потоков, то какая запись наша ? Знание ИД записи достаточно важно — например, нам для этого контакта надо будет добавить данные в какую-то другую таблицу ? Или надо послать эту информацию по почте системному администратору ? Да мало ли зачем еще. В общем, иметь возможность узнать, под каким ИД вставилась запись — важно и очень удобно. Можно сформулировать эту проблему даже шире — узнать значения полей, которые вставлялись автоматически.
При всей очевидности необходимости иметь такой механизм, на многих SQL-серверах это делается совершенно разными способами, что очень затрудняет разработку.

К счастью для Java-разработчиков, при создании JDBC этот важный вопрос был продуман и вы можете использовать стандартное решение. Решение строится на двух моментах.
Во-первых, надо указать список полей, значения которых вы хотите получить после выполнения запроса.
Во-вторых, у объекта **Statement** есть специальный метод получения сгенерированных полей — **getGeneratedKeys()**
Теперь давайте посмотрим код, который демонстрирует этот механизм

```java
package edu.javacourse.database;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class InsertDb
{
    public static void main(String[] args) {
        InsertDb m = new InsertDb();
        try {
            Class.forName("org.postgresql.Driver");
            String url = "jdbc:postgresql://localhost:5432/contactdb";
            String login = "postgres";
            String password = "postgres";
            Connection con = DriverManager.getConnection(url, login, password);
            try {
                // Процедура вставки
                int contactId = m.insert(con, "FirstName", "LastName", "phone", "email");
                System.out.println("CONTACT_ID:" + (Integer) contactId);
            } finally {
                con.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private long insert(Connection con, String firstName, String lastName, String phone, String email) throws SQLExcept
        // Объявили переменную для хранения ИД
        int contactId = -1;

        // Вторым параметром передаем массив полей, значниея которых нам нужны
        PreparedStatement stmt = con.prepareStatement("INSERT INTO jc_contact (first_name, last_name, phone, email) VAI
        stmt.setString(1, firstName);
        stmt.setString(2, lastName);
        stmt.setString(3, phone);
        stmt.setString(4, email);
        stmt.executeUpdate();


        // Получаем список данных дял сгенерированных ключей
        ResultSet gk = stmt.getGeneratedKeys();
        if(gk.next()) {
```

```
47              // Получаем поле contact_id
48              contactId = gk.getInt("contact_id");
49          }
50          stmt.close();
51
52          return contactId;
53      }
54  }
```

Все самое главное находится в методе **insert**. Мы объявили переменную для нового ИД. После создаем **PreparedStatement** и в качестве уже двух параметров передаем строку с запросом (как и раньше) и массив имен столбцов, данные которых нам нужны. У нас только одно поле — **contact_id** — поэтому массив состоит из одного элемента.
**ВАЖНО !!!**Обращаю ваше внимание на то, что JDBC PostgreSQL различает большие и маленькие буквы в этом месте. Это его особенность. В остальных местах регистр не важен. Почему так — не знаю, не разбирался.

В конце используем код для получения значений для поля **contact_id**

```
1           ResultSet gk = stmt.getGeneratedKeys();
2           if(gk.next()) {
3               // Получаем поле contact_id
4               contactId = gk.getLong("contact_id");
5           }
6           stmt.close();
```

Полный пример можно оскачать здесь: InsertDb.

## Транзакция

В общем виде транзакция — это механизм, который позволяет выполнить несколько запросов на изменение данных как одно целое. Т.е. либо все запросы пройдут, либо ни один. Например, вы вставляете данные о пользователе в разные таблицы. В этом случае будет плохо, если вы половину таблиц заполните, а вторая половина останется пустой из-за того, что при исполнении какого-то запроса произойдет ошибка. Тут уж либо во все таблицы надо вставить данные, либо совсем никаких изменений быть не должно.
На мой взгляд достаточно очевидное желание. Часто приводят в пример вариант пересылки денег с одного счета на счет — с одного счета денежку сняли, на другой счет зачислили. И несмотря на то, что это две разные операции, они должны пройти либо обе, либо ни одна из них.
Если немного поразмышлять, то все становится еще загадочнее в случае, когда у вас все это происходит не в одном банке в рамках одной базы данных, а в разных банках — само собой там и базы данных тоже разные. Это уже приближается к понятию “распределенная транзакция”, но об этом мы в этом разделе говорить не будем — слишком сложно и глубоко погружаться придется. Хотя такие задачи существуют и программисты имеют механизмы для их решения.

Мы рассмотрим несложный пример использования транзакции в виде вставки нескольких записей в одну и ту же таблицу, но даже в этом случае мы сможем показать, что либо все несколько записей добавятся, либо ни одна. Давайте смотреть — я специально использовал практически тот же пример, что и в предыдущем разделе — просто добавил цикл для вставки сразу 5 записей и сделал это в рамках одной транзакции, так что все изменения надо смотреть в методе **main**:

```
1  package edu.javacourse.database;
2
3  import java.sql.Connection;
4  import java.sql.DriverManager;
5  import java.sql.PreparedStatement;
6  import java.sql.ResultSet;
7  import java.sql.SQLException;
8  import java.sql.Statement;
9
10 public class InsertTransactionDb
11 {
12     public static void main(String[] args) {
13         InsertTransactionDb m = new InsertTransactionDb();
14
15         try {
16             Class.forName("org.postgresql.Driver");
17             String url = "jdbc:postgresql://localhost:5432/contactdb";
18             String login = "postgres";
19             String password = "postgres";
20             Connection con = DriverManager.getConnection(url, login, password);
21
22             // Демонстрация использования транзакций
23             try {
24                 // Шаг первый - выставляем свойство AutoCommit в false
25                 con.setAutoCommit(false);
26                 // В цикле вставлем несколько записей
27                 for (int i = 0; i < 5; i++) {
28                     long contactId = m.insert(con, "FirstName_" + i, "LastName_" + i, "phone", "email");
29                     System.out.println("CONTACT_ID:" + contactId);
30                 }
31                 // Завершаем транзакцию - подтверждаем
32                 con.commit();
33                 // Вызов rollback отменит все внесенные изменения
34                 //con.rollback();
35
36                 // Возвращаем свойство AutoCommit в true
37                 con.setAutoCommit(true);
38
39                 // Можно проверить результат
40                 m.select(con);
```

```
            } finally {
                con.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private long insert(Connection con, String firstName, String lastName, String phone, String email) throws SQLExcept
        long contactId = -1;
        PreparedStatement stmt = con.prepareStatement(
            "INSERT INTO jc_contact (first_name, last_name, phone, email) VALUES (?, ?, ?, ?)",      new String[]{"conta
        stmt.setString(1, firstName);
        stmt.setString(2, lastName);
        stmt.setString(3, phone);
        stmt.setString(4, email);
        stmt.executeUpdate();

        ResultSet gk = stmt.getGeneratedKeys();
        while (gk.next()) {
            contactId = gk.getLong("CONTACT_ID");
        }
        stmt.close();

        return contactId;
    }

    private void select(Connection con) throws SQLException {
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * FROM jc_contact");
        while (rs.next()) {
            String str = rs.getString("contact_id") + ":" + rs.getString(2);
            System.out.println("Contact:" + str);
        }
        rs.close();
        stmt.close();
    }
}
```

Как я уже говорил всек самое интересное находится в самом начале программы — в методе **main**

```
// Шаг первый - выставляем свойство AutoCommit в false
con.setAutoCommit(false);
// В цикле вставлем несколько записей
```

```
for (int i = 0; i < 5; i++) {
    long contactId = m.insert(con, "FirstName_" + i, "LastName_" + i, "phone", "email");
    System.out.println("CONTACT_ID:" + contactId);
}
// Завершаем транзакцию - подтверждаем
con.commit();
// Вызов rollback отменит все внесенные изменения
//con.rollback();
```

Первый шаг — это выставление свойства соединения **autoCommit** в **false**. Это свойство говорит о том, надо ли автоматически помещать исполнение каждого запроса (Statement) как-бы в рамки одной транзакции. Т.е. Получается такая маленькая компактная одноразовая транзакция — сделал запрос и транзакция в рамках этого запроса и началась и закончилась. Если запрос выполнился успешно — транзакция завершилась подтверждением — **commit**. Выполнился запрос неудачно — транзакция откатывается — **rollback**. Так вот по умолчанию это свойство стоит в **true**. Для того, чтобы самим начать управлять транзакциями надо это свойство поставить под свое управление — запретить думать за вас. С этого момента все ваши запросы вам надо будет либо подтвердить — что сделано сразу после цикла путем вызова **con.commit();**, либо вам надо отменять все ваши запросы — вариант вызова чуть ниже (закоментирован) — **con.rollback();**.
Я рекомендую сначала запустить прирме как есть — и в конце за счет кода

```
// Можно проверить результат
m.select(con);
```

можно посмотреть, что записи вставились (конечно, если пример успешно завершился).
После этого закомментировать вызов **con.commit();** и раскомментировать вызов **con.rollback();**. Теперь записи не будут вставлены, хотя запросы на вставку будут выполнены.
Вы даже это сможете увидеть — буду выводится строки с CONTACT_ID. А вот новых записей не будет.
Подведем итог разговора о транзакциях. Во-первых, конечно же я очень упрощенно рассказал об этом весьма любопытном, сложном и важном механизме. Во-вторых, логика чередования вызовов **commit** и **rollback** ложится на плечи программиста. Использование более сложных пакетов и технологий в большинстве случаев позволяет сделать эту работу удобнее и легче, но тем не менее алгоритм, логика — все на ваше усмотрение.
Достаточно часто используется конструкция подобная такой:

```
try {
    // устанавливаем ручное управление транзакциями
    con.setAutoCommit(false);

    …. .. вот тут выполняется много всяких запросо на модификацию

    // Если все прошло успешно - делаем commit
    con.commit();
```

```
 9 catch(Exception ex) {
10     // Если не прошло успешно и мы вылетели на каком-то запросе - делаем rollback
11     con.rollback();
12 }
```

Эта конструкция достаточно удобная и покрывает большое количество ситуаций, но не факт, что она выручит вас в каком-то сложном случае.

Полный пример можно скачать здесь: InsertTransactionDb.

И теперь нас ждет следующая статья: JDBC — групповые операции

## 2 comments to *Возможности JDBC — второй этап*

- 
  Декабрь 11, 2017 at 18:54
  *siriosca* says:

  Спасибо огромное за статью, по моему мнению лучше читать такие статьи чем смотреть видео уроки всяких неуч.

  P.S: добавьте метку времени актуальности статьи.

  Reply

  - 
    Декабрь 11, 2017 at 23:21
    *admin* says:

    Спасибо за отзыв/ И за идею — наверно есть смысл писать дату публикации.

    Reply

## Leave a reply

Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code class="" title="" data-url=""> <del datetime=""> <em> <i> <q cite=""> <s> <strike> <strong> <pre class="" title="" data-url=""> <span class="" title="" data-url="">`

Имя *

E-mail *

Сайт

1 × = семь

Add comment