

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

«Вятский государственный университет»

(ФГБОУ ВПО «ВятГУ»)

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

Методические указания
к самостоятельным и лабораторным работам
по дисциплине «Базы данных»

Киров 2019

Лабораторная работа №4

Связывание приложения на Java с базой данных под управлением PostgreSQL

Цели лабораторной работы:

- Познакомиться со стандартным API в Java для связывания приложения с БД
- Изучить некоторые шаблоны проектирования, связанные с работой с БД
- Освоить на практике основы взаимодействия с БД под управлением PostgreSQL в приложении на Java

Задание на лабораторную работу:

Создать приложение с графическим приложением на языке Java. Приложение должно использовать БД, разработанную в предыдущих лабораторных работах.

Для любой одной таблицы, которая содержит внешний ключ на другую таблицу, приложение должно выполнять следующее:

- Выводить строки таблицы
- Предоставлять любой фильтр по значению строк. (Например, «Дата с ... по ...» или «Имя содержит ...»)
- Предоставлять возможность добавления новых строк
- Предоставлять возможность удаления строки

Требования к реализации:

- Заголовки должны быть осмысленными. Например, вместо «name» в таблице должен быть заголовок «Имя»
- При добавлении новой строки внешний ключ выбирается из списка
- Сохранение или удаление строки должно быть реализовано с помощью функции PL/pgSQL
- Фильтрация значений при поиске должна производиться через запрос, а не в полученной коллекции
- Разрешается использование любого фреймворка
- При разработке нужно использовать шаблоны проектирования, связанные с работой с БД

Отчет должен содержать:

- Диаграмму классов, организующих работу с БД
- Их исходный код

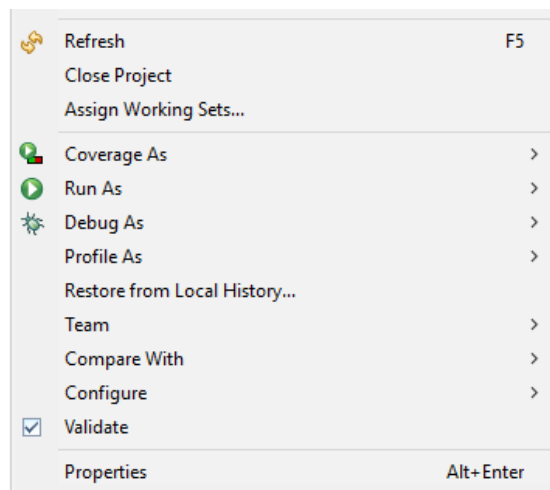
- Экранные формы

Подготовка среды

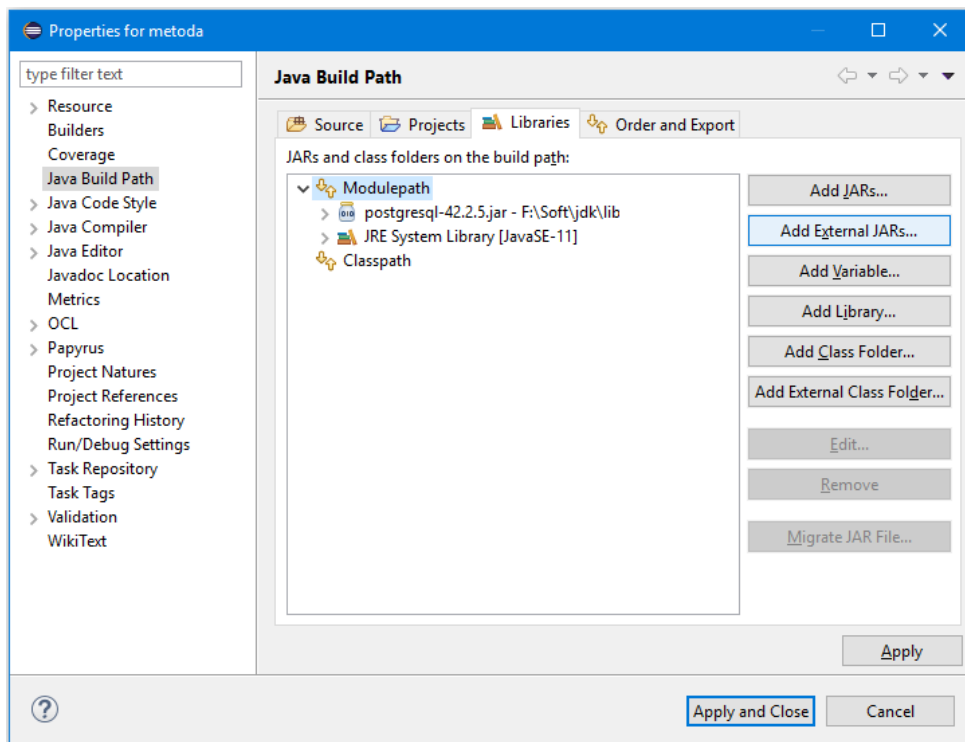
JDBC предоставляет простой способ подключения к базам данных. Для начала работы нам потребуется драйвер для базы данных. Для этого нужно перейти на <https://jdbc.postgresql.org/download.html> и скачать его.

В примерах используется IDE Eclipse. Необходимые действия могут изменяться в зависимости от среды разработки.

После загрузки необходимо добавить драйвер в проект. Для этого нужно зайти в свойства проекта через пункт Properties.

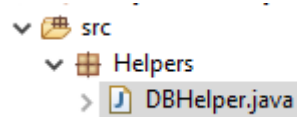


Далее Java Build Path -> Modulepath и нажать в меню справа «Add External JARs». Далее нужно выбрать загруженный драйвер. В результате Modulepath должен иметь следующий вид.



Подключение к базе данных

Чтобы подключиться к базе данных, добавим пакет `Helpers` и создадим в нем класс `DBHelper`. Через этот класс мы будем осуществлять все взаимодействие с базой данных.



Чтобы выполнить подключение нужно объяснить драйверу, куда мы хотим попасть. Строка подключения выглядит следующим образом:

```
"jdbc:postgresql://localhost:5432/lab_db";
```

Разберемся что есть что в этой строке

jdbc:postgresql – указывает на то, что мы подключаемся к субд postgresql

://localhost – адрес базы данных

5432 – Порт для подключения

lab_db – имя БД

Чтобы зарегистрировать драйвер postgresql в системе, вызовем следующий метод:

```
Class.forName("org.postgresql.Driver");
```

Это позволит выполнить проверку на наличие драйвера в системе. Далее можно будет подключиться к базе. Для этого используется класс *DriverManager*.

Для соединения необходимо вызвать

DriverManager.getConnection() и передать туда адрес базы, логин и пароль.

В итоге для подключения потребуется следующий код:

```
package Helpers;

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

public class DBHelper {

    //Объект соединения с БД
    private Connection connection;

    //Строка подключения к БД
    private static String DB_URL = "jdbc:postgresql://localhost:5432/lab_db";

    //Имя пользователя и пароль для подключения
    private static String DB_USER = "lab_user";
    private static String DB_PASSWORD = "lab_user";

    public DBHelper() {
        try {
            //Регистрируем драйвер
            Class.forName("org.postgresql.Driver");

            //Соединяемся с базой
            connection = DriverManager.getConnection(DB_URL, DB_USER,
DB_PASSWORD);

            System.out.println("Успешное подключение");
        } catch (ClassNotFoundException e) {
            //Сюда мы попадем, если драйвер не будет установлен
            System.out.println("Не удалось загрузить драйвер");
        } catch (SQLException e) {
            //Сюда мы попадем, если возникнет проблема при соединении
            System.out.println("Не удалось подключиться к БД");
        }
    }

    //Закрытие соединения
    public void closeConnection()
    {
        try {
            if (connection != null && !connection.isClosed()) {
                connection.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Создадим так же класс, который будет обращаться к БД через созданный только что класс.

```
package MainPackage;

import Helpers.DBHelper;

public class Main {

    public static void main(String[] args) {

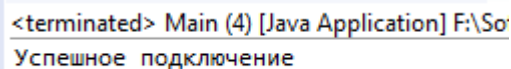
        DBHelper db = new DBHelper();
        db.closeConnection();
    }
}
```

В результате будет происходить следующее:

- Создается объект DBHelper
- При создании происходит попытка подключиться к БД.
- Если соединение успешно, будет выведено сообщение об успехе.
- Если подключиться не выйдет, будет выведено сообщение об ошибке.

Не забывайте закрывать за собой подключение. В примере для этого используется метод `closeConnection()`. Если не закрыть соединение, оно может повиснуть на стороне СУБД, и привести к ошибкам.

Если все сделано правильно, вы получите следующий вывод:



```
<terminated> Main (4) [Java Application] F:\So
Успешное подключение
```

Выполнение запросов к БД

Для выполнения обращений к БД используется специальный интерфейс Statement. Создать его можно через существующее подключение.

```
connection.createStatement();
```

Statement представляет собой строку запроса, которую можно будет передать к БД. Есть 3 метода, через которые можно выполнить запрос к БД

- `execute(String sql)`. Возвращает true, если запрос что то вернул.
- `executeUpdate(String sql)`. Возвращает количество строк, которых коснулось выполнение. Обычно используется для запросов insert, update, delete
- `executeQuery(String sql)`. Возвращает экземпляр ResultSet, через который можно посмотреть, что вернула СУБД. Обычно используется для запросов select.

После выполнения запроса необходимо его закрыть.

Выполним простой запрос к базе данных. Например, считаем данные из таблицы group. Запрос будет иметь следующий вид:

```
"SELECT * FROM public.group"
```

Добавим в класс DBHelper следующий код:

```
public void simpleSelect() {
    //Объявляем объекты запроса и результата
    Statement statement;
    ResultSet result;
    try {
        //Создаем экземпляр запроса
        statement = connection.createStatement();

        //Выполняем запрос к БД
        result = statement.executeQuery("SELECT * FROM public.group");

        //Выполняем проход по строкам таблицы и выводим в консоль
        while(result.next())
        {
            System.out.printf("Группа: %s\nЧисло студентов: %d\n\n",
                               result.getString(2),
                               result.getInt(3));
        }

        //Закрываем соединение
        statement.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Добавим в главный класс следующую строчку до закрытия соединения:

```
db.simpleSelect();
```


Если все сделано правильно, будет выведен результат запроса

```
Успешное подключение
Группа: ИВТ6-3301-01-00
Число студентов: 15

Группа: ИВТ6-3302-02-00
Число студентов: 20

Группа: ИВТ6-4301-01-00
Число студентов: 12
```

Объект `ResultSet` содержит в себе курсор, который перемещается внутри результата. Изначально он стоит на позиции до первой строки. Метод `next()` позволяет перемещать курсор вперед.

Для получения значений в строке используются методы `getXXX()`, где `XXX` – возвращаемый тип.

Важно: нумерация столбцов таблицы происходит с 1, а не с 0!

В предыдущем примере в методах `get...` использованы номера столбцов. Вместо них можно использовать их имена. Это позволит улучшить читаемость кода и упростит работу с таблицей.

Следующий код будет выполнять то же самое, но обращение будет через имена столбцов:

```
public void simpleSelect() {
    //Объявляем объекты запроса и результата
    Statement statement;
    ResultSet result;
    try {
        //Создаем экземпляр запроса
        statement = connection.createStatement();

        //Выполняем запрос к БД
        result = statement.executeQuery("SELECT * FROM public.group");

        //Выполняем проход по строкам таблицы и выводим в консоль
        while(result.next())
        {
            System.out.printf("Группа: %s\nЧисло студентов: %d\n\n",
                               result.getString("name"),
                               result.getInt("number_of_students"));
        }

        //Закрываем соединение
        statement.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Теперь можно заметить, что после слова «группа» выводится значение столбца с именем «name», а не какого-то столбца с номером 2.

Существует такой тип запросов, как подготовленные запросы. В такие запросы можно передавать параметры, и они могут дать выигрыш в скорости работы при частом использовании. Пример кода приведен ниже:

```
public void simplePreparedSelect() {
    //Объявляем объекты запроса и результата
    PreparedStatement preparedStatement;
    ResultSet result;
    try {
        //Создаем экземпляр запроса
        preparedStatement = connection.prepareStatement("SELECT * FROM
public.group WHERE number_of_students > ?");

        //Добавление параметров
        preparedStatement.setInt(1, 14);

        //Выполняем запрос к БД
        result = preparedStatement.executeQuery();

        //Выполняем проход по строкам таблицы и выводим в консоль
        while(result.next())
        {
            System.out.printf("Группа: %s\nЧисло студентов: %d\n\n",
                result.getString("name"),
                result.getInt("number_of_students"));
        }

        //Закрываем соединение
        preparedStatement.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

В данном случае к запросу добавляется ограничение. Будут выводиться только те группы, где число учеников больше 14.

Для добавления параметра используются методы setXXX, где XXX – так же тип значения. В качестве параметров передаются положение параметра в запросе и его значение. Все параметры в запросе должны быть заполнены, иначе будет поднято исключение. Однако значения параметров могут быть null.

На самом деле, добавлять свои параметры можно и в обычный Statement, но лучше используйте PreparedStatement

Важно: как и в объекте ResultSet, нумерация идет с 1, а не с 0!

При работе с СУБД для обозначения дат и времени используются классы из пакета java.sql. В приложении лучше использовать классы из пакета java.util, поэтому следует выполнять приведения.

```
java.util.Date date = resultSet.getDate(1);
java.util.Date time = resultSet.getTime(1);
java.util.Date dateTme = resultSet.getTimestamp(1);
```

Выполнение запросов на изменение/удаление

Запросы для удаления или изменения элементов таблиц не сильно отличаются от select запроса, и выполняются с использованием тех же объектов Statement/PreparedStatement.

Для примера выполнения запросов insert/update/delete поработаем с таблицей subject. Изначально таблица имеет следующие данные:

id	name	number_of_hours
1	Технологии программирования	18
2	История	36

Для начала, добавим новый элемент в таблицу. Ниже представлен код для добавления:

```
public void simpleInsert() {  
    //Объявляем объекты запроса и результата  
    PreparedStatement preparedStatement;  
    try {  
        //Создаем экземпляр запроса  
        preparedStatement = connection.prepareStatement("INSERT INTO  
subject(name, number_of_hours) VALUES(?, ?)");  
  
        //Добавление параметров  
        preparedStatement.setString(1, "Русский язык");  
        preparedStatement.setInt(2, 100500);  
  
        //Выполняем запрос к БД  
        preparedStatement.execute();  
  
        //Закрываем соединение  
        preparedStatement.close();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

Как можно заметить, формат метода не изменился относительно select запроса. Но теперь вместо executeQuery используется обычный execute. Почему? Потому что читать после вызова insert нечего, да и не всегда нужно.

После добавления можно увидеть, как таблица пополнилась

id	name	number_of_hours
1	Технологии программирования	18
2	История	36
3	Русский язык	100500

Теперь изменим последнюю строку в таблице. Для этого необходимо выполнить запрос update. Ниже представлен код для изменения данных.

```
public void simpleUpdate() {
    //Объявляем объекты запроса и результата
    PreparedStatement preparedStatement;
    try {
        //Создаем экземпляр запроса
        preparedStatement = connection.prepareStatement("UPDATE subject
SET name = ?, number_of_hours = ? WHERE id = 3");

        //Добавление параметров
        preparedStatement.setString(1, "Физика");
        preparedStatement.setInt(2, 30);

        //Выполняем запрос к БД
        preparedStatement.execute();

        //Закрываем соединение
        preparedStatement.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Аналогичным образом можно выполнить и delete запрос.

Тут так же можно использовать и обычный Statement, а параметры подставить, например, через String.format

Вызовы функций

Теперь попробуем вызвать функцию, которая была добавлена в базу в рамках предыдущей работы. Возьмем, например, вот эту:

```
CREATE OR REPLACE FUNCTION foo ()  
RETURNS SETOF subject  
AS $$  
BEGIN RETURN QUERY (SELECT * FROM subject);  
END;  
$$ LANGUAGE PL/pgSQL;
```

Вызывать функции можно и с использованием обычного select.

```
public void simpleFunctionSelect() {  
    //Объявляем объекты запроса и результата  
    Statement statement;  
    try {  
        //Создаем экземпляр запроса  
        statement = connection.createStatement();  
  
        //Выполняем запрос к БД  
        ResultSet result = statement.executeQuery("SELECT * FROM foo()");  
  
        while(result.next()) {  
            System.out.printf("Предмет: %s\nЧисло часов: %d\n\n",  
                               result.getString("name"),  
                               result.getInt("number_of_hours"));  
        }  
  
        //Закрываем соединение  
        statement.close();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

В таком случае в select запрос просто подставляется вызов функции.

Но есть и другой вариант. Для вызова процедур используется интерфейс CallableStatement. Если использовать его, код примет следующий вид:

```
public void simpleFunctionCall() {
    //Объявляем объекты запроса и результата
    CallableStatement preparedStatement;
    try {
        //Создаем экземпляр запроса
        preparedStatement = connection.prepareCall("{call foo()}");

        //Выполняем запрос к БД
        ResultSet result = preparedStatement.executeQuery();

        while(result.next()) {
            System.out.printf("Предмет: %s\nЧисло часов: %d\n\n",
                               result.getString("name"),
                               result.getInt("number_of_hours"));
        }

        //Закрываем соединение
        preparedStatement.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Также в вызовы функций можно подставлять свои параметры, и получать значения. Возьмем для этого функцию save_subject

```
CREATE OR REPLACE FUNCTION save_subject (
    _id BIGINT,
    _name VARCHAR(50),
    _num_of_hours INT
)
RETURNS BIGINT
AS $$
DECLARE
    used_id BIGINT;

BEGIN
    IF _id IS NULL THEN

        INSERT INTO subject (name, number_of_hours)
        VALUES (_name, _num_of_hours)
        RETURNING id /* Конструкция позволяет вернуть id нового элемента */
        INTO used_id; /* id нового элемента записывается в переменную used_id */
    ELSE
        UPDATE subject SET
            name = _name,
            number_of_hours = _num_of_hours
        WHERE id = _id;

        used_id := _id; /* Нам уже известен id, поэтому просто присвоим его */
    END IF;

    RETURN used_id;
END;
$$ LANGUAGE PL/pgSQL;
```

В этой функции есть и входные параметры, и выходные. А значит можно посмотреть, как ими пользоваться. Код будет выглядеть следующим образом.

```
public void functionCall() {
    //Объявляем объекты запроса и результата
    CallableStatement callableStatement;
    try {
        //Создаем экземпляр запроса
        callableStatement = connection.prepareCall("{? = call
save_subject(?, ?, ?)}");

        //Регистрируем выходное значение
        callableStatement.registerOutParameter(1, Types.BIGINT);

        //Добавим параметры
        callableStatement.setInt(2, 2);
        callableStatement.setString(3, "Любой предмет");
        callableStatement.setInt(4, 42);

        //Выполняем запрос к БД
        callableStatement.execute();

        System.out.printf("Изменена строка %d\n",
callableStatement.getLong(1));

        //Закрываем соединение
        callableStatement.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Изменился формат вызова функции. Теперь мы требуем возврата значения конструкцией "? = " в начале вызова.

Далее необходимо указать, какой тип данных мы будем получать. Для этого применяется метод `registerOutParameter()`, который принимает номер параметра в выражении, и номер типа этого параметра. Все возвращаемые типы хранятся в `java.sql.Types`

Далее через методы `set` как в обычном `PreparedStatement` указываются входные параметры функции.

Значения из запроса мы будем получать через параметр, поэтому `ResultSet` здесь не понадобится. Выходные параметры можно получить через `getXXX` методы, как и в других запросах.

Немного дополнительной информации

Транзакции

При работе с СУБД есть такое понятие как транзакция. Это по сути любое действие, совершаемое с БД. Проблема в том, что создание транзакций – достаточно тяжелая для СУБД операция, и при большом количестве запросов может снижать ее быстродействие.

Например, у вас есть 10 запросов к БД, которые выполняются друг за другом, и если в одном из них возникнет ошибка, откатить все изменения.

Если каждый запрос будет проходить в отдельной транзакции, то СУБД создаст 10 транзакций, а нам по сути надо сделать одно действие. Упаковка всех действий в одну транзакцию позволит повысить быстродействие.

Если в одном из запросов возникнет ошибка, откатить все предыдущие вызовы будет нелегко, если они были в отдельных транзакциях. Но если все выполняется в пределах одной, то достаточно будет выполнить функцию *rollback()*, которая откатит изменения, произошедшие в транзакции.

Итак, использование транзакций позволяет достичь как минимум двух вещей:

- Безопасность при выполнении нескольких запросов, если между ними есть зависимость
- Повышение быстродействия путем упаковки всех действий в одну транзакцию.

Для того, чтобы начать управлять транзакциями, достаточно вызвать

```
connection.setAutoCommit(false);
```

Это выключит автоматическое применение изменений при выполнении запросов. Выполнять данное действие стоит до того, как вы начнете что-то менять.

Чтобы применить выполненные изменения, нужно вызвать

```
connection.commit();
```

А если у вас что-то пошло не так и надо все вернуть назад, то нужно вызвать

```
connection.rollback();
```

И, конечно же, не забывайте закрывать за собой обращения к БД.

Шаблоны проектирования:

Для упрощения своей жизни при разработке приложений используются различные шаблоны проектирования. Для работы с БД это могут быть, например, шаблоны DTO(Data transfer object), DAO (Data Access Object), Query Object Service.

DTO – это шаблон для передачи данных внутри приложения. На самом деле это просто обертка для сущности таблицы. DTO содержит в себе поля, однозначно соответствующие полям таблицы. Все, что он делает – хранит в себе значение какой-либо сущности в БД, и может быть иногда выполняет какую-то мелкую логику, но основная задача DTO – представление данных таблицы.

Пример DTO:

```
public class Item {  
  
    private Long id;  
    private String code;  
    private String name;  
    private Double price;  
    // Ниже геттеры и сеттеры  
    // ...  
}
```

DAO – еще один шаблон проектирования. Все примеры выше как раз таки используют DAO, но не совсем.

DAO представляет собой интерфейс для взаимодействия с различными БД. Обычно он предоставляет какой-то базовый функционал: чтение, создание, удаление, изменение, поиск. Использование DAO позволяет практически на лету менять базы. Так как DAO представляет собой интерфейс(а это важно), можно подставлять различные реализации обращений к различным бд, просто подставляя в объект с типом интерфейса нужные реализации(*чтобы понять как это работает, нужно знать, как работает полиморфизм*).

Если, допустим, сделать некий интерфейс DAO, а в классе DBHelper реализовать функции этого интерфейса, то класс станет одной из реализаций.

Еще один плюс DAO – единая точка входа. Всегда понятно, где лежит код для взаимодействия с БД.