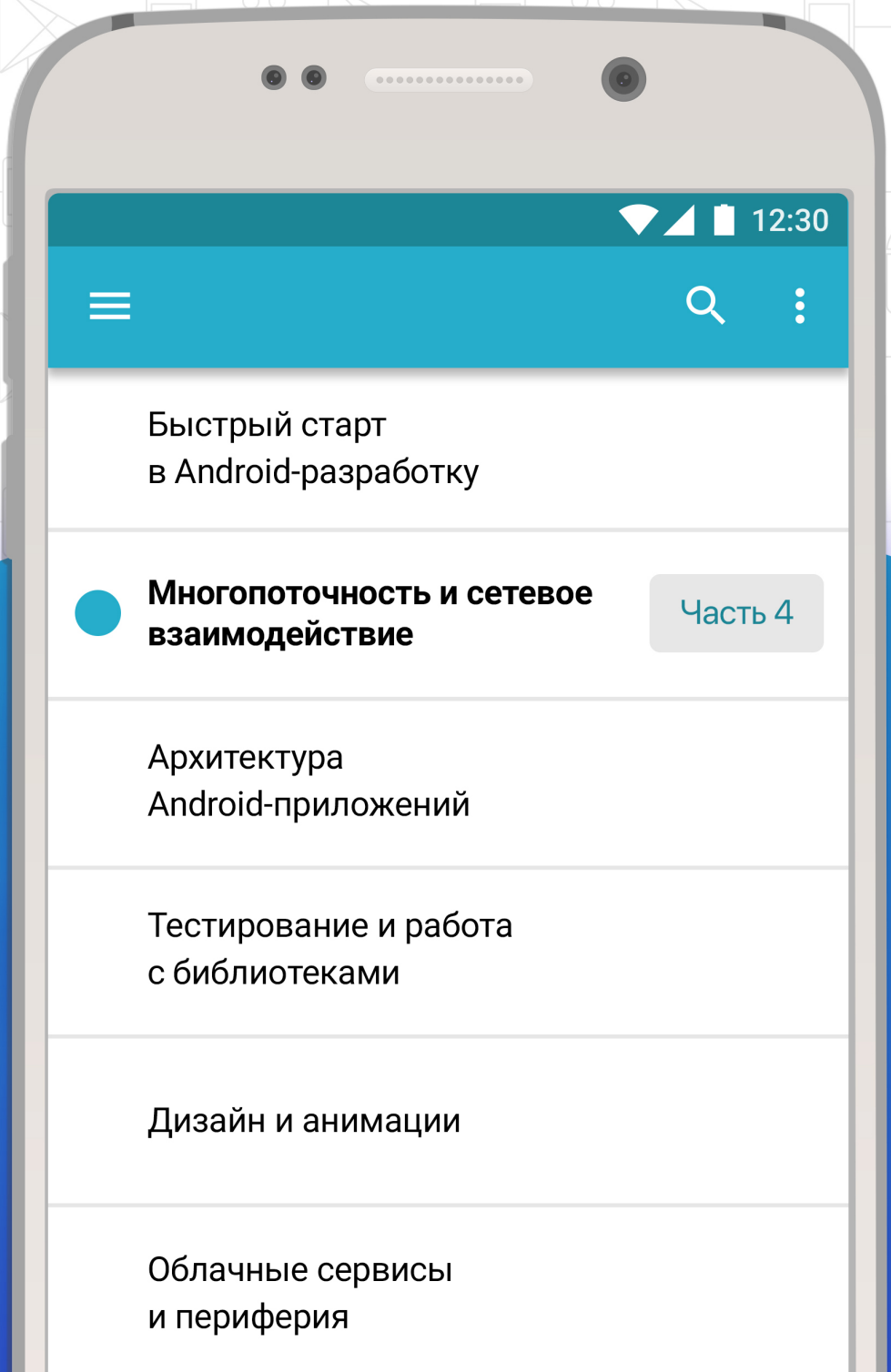
фонд развития
онлайн образования
eldf.ru

e·legion

academy.e-legion.com

Программа Android-разработчик

Конспект



Оглавление

1 НЕДЕЛЯ 4 **2**

1.1 Первые шаги к тому, чтобы быть реактивным 2

1.1.1 Знакомство с реактивным программированием 2

1.1.2 Знакомство с RxJava2 6

1.1.3 Инициализация RxJava2. Добавление Call Adapter. Обновление регистрации 17

1.1.4 Получение альбомов с сервера с помощью RxJava2 18

1.1.5 Обновление проекта для работы с объектами без ”data” обертки 19

1.2 Добавление БД и комментариев 20

1.2.1 Объединение БД модели и view модели альбома в одну сущность 20

1.2.2 Логика сохранения и получения альбомов с помощью Room 21

1.2.3 Обзорное видео по курсовому проекту 23

1.2.4 Обзорное видео по курсу 23

Глава 1

НЕДЕЛЯ 4

1.1. Первые шаги к тому, чтобы быть реактивным

1.1.1. Знакомство с реактивным программированием

Наверняка многие из вас уже слышали такое слово, как «реактивщина» или реактивное программирование, и также большинство из вас не знает, что это. Давайте это исправлять. Становимся на путь реактивный!

Реактивное программирование. Для чего?

Суть реактивного подхода заключается в том, чтобы рассматривать абсолютно все данные как поток/потоки. Т. е. должна быть какая-то инстанция, которая является источником данных(поток), которая будет оповещать нас о том, что происходит с данными, а мы на основе этой информации решаем, что с ними делать.

Чтобы вы не запутались, под потоками подразумевается Streams, а не Threads. Это два разных понятия. Stream – поток данных, Thread – в языке Java поток представляется в виде объекта-потомка класса Thread.

Вам предоставляется возможность создавать потоки чего-либо. Потоки легковесны и используются практически везде:

- Переменные;

- Пользовательский ввод;
- Свойства;
- Кэш;
- Структуры данных;
- и т. д.

Например, лента в Instagram может быть потоком данных, так же как и события пользовательского интерфейса. То есть можно слушать поток и реагировать на события в нем. Помимо того, что вы можете просто слушать поток, вы также можете творить с ним все, что захотите: изменять, добавлять, вырезать, фильтровать, преобразовывать данные в другой тип и т.д. Реализуется это все с помощью функций, которые работают с потоком.

Функции для работы с потоком предоставляет инструмент (библиотека), который реализовал этот реактивный подход. Известных реализаций несколько штук:

- ReactJS – созданная в Facebook JavaScript-библиотека разработки пользовательских интерфейсов;
- Bacon.js – небольшая функциональная библиотека для JavaScript;
- ReactiveX – мультиплатформенная реализация FRP для Java, JS, C#, Scala, Clojure, Swift и др.
- и др.

Мы же с вами в будущем будем использовать реализацию от ReactiveX.

Реактивное программирование. В чем плюсы?

Кстати, в нашем случае мы будем рассматривать не чисто реактивное программирование, а объектно-ориентированное реактивное программирование или объектно-реактивное программирование (ОРП). Согласен, звучит необычно.

Если вы не хотите...	**То в реактивном подходе...**
...писать много, а делать мало	...пиши мало, делай много!
...часами дебажить простую логику	...обеспечение связи данных друг с другом!
...чтобы все по умолчанию тупило	...оптимизация потоков данных по умолчанию!
...чтобы приложение падало целиком и полностью	...никакой exception не пройдет мимо!
...проводить махинации с индикаторами ожидания	...индикаторы ожидания сами появляются, где надо!
...создавать переиспользуемые компоненты	...компоненты будут переиспользуемыми по умолчанию!

Реактивный подход vs стандартный подход. Теория

Есть два варианта получить данные.

1. Pull – когда мы сами делаем запрос на получение и нам приходит ответ, который мы обрабатываем;
2. Push – когда поток сам уведомляет нас об изменениях и отправляет нам данные.

Реактивное приложение – приложение, которое само извещает нас об изменении своего состояния. Не мы делаем запрос и проверяем, а не изменилось ли там что-то, а приложение само нам сигнализирует. Эти события и эти сигналы мы можем обрабатывать, так как захотим.

Пример. Мы можем взять обычную коллекцию, преобразовать ее в реактивную, и тогда мы будем иметь коллекцию событий об изменении данных в ней. Мы очень просто получаем только те данные, которые изменились. По такой коллекции мы можем делать выборку, фильтровать ее и т.д. Если бы мы это делали традиционным способом, то нам нужно было бы закэшировать (временно сохранить) текущие данные, потом делать запрос на получение новых данных, потом сравнить их с кэшем и разница будет равна этим изменениям.

Реактивный подход vs стандартный подход. Практика

Представим ситуацию, что нам нужно из БД получать 300 фото, а после получения отображать их на экране.

Стандартный подход:

1. Делаем запрос к БД;

2. Ждем 2 секунды;
3. Получаем массив из 300 фото;
4. Пробегаемся по массиву и выводим каждое фото на экран.

```
photos = getPhotosFromDatabase();
for(Photo p : photos)  {
    showPhoto(p);
}
```

Реактивный подход:

1. Делаем запрос к БД;
2. Передаем в него callback, который будет обрабатывать каждый последующий элемент массива.

```
photos = getPhotosFromDatabase(p -> showPhoto(p));
```

В первом случае мы сами явно вытягивали следующий элемент списка (pull), а во втором случае источник данных сам давал нам следующий элемент, когда он был готов (push).

В первом случае мы ждем, пока сформируется источник данных (как правило занимаем тред) и после этого сами ручками просматриваем результат, во втором случае источник данных сам нас уведомит, когда будет готов.

Что это нам дает?

- Асинхронность – в UI это дает отзывчивость.
- Масштабируемость – источник данных (коллекция фото) и приемник(showPhoto(p)) не связаны. Отсутствие связи дает нам возможность подключить хоть 10 обработчиков фото. Например, один выводит в черно-белом, а другой накладывает сепию.
- Отказоустойчивость достигается тем, что при возникновении ошибки в работе потока приложение не упадет, а выведет в логи нашу ошибку.

- Реактивность дает слабую связанность.
- В некоторых случаях это дает возможность писать более простой и понятный код.
- Отсутствие callback hell – ситуации в императивном подходе, когда каждое новое действие должно оборачиваться в свой колбек.

Дополнительная информация:

1. https://tproger.ru/translations/reactive-programming/
2. https://habrahabr.ru/post/279715/
3. https://habrahabr.ru/post/140719/
4. http://www.pvsm.ru/cat/reaktivnoe-programmirovanie
5. https://ru.wikipedia.org/?oldid=91682928

1.1.2. Знакомство с RxJava2

Цель данного материала заключается в том, чтобы лишь познакомить и научить вас самым основам реактивного программирования, а не сделать из вас гуру реактивщины. Глубже познакомиться с RxJava2 вы сможете самостоятельно и по ходу прохождения программы.

Немного об RxJava2

Мы будем использовать решение от ReactiveX – RxJava2. Именно вторую версию, т.к. она очень сильно отличается от первой. Первый релиз RxJava2 вышел в октябре 2016, все новые проекты создаются именно на ней. Разумеется, концепции RxJava1 и RxJava2 схожи, но реализации отличаются достаточно сильно.

RxJava2 была полностью переписана с нуля, поверх спецификации Reactive Streams. Сама спецификация эволюционировала из RxJava1 и обеспечивает общую базу для реактивных систем и библиотек. Поскольку Reactive Streams имеет другую архитектуру, она предусматривает изменения некоторых известных типов RxJava.

Если интересно, в чем именно различия первой и второй версии, то можете посмотреть здесь: https://github.com/ReactiveX/RxJava/wiki/What%27s-different-in-2.0.

Подходы в реактивном программировании в web-разработке значительно отличаются от подходов в Android-разработке, мы с вами будем рассматривать именно Android.

RxJava2. Введение

Основными классами для работы реактивного кода в связке с RxJava2 являются Observable, Flowable и Observer, Subscriber. Observable и Flowable являются источниками данных, а Observer и Subscriber – приемниками. Далее я буду называть Observable и Flowable – источником, а Observer и Subscriber – приемниками, чтобы постоянно не употреблять пары этих названий. На самом деле источников намного больше, просто Observable и Flowable являются самыми часто используемыми.

Список всех источников:

Тип	Описание
Flowable<T>	Излучает 0 или n элементов и завершается с успехом или ошибкой. Поддерживает backpressure, с его помощью можно контролировать, как быстро источник испускает элементы.
Observable<T>	Излучает 0 или n элементов и завершается с успехом или ошибкой. Не поддерживает backpressure.
Single<T>	Излучает либо один элемент, либо событие ошибки. Реактивная версия вызова метода.
Maybe<T>	Успешно с о элементом, или без элемента, или без ошибок. Реактивная версия типа Optional.
Completable	Либо завершается с успехом, либо с событием ошибки. Он никогда не излучает элементы. Реактивная версия типа Runnable.

RxJava2. Реактивная схема данных

Порождение данных через источник, когда у него есть приемник, всегда происходит в одном и том же порядке:

1. Приемник подписывается на источник;
2. Источник «излучает» некоторое количество данных (может ничего не излучать);

3. Для каждого приемника, который подписан на источник, вызывается метод onNext() для каждого элемента потока данных. Т.е. каждый раз, когда источник «излучает» данные;
4. Финальные действия:
   (a) Если во время «излучения» данных произошла ошибка, то приемник завершает свою работу с ошибкой – onError();
   (b) Если ошибки не было, то завершает свою работу успешно – onComplete().

RxJava2. Источники

Вышеупомянутая схема напоминает паттерн Observer (Наблюдатель). У нас есть что-то, что может генерировать данные и есть описание того, как должны выглядеть эти данные. И мы хотим наблюдать за ними. Мы хотим добавить слушателя и получать уведомления, когда что-то происходит.

Но есть одно важное но: есть такие источники, которые не начинают порождать данные до тех пор, пока кто-нибудь явно не подписывается на них. Ведь если вы будете кричать, а рядом никого не будет, то кто услышит, что вы кричите?

Такие источники разделяются на два типа:

- Горячие(hot) – будут постоянно работать и излучать данные;
- Холодные(cold) – будут отдавать данные только тогда, когда у них есть хотя бы один подписчик.

Подробнее об этом: https://github.com/Froussios/Intro-To-RxJava/blob/master/Part%203%20-%20Taming%20the%20sequence/6.%20Hot%20and%20Cold%20observables.md

Источники могут быть пустыми. Это концепция источника данных, который не содержит никаких элементов и работа которого либо успешно выполняется, либо завершается сбоем. Представьте, что вы записываете данные в базу данных или в файл. Они не возвращают вам элементы. Запись либо успешна, либо нет. В RxJava источники моделируют этот подход «выполнения или отказа» с помощью onComplete() и onError(). Это аналогично методу, который либо возвращает ответ, либо бросает исключение.

Завершения может и не быть. К примеру, вы двигаете мышь по столу, курсор двигается вслед за ней. И для курсора нет какого-то окончательного последнего значения. В любой следующий момент времени мышь может быть сдвинута.

Источник может работать как синхронно, так и асинхронно. Например, блокирующий сетевой запрос, выполняющийся в фоновом потоке, либо что-то чисто асинхронное, вроде обращения к Android и ожидания onActivityResult. Источник может выдавать один или несколько элементов, это зависит от того, как вы настроите работу источника. Сетевой запрос вернет один ответ. Но пока работает ваш UI, поток нажатий на кнопки потенциально бесконечен, даже если вы подписаны на единственную кнопку.

RxJava2. Observable vs Flowable. Backpressure

Observable и Flowable устроены одинаково, но есть такая «замечательная» вещь как backpressure. Backpressure – это ситуация, когда источник выдает данные слишком быстро, а приемник не успевает их обрабатывать, из-за этого чем дольше работает поток, тем больше данных ждут своей обработки. Они копятся, копятся, копятся и в конце концов выбрасывается ошибка.

Самый простой пример – источник выдает каждые 100 мс случайное пятизначное число, а приемник вычисляет из него факториал. Понятное дело, что факториал за 100 мс не посчитать. Происходит backpressure.

Способов борьбы много. Можно обрабатывать каждый n-ный элемент, или обрабатывать каждый последний элемент спустя какой-то промежуток времени. Пока что, акцентироваться на этом не будем, при желании можете прочитать подробнее из других источников.

В RxJava2 просто вырезали поддержку backpressure из Observable и создали Flowable, в котором эта поддержка есть.

Подробнее о backpressure: https://habrahabr.ru/post/336268/

RxJava2. Observable. Когда применять?

Observable стоит использовать в тех случаях, когда:

- У вас за один раз передается не более 1000 элементов и это самый пессимистичный сценарий;

- Вы считаете, что в вашем случае при работе с данными никогда не выскочит OutOfMemory Exception;
- Вам не требуется backpressure.

Такой вариант подходит для работы с UI, отслеживание скролла, нажатие на кнопки и другие UI-ные ивенты и для работы с не очень большим количеством данных.

RxJava2. Flowable. Когда применять?

Flowable стоит использовать, когда нужно:

- Работать с таким количеством данных, что устройство просто не способно с ним справиться;
- Генерировать данные;
- Парсить (сериализация/десериализация любых форматов данных);
- Работать с Internal и External storage;
- Делать сетевые запросы;
- Работать локально с БД;
- Делать любую вещь, которая будет блочить ваш UI поток.

RxJava2. Observer и Subscriber

Observer используется только вместе с Observable, а Subscriber используется только вместе с Flowable.

Так выглядит интерфейс Observer:

```
1  public interface Observer<T> {
2      void onSubscribe(@NonNull Disposable d);
3
```

```
4       void onNext(@NonNull T t);
5
6       void onError(@NonNull Throwable e);
7
8       void onComplete();
9   }
```

А так Subscriber:

```
1   public interface Subscriber<T> {
2       void onSubscribe(@NonNull Subscription s);
3
4       void onNext(@NonNull T t);
5
6       void onError(@NonNull Throwable e);
7
8       void onComplete();
9   }
```

В Observer.onSubscribe(Disposable d) тип Disposable позволяет вызывать метод dispose, означающий «Я закончил работать с этим ресурсом, мне больше не нужны данные». Если у вас есть сетевой запрос, то он может быть отменен. Если вы прослушивали бесконечный поток нажатий кнопок, то это будет означать, что вы больше не хотите получать эти события, в таком случае можно удалить OnClickListener у View.

Все это работает и в Subscription. Вместо метода dispose() у него есть метод cancel(). И в нем также есть второй метод request(long r), с помощью которого backpressure проявляется в API. Через этот метод мы сообщаем Flowable, о том, что нам нужно больше элементов.

RxJava2. Немного практики

Если в примерах показывается Observable, то все это применимо и к Flowable.

Для начала добавим RxJava2 в наш проект:

```
implementation 'io.reactivex.rxjava2:rxjava:2.1.8'
```

И RxAndroid:

```
implementation 'io.reactivex.rxjava2:rxandroid:2.0.1'
```

RxAndroid – это расширение RxJava, созданное специально для Android, в котором есть определенные механизмы, которые упрощают работу с RxJava в Android.

Создать источник можно несколькими способами:

- Observable.just(”Привет!”) – создает обертку над другими типами данных;
- Observable.fromIterable() – принимает java.lang.Iterable<T> и излучает свои значения в порядке их расположения в структуре данных;
- Observable.fromArray() – принимает массив и испускает свои значения в порядке их расположения в структуре данных;
- Observable.fromCallable() – позволяет создавать источник для java.util.concurrent.Callable<V>;
- Observable.fromFuture() – позволяет создавать источник для java.util.concurrent.Future;
- Observable.interval() – создает источник, который излучает Long объекты в данном интервале;

Аналогичные методы существуют для других источников: Flowable, Maybe и Single.

Подписаться на источник также можно несколькими способами:

- subscribe() – возвращает Disposable;
- subscribe(Consumer<? super T> onNext) – с передачей Consumer для обработки результатов работы источника, возвращает Disposable;
- subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError) – как предыдущий, но добавлен Consumer для ошибки, возвращает Disposable;

- subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError, Action onComplete) – как предыдущий, но добавлен Action для оповещения о завершении работы источника, возвращает Disposable;
- subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError, Action onComplete, Consumer<? super Disposable> onSubscribe) – как предыдущий, но добавлен Consumer для оповещения о том, что наш приемник успешно подписался на источник, возвращает Disposable;
- subscribe(Observer/Subscriber observer) – подписка с передачей в него приемника данных, ничего не возвращает;
- subscribeWith(Observer/Subscriber observer) – как предыдущий, но возвращает Observer/Subscriber;
- blockingSubscribe() – такой же метод как и все subscribe(), описанные выше, но.

Мы можем указать, в каких потоках должен выполнять свою работу источник. Делается это так:

```
observable
.subscribeOn(Schedulers.io()) //указываем пул потоков в котором выполняется работа
.observeOn(AndroidSchedulers.mainThread()) //указываем пул потоков в котором
    ↪ вызываются onSubscribe, onNext, onError, onComplete
```

RxJava2. Пример работы с источником

Есть метод API, который возвращает Flowable со списком городов:

```
@GET("cities/russia/")
Flowable<List<City>> getCities();
```

Допустим, что после получения наших данных от сервера нам нужно:

- Чтобы источник не отдавал данные, пока у него не будет хотя бы одного подписчика;
- Отобрать из них те, которые начинаются на букву «D»;

- Отсортировать в нужном нам порядке;
- В каждый объект города добавить локальный ID;
- Выдавать их не всем списком, а порциями по 10 штук;
- Создать приемник и подписаться на источник.

Создадим получим источник и обработаем данные, которые он излучает:

```
public Flowable<List<City>> getFilteredCities() {
  //defer - метод, который начинает работу источника,
  //только тогда, когда у него есть хотя бы один подписчик
  return Flowable.defer(new Callable<Publisher<? extends List<City>>>() {
     @Override
     public Publisher<? extends List<City>> call() throws Exception {
return mApi.getCities()
        .flatMap(new Function<List<City>, Publisher<City>>() {
                @Override
                public Publisher<City> apply(List<City> cities) throws Exception {
                        //превращаем поток списков городов в поток городов
                        return Flowable.fromIterable(cities);
                }
        })
        .filter(new Predicate<City>() {
                @Override
                public boolean test(City city) throws Exception {
                        //отбираем те города, которые начинаются на букву "D"
                        return city.getName().startsWith("D");
                }
        })
        .map(new Function<City, City>() {
                @Override
                public City apply(City city) throws Exception {
                        //добавляем локальный id в каждый объект города
                        city.setId(UUID.randomUUID().toString());
                        return city;
                }
        })
        .sorted(new Comparator<City>() {
```

```
                @Override
                public int compare(City city1, City city2) {
                        //сортируем города в алфавитном порядке по названиям
                        return city1.getName().compareTo(city2.getName());
                }
        })
        //превращаем поток городов в поток списков, которые хранят в себе по 10
         ↪  городов
        .buffer(10);
    }
  });
}
```

То же самое, но с использованием лямбда-выражений:

```
public Flowable<List<City>> getFilteredCities() {
  //defer - метод, который начинает работу источника,
  //только тогда, когда у него есть хотя бы один подписчик
   return Flowable.defer(() -> mApi.getCities()
        //превращаем поток списков городов в поток городов
        .flatMap(Flowable::fromIterable)

        //отбираем те города, которые начинаются на букву "D"
        .filter(city -> city.getName().startsWith("D"))

        //добавляем локальный id в каждый объект города
        .map(city -> {
                city.setId(UUID.randomUUID().toString());
                return city;
        })

        //сортируем города в алфавитном порядке по названиям
        .sorted((city1, city2) -> city1.getName().compareTo(city2.getName()))

        //превращаем поток городов в поток списков, которые хранят в себе по 10
         ↪  городов
        .buffer(10));
}
```

Теперь на источник можно подписаться и получить из него данные. Создадим приемник и подпишемся на наш источник:

```
getFilteredCities()
.subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Subscriber<List<City>>() {
                @Override
                public void onSubscribe(Subscription s) {
                        //todo обработка новой подписки
                }
                @Override
                public void onNext(List<City> cities) {
                        showCities(cities);
                }
                @Override
                public void onError(Throwable t) {
                        //todo обработка ошибки
                }
                @Override
                public void onComplete() {
                        //todo обработка завершения работы потока
                }
        });
```

Список часто используемых операторов для работы над данными потоками: map, defer, zip, take, reduce, flatMap, filter, buffer, skip, merge, concat, replay и многие другие.

Обязательно к прочтению, если вы хотите стать тем, кто будет знать больше нас:

1. Официальная wiki
2. Список всех операторов
3. Отличные примеры работы с RxJava2
4. Разница между RxJava1 и RxJava2
5. Отличный туториал
6. RxJava1:

(a) Грокаем RxJava. 1 часть
(b) Грокаем RxJava. 2 часть
(c) Грокаем RxJava. 3 часть
(d) Грокаем RxJava. 4 часть
(e) Почему следует использовать RxJava
(f) Хороший туториал с подробным разбором

7. RxJava2:

(a) Коротенькая статья о некоторых особенностях RxJava
(b) Отлично расписано о backpressure
(c) Неплохое сравнение RxJava1 и RxJava2

1.1.3. Инициализация RxJava2. Добавление Call Adapter. Обновление регистрации

В данном занятии мы переделаем регистрацию так, чтобы она работала с помощью RxJava2. Перейдем в проект. Откроем build.gradle(Module: app). Добавим три зависимости. Это adapter-rxjava2:2.3.0, rxjava2:rxjava:2.1.6, rxjava2:rxandroid:2.0.1. Далее перейдем в ApiUtils, добавим .add CallAdapterFactory() в Instance Retrofit'a. В него передадим RxJava2CallAdapterFactory.create(). Это нужно, чтобы наши ретрофитовские запросы и response'ы изменялись в сущности RxJava2. Теперь перейдем в AcademyApi. Call<Void> изменим на Completable, нажмем Enter и перейдем в RegistrationFragment. Здесь в mOnRegistrationClickListener сделаем следующее. enqueue у нас больше не работает, так как registration возвращает completable. И все верно. Теперь:

```
1  .subscribeOn(Schedulers.io())
2  .observeOn(AndroidSchedulers.mainThread())
```

Сделаем

```
1  .subscribe()
```

в который передадим onComplete и onError. Соответственно это будет new Action, в котором будет onComplete, и будет new Consumer<Throwable>. Это будет наш onError. В onComplete

передадим, добавим следующий код. То есть что будет, когда нам вернется успешный запрос. Если к нам придет ошибка, то выполню вот этот код, но уже без main handler'a, так как наш onComplete и onError будет работать в main потоке. Сотрем весь код с enqueue.

Переделаем на лямбды, чтобы это было более читабельно. Вот и все. Теперь запустим эмулятор и посмотрим, ничего ли мы не сломали. Нажмем «зарегистрироваться». Введем какой-нибудь email. Введем какое-нибудь имя. Введем наш любимый пароль. 12345678. Введем наш любимый пароль еще раз. И нажмем «зарегистрироваться». Как вы видите регистрация прошла успешно, а значит мы ничего не сломали.

В данном занятии мы переделали регистрацию на RxJava2.

1.1.4. Получение альбомов с сервера с помощью RxJava2

В данном занятии мы с вами переделаем получение альбомов с Retrofit2 на RxJava2, чтобы вы лучше понимали всю прелесть реактивного программирования. Теперь перейдем в проект, откроем AlbumsFragment и откроем AcademyApi. В методе getAlbums переделаем возвращаемый тип на Single<Albums>. Далее перейдем в AlbumsFragment и, поскольку getAlbums возвращает Single, сделаем следующее:

```
1  .subscribeOn(schedulers.io())
2  .observeOn(AndroidSchedules.mainThread())
3  Далее делаем
4  .subscribe(new Consumer<Albums>(){
5  ...
6  }, new Consumer<Throwable>() {
7  ...
8  })
```

Копируем код onResponse successful. Вставляем его в соответствующий метод. И то же делаем с onFailure. Далее копируем setRefreshing(false) и вставим его сюда, чтобы у нас останавливался показ загрузки. Далее стираем весь enqueue, он нам больше не пригодится, переделаем на лямбды. Добавим следующую вещь:

```
1  .doOnSubscribe(new Consumer<Disposable>() {
2  ...
3  })
```

и передадим в него

```
mRefresher.setRefreshing(true);
```

Переделаем опять-таки на лямбды. Как мы видим, у нас в каждом методе есть setRefreshing(false). Давайте избавимся от этого. Добавим метод

```
.doFinally(new Action() {
...
}
```

в который мы будем передавать Action, в котором мы будем останавливать показ загрузки. И сотрем строчку isOnError. Далее переделаем на лямбду. response.body().getData() у нас теперь больше нет. Теперь вместо response.body нам приходит Albums. Вот и все.

Давайте посмотрим, как это работает на эмуляторе. Введем логин. Введем пароль. Жмем «войти». Как мы видим, все альбомы успешно загрузились, значит, мы ничего не сломали. В данном занятии мы успешно переделали получение альбомов с Retrofit2 на RxJava2.

1.1.5. Обновление проекта для работы с объектами без ”data” обертки

В данном занятии мы избавимся от data-обертки, которая возвращается с сервера. Ведь вы согласны с тем, что неудобно внутри приложения постоянно использовать DataBean’ы. И мы это сейчас исправим. Но сначала разберемся с тем, что мы уже добавили в проект.

Откроем проект. Перейдем в module, класс Data. Этот класс по факту и есть наша обертка, от которой мы собираемся избавиться. Далее откроем package converter, DataConverterFactory. Эта фабрика на самом деле позволяет избавиться от data-обертки, без нее мы бы не смогли этого добиться. Также перейдем в ApiUtils и увидим, что здесь добавилась еще одна ConverterFactory, которую мы написали. Теперь перейдем в модель Album, стираем DataBean и все наши модели будут выглядеть таким образом. В Songs DataBean у нас тоже не будет, у нас будет просто Song. Здесь это мы тоже исправим.

По факту на данный момент мы с вами занимаемся рефакторингом приложения. Такое происхо-

дит достаточно часто, и это вполне нормальная тема. Класс Albums нам не нужен, теперь у нас будет возвращаться список альбомов. Перейдем в класс Song, также избавимся от DataBean'а. Songs нам не нужен. Как вы заметили, количество классов изменилось, но теперь давайте поправим весь код внутри приложения. Для этого сделаем build и увидим, что у нас есть ошибки. Давайте их исправлять.

Теперь у нас нет Albums, теперь у нас возвращается List<Album>. У нас нет Songs, у нас есть список песен. Вместо DataBean у нас теперь есть просто Song. И так далее мы работаем с каждой ошибкой. Изменим OnItemClick, он теперь тоже получает Album. Изменим тип списка, идем дальше по ошибкам. Убираем DataBean'ы. В принципе это достаточно монотонная работа, но она приносит свои плоды, в будущем нам будет гораздо проще работать с нашими объектами. И также это занимает достаточное количество времени. Теперь у нас нет метода getData, теперь мы просто можем сделать метод getSong.

Посмотрим, что у нас здесь, убираем DataBean. DataBean. DataBean. Где еще есть ошибки? Чтобы было проще, можно сделать build еще раз, и пройтись по списку ошибок целиком. Их всего две. put Serializable. Все верно. В Album теперь нужно добавить implements Serializable. И cannot find symbol method getData, его также сотрем. Сделаем build. Build прошел успешно.

Теперь давайте посмотрим, как работает наше приложение после маленького рефакторинга. Запустим эмулятор. Введем логин. Введем наш любимый пароль. Нажмем «войти». Обновимся для уверенности. Нажмем альбом. Обновимся еще раз, пролистаем список, обновимся, нажмем «назад». Все работает. В данном занятии мы успешно провели рефакторинг приложения и избавились от обертки DataBean.

1.2. Добавление БД и комментариев

1.2.1. Объединение БД модели и view модели альбома в одну сущность

В данном занятии мы подготовим наше приложение для того, чтобы работать с базой данных, но для начала давайте посмотрим, что мы уже добавили.

Перейдем в проект. java/com.elegion.myfirstapplication. Увидим, что мы добавили package db. Это точно такой же package для работы с базой данных, который был у Азрета. Он слегка упрощен и в нем нет работы с некоторыми сущностями. Зайдем в model, посмотрим AlbumDb. Как вы видите, он очень похож на Album, который есть в нашей модели. Я бы даже сказал, что он в

принципе идентичный. Перейдем в DataBase, getMusicDao и MusicDao. В них описана только работа с альбомами. Всю остальную работу вы будете делать в домашнем задании.

Далее перейдем в AlbumDb и будем копировать все аннотации, которые используются для работы с базой данных и переносить их в наш Album. Это нужно для того, чтобы объединить модель базы данных и модель, которую мы используем внутри приложения, чтобы у нас не было нескольких разных сущностей.

Согласитесь, неудобно будет использовать deserialiser'ы и serialiser'ы для того, чтобы работать с приложением. То есть в базе данных хранятся одни сущности, а используем внутри приложения мы другие. В нашем случае мы можем их объединить, поэтому именно этим мы и занимаемся. Далее скопируем аннотацию Entity, вставим ее в альбом, сам AlbumDb можно удалять. Нажмем OK. Так же как и стереть model. Теперь перейдем к DataBase и исправим на использование album. Перейдем в MusicDao и не забудем поправить все наименования таблиц. Сделаем build, чтобы убедиться, что все работает правильно. Столкнулись с ошибкой. Cannot figure out how to save this field into database.

Все верно. Для того, чтобы сохранять списки, нам нужна таблица связей. У нас ее не будет. Вы ее добавите самостоятельно в домашнем задании. А пока что мы добавим аннотацию ignore, чтобы избежать этой проблемы. Сделаем build еще раз. Вполне нормальная ошибка. Если у вас возникает такое, просто делаем build, clean project, ждем пока он очистится, делаем build. Как мы видим, наш проект успешно построился, а значит мы успешно объединили в себе модели базы данных и модель, которую мы используем внутри приложения.

1.2.2. Логика сохранения и получения альбомов с помощью Room

В данном занятии мы добавим взаимодействие с базой данных с помощью RxJava2 на экране всех альбомов. Для этого перейдем в проект, откроем AlbumsFragment и обязательно перед методом observeOn добавим следующий код:

```
1  .doOnSuccess(new Consumer<List<Album>>() {
2  ...
3  }
```

Далее нам нужно получить сущность базы данных. Для этого добавим

```
private MusicDao getMusicDao() {
        return ((App) getActivity().getApplication().getDatabase().getMusicDao();
}
```

Далее перейдем в doOnSuccess и в него добавим

```
getMusicDao.insertAlbums(albums);
```

То есть если с сервера успешно вернулся запрос, и мы получили наши альбомы, мы сохраняем их в базу данных. Идем дальше:

```
.onErrorReturn(new Function<Throwable,List<Album>>() {
...
}
```

Сделаем следующее:

```
if (ApiUtils.NETWORK_EXEPTIONS.contains(throwable.getClass())) {

}
```

Давайте посмотрим, что такое NETWORK_EXCEPTIONS. Это список всех exception'ов, которые часто встречаются при возникновении проблем с интернетом. Далее:

```
if (ApiUtils.NETWORK_EXCEPTIONS.contains(throwable.getClass())){
        return getMusicDao().getAlbums();
} else return null;
```

Ctrl+Alt+L. Вот и все. Мы успешно добавили логику взаимодействия с базой данных. Теперь давайте посмотрим, как это работает в эмуляторе. Запустим его. Введем логин. Введем пароль. Нажмем «войти». Наши альбомы успешно подгрузились с сервера. Теперь отключаем интернет. Подгружаем их еще раз. Как вы видите, наши данные успешно загрузились из базы данных. В данном занятии мы успешно добавили логику работы с базой данных с помощью RxJava2.

1.2.3. Обзорное видео по курсовому проекту

Вот и подошел к концу наш курсовой проект. В нем мы научились использовать пакет OkHttp3 и Retrofit2 для сетевого взаимодействия. Также мы научились работать с json'ом и библиотекой gson. Научились выводить данные с помощью RecyclerView. Познакомились с Adapter'ом и Holder'ом. Также мы немножечко познали реактивное программирование и всю его мощь. Познакомились с Room и в целом с базами данных. Написали достаточно неплохое приложение, которое можно добавить в свое портфолио, если сделать что-то аналогичное. Также помимо теоретических знаний мы постоянно подкрепляли их практическими. Мы добавляли вам задания, чтобы вы могли самостоятельно сделать что-то свое. Хорошего кодинга вам.

1.2.4. Обзорное видео по курсу

Поздравляю с окончанием второго блока! Давайте теперь остановимся, отдохнем и вспомним, чего мы достигли, чему мы научились. Мы освоили многопоток. Узнали несколько способов обработки тяжелых операций. Теперь вопрос из разряда «Какие способы обеспечения многопоточности на Android вы знаете?» нас больше не смущает. Мы освоили persistance и узнали про работу с файлами и базами данных, в частности, мы поработали с Room. Также мы поиграли с ContentProvider'ом, а это уже межпроцессное взаимодействие. Мы узнали основы работы со списками, с RecyclerView. Теперь самый распространенный экран в Android-приложениях подвластен нам. Мы научились сетевому взаимодействию с помощью библиотек OkHttp и Retrofit. Теперь мы в состоянии писать клиент-серверные приложения. Это ли не круто? И мы слегка копнули в сторону реактивщины. Уверяю, то, что было в этом курсе – это всего лишь вершина айсберга. Книги, статьи, видео, код – пробуйте, изучайте все. Что дальше? Дальше вам нужно, если вы этого еще не сделали, начать писать приложения для портфолио. Приложение должно быть клиент-серверным. Я понимаю, что выбрать сложно, поэтому мы предлагаем некоторые варианты на случай, если у вас вообще нет идей. Уделите этому приложению много времени. Шлифуйте как интерфейс, так и логику. Приложение не должно быть уникальным в своем роде, оно просто должно быть рабочим. Не бойтесь экспериментировать, подключите GitHub, пишите в новых ветках, вливайте, откатывайте изменения. Вы также можете собраться с сокурсниками вдвоем-втроем и писать одно, но большое приложение. Разделение обязанностей здорово подстегивает. Удачи. Ждем вас в третьем блоке.

О проекте
Академия e-Legion – это образовательная платформа для повышения квалификации в мобильной разработке. Слушайте лекции топовых разработчиков, выполняйте практические задания и прокачивайте свои скиллы. Получите высокооплачиваемую профессию – разработчик мобильных приложений.

Программа “Многопоточность и сетевое взаимодействие”

Блок 1. Обзор средств для обеспечения многопоточности

- Знакомство с курсом
- Многопоточность и параллельное программирование
- Обзор инструментов для обеспечения многопоточности в Java (Thread, Runnable, Callable, Future, Executors)
- Обзор инструментов для обеспечения многопоточности в Android (IntentService + BroadcastReceiver, HaMeR, AsyncTask, Loaders)
- Материалы для самостоятельного изучения
- Задание на программирование. Многопоточность.

Блок 2. Service + BroadcastReceiver

- Знакомство с Service, IntentService
- Создание Service
- Бродкастресивер, знакомство
- Создание BroadcastReceiver
- Связка Activity-Service-BroadcastReceiver-Activity
- Материалы для самостоятельного изучения
- Задание на программирование. Многопоточность.

Блок 3. Многопоточность в Android

- AsyncTask, знакомство
- AsyncTask, работа
- HaMeR
- Пример работы HaMeR
- Loader, знакомство

- ContentProvider, знакомство
- Материалы для самостоятельного изучения
- Задание на программирование. Многопоточность.