

Модуль 5. Основы разработки серверной части мобильных приложений

Тема 5.5. Серверные СУБД

6 часов

Оглавление

5.5. Серверные СУБД.....	2
5.5.1. Клиент-серверные архитектуры. Серверные СУБД.....	2
5.5.2. Настройка PostgreSQL и подключение к БД.....	4
Установка в Heroku СУБД PostgreSQL.....	4
Установка интерфейса к СУБД pgAdmin.....	5
Создание таблиц в БД.....	8
Командная строка PostgreSQL.....	11
5.5.3 Реализация back end части приложения на языке Java.....	11
*5.5.4 Реализация back end части приложения на языке PHP.....	21
5.5.5. Практикум.....	24
Задание.....	24
Реализация. Серверная часть.....	25
Реализация. Клиентская часть.....	28
Задание 5.5.1.....	40
*5.5.6 Синхронизация баз данных (локальной и серверной).....	41
*5.5.7. Индексы базы данных.....	43
Создание индекса.....	43
Применение индексов.....	45
Источники.....	46
Благодарности.....	46

5.5. Серверные СУБД

5.5.1. Клиент-серверные архитектуры. Серверные СУБД

С клиент-серверной архитектурой в общем мы познакомились в теме 5.3, мы знакомы с локальной системой управления базами данных SQLite и изучили, как Android-приложение работает в качестве клиента.

Преимущества использования баз данных абсолютно понятно - это централизованность (данные хранятся в одном месте) и структурированность (данные логически связаны между собой, образуя определенную структуру, которая позволяет хранить информацию о некоторой предметной области). Рассмотрим различные варианты архитектуры построения приложений (рис.1).

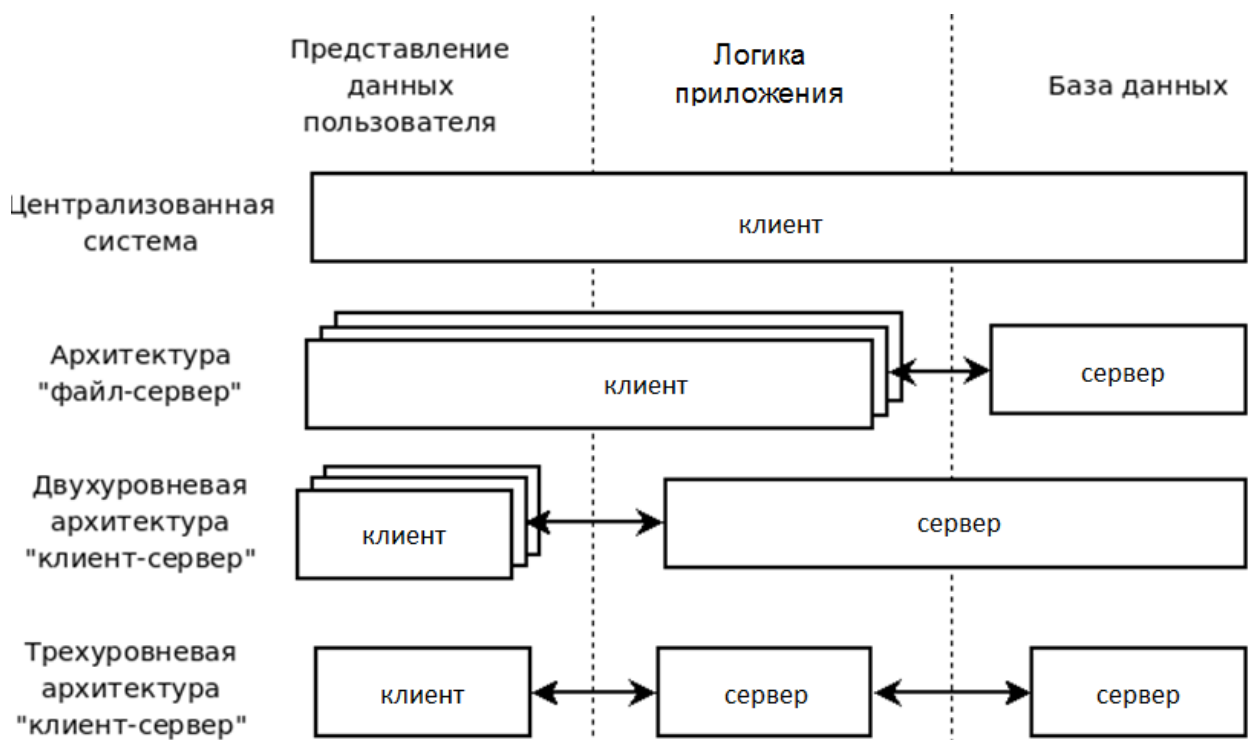


Рисунок 1 - Основные архитектуры приложений БД

С **централизованной архитектурой** мы уже знакомы - это наши предыдущие приложения с SQLite. В чем недостатки таких систем? Представим, что мы создали Android приложение работающее с базой данных, в которую положили необходимую информацию. Это значит, что когда нам потребуется изменить данные в этой базе, мы сможем их изменить, только, обновив все приложение. К тому же, вся логика работы с БД, а следовательно и вычислительная нагрузка ложатся на смартфон пользователя. Согласитесь, не очень хорошо заставлять пользователя постоянно скачивать такие мало полезные обновления и нагружать его телефон.

Файл-серверная архитектура не получила распространения среди мобильных приложений по причине того, что хотя данные и лежат на сервере, но копия СУБД остается на устройстве и "ест" его ресурсы.

Двухуровневая клиент-серверная архитектура предполагает использование клиент-серверной СУБД. Такая СУБД использует технологию «клиент-сервер», размещается на сервере и позволяет обмениваться клиенту и серверу минимально необходимыми объёмами информации. При этом основная вычислительная нагрузка ложится на сервер. Клиент может выполнять функции предварительной обработки перед передачей информации серверу, но в основном его функции заключаются в организации доступа пользователя к серверу. Сервер баз данных даёт возможность отказаться от пересылки по сети файлов данных и передавать только ту выборку из базы данных, которая удовлетворяет запросу пользователя. Логика, необходимая для работы с базой данных остаётся всё ещё на клиентских устройствах пользователей, что в случае смартфона является большим недостатком, кроме того велика вероятность возникновения ошибок и других проблем. Здесь же стоит упомянуть, что если необходимо клиентские приложения под различные платформы, то придётся продублировать много одинакового по своей логике кода.

Трёхуровневая (трехзвенная) архитектура решает эту проблему путем переноса всей работы с СУБД в отдельный слой. В этом случае клиентское приложение будет являться лишь средством ввода и отображения информации, а все «мозги» нашей системы будут располагаться на сервере.

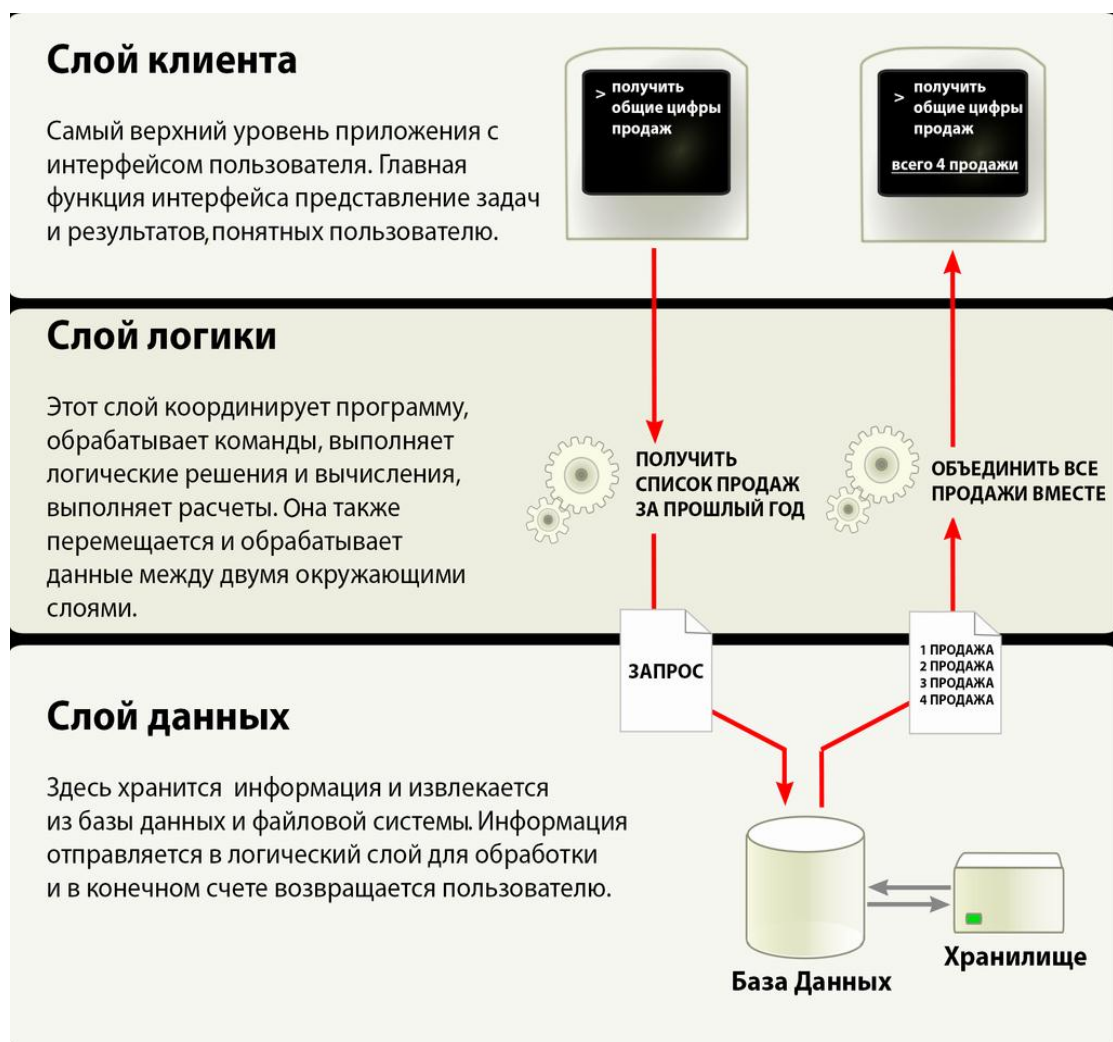


Рисунок 2 Трёхзвенная архитектура приложения

Рисунок 2 в полной мере описывает работу трёхзвенной архитектуры приложения, для каждого слоя слева располагается поясняющая информация о каждом уровне.

С развитием сетей и интернета данная трехуровневая архитектура построения приложений приобретала всё более популярный характер, что повлекло за собой появление различных новых программных комплексов, качественно реализующих её. Однако, в более сложных приложениях используют более масштабируемую **многоуровневую архитектуру**.

На рынке серверного ПО существует достаточно много СУБД. Наиболее распространенными являются MySQL, PostgreSQL, Oracle, SQL Server.

На прошлых занятиях мы познакомились и зарегистрировались на облачной платформе Heroku. Она как раз является одним из многих решений для размещения серверных частей приложений. Это достаточно мощная платформа которая поддерживает около семи языков программирования, сервера приложений (мы используем веб-сервер Tomcat) и может работать с пятью СУБД, где основной является СУБД PostgreSQL. Данная платформа подходит нам для создания учебных приложений, потому что поддерживает необходимое ПО и дает возможность её бесплатного использования.

5.5.2. Настройка PostgreSQL и подключение к БД

Как уже было сказано, в конце предыдущей главы, в качестве серверной платформы выбрана <https://www.heroku.com/>. В контексте данной главы, нашей задачей является создание базы данных в СУБД PostgreSQL Heroku, и подключение к ней для последующей работы.

Для начала нам потребуется:

1. Успешно зарегистрированный аккаунт на <https://www.heroku.com/>.
2. Успешно установленная СУБД <http://www.postgresql.org/download/>.

Установка в Heroku СУБД PostgreSQL

Если всё это есть, можно смело переходить к созданию базы данных. К слову сказать, это будет самое простое создание базы данных, которое вы когда-либо видели, потому что для этого необходимо всего четыре клика мышкой. Рисунки 3-6 демонстрируют этот процесс.

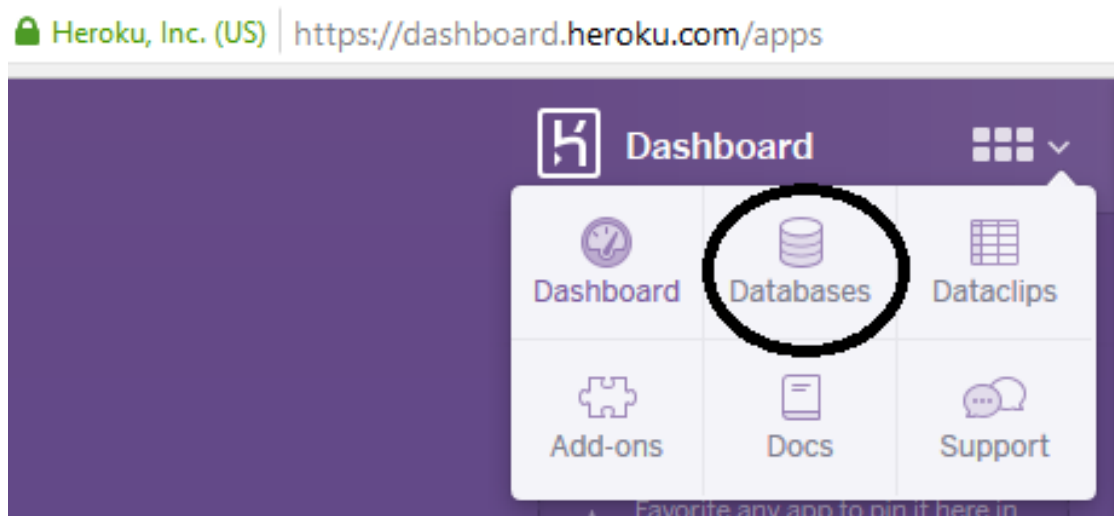


Рисунок 3 - переход в раздел Databases (базы данных)

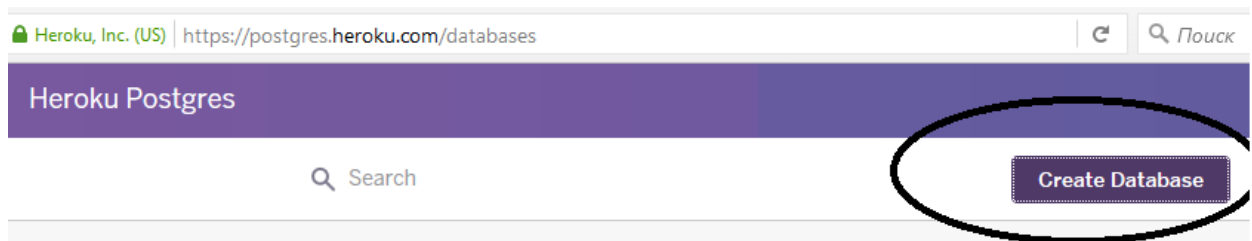


Рисунок 4 - нажатие на кнопку Create Database (создать базу данных)

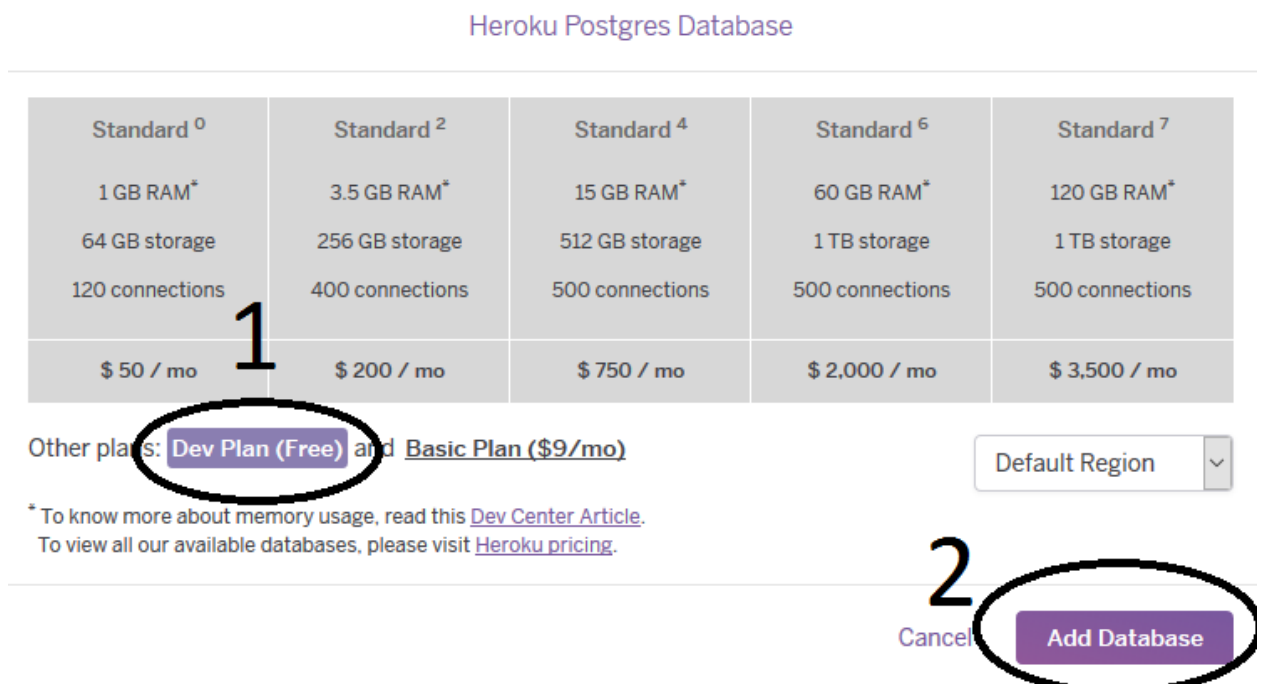


Рисунок 5 - выбор Dev Plan (Free) (бесплатного плана разработчика) и нажатие на кнопку Add Database (добавить базу данных)

После проделанных выше действий, необходимо подождать некоторое время, приблизительно 30-60 секунд для того, что бы СУБД успешно установилось. Подтверждением этого будет являться её статус Available (доступно) как показано на рисунке 6.

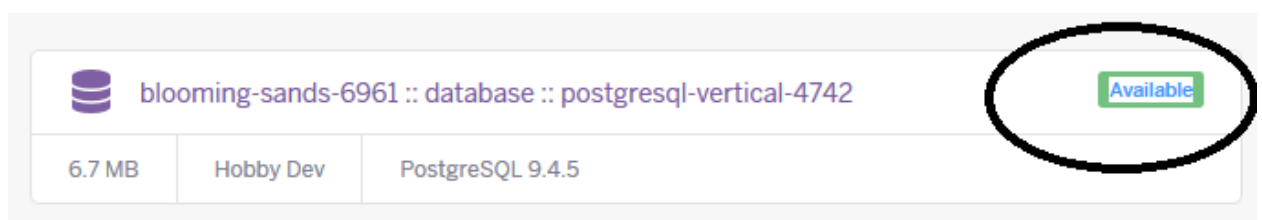


Рисунок 6 - завершение создания БД, статус Available (доступно)

Установка интерфейса к СУБД pgAdmin

Можем себя поздравить с успешно созданной базой данных. Однако, к большому сожалению, на платформе Heroku нет полноценного web-интерфейса для управления созданной нами базой данных. Поэтому мы воспользуемся pgAdmin - свободно распространяемой программой для создания пользовательского интерфейса к СУБД PostgreSQL, которую необходимо установить на

компьютер (скачать здесь <http://www.pgadmin.org/download/>).

Запустим на своём устройстве pgAdmin и создадим подключение к нашему серверу на Heroku. Для этого в запущенном pgAdmin выполним следующие действия, выберем пункт в главном меню “Файл”, а затем “Добавить сервер”, перед нами откроется окно “Новая регистрация сервера” изображенное на рисунке 7.

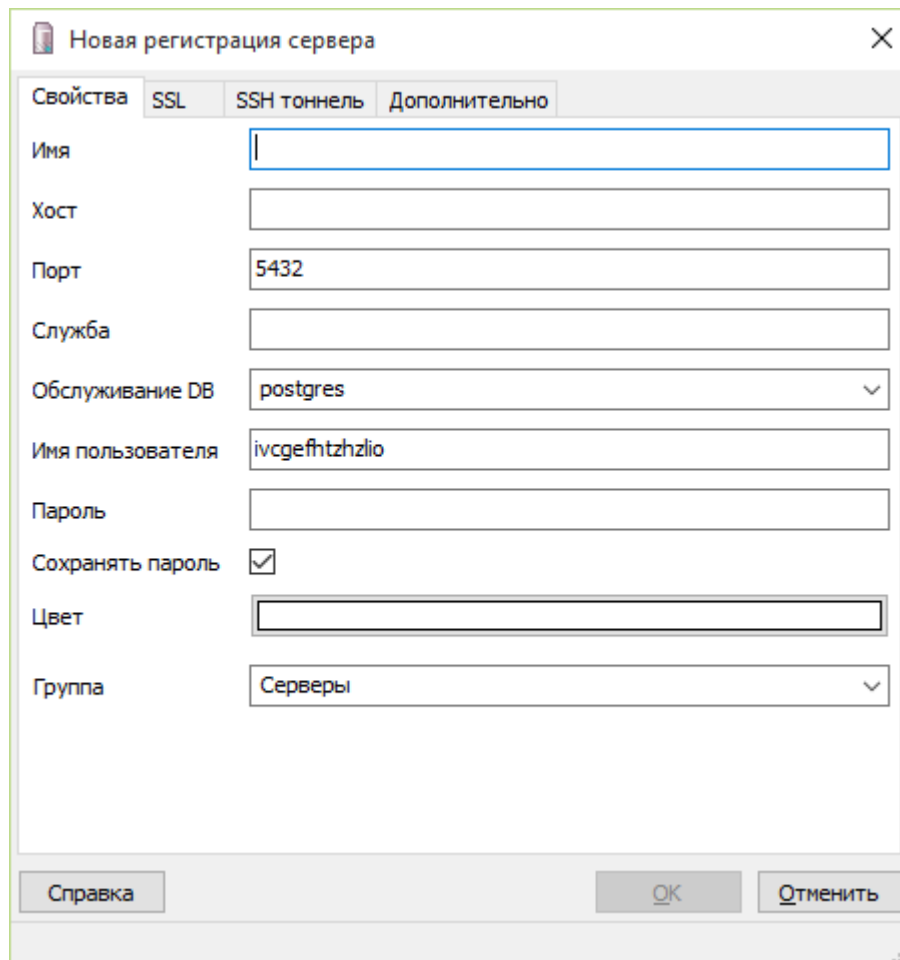


Рисунок 7 - Новая регистрация сервера

Теперь заполним недостающие значения и где необходимо изменим существующие. Встаёт вопрос, откуда взять данные для заполнения?! Эта информация доступна нам на сервисе heroku, если мы перейдём в подробное описание базы данных - клик на её название (Рисунок 8).

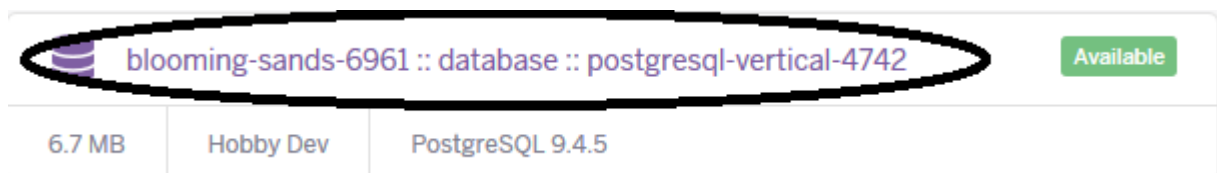


Рисунок 8 - выполнение клика по названию базы данных

Перенесём необходимые параметры из раздела Connection Settings (настройки соединения) в окно pgAdmin “Новая регистрация сервера” следующим образом:

- имя задаём произвольное, например Heroku;

- Host - Хост;
- Database - Обслуживание DB;
- User - Имя пользователя;
- Port - Порт;
- Password - Пароль;

Пустым остаётся лишь пункт в pgAdmin “Новая регистрация сервера” - “Служба”. Результат вышеописанных действий изображен на рисунке 9.

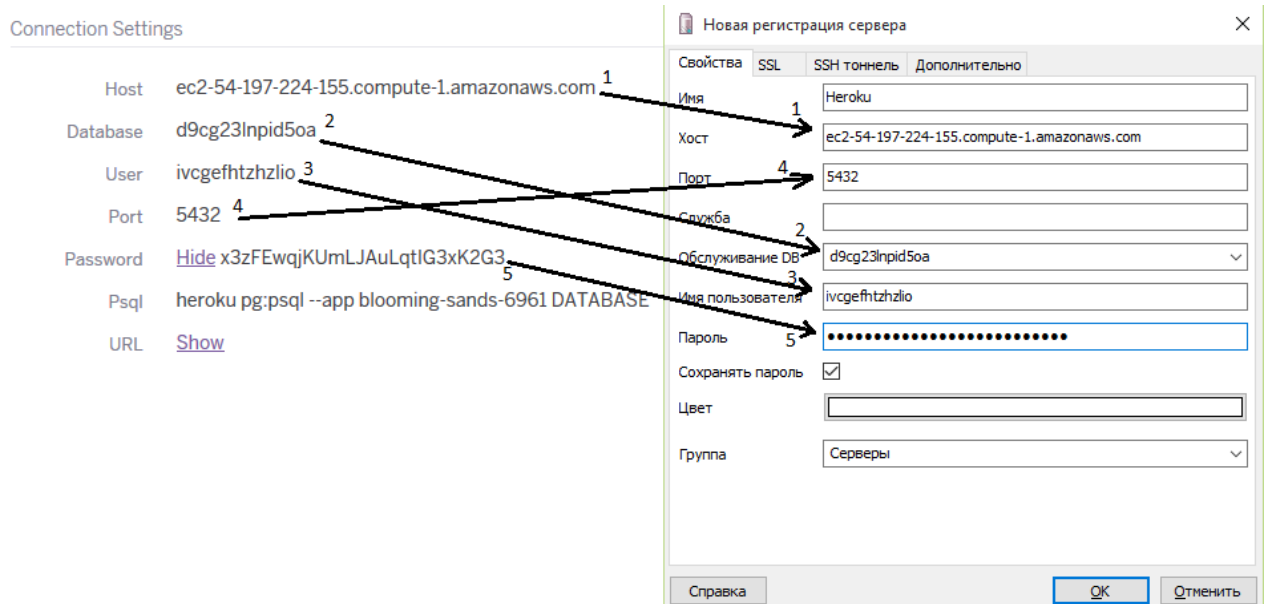


Рисунок 9 - перенос настроек, новая регистрация сервера.

После того как все настройки перенесены, нужно всего лишь нажать кнопку “Ок” и Heroku появится в списке серверов, расположенном в браузере объектов (Рисунок 10).

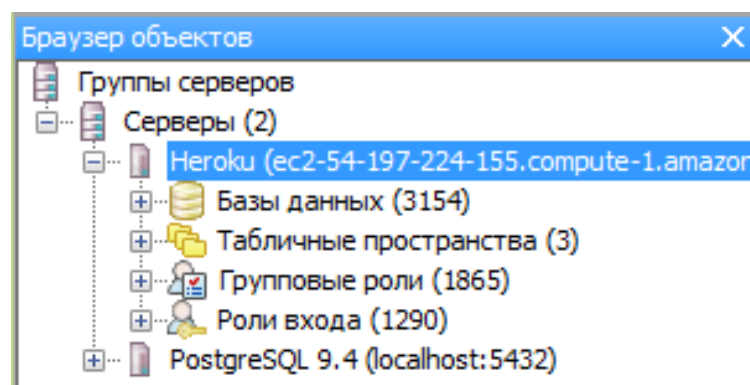


Рисунок 10 - браузер объектов, отображается сервер Heroku

На рисунке 10 в разделе “Базы данных” мы видим удивительно много объектов. Откуда они? На самом деле нам принадлежит, конечно, только одна база данных, которую мы создавали, а остальные - это объекты Heroku.

Найдем свою БД по названию, которое мы указывали на рисунке 9 под пунктом 2. На этом настройка PostgreSQL и подключение к БД закончены. Следующим шагом создадим таблицы в

этой базе данных и напишем для работы с ней серверную часть приложения.

Создание таблиц в БД

В качестве предметной области для создания таблиц в базе данных выберем телефонный справочник. Это интересный и простой для понимания пример работы с БД и поэтому его часто используют даже для демонстрации работ со сложными соединениями и нагрузками.

В самом простом случае, который мы и возьмём за основу, будет две таблицы: люди (которым принадлежит какой-то номер) и номер телефона. Таблицы будут соединены связью один ко многим (1:M), так как одному человеку может принадлежать несколько номеров, но в свою очередь за одним номером закрепляется только один человек. На рисунке 11 представлена модель данных.



Рисунок 11 - модель данных телефонного справочника

Для того, что бы создать таблицы, изображенные на рисунке 11, необходимо в pgAdmin в браузере объектов найти свою базу данных и нажать слева от её названия на плюсик, затем таким же способом раскрыть объекты Схема и public, в результате чего мы увидим много возможностей, которые можно проводить с выбранной схемой. Нас конечно же интересует пункт Таблицы и возможность создания новой, для этого правой кнопкой мыши вызываем контекстное меню пункта “Таблицы”, затем “Новый объект” - “Новая таблица” (рисунок 12).

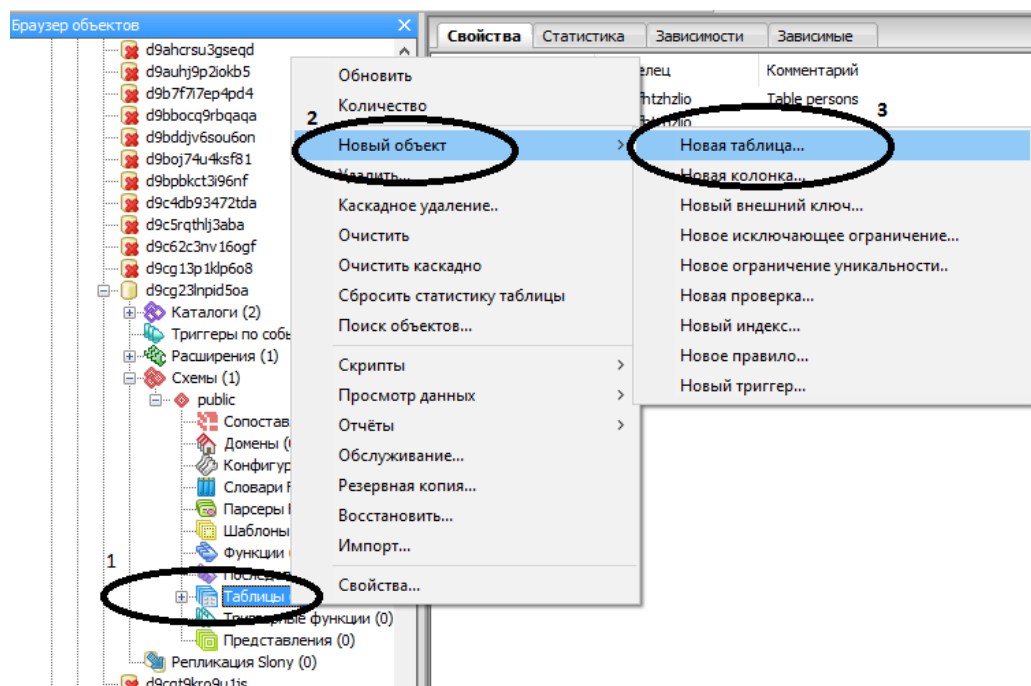


Рисунок 12 - выбор пункта “Новая таблица”

В результате выполненных действий откроется окно “Новая таблица”. Что бы было видно все верхние вкладки данное окно необходимо расширить, либо переключать их специальными стрелочками, расположенными в правом верхнем углу. Заполним важные для нас поля на примере создания таблицы PERSONS. На рисунке 13 отображена вкладка “Свойства” и написано имя таблицы. Так же черными овалами выделены пункты которые нас больше остальных интересуют.

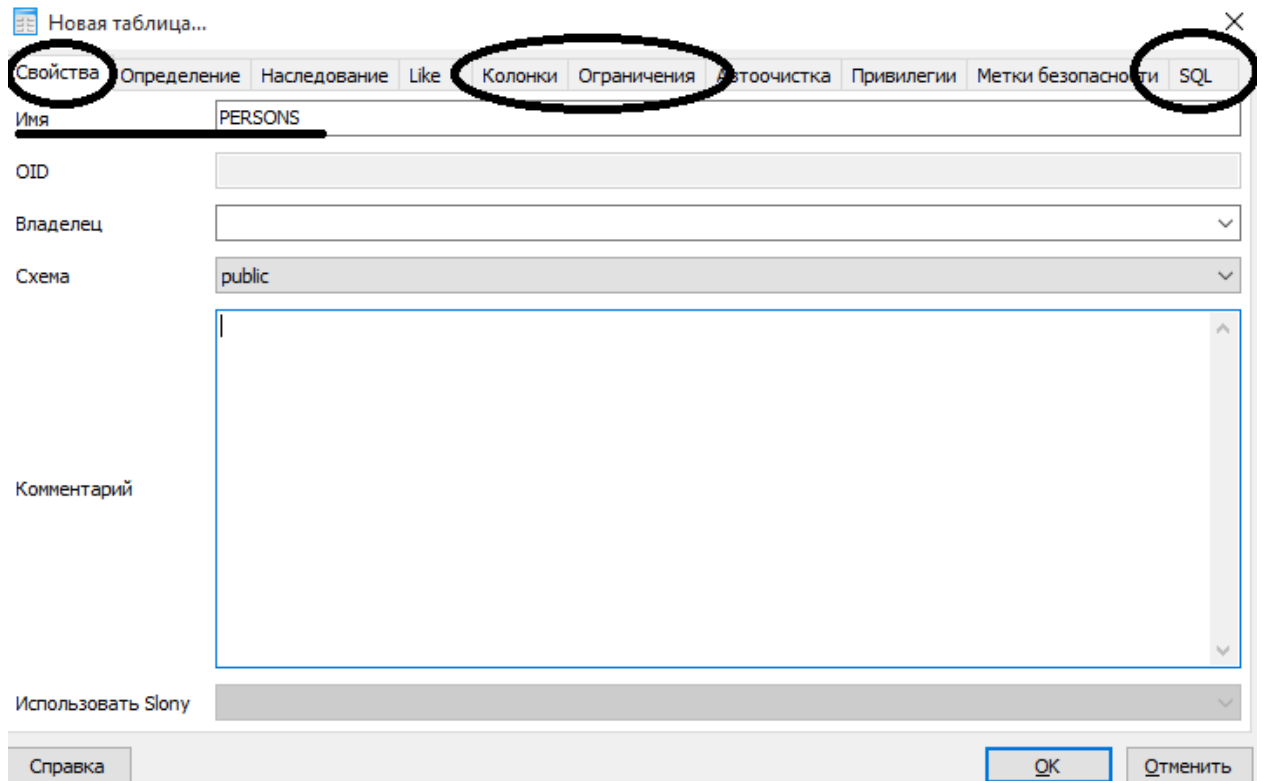


Рисунок 13 - окно “Новая таблица”, вкладка “Свойства”

Если перейти во вкладку “Колонки” и в правом нижнем углу нажать на кнопку “Добавить”, то в результате откроется окно “Новая колонка” для добавления полей в таблицу (Рисунок 14). Опять же, самыми интересными для нас являются два поля: “Имя” и “Тип данных”, а так же две вкладки: “Свойства” и “Определение”. На вкладке “Определение” можно поставить галочку на разрешение Null-значения, либо задать значение по умолчанию. После нажатия на кнопку “ОК”, произойдёт закрытие данного окна и мы снова окажемся во вкладке “Колонки”, в которой уже будет присутствовать только что добавленное нами поле.

Далее переходим во вкладку “Ограничения”. Она нам полезна, потому что позволяет создавать первичные и внешние ключи. Разберём их создание на примере установки поля ID таблицы PERSONS первичным ключом. Выбираем в нижнем выпадающем списке “Первичный ключ” и нажимаем кнопку “Добавить” откроется окно “Новый первичный ключ”. Задаем ему имя, например “ID_PERSONS” и переходим во вкладку “Колонки”, затем в выпадающем списке выбираем наше поле “ID” и нажимаем “Добавить”, а потом на кнопку “ОК”. В результате данных манипуляций поле “ID” станет первичным ключом.

Последняя вкладка SQL позволяет просмотреть получающийся скрипт.

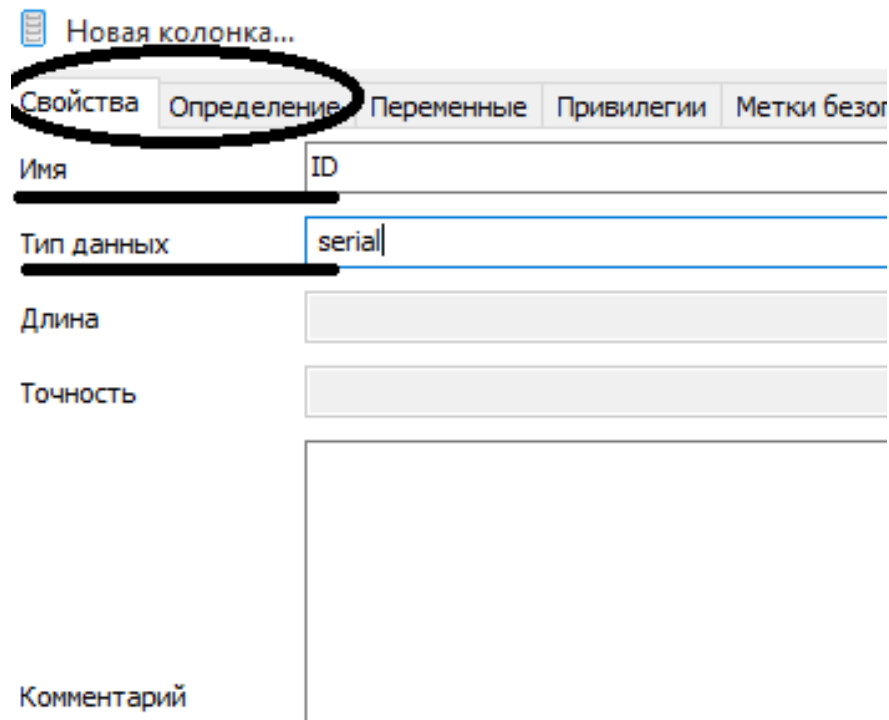


Рисунок 14 - окно "Новая колонка"

Обратите внимание, что у идентификатора устанавливается тип данных "serial" для его автоматической инкрементации.

Итогом создания двух таблиц должна получиться структура в браузере объектов, изображенная на рисунке 15.

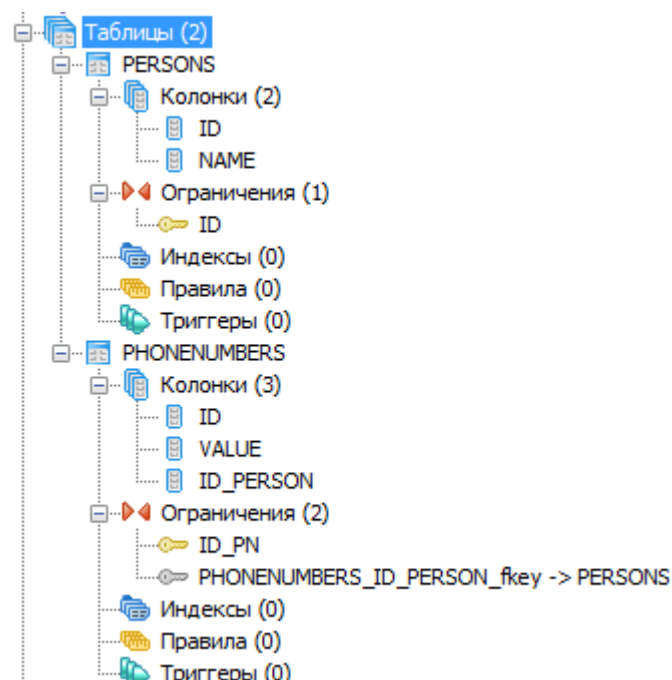


Рисунок 15 - финальное дерево объектов с созданными таблицами и ограничениями

Командная строка PostgreSQL

Тот же результат по созданию БД и таблиц предыдущего раздела можно получить с помощью командной строки PostgreSQL. Для многих, кто хорошо владеет SQL,- это более удобный способ, потому что не требует установки дополнительного приложения. Но и для начинающих разработчиков это полезно, к примеру, в случае создания пустой копии существующей уже БД, потому что необходимо просто запустить уже сгенеренный SQL скрипт.

Для того, что бы создать таблицы в базе данных heroku через командную строку, необходимо к ней подключиться. Для этого в консоли пишем команду:

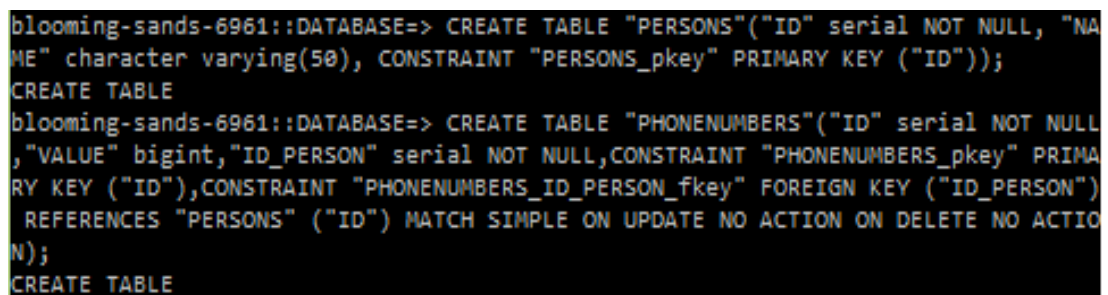
```
heroku pg:psql --app НАЗВАНИЕ_БАЗЫ_ДААННЫХ DATABASE
```

Данную команду можно скопировать из веб-интерфейса Heroku Postgres. Она доступна при просмотре подробных сведений о базе данных в строке с название "psql". При выполнении команды осуществляется переход в раздел управления базой данных. Теперь нужно всего лишь запустить sql-скрипты на выполнение. SQL-скрипты представлены ниже.

```
CREATE TABLE "PERSONS"("ID" serial NOT NULL, "NAME" character varying(50), CONSTRAINT "PERSONS_pkey" PRIMARY KEY ("ID"));

CREATE TABLE "PHONENUMBERS"("ID" serial NOT NULL,"VALUE" bigint,"ID_PERSON" serial NOT NULL,CONSTRAINT "PHONENUMBERS_pkey" PRIMARY KEY ("ID"),CONSTRAINT "PHONENUMBERS_ID_PERSON_fkey" FOREIGN KEY ("ID_PERSON") REFERENCES "PERSONS" ("ID") MATCH SIMPLE ON UPDATE NO ACTION ON DELETE NO ACTION);
```

При успешном исполнении sql-скрипта на создание таблицы в консоли выводится надпись "CREATE TABLE". Пример результата представлен ниже на рисунке 16.



```
blooming-sands-6961::DATABASE=> CREATE TABLE "PERSONS"("ID" serial NOT NULL, "NAME" character varying(50), CONSTRAINT "PERSONS_pkey" PRIMARY KEY ("ID"));
CREATE TABLE
blooming-sands-6961::DATABASE=> CREATE TABLE "PHONENUMBERS"("ID" serial NOT NULL, "VALUE" bigint, "ID_PERSON" serial NOT NULL, CONSTRAINT "PHONENUMBERS_pkey" PRIMARY KEY ("ID"), CONSTRAINT "PHONENUMBERS_ID_PERSON_fkey" FOREIGN KEY ("ID_PERSON") REFERENCES "PERSONS" ("ID") MATCH SIMPLE ON UPDATE NO ACTION ON DELETE NO ACTION);
CREATE TABLE
```

Рисунок 16 - исполнение sql-скриптов в командной строке

5.5.3 Реализация back end части приложения на языке Java

Создадим проект Spring Starter Project для написания back end функционала. Для этого выполните следующие действия: **File** ⇒ **New** ⇒ **Project**. Затем в списке выбираем раздел Spring и далее пункт Spring Starter Project. Перед вами появится окно создания нового стартового проекта Spring. Заполним необходимые поля (рисунок 18) и нажмём кнопку Finish:

- Name (название проекта) - PhoneBook;
- Type (система сборки проекта) - Maven Project;

- Java Version - 1.8;
- Boot Version (версия фреймворка Spring Boot) - 1.3.1;
- Packaging - jar;
- Language - Java;
- Group (группа, команда, можно указать сайт разработчика) - ru.myitschool;
- Artifact (задаем такой же, как название приложения) - PhoneBook;
- Version - 0.0.1-SNAPSHOT;
- Description - Phone book project for Spring Boot;
- Package - ru.myitschool;
- Dependencies - Web.

Завершая создание проекта кнопкой Finish, в Package Explorer появится наш проект (рисунок 19).

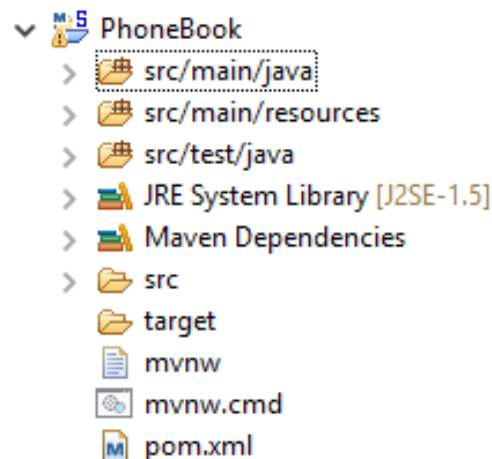


Рисунок 19 - структура проекта PhoneBook

Теперь необходимо убедиться в том, что проект корректно запускается, но если мы это сделаем прямо сейчас, то увидим скучную 404 страничку в браузере. Поэтому раскроем самую первую ветку (src/main/java), затем наш пакет (ru.myitschool) и создадим в нём класс HomeController. Напишем метод index, возвращающий строковое значение "Привет, мир!". Ниже приведён код.

```
package ru.myitschool;

import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
public class HomeController {

    @RequestMapping("/")
    public String index() {
        return "Привет, мир!";
    }
}
```

Нажимаем правой кнопкой мыши по названию проект, **Run As ⇒ Spring Boot App**. В консоль выводится процесс запуска приложения. Если всё прошло успешно, в последней строке в консоли

будет написана фраза “Started PhoneBookApplication” и указано время за которое это случилось. Открываем любимый браузер и в адресной строке вводим <http://localhost:8080/>. В результате чего должна загрузиться страница с надписью “Привет, мир”. Даже в таком простом примере, с единственным методом вывода строкового значения, есть интересные особенности фреймворка Spring, а именно аннотации (`@RestController`, `@RequestMapping("/")`). Они помогают простым и удобным способом обозначить назначение какой-либо структуры. Например, аннотация “`@RestController`” говорит о том, что класс `HomeController` является контроллером, а “`@RequestMapping("/")`” позволяет задавать пути запросов. Подробнее об аннотациях в Spring можно прочитать в источниках [\[1\]](#) и [\[2\]](#).

Последним подготовительным шагом, для успешной дальнейшей работы, является добавление дополнительных библиотек. Так как используется система сборки проекта Maven, сделать это достаточно просто, нужно лишь добавить необходимые зависимости в файле `pom.xml` (находится в корне проекта) внутри тега “`<dependencies>`”. Ниже указаны используемые зависимости.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
</dependency>
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>9.4-1203-jdbc42</version>
</dependency>
<dependency>
  <groupId>org.json</groupId>
  <artifactId>json</artifactId>
  <version>20151123</version>
</dependency>
```

Все подготовительные шаги уже выполнены и теперь можно приступить к написанию “боевых” функций сервера. Для этого мы создадим базовый CRUD (Create, Read, Update, Delete) функционал для наших сущностей базы данных, а именно `Persons` и `PhoneNumbers`. Посмотрим на те шаги, которые нам нужно выполнить для решения данной задачи:

1. Создать подключение к базе данных, располагающейся на сервисе heroku;
2. Представить сущности `Persons` и `PhoneNumbers` в виде java-классов;

3. Реализовать на их основе репозитории;
4. Написать CRUD контроллеры.

1. Создать подключение к базе данных, располагающейся на сервисе heroku.

Для этого раскроем ветку нашего проекта “src/main/resource” и откроем файл “application.properties”. Затем добавим следующие строки:

```
spring.datasource.url=jdbc:postgresql://host:port/database?sslmode=require
spring.datasource.username=
spring.datasource.password=
spring.datasource.driver-class-name=org.postgresql.Driver
```

В первой строке, указывая url (строку подключения), перенесём данные со страницы параметров базы данных, как это было сделано на рисунках 8 и 9, при подключении к серверу через pgAdmin. В результате получится строка подключения как на рисунке 20 (только с вашими настройками).

Рисунок 20 - строка подключения (spring.datasource.url)

Рисунок 20 - строка подключения (spring.datasource.url)

Перенесём данные об имени пользователя и пароле аналогичным образом. Особое внимание нужно уделить наличию параметра “sslmode=require”, так как при его отсутствии подключение к базе данных будет невозможно (это специфическая особенность Heroku, да и вообще многих других облачных сервисов). В последней строке указывается название драйвера базы данных.

2. Представить сущности Persons и PhoneNumbers в виде java-классов

Данные классы лучше расположить в отдельном пакете ru.myitschool.entity. Код сущностей представлен ниже, сначала Persons, затем PhoneNumbers.

```
package ru.myitschool.entity;

public class Persons {
    private Integer id;
    private String name;
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

```
package ru.myitschool.entity;

public class PhoneNumbers {
    private Integer id;
    private Long value;
    private Integer idPerson;
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public Long getValue() {
        return value;
    }
    public void setValue(Long value) {
        this.value = value;
    }
    public Integer getIdPerson() {
        return idPerson;
    }
    public void setIdPerson(Integer idPerson) {
        this.idPerson = idPerson;
    }
}
```

3. Реализовать репозитории для сущностей

В данном случае, для упрощения, мы не будем реализовывать паттерн репозиторий в привычном понимании. Создадим классы с методами позволяющими выполнять SQL-запросы к базе данных. Так же создадим объект класса `JdbcTemplate`, для того, что бы не прописывать ручным способом работу с соединением и источником данных. Код класса `PersonRepository` представлен ниже.

```
package ru.myitschool.repositories;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;
import ru.myitschool.entity.Persons;

@Component
public class PersonsRepository {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public int createPerson(String name){
        return jdbcTemplate.update("INSERT INTO \"PERSONS\" (\"NAME\") VALUES (?)", name);
    }

    public int updatePerson(Persons person){
        return jdbcTemplate.update("UPDATE \"PERSONS\" SET \"NAME\" = ? WHERE \"ID\" = ?", person.getName(), person.getId());
    }
}
```



```
}

public int deletePerson(Integer id){
    return jdbcTemplate.update("DELETE FROM \"PERSONS\" WHERE \"ID\" = ?",
        id);
}

public Persons getPerson(Integer id){
    return jdbcTemplate.queryForObject("SELECT * FROM \"PERSONS\" WHERE
        \"ID\"=?", new PersonsMapper(), id);
}

public List<Persons> getPersons(){
    return jdbcTemplate.query("SELECT * FROM \"PERSONS\"", new
        PersonsMapper());
}
}
```

В методах `createPerson`, `updatePerson` и `deletePerson` нет ничего примечательно. У объекта `jdbcTemplate` просто вызывается метод `update()`, принимающий в качестве параметров SQL-запрос и переменные для этого запроса, значение которых подставляется вместо вопросительного знака. Наибольший интерес вызывают методы получения одной персоны и списка персон. В первом случае, для выполнения запроса используется метод `queryForObject()`, во втором `query()`. В отличие от метода `update`, примечательно в них то, что они принимают еще один дополнительный объект типа `RowMapper<T>`, функцией которого является преобразование ответа от базы данных в требуемый для разработчика вид. На самом деле `RowMapper<T>` это интерфейс, по этому для того, что бы создать на его основе объект, сначала создадим класс, который его имплементирует. Код класса `PersonsMapper` представлен ниже.

```
package ru.myitschool.repositories;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;
import ru.myitschool.entity.Persons;

public class PersonsMapper implements RowMapper<Persons> {

    public Persons mapRow(ResultSet rs, int rowNum) throws SQLException {
        Persons person = new Persons();
        person.setId(rs.getInt("id"));
        person.setName(rs.getString("name"));
        return person;
    }
}
```

В результате имплементации интерфейса `RowMapper<T>`, необходимо реализовать метод `mapRow()`. У него имеется два параметра: `ResultSet` (результатирующий набор, сюда приходит результат выполнения SQL-запроса) и целочисленная переменная `rowNum` (хранит номер строки).

Создаём объект класса Persons и приваиваем значения его атрибутам.

Рассмотрим класс PhoneNumberRepository. У него есть интересный метод getPhoneBook(), позволяющий получить все записи (люди и их телефонные номера) из базы данных и преобразовать в вид json-строки. Для этого сначала создаёт два json объекта. Объект класса JSONObject позволит поместить каждую конкретную запись, а JSONArray запакует всё в массив. Далее создадим объекты классов Connection и Statement, напомним SQL-запрос и выполним его при помощи метода executeQuery объекта stmt, результат которого присвоим объекту класса ResultSet. При помощи цикла while делается обход всего ответа и методами put() он запаковывается в json. Ниже приведён код данного класса.

```
package ru.myitschool.repositories;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;
import ru.myitschool.entity.PhoneNumbers;

@Component
public class PhoneNumbersRepository {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public int createPhoneNumber(Long value, Integer idPerson){
        return jdbcTemplate.update("INSERT INTO \"PHONENUMBERS\" (\"VALUE\", \"ID_PERSON\") VALUES (?,?)", value, idPerson);
    }

    public int updatePhoneNumber(PhoneNumbers phoneNumbers){
        return jdbcTemplate.update("UPDATE \"PHONENUMBERS\" SET \"VALUE\" = ? WHERE \"ID\" = ?", phoneNumbers.getValue(), phoneNumbers.getId());
    }

    public int deletePhoneNumbers(Integer id){
        return jdbcTemplate.update("DELETE FROM \"PHONENUMBERS\" WHERE \"ID\" = ?", id);
    }

    public JSONArray getPhoneBook(){
        JSONObject json;
        JSONArray jsonArr = new JSONArray();
        try {
```

```

        Connection conn=jdbcTemplate.getDataSource().getConnection();
        Statement stmt = conn.createStatement();
        String sql = "SELECT  \"PHONENUMBERS\".\"ID\" AS \"ID\",
        \"PHONENUMBERS\".\"VALUE\" AS \"NUMBER\", \"PERSONS\".\"NAME\"
        AS \"NAMEPERSON\" FROM \"PHONENUMBERS\" LEFT JOIN \"PERSONS\"
        ON \"PERSONS\".\"ID\" = \"PHONENUMBERS\".\"ID_PERSON\";";
        ResultSet rs = stmt.executeQuery(sql);
        while(rs.next()){
            json = new JSONObject();
            json.put("name", rs.getString("NAMEPERSON"));
            json.put("value", rs.getString("NUMBER"));
            jsonArr.put(json);
        }
    } catch (SQLException e) {
        e.getLocalizedMessage();
        return null;
    } catch (JSONException e) {
        e.getLocalizedMessage();
        return null;
    }
    return jsonArr;
}
}

```

4 Написать CRUD контроллеры

Создадим еще один пакет с названием ru.myitschool.controllers и два класса: PersonsController и PhoneNumberController. Код обоих классов представлен ниже.

```

package ru.myitschool.controllers;

import java.util.List;
import org.json.JSONException;
import org.json.JSONObject;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import ru.myitschool.entity.Persons;
import ru.myitschool.repositories.PersonsRepository;

@RestController
@RequestMapping("person")
public class PersonsController {
    @Autowired
    private PersonsRepository person;
}

```

```
@RequestMapping(value = "/create", method=RequestMethod.PUT,
consumes="text/plain")
public int createPerson(@RequestBody String param){
    String name = null;
    try{
        JSONObject json = new JSONObject(param);
        name = json.getString("name");
    }catch(JSONException e){
        e.getLocalizedMessage();
        return 0;
    }
    return person.createPerson(name);
}

@RequestMapping(value = "update", method=RequestMethod.POST,
consumes="text/plain")
public int updatePerson(@RequestBody String param){
    Persons p = new Persons();
    try{
        JSONObject json = new JSONObject(param);
        p.setId(json.getInt("id"));
        p.setName(json.getString("name"));
    }catch(JSONException e){
        e.getLocalizedMessage();
        return 0;
    }
    return person.updatePerson(p);
}

@RequestMapping(value="{id}", method=RequestMethod.DELETE)
public int deletePerson(@PathVariable Integer id){
    return person.deletePerson(id);
}

@RequestMapping(value = "/getperson", method=RequestMethod.GET)
public Persons getPerson(@RequestParam("id") Integer id){
    return person.getPerson(id);
}

@RequestMapping(value = "/getpersons", method=RequestMethod.GET)
public List<Persons> getPersons(){
    return person.getPersons();
}
}
```

Код класса PhoneNumbersController:

```
package ru.myitschool.controllers;

import org.json.JSONException;
import org.json.JSONObject;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import ru.myitschool.entity.PhoneNumbers;
import ru.myitschool.repositories.PhoneNumbersRepository;

@RestController
@RequestMapping("pn")
public class PhoneNumberController {

    @Autowired
    private PhoneNumbersRepository phoneNumbersRepository;

    @RequestMapping(value = "/create", method=RequestMethod.PUT,
consumes="text/plain")
    public int createPhoneNumber(@RequestBody String param){
        Long value = null;
        Integer id = null;
        try{
            JSONObject json = new JSONObject(param);
            value = json.getLong("value");
            id = json.getInt("idperson");
        }catch(JSONException e){
            e.getLocalizedMessage();
            return 0;
        }
        return phoneNumbersRepository.createPhoneNumber(value, id);
    }

    @RequestMapping(value = "update", method=RequestMethod.POST,
consumes="text/plain")
    public int updatePhoneNumber(@RequestBody String param){
        PhoneNumbers pn = new PhoneNumbers();
        try{
            JSONObject json = new JSONObject(param);
            pn.setId(json.getInt("id"));
            pn.setValue(json.getLong("value"));
            pn.setIdPerson(json.getInt("idperson"));
        }catch(JSONException e){
            e.getLocalizedMessage();
            return 0;
        }
        return phoneNumbersRepository.updatePhoneNumber(pn);
    }

    @RequestMapping(value="{id}", method=RequestMethod.DELETE)
    public int deletePhoneNumber(@PathVariable Integer id){
        return phoneNumbersRepository.deletePhoneNumbers(id);
    }
}
```

```
    }

    @RequestMapping(value = "/getpb", method=RequestMethod.GET)
    public String getItemPhoneBook(){
        return phoneNumbersRepository.getPhoneBook().toString();
    }
}
```

Так как оба класса похожи друг на друга, рассмотрим последний из них. Для класса `PhoneNumbersController` указано две аннотации. `@RestController` указывает на то, что данный класс является контроллером по архитектуре REST. `@RequestMapping("pn")`, означает, что для доступа к методам данного класса URL адрес будет иметь вид: <http://localhost:8080/pn/>. Затем идёт определение объекта репозитория, на базе созданного ранее класса. У данного объекта имеется аннотация `@Autowired`, позволяющая автоматически устанавливать значение поля. Далее определены четыре метода-контроллера. У каждого из них присутствует аннотация `@RequestMapping` позволяющая указать: путь (по которому данный метод будет доступен), тип запроса по архитектуре REST (GET,POST,PUT,DELETE и т.д.). У двух методов указан еще один параметр для данной аннотации - `consumes`, со значением `"text/plain"`, сигнализирующий о том, что входящим параметром является простая строка (текст).

Суть написанных методов достаточно проста. Они должны принять какую-либо информацию через параметры, возможно немного её подготовить а затем вызвать метод(ы), например репозитория, что бы получить от них ответ и отправить его клиенту. Если обратить внимание на параметры данных методов, то можно увидеть, что там так же используются аннотации:

- `@RequestBody` - информация передаётся в теле запроса.
- `@PathVariable` - значение является частью пути (<http://localhost:8080/pn/8>)
- `@RequestParam` - передача значения через переменную пути (<http://localhost:8080/pn/get?id=8>)

Использование аннотаций - одно из условий, которое ставит SpringBoot для обеспечения простоты написания кода, на основе данного фреймворка. В качестве дополнительного материала рекомендуется почитать статьи [на официальном сайте данного фреймворка \[3\]](#). А [здесь \[4\]](#) можно посмотреть как писать на SpringBoot без использования строк запросов на голем SQL.

*5.5.4 Реализация back end части приложения на языке PHP

В данном разделе рассмотрим как реализовать серверную часть приложения, выполненную ранее на языке Java, при помощи скриптового языка программирования PHP.

Для написания кода серверной части есть несколько устоявшихся подходов, один из которых использовался в предыдущем разделе, называется MVC – Model View Controller. Этот подход служит для разделения кода работы с базой данных от кода непосредственно серверной части и от интерфейса пользователя. Согласно этому подходу весь код, реализующий CRUD-функционал размещается в моделях, код необходимый для организации работы сервера размещается в контроллерах и соответственно интерфейс размещается в представлениях. Модель – это класс, представляющий собой описание сущности базы данных (например, таблица). В нашем случае

будет две модели – персоны и телефонные номера. Также добавим еще одну служебную модель – dbConfig в которой будем описывать методы для работы непосредственно с базой данных (вызов функций, подключение к базе и т.д.).

Первая модель будет называться «persons», она описывает сущность «Персоны». Приведем часть кода этой модели:

```
include "dbFunc.php";
class persons {
    var $id;
    var $name;
    var $result;
    var $objDb;

    function persons() {
        //конструктор
        $this->objDb = new dbConfig();
        if($this->objDb->getResult()!=1) {
            echo "Database connection error.";
            exit;
        }
    }
}
```

Здесь первой строкой мы подключаем служебную модель dbConfig, которая находится в файле dbFunc.php. Для подключения сторонних файлов в скрипт в PHP есть функция «include». Далее мы объявляем класс persons и объявляем его атрибуты. Далее описываем конструктор класса в котором создаем подключение к базе данных и сообщаем об ошибке в случае неудачного соединения с базой.

Также этот класс имеет методы для создания, изменения, удаления и чтения данных. Все эти методы имеют одинаковую структуру и отличаются только названием и кодом для работы с базой. Приведем для примера код метода добавления новой записи в таблицу.

```
function newPersons($name) {
    //метод для добавления новой персоны
    $textQuery = "INSERT INTO \"PERSONS\" (\"NAME\") VALUES ('".$name."')";
    $this->objDb->executeQuery($textQuery);
    $this->result = $this->objDb->getResult();
}
```

Этот метод принимает переменную \$name в которой содержится имя добавляемой персоны. Далее в переменную \$textQuery мы записываем запрос на языке SQL для вставки данных в таблицу. Потом этот запрос мы передаем в функцию executeQuery нашей служебной модели dbConfig. Именно эта функция посылает запрос в базу и возвращает результат. Результат операции мы получаем с помощью функции getResult() нашей служебной модели. Таким образом, меняя код запроса можно реализовать весь CRUD функционал.

Модель «phoneNumber» выглядит точно также как и модель «persons», только с атрибутами для сущности Персоны. Интересной для рассмотрения является модель dbConfig. Рассмотрим ее подробнее и приведем весь ее код.

```
<?php
class dbConfig {
//класс-конфиг для работы с базой данных
    var $result;
    var $db;

    function dbConfig() {
        $this->result=0;
        $connectionString="host=ec2-54-217-231-152.eu-west-.compute.amazonaws.com
                                port=5432
                                dbname=d80fg54admdmct
                                user=usyzfnmscfzbm
                                password=gtRV3ci01pbD_Zqqjlc-I6FrRe
                                sslmode=require";

        # Установка соединения с базой данных
        $this->db = pg_connect($connectionString);
        if (!$this->db) {
            $this->result=0;
            exit;
        }
        else {
            $this->result=1;
        }
    }
    function executeQuery($textQuery) {
        $resultQuery = pg_query($this->db, $textQuery);
        if (!$resultQuery) {
            $this->result=0;
            exit;
        }
        else {
            $this->result=1;
            return $resultQuery;
        }
    }
    function getResult() {
        return $this->result;
    }
}
?>
```

Все стандартно начинается с объявления класса dbConfig, далее описываются атрибуты и конструктор. В конструкторе задается строка подключения к базе данных, расположенной в сервисе Heroku и производится попытка соединения с ней. Функция executeQuery принимает текст SQL-запроса и с помощью функции pg_query, которая работает с базой данных отправляет этот запрос на сервер.

На этом рассмотрение моделей можно закончить и перейти к контроллерам. Контроллеры в нашем серверном приложении будут являться чем-то вроде шлюзов, их задачей будет принять запрос от клиента и отправить его модели, после получения ответа от модели вернуть его клиенту. У нас присутствует две сущности – Персоны и Телефонные номера, значит необходимо сделать

два контроллера. Их структура абсолютно одинакова, различаются только имена переменных и функций. Приведем начало контроллера для работы с сущностью «Персоны».

```
<?php
include "personsModel.php";
$action = $_GET["action"];
$objPerson = new persons();
switch($action) {
    case 'get':
        $masPerson = $objPerson->listPersons();
        echo "List data: ".json_encode($masPerson);
        break;
    case 'getDetails':
        $itemPerson = $objPerson->getPerson($_GET["id"]);
        echo "Current item: ".json_encode($itemPerson);
        break;
    case 'post':
        $objPerson->newPersons($_POST["json"]);
        echo ($objPerson->getResult()!=1)?"Error":"Success";
        break;
}
}??>
```

Сначала мы подключаем файл с моделью persons уже известной нам командой include. Далее мы считываем из служебной переменной \$_GET с указанием ключа, идентификатор действия, которое контроллер должен выполнить. Стоит привести пример запроса, который будет отправляться на сервер: ~/personController.php?action=ДЕЙСТВИЕ. Здесь как мы видим в контроллер передается переменная action с определенным значением. Согласно правилам построения клиент-серверных приложений определено 4 типа действий:

- post – создание объекта;
- put – изменение объекта;
- delete – удаление объекта;
- get – получение объекта.

Эти 4 действия позволяют организовать так называемый API. Однако, язык PHP в силу своих особенностей не позволяет в полной мере реализовать API, поэтому мы добавим некоторые свои типы действий, например действий getDetails для получения сведений о конкретном объекте. После получения типа действия мы запускаем условный оператор switch в котором проверяем значение переменной \$action и выполняем требуемое действие. Таким образом реализуя контроллеры для наших сущностей мы добьемся выполнения запланированных функций по работе с базой данных и изменения данных в ней.

5.5.5. Практикум

Задание

Создать клиент-серверное приложение, которое позволяет:

- авторизоваться через OAuth-авторизацию социальной сети ВКонтакте
- сохранять информацию о пользователе и его местоположении в БД на сервере (повторите это дома, чтобы более наглядно увидеть расположение на карте города)
- отображать информацию о других пользователях.

- отображать местоположение всех пользователей на карте.

Реализация. Серверная часть

В классической схеме при разработке клиент-серверных приложений принято начинать с модели данных, затем построении самой БД, серверной части и клиентской. Однако, современные средства позволяют совместить некоторые этапы и ускорить процесс разработки. В частности, используя сильные стороны Spring Boot, можно объединить создание модели, базы и сервера. Рассмотрим как это сделать, но для начала, определимся какие данные нам нужно хранить. В самом простом случае, не используя лучшие практики, нам достаточно одной таблицы Users со следующими полями:

- id - идентификатор записи, первичный ключ;
- vkUserId - идентификатор пользователя социальной сети Вконтакте;
- vkUserName - текстовое поле для хранения имени и фамилии пользователя, получаемой из социальной сети Вконтакте;
- latitude - целочисленное поле для хранения широты (последнее местоположения пользователя);
- longitude - целочисленное поле для хранения долготы (последнее местоположения пользователя);

Если бы использовалась, упомянутая выше, классическая схема разработки, следующим шагом необходимо было бы зайти в pgAdmin, подключиться к серверу Heroku и создать базу данных со всеми таблицами и полями. Но можно пойти по более современному пути и начать сразу разрабатывать сервер, описав в нём необходимые сущности.

Создадим начальный проект Spring Boot с названием “socialmaps” и пакетом “ru.myitschool”, так же, как это делалось в разделе 5.5.3. Добавим новый пакет “ru.myitschool.entity” для размещения в нём класса, описывающего сущность Users. И соответственно создадим сам класс. Код данного класса представлен ниже. Обратите внимание на последний комментарий! Для экономии места настоящего материала, get и set методы для свойств класса необходимо создать самостоятельно.

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "users")
public class Users{

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;

    @Column(nullable = false)
    private long vkUserId;
```

```
@Column(nullable = false)
private String vkUserName;

@Column(nullable = false)
private double latitude;

@Column(nullable = false)
private double longitude;

public Users() {}

public Users(long vkUserId, String vkUserName, double latitude, double
longitude) {
    super();
    this.vkUserId = vkUserId;
    this.vkUserName = vkUserName;
    this.latitude = latitude;
    this.longitude = longitude;
}

/*Generate Getters&Setters methods*/
}
```

Используя аннотации, данный класс становится средством создания указанной таблицы в базе данных на сервере Heroku, а не просто представлением сущности на сервере. Правда для этого необходимо добавить несколько строк в файл “application.properties”, и разместить их ниже описания подключения к базе данных heroku. Конечный вариант файла “application.properties” представлен ниже.

```
spring.datasource.url={Указывается строка подключения к БД}
spring.datasource.username={Указывается логин для подключения к БД}
spring.datasource.password={Указывается пароль для подключения к БД}
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.maxActive=10
spring.datasource.maxIdle=5
spring.datasource.minIdle=2
spring.datasource.initialSize=5
spring.datasource.removeAbandoned=true

spring.jpa.show-sql = true
spring.jpa.hibernate.ddl-auto = update
spring.jpa.hibernate.naming-strategy=org.hibernate.cfg.ImprovedNamingStrategy
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

Теперь, при совершении первого запроса к базе данных, таблица Users создастся автоматически, со всеми указанными полями. Соответственно возникает необходимость в создании самих запросов. В разделе 5.5.3 мы использовали подход с написанием “голых” sql-запросов. В текущем практикуме будет использоваться иной подход. Создадим репозиторий “UsersRepository” в пакете

“ru.myitschool.dao”, интерфейс наследуемый от `CrudRepository<T,ID extends Serializable>`. Код представлен ниже.

```
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import ru.myitschool.entity.Users;

@Repository
@Transactional
public interface UsersRepository extends CrudRepository<Users, Long> {

}
```

Подробнее об этом интерфейсе можно прочитать [здесь \[8\]](#). Суть такого репозитория в том, что он предоставляет доступ к методам, реализующим основные CRUD-операции. В добавок к этому, есть возможность описать своим собственные методы, но естественно без реализации. Spring сам, на основании указанного имени метода, поймёт какой запрос нужно сформировать.

Остаётся только написать контроллеры и вызвать правильные методы. Для этого создадим класс “UsersController” с аннотацией “@Controller” и в нём два метода-контроллера для сохранения данных о пользователе и получения списка всех пользователей. Код представлен ниже.

```
import org.json.JSONObject;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import ru.myitschool.dao.UsersRepository;
import ru.myitschool.entity.Users;

@Controller
public class UsersController {

    @Autowired
    private UsersRepository userRepository;

    @RequestMapping(value="/save", method=RequestMethod.PUT,
        consumes="text/plain")
    @ResponseBody
    public String save(@RequestBody String param){
        try{
            JSONObject json = new JSONObject(param);
            Users user = new Users(json.getLong("vkUserId"),
                json.getString("vkUserName"), json.getDouble("latitude"),
                json.getDouble("longitude"));
            userRepository.save(user);
        }catch(Exception e){

}
```

```
        return "Error: "+e.getLocalizedMessage();
    }
    return "1";
}

@RequestMapping(value="/getall", method=RequestMethod.GET)
@ResponseBody
public Iterable<Users> getUserList(){
    return userRepository.findAll();
}
}
```

Если посмотреть внимательно на код, то можно увидеть вызов двух методов репозитория: `save()` и `findAll()`. Вот так просто, не написав самостоятельно никаких SQL-команд, можно работать с базой данных в Spring.

Для того, что бы дальше можно было спокойно и удобно разрабатывать клиентскую часть, необходимо загрузить серверный код на платформу Heroku, как это сделать описано в [официальной документации \[9\]](#).

Реализация. Клиентская часть

Для реализации клиентской части создадим Android-проект с одной главной активностью "MainActivity". Используемый шаблон - "Empty Activity". Первым делом необходимо написать код для авторизации через социальную сеть Вконтакте по протоколу OAuth. Механизм авторизации уже известен из предыдущей темы 5.4, однако в данном примере он будет выглядеть несколько проще. Код файла интерфейса `activity_main.xml` представлен ниже.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <WebView
        android:id="@+id/wvAuth"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

Интерфейс содержит достаточно простую разметку: базовый слой `LinearLayout` с вертикальной ориентацией и компонент `WebView`. Далее необходимо создать новый java-класс "VKWebViewClient" унаследованный `WebViewClient`. Но перед этим, хорошим тоном будет добавить класс "Users", точно такой же, как был сделан на сервере, только без аннотаций. "VKWebViewClient" будет отвечать за авторизацию и получение имени пользователя из социальной сети Вконтакте. Листинг класса "VKWebViewClient" представлен ниже.

```
import java.io.BufferedReader;
import java.io.IOException;
```

```

import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;
import android.content.Context;
import android.os.AsyncTask;
import android.util.Log;
import android.webkit.WebView;
import android.webkit.WebViewClient;
import android.widget.Toast;

public class VKWebViewClient extends WebViewClient {

    Context ctx;
    private OnChangeVkUserIdListener vkUserIdListener;
    private Users user;

    public VKWebViewClient(Context ctx, OnChangeVkUserIdListener
vkUserIdListener){
        this.ctx = ctx;
        this.vkUserIdListener = vkUserIdListener;
        user = new Users();
    }

    @Override
    public void onPageFinished(WebView view, String url) {
        super.onPageFinished(view, url);
        if(url.contains("oauth.vk.com/blank.html")&url.contains("error")){
            Toast.makeText(ctx, "Произошла ошибка в процессе авторизации!",
Toast.LENGTH_LONG).show();
        }else{
            String urlRequest = url.substring(url.indexOf("#")+1);
            String[] massUrlRequestParam = urlRequest.split("&");
            String vkUserId = massUrlRequestParam[2];

            user.setVkUserId(Long.valueOf(vkUserId.substring(vkUserId.indexOf("=")+1)));
            Toast.makeText(ctx, "Вход через Вконтакте выполнен успешно!",
Toast.LENGTH_LONG).show();
            new
HttpInfoUserTask().execute("https://api.vk.com/method/users.get?user_id="+String.va
lueOf(user.getVkUserId())+"&v=5.45");
        }
    }

    private class HttpInfoUserTask extends AsyncTask<String, Void, String>{

        @Override

```



```

        protected String doInBackground(String... params) {
            InputStream inputStream;
            HttpURLConnection urlConnection;
            String result = "";
            try {
                URL url = new URL(params[0]);
                urlConnection = (HttpURLConnection) url.openConnection();
                urlConnection.setRequestMethod("GET");
                urlConnection.connect();
                inputStream = new BufferedInputStream(urlConnection.getInputStream());
                String response = convertInputStreamToString(inputStream);
                urlConnection.disconnect();
                result = response;
            } catch (MalformedURLException e) {
                Log.e("error_malformed", e.getMessage());
            } catch (IOException e){
                Log.e("error_ioexception", e.getMessage());
            }
            return result;
        }

        @Override
        protected void onPostExecute(String result) {
            super.onPostExecute(result);
            try {
                JSONObject jsonObject = new JSONObject(result);
                JSONArray jsonArray =
jsonObject.getJSONArray("response");
                JSONObject jsonObject2 = jsonArray.getJSONObject(0);
                user.setVkUserName(jsonObject2.getString("first_name")+"
"+jsonObject2.getString("last_name"));
                vkUserIdListener.onChangeVkUserId(user);
            } catch (JSONException e) {
                Log.e("error_jsonexpection",e.getMessage());
            }
        }
    }

    private String convertInputStreamToString(InputStream inputStream) throws
IOException {
        BufferedReader bufferedReader = new BufferedReader( new
InputStreamReader(inputStream));
        String line;
        String result = "";
        while((line = bufferedReader.readLine()) != null){
            result += line;
        }
        if(inputStream!=null){
            inputStream.close();
        }
        return result;
    }

```

```
}
```

Что бы передать в класс MainActivity.java полученные данные из социальной сети Вконтакте (идентификатор пользователя и его имя), используется шаблон проектирования Наблюдатель (Observer), прочитав про него подробнее можно [здесь \[10\]](#). Для его реализации был создан интерфейс с одним единственным методом. Ниже представлен код интерфейса.

```
public interface OnChangeListener {  
    public void onChangeUser(Users user);  
}
```

Теперь воспользуемся созданным классом и интерфейсом в главной активности MainActivity.java, а так же напомним код для сохранения пользовательских данных путём отправки запроса на сервер, ранее реализованный и успешно загруженный на платформу Heroku. Код класса MainActivity.java представлен ниже.

```
package ru.myitschool;  
  
import java.io.BufferedInputStream;  
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStream;  
import java.io.InputStreamReader;  
import java.net.HttpURLConnection;  
import java.net.MalformedURLException;  
import java.net.URL;  
import org.json.JSONException;  
import org.json.JSONObject;  
import android.app.Activity;  
import android.content.Context;  
import android.location.Location;  
import android.location.LocationManager;  
import android.os.AsyncTask;  
import android.os.Bundle;  
import android.util.Log;  
import android.webkit.WebView;  
import android.widget.Toast;  
  
public class MainActivity extends Activity {  
  
    private WebView wvAuth;  
    private LocationManager locationManager;  
    private Users user;  
    private Context ctx;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
    }  
}
```

```

        ctx = MainActivity.this;

        locationManager = (LocationManager) getSystemService(LOCATION_SERVICE);
        wvAuth = (WebView) findViewById(R.id.wvAuth);
        wvAuth.setWebViewClient(new VKWebViewClient(ctx, new
OnChangeVkUserIdListener() {
            @Override
            public void onChangeVkUserId(Users user) {
                MainActivity.this.user = user;
                Location location =
locationManager.getLastKnownLocation(LocationManager.NETWORK_PROVIDER);

                MainActivity.this.user.setLatitude(location.getLatitude());

                MainActivity.this.user.setLongitude(location.getLongitude());
                new HttpSaveUserInfoTask().execute("http://pure-inlet-
35553.herokuapp.com/save");
            }
        }));

        wvAuth.loadUrl("https://oauth.vk.com/authorize?client_id="+getString(R.strin
g.vk_app_id)+"&display=mobile&redirect_uri=https://oauth.vk.com/blank.html&response
_type=token&v=5.44");
    }

    private class HttpSaveUserInfoTask extends AsyncTask<String, Void, String>{

        private JSONObject json;

        @Override
        protected void onPreExecute() {
            super.onPreExecute();
            try {
                json = new JSONObject();
                json.put("vkUserId", user.getVkUserId());
                json.put("vkUserName", user.getVkUserName());
                json.put("latitude", user.getLatitude());
                json.put("longitude", user.getLongitude());
            } catch (JSONException e) {
                Log.e("error_json", e.getMessage());
            }
        }

        @Override
        protected String doInBackground(String... params) {
            InputStream inputStream;
            HttpURLConnection urlConnection;
            String result = "";
            String jsonToString = json.toString();
            try {
                URL url = new URL(params[0]);

```

```

        urlConnection = (HttpURLConnection) url.openConnection();
        urlConnection.setRequestMethod("PUT");
        urlConnection.setDoInput(true);
        urlConnection.setDoOutput(true);
        urlConnection.setRequestProperty("Content-Type", "text/plain; charset=utf-8");
        urlConnection.connect();
        urlConnection.getOutputStream().write(jsonToString.getBytes());
        inputStream = new BufferedInputStream(urlConnection.getInputStream());
        String response = convertInputStreamToString(inputStream);
        urlConnection.disconnect();
        result = response;
    } catch (MalformedURLException e) {
        Log.e("error_malformed", e.getLocalizedMessage());
    } catch (IOException e){
        Log.e("error_ioexception", e.getLocalizedMessage());
    }
    return result;
}

@Override
protected void onPostExecute(String result) {
    super.onPostExecute(result);
    if(result.equals("1")){
        Toast.makeText(ctx, "Данные пользователя успешно сохранены на сервере", Toast.LENGTH_LONG).show();
    }else{
        Toast.makeText(ctx, "При сохранении пользовательских данных на сервере произошла ошибка", Toast.LENGTH_LONG).show();
    }
}

private String convertInputStreamToString(InputStream inputStream) throws IOException {
    BufferedReader bufferedReader = new BufferedReader( new
    InputStreamReader(inputStream));
    String line;
    String result = "";
    while((line = bufferedReader.readLine()) != null){
        result += line;
    }
    if(inputStream!=null){
        inputStream.close();
    }
    return result;
}
}

```

На текущем этапе имеется полностью работающее приложение, которое позволяет авторизоваться через социальную сеть Вконтакте, получает оттуда имя пользователя, определяет его текущее место положение и сохраняет это всё в базе данных Heroku при помощи сервера.

Теперь необходимо показать пользователю его местонахождение на карте. Для этого к текущему Android-проекту подключим библиотеку от компании Яндекс “Yandex Map Kit for Android”. Она доступна на github по этой [ссылке \[11\]](#). При внимательном ознакомлении с данной ссылкой, можно увидеть проект-пример по работе с данной библиотекой.

Создадим новую активность с названием “MapActivity”, и в файл разметки добавим компонент “ru.yandex.yandexmapkit.MapView”, где базовым слоем является вертикальный LinearLayout. Укажем id со значением “@+id/map” и добавим параметры ширины и высоты со значением по родительскому контейнеру. Остался еще один важный параметр, который называется “apiKey”. Это ключ, который выдаётся компанией Яндекс, путём отправки письма на их почту: support@mobmaps.yandex.ru. По крайней мере, так утверждают не многочисленные источники в интернете. Но есть и более простое решение, можно установить тестовое значение ключа: “1234567890”. Для простых экспериментов с картами его должно быть достаточно. Разметка приобретёт следующий вид.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="ru.myitschool.MapActivity" >

    <ru.yandex.yandexmapkit.MapView
        android:id="@+id/map"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:apiKey="1234567890"/>

</LinearLayout>
```

Теперь займёмся написанием кода в MapActivity для пока пользователю его местоположения. Для этого, первым делом, добавим в MainActivity код, отвечающий за открытие MapActivity, привязав к намерению данные о пользователе. Код необходимо вставить в реализации слушателя OnChangeListener, перед отправкой запроса на сервер для сохранения информации.

```
startActivity(new Intent(ctx, MapActivity.class).
    putExtra("vkUserId", mUser.getVkUserId()).
    putExtra("vkUserName", mUser.getVkUserName()).
    putExtra("latitude", mUser.getLatitude()).
    putExtra("longitude", mUser.getLongitude()));
```

В классе MapActivity определим три закрытые переменные типов MapView, MapController и Users, а затем инициализируем их в методе onCreate.

```
import ru.yandex.yandexmapkit.MapController;
import ru.yandex.yandexmapkit.MapView;
import android.app.Activity;
import android.os.Bundle;

public class MapActivity extends Activity {

    private MapView mapView;
    private MapController mapController;
    private Users mUser;
    private List<Users> listUsers;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_map);
        mapView = (MapView) findViewById(R.id.map);
        mapController = mapView.getMapController();

        Bundle bundle = getIntent().getExtras();
        mUser = new Users(bundle.getLong("vkUserId"),
                           bundle.getString("vkUserName"),
                           bundle.getDouble("latitude"),
                           bundle.getDouble("longitude"));
    }
}
```

Поставим точку на карте с обозначением местоположения пользователя и при нажатии на неё будем выводить информацию об имени с идентификатором в социальной сети Вконтакте. В качестве изображения точки используется стандартная иконка приложения Android. Для этого напишем отдельный метод `showUserOnMap` и вызовем его в самом конце метода `onCreate`. Код метода прилагается ниже.

```
private void showUserOnMap(){
    OverlayManager overlayManager = mapController.getOverlayManager();
    GeoPoint geoPoint = new GeoPoint(mUser.getLatitude(), mUser.getLongitude());
    mapController.setPositionAnimationTo(geoPoint);
    Overlay overlay = new Overlay(mapController);
        OverlayItem overlayItem = new OverlayItem(geoPoint,
            getResources().getDrawable(R.drawable.ic_launcher));
    BalloonItem balloonItem = new BalloonItem(MapActivity.this, geoPoint);
    balloonItem.setText("Меня зовут: "+mUser.getVkUserName()
        +"\nМой id в соц.сети Вконтакте: "
        +String.valueOf(mUser.getVkUserId()));
    overlayItem.setBalloonItem(balloonItem);
    overlay.addOverlayItem(overlayItem);
    overlayManager.addOverlay(overlay);
}
```

Хоть и используется много незнакомых классов, код легко читаем и понятен. Сначала создаётся объект менеджер, который управляет всем балом (ставит точку, добавляет иконки, информацию,

приближает карту и т.д.). Затем определяется объект с координатами. Далее меняется позиция карты в соответствии с заданными координатами. Затем создаётся “место” и сам объект/точка. И в самом конце всё это отдаётся менеджеру для размещения. Запустив приложение, можно увидеть иконку андроида, демонстрирующую местоположение пользователя. При нажатии на неё выводится подробная информация: имя и идентификатор в социальной сети Вконтакте. Пример полученного результата изображён на рисунке 20.

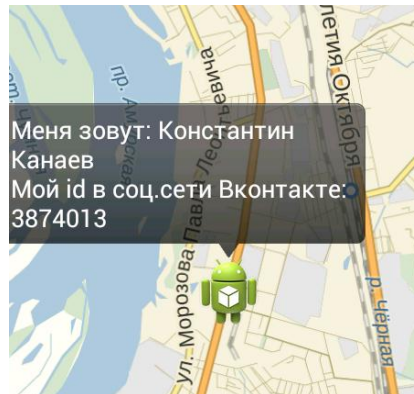


Рисунок 20 - демонстрация местоположение пользователя

Добавим возможность просмотра всех пользователей приложения на карте. Для этого отправим запрос на сервер, для получения всех записей из таблицы Users и напомним новый метод, добавляющий их на карту. Запрос будет отправляться при загрузке MapActivity, поэтому его вызов определяем в конце метода onCreate. Что бы отобразить всех пользователей, используем клик по элементу меню. Весь код класса MapActivity представлен ниже.

```
import java.io.BufferedInputStream;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;
import org.json.JSONArray;
import org.json.JSONException;
import ru.yandex.yandexmapkit.MapController;
import ru.yandex.yandexmapkit.MapView;
import ru.yandex.yandexmapkit.OverlayManager;
import ru.yandex.yandexmapkit.overlay.Overlay;
import ru.yandex.yandexmapkit.overlay.OverlayItem;
import ru.yandex.yandexmapkit.overlay.balloon.BalloonItem;
import ru.yandex.yandexmapkit.utils.GeoPoint;
import android.app.Activity;
import android.app.ProgressDialog;
import android.os.AsyncTask;
import android.os.Bundle;
import android.util.Log;
import android.view.Menu;
```



```
import android.view.MenuItem;

public class MapActivity extends Activity {

    private MapView mapView;
    private MapController mapController;
    private Users mUser;
    private List<Users> listUsers;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_map);
        mapView = (MapView) findViewById(R.id.map);
        mapController = mapView.getMapController();

        Bundle bundle = getIntent().getExtras();
        mUser = new Users(bundle.getLong("vkUserId"),
                           bundle.getString("vkUserName"),
                           bundle.getDouble("latitude"),
                           bundle.getDouble("longitude"));
        new HttpGetAllUsersTask().execute
            ("http://pure-inlet-35553.herokuapp.com/getall");
        showUserOnMap();
    }

    private void showUserOnMap(){
        OverlayManager overlayManager = mapController.getOverlayManager();
        GeoPoint geoPoint = new GeoPoint(mUser.getLatitude(),
                                           mUser.getLongitude());
        mapController.setPositionAnimationTo(geoPoint);
        Overlay overlay = new Overlay(mapController);
        OverlayItem overlayItem = new OverlayItem(geoPoint,
            getResources().getDrawable(R.drawable.ic_launcher));
        BalloonItem balloonItem = new BalloonItem(MapActivity.this, geoPoint);
        balloonItem.setText("Меня зовут: "
                            +mUser.getVkUserName()
                            +"\nМой id в соц.сети Вконтакте: "
                            +String.valueOf(mUser.getVkUserId()));
        overlayItem.setBalloonItem(balloonItem);
        overlay.addOverlayItem(overlayItem);
        overlayManager.addOverlay(overlay);
    }

    private void showAllUsersOnMap(){
        OverlayManager overlayManager = mapController.getOverlayManager();
        Overlay overlay = new Overlay(mapController);
        GeoPoint geoPoint;
        OverlayItem overlayItem;
        BalloonItem balloonItem;
        Users user;
```

```

        for(int i=0; i<listUsers.size(); i++){
            user = listUsers.get(i);
            geoPoint = new GeoPoint(user.getLatitude(),
user.getLongitude());
            overlayItem = new OverlayItem(geoPoint,
getResources().getDrawable(R.drawable.ic_launcher));
            balloonItem = new BalloonItem(MapActivity.this, geoPoint);

            balloonItem.setText("Меня зовут: "
                +user.getVkUserName()
                +"\nМой id в соц.сети Вконтакте: "
                +String.valueOf(user.getVkUserId()));
            overlayItem.setBalloonItem(balloonItem);
            overlay.addOverlayItem(overlayItem);
            overlayManager.addOverlay(overlay);
        }
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.map, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();
        if (id == R.id.action_settings) {
            showAllUsersOnMap();
            return true;
        }
        return super.onOptionsItemSelected(item);
    }

    private class HttpGetAllUsersTask extends AsyncTask<String, Void, String>{

        ProgressDialog dialog;

        @Override
        protected void onPreExecute() {
            super.onPreExecute();
            dialog = new ProgressDialog(MapActivity.this);
            dialog.setTitle("Загружаем информацию...");
            dialog.setCancelable(false);
            dialog.setIndeterminate(true);
            dialog.show();
        }

        @Override
        protected String doInBackground(String... params) {
            InputStream inputStream;

```

```

        HttpURLConnection urlConnection;
        String result = "";
        try {
            URL url = new URL(params[0]);
            urlConnection = (HttpURLConnection) url.openConnection();
            urlConnection.setRequestMethod("GET");
            urlConnection.connect();
            inputStream = new BufferedInputStream(urlConnection.getInputStream());
            String response = convertInputStreamToString(inputStream);
            urlConnection.disconnect();
            result = response;
        } catch (MalformedURLException e) {
            Log.e("error_malformed", e.getLocalizedMessage());
        } catch (IOException e){
            Log.e("error_ioexception", e.getLocalizedMessage());
        }
        return result;
    }

    @Override
    protected void onPostExecute(String result) {
        super.onPostExecute(result);
        Users user = null;
        listUsers = new ArrayList<Users>();
        try {
            JSONArray array = new JSONArray(result);
            for(int i=0; i<array.length(); i++){
                user = new
Users(array.getJSONObject(i).getLong("vkUserId"),
array.getJSONObject(i).getString("vkUserName"),
array.getJSONObject(i).getDouble("latitude"),
array.getJSONObject(i).getDouble("longitude"));
                listUsers.add(user);
            }
        } catch (JSONException e) {
            e.printStackTrace();
        }
        dialog.dismiss();
    }

    private String convertInputStreamToString(InputStream inputStream) throws
IOException {
        BufferedReader bufferedReader = new BufferedReader( new
InputStreamReader(inputStream));
        String line;
        String result = "";
        while((line = bufferedReader.readLine()) != null){
            result += line;
        }
    }

```

```
}  
    if(inputStream!=null){  
        inputStream.close();  
    }  
    return result;  
}  
}
```

Теперь можно попросить товарища, находящегося в другом месте, воспользоваться данным приложением. После того, как им будет пройдена авторизация, данные о нём добавятся в систему и по нажатию на кнопку в меню, для показа всех пользователей, будем видно своё и его местоположение на карте. Результат в виде скриншота экрана можно увидеть на рисунке 21.

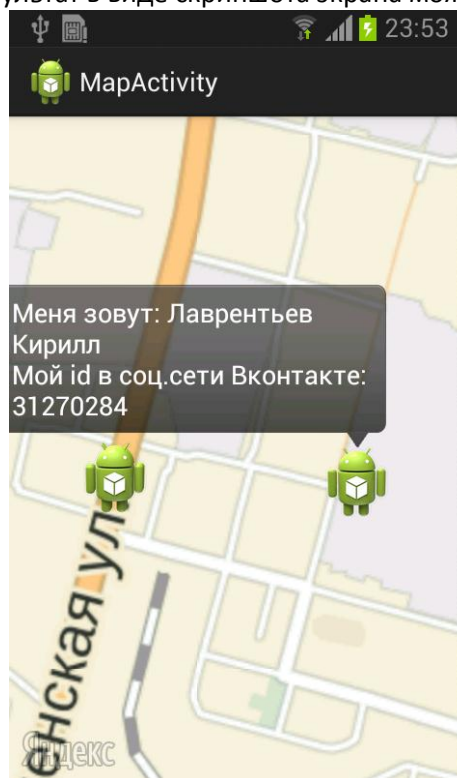


Рисунок 21 - демонстрация отображения местоположения всех пользователей приложения

На этом разработка практического примера закончена.

Задание 5.5.1

1. Измените на сервере метод добавления пользователя таким образом, что бы уже существующие записи в базе данных не дублировались. Для этого, перед сохранением в БД, сделайте запрос для проверки наличия данного пользователя.
2. Добавьте в клиентскую часть метод, отслеживающий наличие интернета и при его отсутствии не пускайте пользователя в приложение.
3. Добавьте в клиентское приложение возможность просматривать на карте ограниченное число пользователей.

*5.5.6 Синхронизация баз данных (локальной и серверной)

При разработке клиент-серверных приложений, иногда возникает необходимость, дать пользователю возможность работать с сервисом при отсутствии доступа к интернету. Для начала следует решить, какая информация должна быть доступна пользователю в любой момент времени:

1. Часть данных из серверной БД.
2. Все данные из серверной БД.

Рассмотрим первый вариант, когда в клиентском приложении нужен доступ только на отдельные функциональные части системы или даже только часть одной функции. Например, сохранение данных о пользователе для входа в систему или сохранение «тяжелой» информацию (медиа файлы) для организации быстрого доступа. Применительно к Android, у него есть четыре общеизвестных способа хранить информацию:

- **SharedPreferences.** Хранение примитивных данных в виде пар ключ-значение. Данный класс обеспечивает общую структуру, позволяющую сохранять и извлекать данные примитивных типов или строк. Информация будет доступна всегда и не зависит от текущего состояния приложения, т.е. даже если телефон был выключен она сохраняется. Чаще всего **SharedPreferences** используется для сохранения доступной (открытой) информации: выбранная мелодия звонка, цвет текста, обои. В общем всё то, что является, так называемыми, пользовательскими предпочтениями.
- **Internal Storage.** Хранение личных данных во внутренней памяти устройства (винчестере). По умолчанию, информация, сохраняемая таким способом, является частной и другие приложения не имеют к ней доступа (как и сами пользователи). Когда приложение удаляется, вместе с ним стираются и данные из внутреннего хранилища. Используется в качестве хранения данных с «повышенным уровнем секретности»: токен, логин, пароль.
- **Кэширование.** Кэш – промежуточный буфер с быстрым доступом, содержащий информацию, которая может быть запрошена с наибольшей вероятностью. Более подробно можно прочитать [здесь \[5\]](#). Кэширование очень часто применяют при работе с медиа-файлами и в частности с графическими изображениями. Очень хороший пример работы кэшем можно посмотреть в одной из лучших библиотек для загрузки изображения из сети интернет [\[6\]](#). Кэшем нужно пользоваться аккуратно, т.к. если устройству будет недостаточно памяти, он его очистит.
- **External Storage.** Хранение открытых данных на внешнем накопителе. Яркий пример такого накопителя – SD-карта. Такую систему хранения можно использовать, например, для выгрузки таблицы со статистикой. В следствии чего, пользователь сможет оперативно воспользоваться этими файлами через USB или извлечение SD-карты и установки её в другое устройство для дальнейшей работы. Важно помнить, что при использовании такого метода хранения, информация становится доступной для всех приложений. Google так же советует, что при покупке контента в вашем приложении, вы использовали данный метод хранения. Конечно если контент, не является «сверх секретным».

Все они могут помочь либо в ускорении работы с данными вашего приложения, либо в двадцати четырёхчасовом доступе для пользователя. Подробнее о каждом из них, можно прочитать [здесь \[7\] в официальной документации](#). Три первых способа хорошо подходят для хранения части данных из серверной БД, которые чаще всего носят статичный характер, т.е. не подвержены постоянным изменениям.

Остался еще один способ хранения данных в Android, уже известная нам СУБД SQLite. В следствии чего рассмотрим второй вариант, когда возникает необходимость построить приложение таким образом, чтобы информация в локальной и серверной базах данных была одинаковой. Способов решить данную задачу огромное количество, следовательно, нужно иметь чёткое представление о конечном результате, которого хочется достичь, чтобы оптимально выбрать решение.

Если приложение построено на rest-архитектуре, то стоит хорошенько подумать перед тем, чтобы использовать дублирующую базу данных на клиентском устройстве, ведь основное преимущество от использования данной архитектуры в том, чтобы отдавать данные только тогда, когда они нужны пользователю. Примерами такого подхода могут быть всем известные крупные сервисы: vkontakte, facebook, twitter, Instagram. Хотя иногда можно заметить, что в случае неустойчивого канала связи, информация с данных сервисов всё еще остаётся доступной. Этого можно достигнуть при использовании кэширования, которое было рассмотрено ранее.

Итак, можно дать совет – если используется rest-архитектура, а приложение функционально большое, наилучшим решением для сохранения какой-то информации, будет использование промежуточного буфера с быстрым доступом.

Иногда случается и так, что приложение не имеет обширного количества функций, база данных сравнительно не большая ($\approx 1-15$ таблиц) и что бы выделиться среди конкурирующих приложений, можно дать возможность пользователю взаимодействовать с сервисом в любой момент времени, не зависимо от наличия сети интернет. Примером такого приложения может быть сервис по планированию личных дел человека. Наверное, будет очень нехорошо, если человек не сможет получить доступ к списку своих дел, которые он утром старательно вносил в приложение. При этом, речь идёт именно о клиент-серверном приложении. В таком случае, стоит действительно задуматься о том, чтобы на устройстве создавалась локальная база данных, которая содержала бы в себе актуальную информацию на момент прекращения работы интернет соединения. Такой механизм работы часто называют односторонняя синхронизация.

Как не сложно догадаться, существует так же двухсторонняя синхронизация. Она основывается на том, что условно «главной» базы данных нет, в следствии чего реализуется функционал для синхронизации баз данных с клиента на сервер, либо в обратную сторону. Двухстороннюю синхронизацию так же можно назвать, с какой-то точки зрения, репликацией данных. Репликация – одна из техник масштабирования баз данных. Состоит эта техника в том, что данные с одного сервера базы данных постоянно копируются (реплицируются) на один или несколько других (называемые репликами). Для приложения появляется возможность использовать не один сервер для обработки всех запросов, а несколько. Таким образом появляется возможность распределить нагрузку с одного сервера на несколько.

Приведём несколько вариантов создания односторонней синхронизации (для всех вариантов характерной особенностью является создание локальной базы данных идентичной серверной):

- Каждый раз, когда посылаем запрос на сервер, чтобы получить какую-нибудь информацию, ответ от сервера выводим не только пользователю, но и сохраняем в локальную базу данных. Здесь есть один существенный недостаток – если какие-то части приложения не использовались, значит и новых данных в базе нет.
- Сделать дополнительное поле или таблицу в базе данных и при каждой новой записи увеличивать там значение. Затем, при входе в приложение, в момент пока красуется

заставка, сравнивать два значения (из локальной и серверной) и, если оно разное, присылать новые данные для всей БД.

- При загрузке приложения отправлять на сервер запрос со всеми последними id в таблицах и если таких нет, или есть следующие по возрастанию, то так же обновлять базы.
- Если в качестве локальной и серверной используется одна и та же СУБД, то при загрузке приложения можно отправлять на клиента скрипты запросов, для пересоздания БД с новыми записями. Не слишком хороший вариант, но очень редко можно встретить и такое.
- Если пользователи никак не влияют на изменение информации, а могут её только читать, то можно каждый раз высылать им обновления, как самого приложения, так и обновленную базу данных.
- Одним из правильных вариантов, является работа с транзакциями. Проверяем отличие и так же загружаем данные в локальную БД.
- И т.д.

Какой способ использовать, зависит от каждого конкретного случая, ведь в разработке программного обеспечения, очень много факторов, влияющих на принимаемые решения/технологии: время, деньги, труд, особенности предметной области и т.д.

*5.5.7. Индексы базы данных

Индекс - структура данных, которая помогает СУБД быстрее обнаружить отдельные записи в файле и сократить время выполнения запросов пользователей.

Вспомните, как мы ищем информацию в книге, используя предметный указатель. Мы ищем нужный термин в предметном указателе, где эти термины расположены в алфавитном порядке, узнаем номер страницы, переходим на нее. Это очень похоже на работу индексов!

Индекс позволяет более эффективно построить поиск данных в таблице БД. Индексы позволяют избежать неэффективный алгоритм последовательного поиска. Как и предметный указатель книги, индекс таблицы базы данных упорядочен по выбранному полю таблицы. Каждый элемент индекса содержит значение поля и один или несколько указателей (идентификаторов записей) на место расположения записи/ей с таким значением поля.

Создание индекса

Проиллюстрируем понятие индекса на примере. Пусть мы имеем таблицу “Contacts”, в которой хранятся данные о сотрудниках министерства:

_id	Last_name	First_name	Position	Phone_number	Email
1	Котов	Андрей	Министр	(4972) 24-51-30	1@example.ru
2	Новикова	Марина	Помощник министра	(4972) 24-51-30	2@example.ru
3	Семичастнова	Дарья	Специалист	(4972) 24-51-30	3@example.ru
4	Агеева	Елена	Начальник отдела государственного заказа	(4972) 24-51-31	4@example.ru
5	Булатова	Елена	Заместитель начальника	(4972) 24-51-31	5@example.ru

			отдела государственного заказа		
6	Семичастнова	Елена	Главный консультант отдела государственного заказа	(4972) 24-51-31	6@example.ru
7	Ивонина	Евгения	Директор департамента финансирования	(4972) 24-51-35	19@example.ru
8	Савичева	Валерия	Референт департамента финансирования	(4972) 24-51-36	20@example.ru
9	Волков	Владислав	Специалист отдела государственного заказа	(4972) 24-51-31	9@example.ru
10	Юров	Николай	Начальник отдела кадровой работы	(4972) 24-51-32	10@example.ru
11	Иванов	Евгений	Начальник отдела мобилизационной подготовки	(4972) 24-51-32	11@example.ru

В данной таблице первичный ключ - поле `_id`. Когда в таблице определяется первичный ключ, то СУБД автоматически создает индекс по этому полю. Одна таблица может иметь несколько связанных с ней индексов.

Для того, чтобы создать свой индекс, нужно использовать инструкцию `CREATE INDEX`:

```
CREATE INDEX index_name ON table_name (column_name[,column_name])
```

Допустим, в нашем примере наиболее востребованный способ поиска записей - это поиск по фамилии сотрудника. Поэтому мы создадим индекс для таблицы "Contacts" по полю "Last_name":

```
CREATE INDEX LstName_idx ON Contacts (Last_name)
```

В БД будет создана отдельная таблица-индекс `LstName_idx`, которая будет связана с исходной таблицей `Contacts` как показано на рис. 22.

Каждая запись индекса хранит указатель на записи исходной таблицы, причем если значения повторяются, то, может хранить несколько указателей (см. "Семичастнова").

Если в инструкции `CREATE INDEX` в скобках указать несколько полей, то будет создан составной индекс: например, индекс на столбцах `Last_name` и `First_name` (полное имя будет уникально, но отдельно возможны дубли в имени или фамилии).

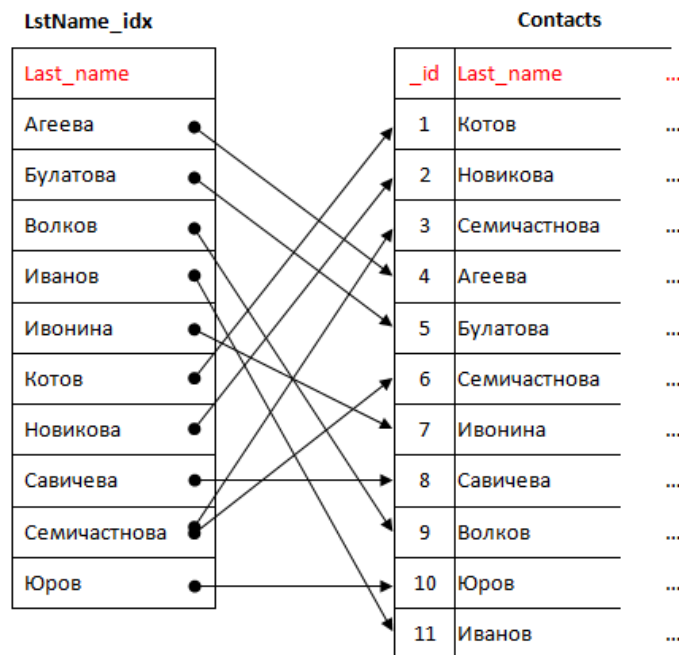


Рисунок 22 – Пример индекса

Применение индексов

Рассмотрим, что произойдет, если в нашем примере запустить на выполнение инструкцию:

```
SELECT * FROM Contacts WHERE Last_name="Савичева"
```

СУБД в индексной таблице LstName_idx найдет запись с значением "Савичева" и затем по указателю перейдет на 8-ю запись.

В чем же выигрыш? Преимущество в том, что индексная таблица упорядочена по нужному нам полю и здесь широко используются уже известные нам эффективные алгоритмы двоичного поиска. Если индекса по искомому полю нет, то нужную запись остается искать последовательным поиском по всей таблице.



Индексы в современных СУБД могут быть организованы с помощью различных структур данных. Используют сбалансированные деревья (B-деревья, B+-деревья, B-деревья) и хеш-таблицы.*

Если исходная таблица огромна, то поиск нужного индекса также становится весьма длительным. Поэтому в промышленных СУБД реализуют многоуровневые индексы: индексная таблица разделяется на несколько таблиц (субиндексов меньшего размера), а над ними создается индекс верхнего уровня, который "знает", какие диапазоны значений хранятся в каждом субиндексе.

Индексы не являются обязательными для таблицы. Если таблица имеет размер, как в приведенном примере, то работа индекса незаметна, потому что мы рассмотрели работу с индексом при простом поиске.

На самом деле, основное назначение индекса - это его использование в затратных по ресурсам запросах по объединению данных из нескольких таблиц. В таких запросах СУБД связывает каждую запись одной таблицы с записями из других таблиц по значению заданного поля и, тогда, рассмотренная нами в примере процедура поиска через индекс выполняется многократно для каждой записи таблицы. На практике, когда мы имеем дело с таблицами в сотни тысяч записей, грамотно построенные индексы могут сократить время выполнения сложных запросов с часов до минут.



Для БД на мобильных устройствах особенно важен грамотный подход к построению индексов.

Индексы ускоряют выборку, но замедляют изменение данных.

Следует быть осторожным с использованием индексов по нескольким полям. Иногда выигрыш от их использования меньше чем потери, связанные с использованием дополнительной памяти под индекс.

Лучший способ проверить - тестировать на конкретном устройстве.

Источники

- [1] HowToDoInJava. Как использовать Spring аннотации. URL: <http://howtodoinjava.com/spring/spring-core/how-to-use-spring-component-repository-service-and-controller-annotations/>
- [2] WikiBooks. Аннотации. URL: https://ru.wikibooks.org/wiki/Spring_Framework_Guide
- [3] Spring. URL: <http://spring.io/>
- [4] Tutorial. Building REST services with Spring. URL: <http://spring.io/guides/tutorials/bookmarks/>
- [5] Кэш. URL: <https://ru.wikipedia.org/wiki/Кэш>
- [6] Square Picasso. URL: <https://github.com/square/picasso>
- [7] Android data storage. URL: <http://developer.android.com/intl/ru/guide/topics/data/data-storage.html>
- [8] CrudRepository (Spring Data Core). URL: <http://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html>
- [9] Deploying Spring Boot Applications to Heroku. URL: <https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku>
- [10] Наблюдатель (шаблон проектирования). URL: [https://ru.wikipedia.org/wiki/Наблюдатель_\(шаблон_проектирования\)](https://ru.wikipedia.org/wiki/Наблюдатель_(шаблон_проектирования))
- [11] Yandex Map Kit for Android. URL: <https://github.com/yandexmobile/yandexmapkit-android>

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю ИТ ШКОЛЫ SAMSUNG Канаеву Константину Александровичу.