6.4. Передача данных по сети. Часть 3

Сайт: Samsung Innovation Campus
Курс: Мобильная разработка на Kotlin
Книга: 6.4. Передача данных по сети. Часть 3

Напечатано:: Murad Rezvan
Дата: понедельник, 3 июня 2024, 19:15

Оглавление

1. 6.4.1 REST-взаимодействие

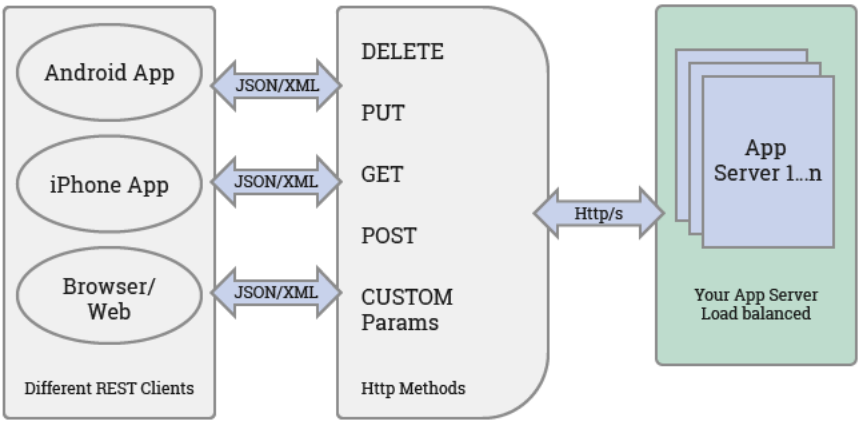
REST (represnetational state transfer) — это распространенная архитектура и принцип взаимодействия в клиент-серверных работающих поверх World Wide Web. Термин REST был предложен 2000 году одним из авторов HTTP-протокола - Роем Филдингом, одним из авторов HTTP-протокола.

REST является максимально простым интерфейсом управления информацией без каких либо дополнительных внутренних прослоек, как это реализовано например в SOAP, XML-RPC или AMF. При этом данные передаются ровно в том объеме и форме, в котором они нужны в рамках выполняемой операции. Сама же передаваемая единица информации определяется конкретным URL. При этом, часто URL является первичным ключем для обрабатываемой порции информации. То есть, например, альбом 12 будет иметь вид /album/12, а трек 15 в этом альбоме — /album/12/track/15 .

Команды управления данными — определяются протоколом передачи данных и внутренней логикой сервиса. Поскольку наиболее распространен протокол HTTP, то действия над данными определяются методом HTTP — GET, PUT, POST, DELETE. Эти четыре операции как раз соответствуют известному набору операций CRUD (Create-Read-Updtae-Delete)

Например:

- GET /album/ — получить список всех альбомов;
- GET /album/3/ — получить альбом номер 3;
- PUT /album/ — добавить альбом (данные в теле запроса);
- POST /album/3 — изменить альбом (данные в теле запроса);
- DELETE /album/3 — удалить альбом 3.



Принципы REST

Клиент-сервер.

В этой системе все взаимодействие происходит в как обмен между клиентом и сервером. Роли четко разделены — клиент делает запрос к серверу и получает ответ. При этом взаимосвязь строго один к одному.

Отсутствие состояния.

Понятие stateless означает, что сервер и клиент не отслеживает состояние друг друга. Конечно у клиента и у сервера может (и скорее всего будет) сохранятся информация о внутреннем состоянии, но при этом, пока нет соединения между клиентом и сервером, сервер не имеет понятия о существовании клиента, а также не ведется учет предыдущих запросов. Каждый новый запрос не имеет исторического контекста.

Единообразие интерфейса.

Этот принцип означает что между серверами и клиентами существует общее API, который позволяет каждой части быть заменяемой или изменяемой, без нарушения целостности системы. Единообразие достигается при выполнении следующих правил

1. Идентификация ресурсов. Ресурс идентифицируется в запросах, например при помощи URI. При этом как ресурсы хранятся на сервере, никоим образом не зависит от той формы в которой клиент его получает

2. Манипуляция ресурсами через представление. Клиенты имея идентификатор ресурса, имеют достаточно данных для модификации или удаления ресурса.
3. «Самодостаточные» сообщения. В сообщении от клиента будет достаточно информации, чтобы понять как его обрабатывать.
4. Гипермедиа как средство изменения состояния сервера. Клиенты могут изменить состояние системы только через действия, которые динамически идентифицируются на сервере посредством гипермедиа (к примеру, гиперссылки в гипертексте, формы связи, флажки, радиокнопки и прочее).

Слои.

Клиент может общаться с сервером не напрямую с сервером, а через промежуточные службы. При этом клиент может не знать об их существовании. Пример слоев — это прокси сервера , балансировщики нагрузки, шлюзы в другие протоколы.

2. 6.4.2 Пример Yandex predictor

Рассматриваемые далее примеры входят в практическую работу, исходный код которой можно скачать по ссылке

Retrofit — библиотека для работы с REST API. В общем случае, при выполнении запроса к REST-серверу, требуется выполнить ряд операций:

- сформировать URL;
- задать HTTP-заголовки;
- выбрать тип HTTP-запроса;
- сформировать тело HTTP-запроса, то есть преобразовать Java-объект в JSON;
- выполнить запрос, воспользовавшись HTTP-клиентом;
- распарсить результаты запроса, то есть преобразовать полученный JSON обратно в Java-объект.

Библиотека Retrofit позволяет описать все перечисленные операции с помощью аннотаций. Продемонстрируем возможности REST-взаимодействия на примере одного из REST API Яндекс https://tech.yandex.ru/#catalog — Предиктора. Яндекс.Предиктор подсказывает наиболее вероятное продолжение слов или фраз, набираемых пользователем. Это упрощает ввод текста, особенно на мобильных устройствах. При этом Предиктор учитывает возможные опечатки.

Описание Web API

Для доступа к API Предиктора по HTTP предлагаются интерфейсы XML, JSON. Все интерфейсы обеспечивают одинаковую функциональность и используют одни и те же входные параметры. XML-интерфейс возвращает ответ в виде XML-документа, JSON-интерфейс вместо XML-элементов возвращает JSON-объекты с теми же именами и семантикой.

Доступ ко всем методам API осуществляется по ключу. Получить ключ можно по ссылке https://tech.yandex.ru/keys/get/?service=pdct.

Метод getLangs

Возвращает список языков, поддерживаемых сервисом. Входные параметры:

Параметр	Тип	Описание
key	string	API-ключ

Пример запроса XML-интерфейс:

```
https://predictor.yandex.net/api/v1/predict/getLangs?key=API-ключ
```

Пример запроса JSON-интерфейс:

```
https://predictor.yandex.net/api/v1/predict.json/getLangs?key=API-ключ
```

Возвращаемое значение - массив строк (ArrayOfString) с названиями доступных языков. Например:

```
["ru","en","pl","uk","de","fr","es","it","tr"]
```

Возможные коды ошибок:

Код	Значение	Описание
ERR_KEY_INVALID	401	Ключ API невалиден
ERR_KEY_BLOCKED	402	Ключ API заблокирован

Метод complete

Возвращает наиболее вероятное продолжение текста, а также признак конца слова. Входные параметры могут передаваться либо с помощью HTTP GET-запроса (см. пример), либо с помощью HTTP POST-запроса, где параметры передаются в body HTTP-запроса. Пример запроса XML-интерфейс:

```
https://predictor.yandex.net/api/v1/predict/complete?key=API-ключ&q=hello+wo&lang=en
```

Пример запроса JSON-интерфейс:

```
https://predictor.yandex.net/api/v1/predict.json/complete?key=API-ключ&q=hello+wo&lang=en
```

Параметры запроса следующие:

Параметр	Тип	Описание
key	string	API-ключ
lang	string	Язык текста (например, «en»).
q	string	Текст, на который указывает курсор пользователя. При подборе подсказки учитывается слово, на которое указывает курсор и 2 слова слева от него, поэтому не требуется передавать текст длиннее трех-четырех слов

Параметр	Тип	Описание
limit	int	Максимальное количество возвращаемых строк (по умолчанию 1). Необязательный параметр

Возвращаемое значение - структура данных содержащих значения:

Элемент	Описание
endOfWord	признак конца слова (true/false).
Pos	позиция в слове, для которого возвращается продолжение. Позиция отсчитывается от последнего символа в запросе, переданном в элементе q.
text	Содержит элементы string с наиболее вероятными вариантами продолжения заданного текста. Если метод не может «продолжить» текст, то элемент не возвращается

Например:

```
{"endOfWord":false,"pos":-2,"text":["world"]}
```

Возможные коды ошибок:

Код	Значение	Описание
ERR_OK	200	Операция выполнена успешно
ERR_KEY_INVALID	401	Ключ API невалиден
ERR_KEY_BLOCKED	402	Ключ API заблокирован
ERR_DAILY_REQ_LIMIT_EXCEEDED	403	Превышено суточное ограничение на количество запросов (с учетом вызовов метода getLangs)
ERR_DAILY_CHAR_LIMIT_EXCEEDED	404	Превышено суточное ограничение на объем подсказанного текста (с учетом вызовов метода getLangs)
ERR_TEXT_TOO_LONG	413	Превышен максимальный размер текста (1000 символов)
ERR_LANG_NOT_SUPPORTED	501	Указанный язык не поддерживается

Пример.

Рассмотрим пример клиент-серверного приложение, использующего библиотеку Retrofit и сервис Яндекс.Предиктор. В приложении необходимо реализовать два текстовых поля. Одно поле для ввода текста, а второе поле для отображения результата работы предикотора. При вводе символов текста в поле ввода, сразу же отправляется запрос предиктору и результат от него записывается в поле вывода.

В начале необходимо объявить структуру данных, в которую будет десериализоваться ответ предиктора.

```
class Model(var  endOfWord:Boolean, var pos:Int, var text:Array<String>)
```

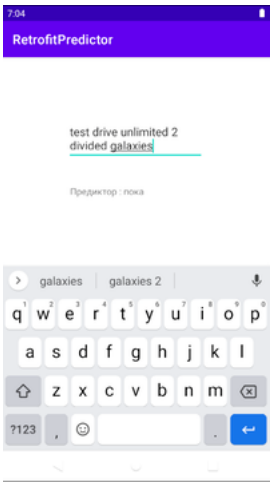
Далее необходимо объявить интерфейс для используемый далее в Retrofit для реализации методов доступа к Web API.

```
interface RestApi {
    @GET("api/v1/predict.json/complete")
    fun predict(@Query("key") key:String , @Query("q") q:String , @Query("lang") lang:String ): Call<Model>
}
```

В коде активности рассматриваемого приложения, для реакции на ввод символов в поле ввода, необходимо подключить специальный объект-слушатель **TextWatcher** на событие изменения текста в поле ввода. Собственно в этом объекте слушателе и формируется запрос к сервису Yandex - **service.predict()**. Непосредственно выполнения запроса осуществляется в асинхронном режиме - путем вызова из объекта запроса метода **enqueue()**. В метод **enqueue()** передается объект слушатель **Callback**, который будет использоваться Андроидом, при получении ответа от сервера - тогда в нем будет вызван метод **onResponse()** или **onFailure()**.

```
 editText.addTextChangedListener(object : TextWatcher {
            override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {}
            override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {}
            override fun afterTextChanged(s: Editable) {
                val call: Call<Model> =
                    service.predict(PREDICTOR_KEY, editText.getText().toString(), "ru");
                call.enqueue(object:Callback<Model> {
                    override fun onResponse(call: Call<Model>, response: Response<Model>) {
                        try {
                            val textWord: String = response.body().text[0].toString();
                            textView.setText("Предиктор : " + textWord);
                        } catch (e: Exception) {
                            e.printStackTrace();
                        }
                    }
                    override fun onFailure(call: Call<Model>, t: Throwable) {
                    }
                });
            }
        })
```

Экран работающего приложения.


7:04
RetrofitPredictor
test drive unlimited 2
divided galaxies
Предиктор : пока
galaxies
galaxies 2

3. 6.4.3 Облачные технологии

Облачные технологии — сегодня это уже не только информационная технология, но и привычная концепция повседневной жизни современного человека. По сути, это перенос в публичную сеть всех видов ресурсов пользователя — от файлов до приложений и сервисов. Доступ к ним может осуществляться через интернет как по отдельности, так и с возможностью их взаимодействия.

Еще совсем недавно пользователь, чтобы просмотреть офисный документ, пересланный по почте, должен был открыть приложение электронной почты, получить письмо, скачать файл из вложения, открыть его при помощи офисного приложения. При использовании облачных технологий пользователь может открыть браузер, зайти на веб-почту, открыть письмо и тут же в браузере просмотреть файл, используя офисное веб-приложение.

Таким образом, можно видеть, что в облачные технологии предполагают перенос файлов (данных) и приложений (вычислений) с компьютера пользователя на интерент-облачные ресурсы.

Достоинства облачных технологий:

- доступность: чтобы получить доступ к своим данным, пользователю достаточно иметь лишь доступ в интернет;
- мобильность: пользователь не привязан к своему рабочему месту, он может работать из любой точки мира, с любого устройства;
- экономичность: пользователю нет необходимости закупать программное обеспечение либо компьютерное оборудование, достаточно иметь любое устройство, вплоть до мобильного, с браузером. При этом он не несет нагрузки по обслуживанию оборудованию, ПО и лицензий;
- арендность: пользователь платит за продукт только в тот момент, когда он ему нужен, и только в том объеме, в котором хочет;
- гибкость: все необходимые ресурсы предоставляются провайдером автоматически;
- высокая технологичность: инфраструктура, на которой работает облако, поддерживается оператором в самом актуальном состоянии. Это означает, что ПО будет самых последних версий, оборудование использует самые мощные и современные решения;
- надежность, которую обеспечивают облачные платформы, обычно гораздо выше, чем у оборудования пользователя, ведь далеко не всякий пользователь может позволить приобрести отказоустойчивый ЦОД.

Недостатки:

- необходимость постоянного соединения к интернет;
- ограничения ПО. ПО, которое разворачивается в облаке, может иметь ограничения, не позволяющие использовать некоторые функции, доступные в локальных версиях аналогичного ПО. Однако следует помнить, что в облачном ПО в то же самое время могут быть и функции, превосходящие возможности локального ПО;
- конфиденциальность: конфиденциальность данных, хранимых в публичных «облаках», часто вызывает беспокойство у пользователей. Однако грамотный подход к вопросам шифрования данных и настроек доступа могут минимизировать эти риски;
- безопасность: облачная платформа сама по себе достаточно безопасна, однако в случае проникновения злоумышленника в сервисы пользователя, он получает доступ сразу к очень большому объему данных. Такое проникновение возможно, например, из-за использования слабых паролей или пользования небезопасной сетью;
- дороговизна оборудования: создание собственного облака требует довольно крупных материальных вложений, в связи с чем это не могут себе позволить небольшие компании или стартапы;
- дальнейшая монетизация ресурса. Вполне возможно, что компании-операторы в дальнейшем решат брать плату с пользователей за предоставляемые услуги, которые в данный момент доступны бесплатно.

Облачные платформы классифицируют по модели развертывания

Частное облако

Частное облако (англ. private cloud) — это облачная ИТ-инфраструктура, созданная частной организацией. Обычно доступ в это облако имеют сотрудники организации и иногда ее клиенты. Таким образом, эта инфраструктура предназначена для пользования закрытыми корпоративными сообществами и не содержащая публичных сервисов.

Общественное облако

Общественное облако (англ. community cloud) — это вид инфраструктуры, предназначенный для использования сообществом пользователей, вне зависимости от принадлежности пользователя к какой-либо компании. Общественное облако обычно находится в собственности коммерческой организации, и она несет затраты по его обслуживанию.

Гибридное облако

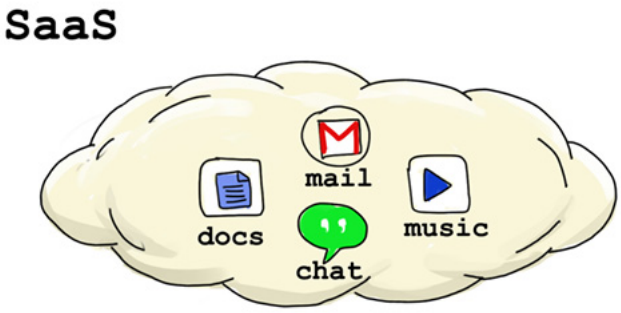
Гибридное облако (англ. hybrid cloud) — это комбинация из двух или более различных облачных инфраструктур (частных, публичных или общественных), остающихся независимыми информационными структурами, но связанных технологиями передачи данных и приложений.


Приватное облако
Гибридное облако
Общественное облако

4. 6.4.4 Модели обслуживания

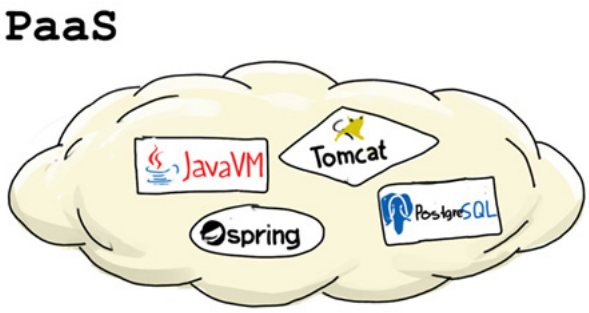
Модель обслуживания определяет, какой вид услуг предоставляет облако.

Программное обеспечение как услуга



Программное обеспечение как услуга (SaaS, англ. Software-as-a-Service) — модель, в которой услугой, предоставляемой пользователю, является доступ к прикладному программному продукту, работающему в облаке. Доступ к предоставляемой программе может осуществляться из браузера. Пример SaaS — веб-почта, Google Docs, Google Calendar и подобные онлайн-программы. Администрирование и управление физической и виртуальной инфраструктурой облака, в том числе сети, серверов, операционных систем, систем хранения, осуществляется облачным провайдером. Доступ пользователю может предоставляться как на платной, так и на бесплатной основе.

Платформа как услуга



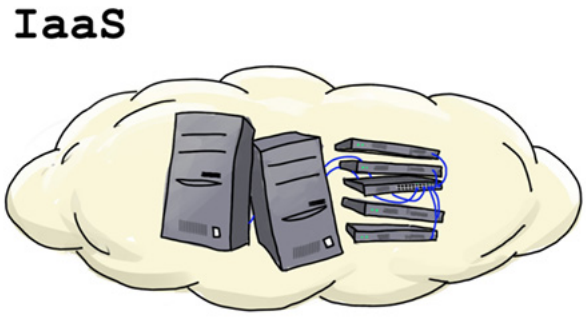
Платформа как услуга (PaaS, англ. Platform-as-a-Service) — модель, когда пользователю предоставляется возможность запускать свое программное обеспечивание на работающем в облаке ПО среднего уровня (MiddleWare) и прочего северного/служебного ПО и фреймворков. Обычно в состав ПО платформы входят:

- сервера СУБД, например, Postgresql, MySql, db2, Oracle DB, MsSql, nosql MongoDB и т. д.;
- сервера приложений/контейнеры, например, Tomcat, Jboss и т. д.;
- среды исполнения, например, Java JVM, GO vm и т. д.;
- фреймворки/библиотеки, например, JDBC, Gson, Spring и т. д.

На этой платформе пользователь может устанавливать как разработанные им приложения, так и существующие ранее. Развертывание, администрирование и управление основной физической и виртуальной инфраструктурой облака, в том числе сети, серверов, операционных систем, систем хранения, а также промежуточного ПО осуществляется облачным провайдером. Иными словами, PAAS — это облачная платформа для разработки, тестирования, развертывания и поддержки собственного ПО пользователя. К 2020 году мировой рынок PaaS предположительно оценивается в $235 млрд. Десятка крупнейших провайдеров:

- Amazon.com (Beanstalk);
- Salesforce.com (Force.com, Heroku,Database.com);
- LongJump;
- Microsoft (Windows Azure);
- IBM (Bluemix);
- Red Hat (OpenShift);
- VMWare (Cloud Foundry);
- Google (App Engine);
- CloudBees;
- Engine Yard.

Инфраструктура как услуга



Инфраструктура как услуга (IaaS, англ. Infrastructure-as-a-Service) предоставляет потребителю виртуальный компьютер (именно как аппаратную часть), с сетью и выходом в интернет. Используя эти облачные аппаратные ресурсы, пользователь может сам установить операционную систему, и любое ПО, которое ему необходимо для его деятельности. Таким образом, предоставляемой услугой фактически является аренда в облаке аппаратной IT-инфраструктуры. Потребитель услуги может устанавливать и запускать произвольное программное обеспечение, которое может включать в себя операционные системы, платформенное и прикладное программное обеспечение. Администрирование и управление основной физической и виртуальной инфраструктурой облака, в том числе сети, серверов, типов используемых операционных систем, систем хранения осуществляется облачным провайдером.

5. 6.4.5 Пример приложения на PAAS Heroku

В предыдущих главах было рассмотрено как с помощью фреймоворка Retrofit можно разработать клиентское мобильное приложение, которое взаимодействует с REST сервисом. Также было рассмотрен пример "Hello world" реализации серверного приложения. Далее рассмотрим пример полного цикла клиент-серверного приложения с использованием PAAS и БД на стороне сервера. В приведенном примере рассматривается реализация телефонного справочника на REST сервисе.

Рассматриваемые далее примеры входят в практическую работу, исходный код которой можно скачать по ссылке

Сервер

Создание проекта

Для начала разработки кода сервера необходимо зарегистрироваться на сервере heroku.com. В процессе регистрации будет предложен выбор языка программирования - "Java", после чего автоматически откроется мини-учебник "Getting Started" в котором нужно изучить первые пять пунктов. В процессе изучения нужно установить:

- Git клиент,
- Heroku Command Line Interface (HeroluCLI),
- помимо требований PAAS, дополнительно потребуется установка плагина Spring Tools 4 (STS 4) в среду разработки Eclipse.

Далее выполняем следующую последовательность действий для создания и настройки серверного проекта.

1. открываем консоль командного интерпретатора - cmd для Windows и bash для Linux
2. логинимся в Heroku

```
heroku login
```

3. переходим в папку с проектами Eclipse

```
cd <путь к папке проектов Eclipse>
```

4. Скачиваем пример Spring проекта от Heroku (консоль не закрываем!)

```
git clone https://github.com/heroku/java-getting-started
```

5. Создаем проект на PASS Heroku на основе полученного примера. После выполнения команд проект появиться в dashboard в на сайте Heroku.

```
cd java-getting-started
heroku create
```

6. Импортируем проект в Eclipse. File -> Import -> Maven -> Existing Maven Project , в открывшемся мастере выбираем папку с проектом и отмечаем галочкой pom.xml в projects.

7. В открывшемся проекте в папке com.example необходимо создать класс PhoneBookContoller, проаннотированный @RestController. Именно основываясь на этих аннотациях, "под капотом" фреймворк связывает классы с сервлетами, работающими в сервере-контейнере. В рассматриваемом случае сервер сервлет-контейнер это Tomcat. В классе необходимо объявить метод hi(), связанный с одноименным URI, при помощи аннотации @RequestMapping . Этот метод далее будем использовать для тестирования доступности создаваемого REST сервиса:

```
@RestController
public class PhoneBookContoller {

@RequestMapping("/hi")
public String hi() {
    return "Hi";
}
}
```

8. В консоли необходимо выполнить команды отправки проекта на сервер Git, в роли которого в данном случае выступает PAAS. При этом, после отправки, в консоли будут выводиться логи по скачиванию на платформе библиотек, компиляции проекта на стороне сервера (должно быть BUILD SUCCESS!), и запуска.

```
git add .
git commit -m "first commit"
git push heroku main
```

9. При необходимости логи запуска и последующих операций на сервере можно просмотреть при помощи команды

```
heroku logs
```

10. На этом первое развертывание (deploy) серверного ПО завершено. Проверить работоспособность можно например зайдя в браузере на URL вашего проекта с URI /hi . Получить путь можно также с помощью команды:

```
heroku open /hi
```

База данных

Здесь рассмотрим добавление в Heroku проект add-on Heroku Postgress и создание таблицы для хранения данных телефонного справочника.

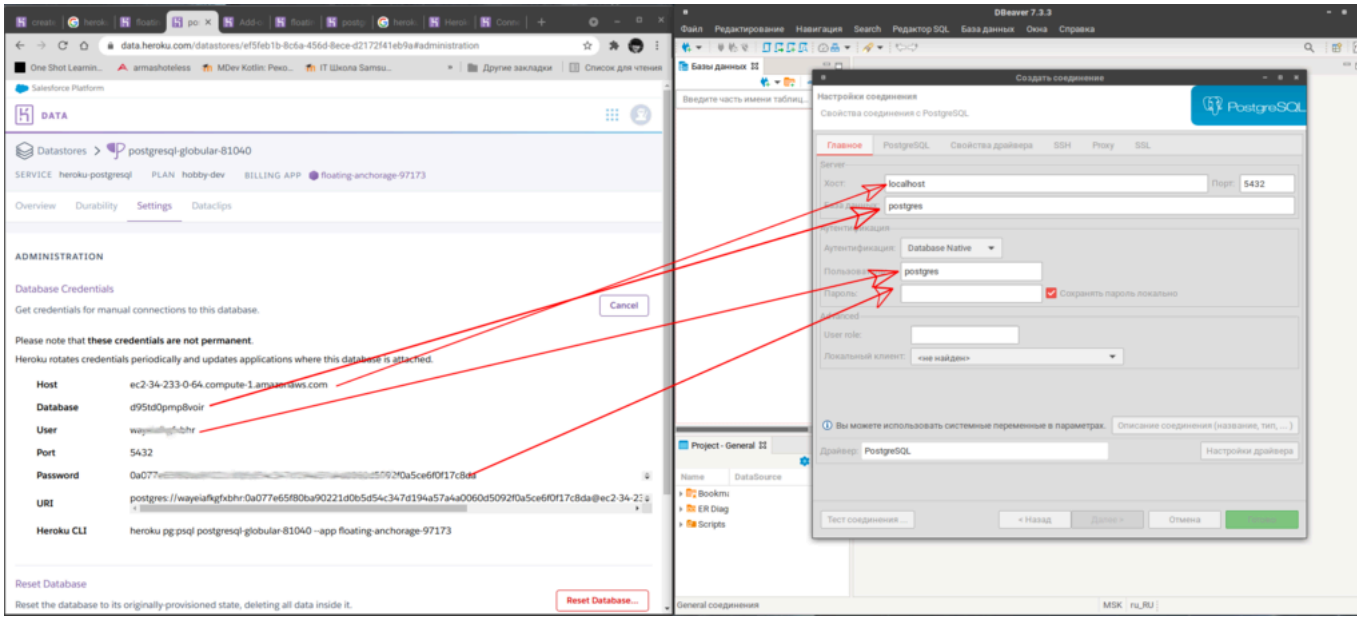
1. Для того чтобы сохранять данные в БД, необходимо подключить к Heroku проекту дополнение с СУБД, например Heroku Postgress addon. Это можно сделать либо в дашборде на сайте Heroku, кликнув по проекту. Если на закладке overview, в разделе installed add-ons нет подключенного расширения с СУБД, то можно нажать "Configure Add-ons" , далее нажать кнопку "Find more add-ons" и выбрать из списка "Heroklu Postgres". Либо тоже самое можно сделать при помощи команд в консоли:

```
heroku addons
heroku addons:create heroku-postgresql:hobby-dev
```

2. Подключение к БД можно выполнить из любой утилиты-клиента - например программы с графическим интерфейсом dbeaver или консольное приложение psql. В случае использования консольного клиента, для подключения достаточно ввести команду:

```
heroku pg:psql
```

При использования dbeaver нужно создать подключение: выбрать в главном меню "Базы данных"-> "Новое соединение", выбрать PostgreSQL и заполнить значимые поля (host, база данных, пользователь, пароль) параметрами подключения взятыми на сайте хероку в разделе dashboard -> проект -> overview -> Installed Add-ons -> heroku postgres -> settings -> View cridentials.



3. Далее, в утилите, необходимо создать таблицу phonebook выполнив в утилите SQL запрос:

```
-- создание таблицы
CREATE TABLE public.phonebook (
id serial NOT NULL,
name varchar NULL,
phone varchar NULL,
CONSTRAINT phonebook_pk PRIMARY KEY (id) );
-- добавление тестовой записи в таблицу
insert into phonebook (name,phone) values ('Ivan','+799912345678');
-- просмотр таблицы
select * from phonebook;
```

Серверный код

Теперь, когда имеется северный Java проект и в проект подключена БД на сервере и имеется таблица phonebook можно писать код, который будет при вызове соответствующих HTTP запросов выполнять классический набор CRUD (create, read, update, delete). Для реализации этой задачи добавим методы в созданный ранее класс PhoneBookContoller:

1. Создать объект для сериализации данных

```
public class PhonebookEntry {
    public long id;
    public String name;
    public String phone;
    public PhonebookEntry(long id,String name,String phone) {
            this.id=id;
            this.name=name;
            this.phone=phone;
    }
}
```

2. Добавить метод PUT для создания записи в телефонную книгу

```
    @RequestMapping(method = RequestMethod.PUT, value = "/create")
    public Boolean create(@RequestBody PhonebookEntry pb) {
            try (Connection connection = dataSource.getConnection()) {
                  PreparedStatement stmt = connection.prepareStatement("insert into phonebook (name,phone) values( ? , ? )");
                  stmt.setString(1, pb.name);
                  stmt.setString(2, pb.phone);
                  int num = stmt.executeUpdate();
                    return num > 0;
                } catch (Exception e) { }
            return false;
    }
```

3. Добавить метод GET для чтения записей из телефонной книги

```
    @RequestMapping(method = RequestMethod.GET,value = "/read")
    public List<PhonebookEntry> read(){
            ArrayList<PhonebookEntry> ls=new ArrayList<PhonebookEntry>();
             try (Connection connection = dataSource.getConnection()) {
                  Statement stmt = connection.createStatement();
                  ResultSet rs = stmt.executeQuery("SELECT * FROM phonebook");
                  while (rs.next()) {
                    ls.add(new PhonebookEntry(rs.getLong("id"),rs.getString("name"), rs.getString("phone")));
                  }
                } catch (Exception e) {   }
            return ls;
    }
```

Проверить работоспособность этого метода (но только после деплоя на PAAS) можно вызвав

```
heroku open /read
```

4. Добавить метод DELETE для удаление записи из телефонной книги:

```
    @RequestMapping(method = RequestMethod.DELETE, value = "/delete")
    public Boolean delete(@RequestBody PhonebookEntry pb) {
            try (Connection connection = dataSource.getConnection()) {
                  PreparedStatement stmt = connection.prepareStatement("delete from phonebook where id=?");
                  stmt.setLong(1, pb.id);
                  int num = stmt.executeUpdate();
                            return num > 0;
                } catch (Exception e) { }
            return false;
    }
```

5. И последний метод POST из набора CRUD - обновляет запись в БД:

```
    @RequestMapping(method = RequestMethod.POST, value = "/update")
    public Boolean update(@RequestBody PhonebookEntry pb) {
            try (Connection connection = dataSource.getConnection()) {
                  PreparedStatement stmt = connection.prepareStatement("UPDATE phonebook SET name = ?,phone = ? WHERE id=?");
                  stmt.setString(1, pb.name);
                  stmt.setString(2, pb.phone);
                  stmt.setLong(3, pb.id);
                  int num = stmt.executeUpdate();
                            return num > 0;
                } catch (Exception e) { }
            return false;
    }
```

6. Важно после всех изменений кода не забывать отправит на сервер новую сборку проекта

```
git add .
git commit -m "first commit"
git push heroku main
```

На этом создание серверной части приложение завершено. После отправки программного кода на сервер, нужно предварительно убедиться в его работоспособности вызвав GET запросы при помощи браузера, а также просмотрев логи сервера.

```
heroku logs
```

Обратите внимание, что в рассматриваемом примере, в функциях, связанных с HTTP запросами PUT, POST и DELETE, передаваемые параметры функций аннотировались **@RequestBody**, так как предполагалась передача объекта сложного типа, сериализованного в JSON, в теле запроса. Однако это не единственный вариант передачи параметров. Например в GET запросе, простые типы данных обычно передают в URI, что указывается аннотацией **@RequestParam**. Кроме того существуют еще множество аннотаций, например часто используются:

- @RequestParam
- @RequestBody
- @PathVariable
- @RequestHeader
- @CookieValue

Документацию по Spring Web можно посмотреть на сайте документации

Клиент

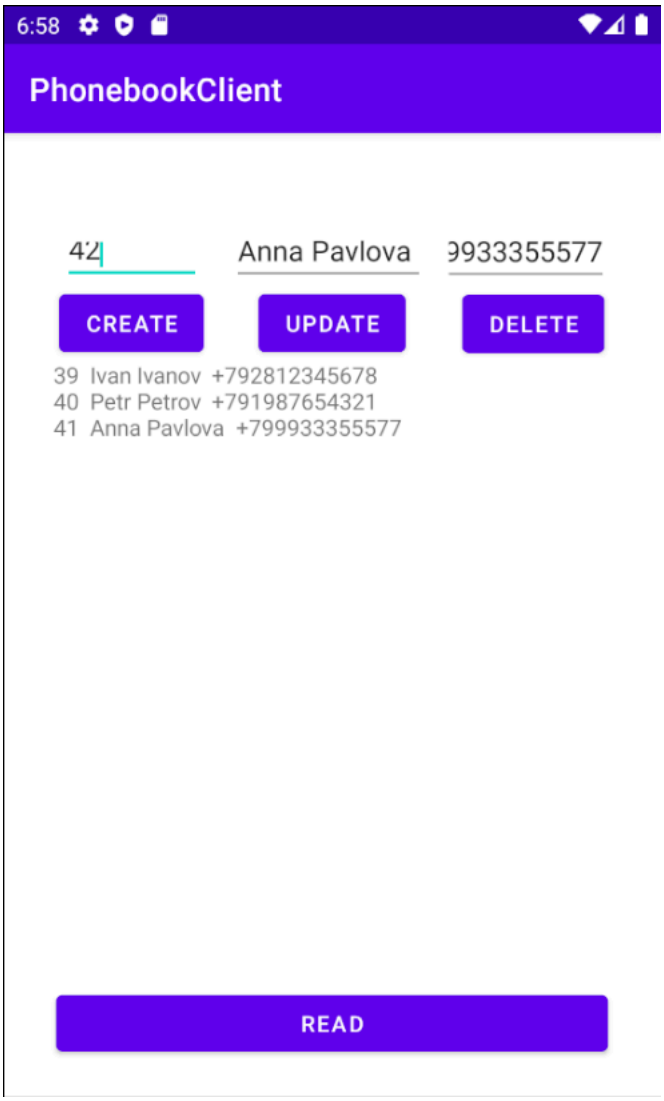
Теперь необходимо разработать программу на Android, которая будет взаимодействовать с серверной частью чтобы выполнить тот же полный цикл CRUD:

- **C**reate - добавить запись в телефонную книгу (имя, телефон),
- **R**ead - вывести на экран записи из книги (id, имя, телефон),
- **U**pdate - изменить запись в книге (id, имя, телефон),
- **D**elete - удалить запись из телефонной книги (id).

Соответственно на интерфейсе должны быть предусмотрены:

- 4е кнопки для вызова операций,
- 3и поля ввода для внесения id, имени и телефона,
- а также одно поле для вывода списка телефонов из книги.

Дизайн макета приложения может выглядеть например так:



Для разработки клиентской Android программы воспользуемся фреймворком Retrofit. Для его подключения в проект необходимо в файл build.gradle (app), в раздел dependencies добавить следующие строки (не забываем нажать sync now):

```
    implementation 'com.google.code.gson:gson:2.8.5'
    implementation 'com.squareup.retrofit2:retrofit:2.4.0'
    implementation 'com.squareup.retrofit2:converter-gson:2.4.0'
```

Ну и конечно не забываем добавить в манифест разрешение на работу с сетью:

```
 <uses-permission android:name="android.permission.INTERNET"/>
```

Прежде чем приступать к разработке основного кода приложения рассмотрим код класса для хранения и передачи записи из записной книжке:

```
class PhonebookEntry (var id:Long, var  name:String, var phone:String)
```

Важно отметить, что он должен полностью соответствовать аналогичному классу серверного кода.

Вторая простая, но важная часть описания клиент-серверного взаимодействия - это интерфейс, описывающий веб (или можно сказать REST) API серверного кода. Он содержит описание функций, которые далее можно будет вызывать для выполнения HTTP запросов к серверу, но также там содержаться и аннотации, благодаря которым Retrofit сможет реализовать эти методы интерфейса, а также укажет ему какие URI соответсвуют какому методу, тип метода, а также набор и размещение параметров в HTTP запросе.

```
interface PhonebookController {
    @PUT("/create")
    fun create(@Body entry:PhonebookEntry): Call<Boolean>
    @GET("/read")
    fun read(): Call<List<PhonebookEntry>>
    @POST("/update")
    fun update(@Body entry:PhonebookEntry): Call<Boolean>
//    @DELETE(value = "/delete")
    @HTTP(method = "DELETE", path = "/delete", hasBody = true)
    fun delete(@Body entry:PhonebookEntry): Call<Boolean>
}
```

Для того чтобы не загромождать главу, не будем здесь рассматривать код макета активности - ее можно посмотреть в приложенном примере практической работы. Упомянем лишь, что там присутствуют все необходимые кнопки, поля ввода, и поле для вывода ответа от сервера. Ниже приведен кусок кода активности, в которой инициализируются соответствующие переменные, но самое главное создается объект retrofit и объект сервис по представленному выше интерфейсу:

```
class MainActivity : AppCompatActivity() {

    // здесь нужно указать URL проекта на Heroku
    val HEROKU_URL: String = "https://floating-anchorage-97173.herokuapp.com/"

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val retrofit: Retrofit = Retrofit.Builder().baseUrl(HEROKU_URL)
            .addConverterFactory(GsonConverterFactory.create()).build();
        val service: PhonebookController = retrofit.create(PhonebookController::class.java);
        val id: EditText = findViewById(R.id.id)
        val name: EditText = findViewById(R.id.name)
        val phone: EditText = findViewById(R.id.phone)
        val result: TextView = findViewById(R.id.result)
        val create: Button = findViewById(R.id.create)
        val update: Button = findViewById(R.id.update)
        val delete: Button = findViewById(R.id.delete)
        val read: Button = findViewById(R.id.read)

        // Здесь еще будет код для обработки
        // нажатий на кнопки и обмен объектами
        // с сервером

        }
}
```

Ну и в завершение, рассмотрим фрагменты кода, котороые будут вызываться при нажатии соответствующих кнопок и будут реализовать функционал CRUD:

- Добавление записи в телефонную книгу

```
    create.setOnClickListener({
        service.create(PhonebookEntry(0,name.text.toString(),phone.text.toString())).enqueue(object: Callback<Boolean>{
            override fun onResponse(call: Call<Boolean>, response: Response<Boolean>) {
                result.text=""+response.body()
            }
            override fun onFailure(call: Call<Boolean>, t: Throwable) { }
        })
    })
```

- Вывод на экран списка телефонов

```
    read.setOnClickListener({
        service.read().enqueue(object: Callback<List<PhonebookEntry>>{
            override fun onResponse(call: Call<List<PhonebookEntry>>, response: Response<List<PhonebookEntry>>) {
                val list=response.body()
                result.text=""
                for (entry in list.orEmpty()) {
                    result.text=result.text.toString() + entry.id+"\t"+entry.name+"\t"+entry.phone+"\n";
                }
            }
            override fun onFailure(call: Call<List<PhonebookEntry>>, t: Throwable) {}
        })
    })
```

- Изменение записи в телефонной книге

```
    update.setOnClickListener({
service.update(PhonebookEntry(id.text.toString().toLong(),name.text.toString(),phone.text.toString())).enqueue(object: Callback<Boolean>{
            override fun onResponse(call: Call<Boolean>, response: Response<Boolean>) {
                result.text=""+response.body()
            }
            override fun onFailure(call: Call<Boolean>, t: Throwable) { }
        })
    })
```

- Удаление записи из телефонной книги

```
    delete.setOnClickListener({
service.delete(PhonebookEntry(id.text.toString().toLong(),name.text.toString(),phone.text.toString())).enqueue(object: Callback<Boolean>{
            override fun onResponse(call: Call<Boolean>, response: Response<Boolean>) {
                result.text=""+response.body()
            }
            override fun onFailure(call: Call<Boolean>, t: Throwable) { }
        })
    })
```

По окончанию разработки приложение должно демонстрировать простейший пример полного цикла клиент-серверного взаимодействия. Поскольку REST взаимодействие стало де-факто стандартом в современной разработке, то этот пример может стать основой для реализации кросс-платформенной версии приложения. Например в качестве клиента может выступать набор HTML страниц размещенных на сервере или приложение на C# и т.д.

Начать тур для пользователя на этой странице