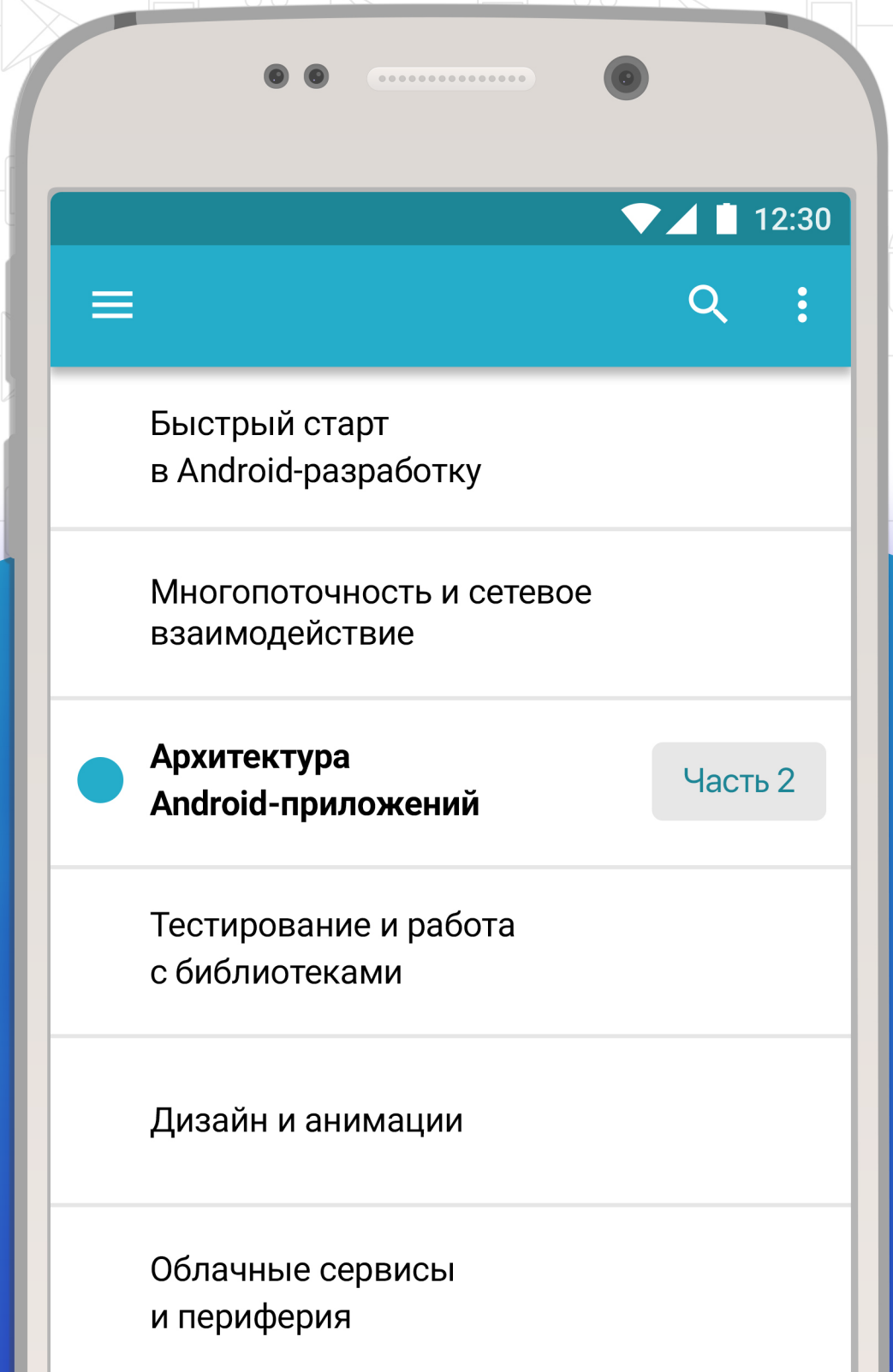
фонд развития
онлайн образования
eldf.ru

e·legion

academy.e-legion.com

# Программа Android-разработчик

## Конспект



# Оглавление

# Глава 1

# НЕДЕЛЯ 2

## 1.1. Паттерн MVVM

### 1.1.1. Обзор Model/View/ViewModel

Привет! В этом видео мы начнем изучать следующий MV-паттерн – Model-View-ViewModel.

MVVM:

- Model – данные и методы их получения, сохранения, обработки;
- View – визуальное представление данных, экран;
- ViewModel – новая компонента – абстракция представления, прослойка между View и Model.

Давайте разберемся, что это значит. Предположим, что у нас есть экран с TextView, в котором хранится имя пользователя. Экран – это View, а в ViewModel (обычный Java-класс) у нас хранится строковое поле. И если значение этого строкового поля равно John Smith, то и текст внутри TextView тоже будет равен JohnSmith. Вот и получается, что ViewModel – это отражение текущего состояния экрана, это абстракция View.

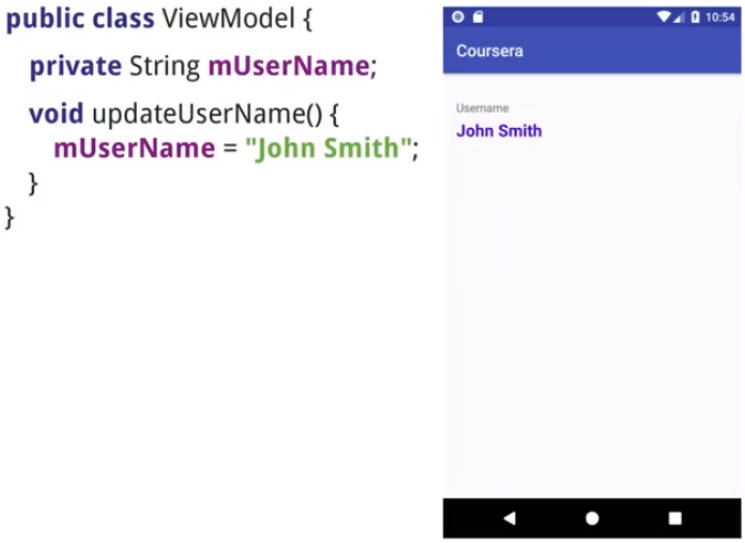



Рис. 1.1: Класс ViewModel

Если немного поразмыслить над этим, то можно прийти к выводу, что ViewModel не нужна View, и это действительно так. ViewModel в схеме взаимодействия находится между View и моделью. ViewModel может вызывать методы модели, она знает об ее существовании. Model, в свою очередь, про ViewModel не знает. Но она может рассылать уведомления об изменении, на которые и подписана ViewModel.

Что касается взаимодействия между ViewModel и View – ViewModel не может вызывать методы View, так как об ее существовании она не знает. ViewModel без разницы, связана ли с ней какая-то View или нет. ViewModel просто меняет свое состояние, меняет значение своих полей. С другой стороны, понятно, что View знает о ViewModel, так как View нужно передавать действия пользователя для обработки. Значит, View просто вызывает методы ViewModel.

А как она получает данные? Для этого используется механизм связывания, databinding. По факту это тот же самый паттерн Observer, и изменения во ViewModel сразу же отражаются во View. В зависимости от реализации связывание также может быть двухсторонним – тогда изменения во View тоже сразу отразятся во ViewModel.

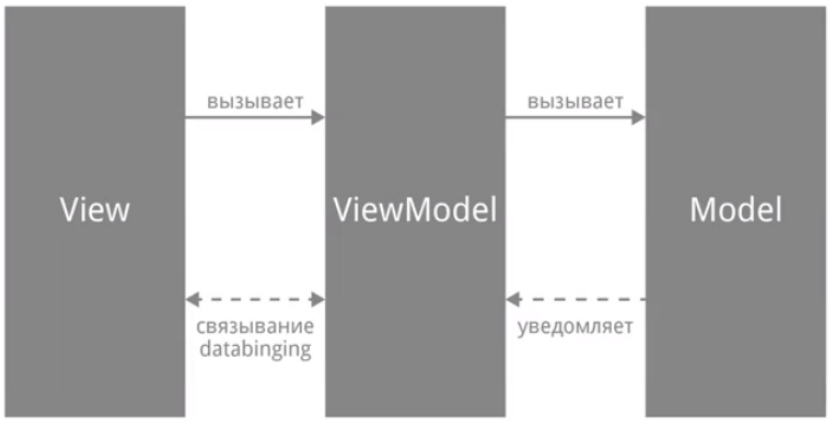



Рис. 1.2: Схема взаимодействия

Приведу простой пример. Предположим, что мы заполняем поле ввода пароля. Каждый новый введенный символ обновляет соответствующее поле во ViewModel, что в свою очередь, дергает валидацию, проверяя, что пароль больше восьми символов, среди которых есть одна цифра, одна строчная буква, один знак препинания, один китайский иероглиф... И в зависимости от всего этого во ViewModel меняется поле состояния ошибки, которое тут же подхватывает и отображает View (или не отображает, если ошибки нет).

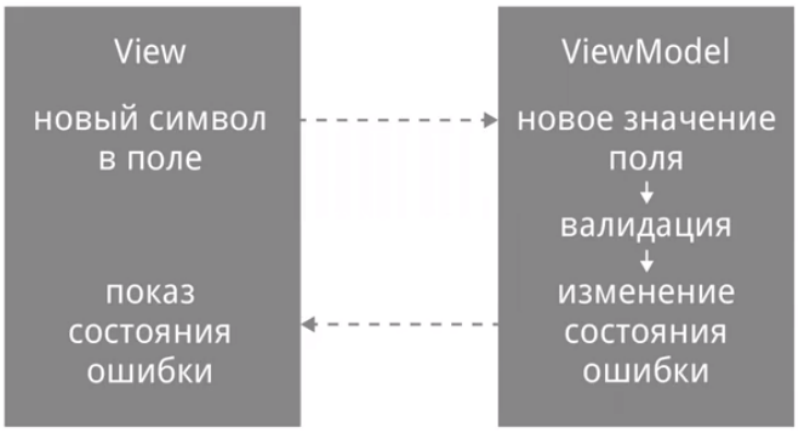



Рис. 1.3: Пример

Хорошо, с MVVM разобрались. В следующем видео разберем плюсы и минусы данного паттерна.

### 1.1.2. Плюсы и минусы MVVM

Давайте теперь попробуем сформировать плюсы и минусы MVVM паттерна.

Плюсы:

- Компоненты слабо связаны;
- Databinding уменьшает количество кода;
- Несколько View –> одна ViewModel.

Минусы:

- Показ Toast и диалогов сложно обработать в MVVM;
- Показ анимаций или данных с задержкой;
- Необходимость обработки команды во View.

Подведем итоги.

Когда использовать MVVM?

Если есть необходимость отображать большое количество данных, причем без каких-либо обработок на стороне View. Например, приложение показывает текущий курс валют. Пользователь не может взаимодействовать с этим курсом, он только наблюдает. View постоянно обновляется вместе с ViewModel.

Дальше. Обработка данных экрана легко производится во ViewModel. Это и уже упомянутая валидация данных, и какие-нибудь фильтры списков – все то, что легко может быть перенесено во ViewModel.

Когда следует воздержаться?

Если экран нагроможден логикой, анимациями, нестандартными View элементами, Toast'ами, диалогами и состояние экрана сложно описать в нескольких переменных, то MVVM вряд ли подойдет. Правда, стоит задуматься, а точно ли все сделано правильно, и не стоит ли разделить один экран на несколько экранов?

### 1.1.3. Обзор DataBinding Library

Привет! В этом видео разберем DataBinding Library – библиотеку, которая связывает xml-разметку с Java-кодом. Во время разбора MVVM я несколько раз упоминал про механизм связывания. DataBinding Library – это гугловская реализация связывания для Android-приложений. Давайте ее изучать.

И первым делом надо библиотеку подключить. Для этого в файл build.gradle уровня app внутри scope'a android нужно добавить следующую команду:

```
dataBinding {
    enabled = true
}
```

Теперь создадим какой-нибудь POJO-класс, например, User. У него будет только одно поле типа String – name. Именно это поле мы и будем связывать с разметкой.

На разметке же определим два TextView: первый – лейбл со словом Username, а во втором уже будет храниться значение. Но нам также придется сделать кое-какие изменения. Рассмотрим верстку. Первое, что бросается в глаза: корневой элемент теперь – нода layout, в ней находится нода data, в которой определена нода variable с именем user и типом класса User. Это переменная, которая будет использоваться далее в разметке. Забегая вперед, скажу, что переменных может быть определено несколько. Дальше идет стандартная разметка, в которой удалены некоторые строки для читабельности. И во втором TextView текст написан необычным образом: @user.name.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android=
"http://schemas.android.com/apk/res/android">
    <data>
        <variable name="user"
        type = "com.elegion.coursera.User"/>
    </data>
```

```
8       <LinearLayout
9           android:layout_width="match_parent"
10          android:layout_height="match_parent"
11          android:orientation="vertical">
12
13          <TextView android:text="Username" />
14          <TextView android:text="@{user.name}" />
15      </LinearLayout>
16  </layout>
```

В Activity следующий код: вместо стандартного setContentView мы вызываем одноименный метод у DataBindingUtil-класса, передавая ему на вход Activity и разметку. Этот метод возвращает нам сгенерированный класс, название которого по умолчанию совпадает с названием xml layout'а, только в CamelCase и с приписанным binding. Но мы можем задать и свое название. Далее мы создаем объект класса User, передавая в конструктор игру слов с именем Агента 007. В конце концов, производим связывание сгенерированного класса и класса User. Запускаем и видим результат: объект user был успешно связан с разметкой. Значение поля name было передано в соответствующий TextView, и все это без findViewById.

```
public class MainActivity extends AppCompatActivity {
  @Override
  protected void onCreate(Bundle
  savedInstanceState) {
    super.onCreate(savedInstance State);
    AcMainBinding binding =
      DataBindingUtil.
      setContentView(this, R.layout.ac_main);
    User user = new User("James Bound");
    binding.setUser(user);
  }
}
```

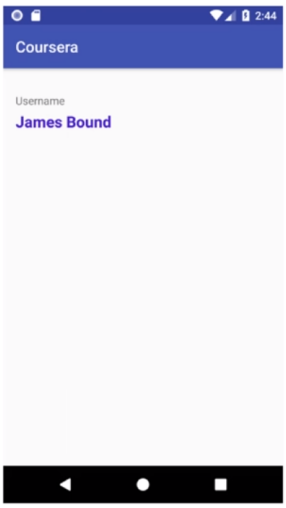



Рис. 1.4: Activity

Просто перенос значения поля, скорее всего, не впечатлит искушенного зрителя, но библиотека может не только это. На самом деле выражений в разметке, которые можно использовать с DataBinding, очень много. Простой и понятный пример: VisibilityView элемента, который выставляется в зависимости от поля объекта. Класс View нужно будет импортировать, совсем как в Java-коде. Делается это простым добавлением ноды import внутрь data. Помимо самого импорта, можно еще и создать псевдоним для этого класса. В моем случае это просто буква V.

```
android:visibility="@{user.shouldShow ? V.VISIBLE : V.GONE}"
<import
    alias="V"
    type="android.view.View" />
```

Другая жизненная ситуация – задать значение, если оно не null. Можно использовать тернарный оператор, по старинке, а можно воспользоваться специальным Null coalescing оператором. Обозначается он двумя вопросительными знаками. Строки кода эквивалентны и говорят сами за себя: берем аватарку пользователя, а если ее нет, то заглушку с ресурса. Да, можно использовать ресурсы в выражениях.

```
android:src="@{user.avatar ?? @drawable/placeholder}"
android:src="@{user.avatar != null ? user.avatar : @drawable/placeholder}"
```

Вообще язык очень обширен и включает в себя поддержку многих операций, а именно математических, логических, бинарных, унарных, тернарных операций, побитовых сдвигов, конкатенацию строк, сравнение, группировку, проверку, приведение типов, вызовы методов, доступ к элементам массива, поддержку литералов и уже знакомый вам доступ к полям. Впечатляет, не правда ли?

Но несмотря на все эти возможности я рекомендую не превращать разметку в код. Это неизбежно приведет к недопониманию и багам. Используйте возможности с умом!

Да, некоторые ограничения все же есть. В разметке отсутствует поддержка слов new, this, super и generic-методов.

Что касается обработки событий нажатий. Как вы уже знаете, в элементах внутри layout'а можно определить атрибут onClick, задав ему название метода, который будет выполняться при нажатии. Но было ограничение: метод должен быть определенной сигнатурой, принимать на вход View и вообще должен быть определен в том компоненте, который использует разметку. DataBinding Library расширяет эти возможности. Обработать нажатия мы можем двумя спосо-

бами. Первый – это ссылка на метод. От описанного выше способа он отличается тем, что можно указать класс, метод которого будет выполняться, то есть нет привязки к Activity. В коде мы описали класс, в который добавили метод, который на вход принимает View.

```
public class UserActions {
    public void onUserClick(View view) {
        //do something
    }
}
```

В разметке мы добавляем переменную actions в ноду data. На View элементе в атрибут onClick добавляем значение @actions::onUserClick. Обратите внимание на стиль ссылки на метод с помощью двух двоеточий – прямо как в Java 8.

```
<data>
    <variable
        name="actions"
        type="com.elegion.coursera.UserActions"/>
</data>
```

И не забываем создать объект класса UserActions и передать его сгенерированному классу, тем самым связав их.

```
UserActions userActions = new UserActions();
binding.setActions(userActions);
```

Второй вариант обработки нажатия производится с помощью передачи лямбды. Главное преимущество этого подхода в том, что можно передавать произвольные данные. Допустим, мы хотим сортировать пользователей по долгому нажатию. У нас есть класс ProfileManager, есть метод sort, который принимает на вход список пользователей, а возвращает нам булево значение. Это обусловлено тем, что метод onLongClick тоже возвращает булево значение, то есть возвращаемые значения должны совпадать.

```
public class ProfileManager{
    public boolean sort(List<User> users){
        //sort users
    }
}
```

Дальше настраиваем импорты и переменные: импортируем типы List и User, добавляем переменные profileManager и users. Обратите внимание на тип переменной users. Это List, но вместо угловых скобок – экранирование. К сожалению, с этим ничего не поделать, придется писать так. Ну и само значение атрибута onLongClick похоже на обычную лямбду. Кстати, есть еще одна особенность, связанная с переданными на вход в лямбду аргументами: можно либо ничего не указывать, либо указывать все. По идее на вход метод onLongClick принимает объект типа View, но так как я его не использую, мне он не нужен. Да, аналогично можно использовать и другие коллекции – массивы, map'ы, sparsearray. Последнее – это аналог map, но с поддержкой примитивных ключей.

```
<data>
    <import type="com.elegion.coursera.User"/>
    <import type="java.util.List"/>

    <variable name="profileManager"
    type="com.elegion/coursera.ProfileManager"/>
    <variable name="users" type="List&lt;User&gt;"/>
</data>

android:onLongClick="@{()->
profileManager.sort(users) }"
```

Представим ситуацию – у пользователя аватарка указана как url картинки. В обычном подходе мы бы просто взяли этот url и загрузили бы картинку через Picasso. А как было бы удобно, если бы можно было в ImageView просто указать этот url в атрибуте, и картинка скачалась бы сама! И как вы уже догадались, с помощью этой библиотеки можно это устроить.

Создаем класс (любой) и указываем аннотацию для метода BindingAdapter, указываем название атрибута. В методе на вход передаем ImageView и строковую переменную. Внутри используем Picasso.

```
public class CustomAdapter {

    @BindingAdapter({"bind:imageUrl"})
    public static void loadImage(ImageView view, String url) {
        Picasso.with(view.getContext()).load(url).into(view);
    }
```

```
}
```

На разметке, соответственно, используем атрибут. Обратите внимание, что namespace не важен – важно только само название атрибута. Поэтому вместо bind можно использовать app.

```
<ImageView
    android:layout_width="@dimen/avatar_size"
    android:layout_height="@dimen/avatar_size"
    app:imageUrl="@{user.avatarUrl}" />
```

Все, что было описано до этого, относилось к неизменяемым объектам. Мы создали пользователя, передали его в связку и просто показываем значения полей. Но это не очень интересно. Гораздо интереснее было бы, если бы изменения в объекте сразу отображались на разметке. И чтобы добиться такого эффекта, нужно использовать наблюдаемые поля, объекты и коллекции, которые предлагает нам библиотека.

Начнем с Observable Objects. Рассмотрим наш класс User. Класс теперь наследуется от BaseObservable из библиотеки DataBinding. У getName теперь есть аннотация Bindable – она нужна, чтобы сгенерировать id для поля name, и этот сгенерированный id мы используем в методе notifyPropertyChanged внутри setName. Обратите внимание, что у сгенерированного библиотекой id свой класс – BR. И... это не самый удобный способ.

```
public class User extends BaseObservable {

    private String mName;

    @Bindable
    public String getName() {
        return mName;
    }

    public void setName(String name) {
        mName = name;
        notifyPropertyChanged(BR.name);
    }
}
```

С наблюдаемыми полями дело обстоит проще. Если вам нужен примитив, то можно просто

использовать нужный обозреваемый тип, например, ObservableInt. Что касается объектов, то можно использовать обертку ObservableField и передать generic нужный тип.

Использовать поля тоже просто: мы обращаемся к обертке и вызываем get или set в зависимости от того, что нам нужно. С коллекциями примерно та же история.

DataBinding Library – это очень интересная библиотека, которая предлагает иной подход к разработке интерфейсов. Конечно, она не без греха, сборки часто совершаются с ошибками, но rebuild иногда спасает. Плюс еще кое-что. Классы генерируются после компиляции. Поэтому сначала добавляем аннотации или ноду layout и data в разметку, собираем, а потом уже используем в Java-коде. Обязательно ознакомьтесь с документацией. Она очень хорошая и подробная. Также можно найти много статей, историй успеха об использовании этой библиотеки. На мой взгляд она стоит того, чтобы хотя бы попробовать с ней поработать, и мы поработаем.

### 1.1.4. Работа с Behancer

На этой неделе мы применяем паттерн MVVM к нашему проекту Behancer.

Однако, есть некоторые изменения в работе с проектом, о которых нужно сказать. Behancer_MVVM – это отдельный проект, initial commit которого совпадает с Behancer-origin, который вы качали в уроке «Знакомство с Behancer».

Как вы уже поняли, в проекте есть гит. Все действия, которые я делаю в каждом из уроков я добавляю в отдельный коммит, который затем добавляется в master. Если вы хотите повторять код за мной, то в любой момент можете взять новую ветку от initial commit (или любого другого) и писать код параллельно.

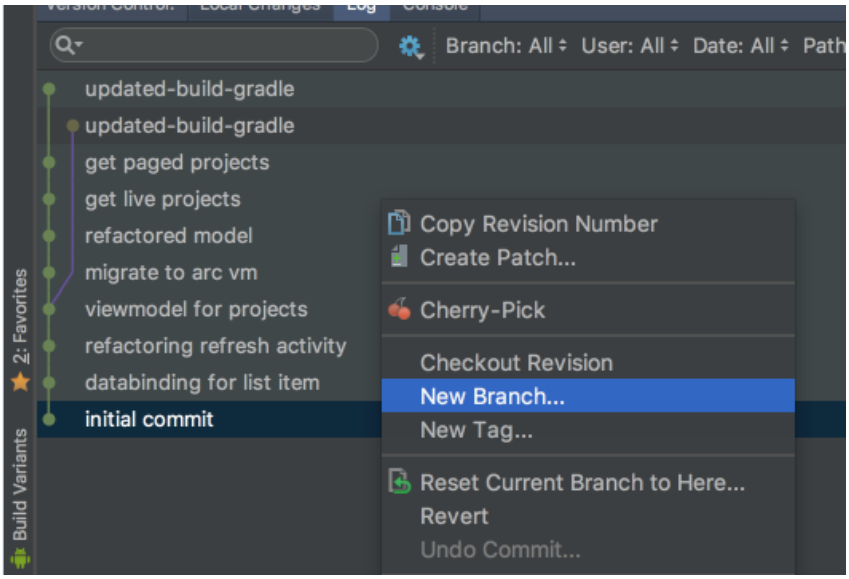



Рис. 1.5: Как взять ветку от initial commit

В этой неделе у вас 2 задания на программирование – я добавил ветки task-1 и task-2 для них. При выполнении заданий, нужно переключаться на соответствующую ветку и писать код уже в ней. Дальше коммитить и передавать на проверку по старой схеме.

При желании, можете слить результаты task-1 в task-2 перед выполнением второго задания.

### 1.1.5. Behancer на MVVM. VM для list item

Привет! Мы изучили архитектуру MVVM. Давайте теперь попробуем применить ее в нашем Behancer. Первым делом очки. Итак, что нужно сделать в первую очередь. В самую первую очередь нам нужно databinding подключить. Мы будем использовать именно его для реализации MVVM. Заходим в build gradle уровня app. И вбиваем

```
1  dataBinding{
2      enabled = true
3  }
```

Щелкаем на «синхронизировать». Хорошо, вроде как подключили. Давайте мы начнем с чуть более простой задачи. Мы добавим databinding для элемента списка, то есть для holder'а.

Переходим в ProjectsHolder, видим вот эту картину. Получается, нам нужно создать ViewModel для элементов списка, которая будет абстракцией экрана. Давайте создадим новый класс и назовем его ProjectListItemViewModel, чтобы точно ни с чем не перепутать, щелкаем OK, добавляем Git. Сделаем Split Vertically, посмотрим на holder. Соответственно у нас есть ImageView, TextView, то есть имя проекта, имя пользователя и дата регистрации. Нам нужно все это перенести во ViewModel. Создаем private string. Мы не можем ссылаться на картинку, так как у нас url, поэтому мы пишем image url, дальше у нас mName, mUsername, mPublisedOn.

Хорошо. Теперь нам нужно каким-то образом получить эти данные. Я сейчас вижу один простой способ получать проект в конструкторе ProjectListItemViewModel и соответственно дергать оттуда данные. Пишем конструктор:

```
1  public ProjectListItemViewModel (Project project) {
2          mImageUrl = project.getCover().getPhotoUrl();
3          mName = project.getName();
4          mUsername = project.getOwners().get(FIRST_OWNER_INDEX).getUsername();
5          mPublishedOn = DateUtils.format(project.getPublisheOn());
6  }
```

Хорошо. Данные сохранили, ViewModel у нас почти что есть, только поля у нас приватные, надо сделать для них геттеры. Так, геттеры создали. Я могу сказать, что ViewModel для элемента списка у нас готова. Закрываем.

Что мы делаем дальше? Следующим шагом станет добавление функциональности databinding в разметку. Переходим в адаптер и находим R.layout.li_projects. Корневой элемент теперь layout, в котором находится data, в котором находится variable:

```
1  <variable
2      name="project"
3      type="com.elegion.test.behancer.ui.projects.ProjectListItemViewModel"/>
```

Просто копируем наш RelativeLayout вот сюда. После сборки проекта она скомпилируется с названием li projects binding, это меня не устраивает. Я задам свое название класса, он будет называться ProjectsBinding. Пробуем собрать.

Вроде собралось без ошибок, хорошо. Теперь давайте связывать. Text view tv name получает-

ся имеет text=”@project.name”. Так, text view username, давайте скопирую, project username. Published on, соответственно, project.publishedOn.

Добавили поля для TextView, с ImageView пока что повременим. Давайте теперь перейдем в наш project адаптер и продолжим добавлять функциональность здесь. Project binding, наш сгенерированный класс, inflate, inflater, parent, false. Полученную связку передаем в holder. Переходим в holder, теперь он принимает project binding. Создаем поле для binding’a.

Также нам нужно добавить метод executePendingBindings. Этот метод заставляет сделать связывание сразу же, как только метод вызвался, а не ждать до следующего кадра. В Recycler View это может вызвать ошибку – если не вызывать этот метод – потому что там View переиспользуются. Давайте это закомментируем и попробуем запустить наш код. Бам. Видим, что тот код, который мы добавили, сработал. Тут нет изображений, потому что мы их не обрабатываем. Давайте удалим закомментированный код, он нам больше не нужен, хотя на самом деле конкретно загрузка с Picasso, наверное, даже пригодится.

Давайте добавим обработку нажатия. Переходим в li_projects и добавляем новую переменную, она будет называться onItemClickListener и будет иметь тип onItemClickListener с projects adapter’a. Хорошо, получается что при нажатии на relative layout on click вызывается метод onItemClickListener. Помните, что databinding library позволяет нам не передавать View, если она нам не нужна, можно ничего не передавать. Вызываем onItemClickListener.onItemClick(project.username). Соберем. Собралось. Переходим в holder и добавляем item click listener, который мы получали на входе в наш binding. Запускаем, щелкаем на элемент списка и видим, что нажатие обрабатывается.

Последнее, что нам осталось сделать – это добавить загрузку изображений. В лекции я упоминал, что можно сделать собственный binding adapter. Давайте этим займемся. Переходим в utils и создаем новый класс. Обзываем его CustomBindingAdapter. Давайте создадим метод

```
1  public static void loadImage (ImageView imageView, String urlImage) {
2      Picasso.with(imageView.getContext()).load(urlImage).into(imageView);
3  }
```

У метода loadImage должна быть аннотация со значением атрибута, который мы хотим использовать. Аннотация называется BindingAdapter и внутри пускай будет уже знакомый нам bind:imageUrl. Дальше переходим обратно в нашу разметку и в ImageView указываем

```
app:bind:imageUrl="@{project.imageUrl}"
```

Кажется, так называлось поле в адаптере. Запустим. Щелкаем OK.

Магия. Картинки загружаются, я очень рад. С адаптером OK, этот код из холдера можно удалить, то есть каким он был, и каким он стал. На этом первый урок по MVVM и databinding заканчивается, встретимся в следующем уроке.

### 1.1.6. Выделение логики обновления в RefreshActivity

Продолжаем. Вообще следующим нашим шагом должно стать добавление ViewModel для экрана с проектами, то есть для projects fragment'a. Если вы помните, он устроен так, что обновление экрана, точнее SwipeRefreshLayout и вся логика, связанная с ним, находится в Activity, в то время как данные находятся во фрагменте. Если я хочу, чтобы одна ViewModel полностью отвечала за экран с проектами, то мне нужно как-то переделать single fragment activity, которая на данном этапе отвечает за обновление.

Давайте начнем. Создаем класс, назовем его RefreshActivity, и суперклассом у него будет SingleFragmentActivity. Щелкаем OK, добавляем Git. OK. Опять делаем Split Vertically, сворачиваем, переключаемся на SingleFragmentActivity. Всю логику, которая отвечает за обновление переносим в RefreshActivity. Метод onRefresh. Ctrl+C, Ctrl+V. Здесь, кстати, можно его удалить. Set refresh state, Ctrl+X, Ctrl+V. obtainStorage, я полагаю, тоже нужно удалить, чтобы конкретные реализации Activity уже реализовывали этот интерфейс. Я его сейчас просто закомменчу.

Класс должен быть абстрактным. abstract class RefreshActivity, потому что его реализации также должны будут реализовывать метод getFragment, более того, мы добавим метод абстрактный getLayout. protected – не абстрактный, просто protected – protected int getLayout() и в setContentView будет вызываться именно он. Пока что перенесем layout сюда. Стандартная реализация должна будет возвращать layout просто с контейнером. Идем в project, в ресурсы, layout и добавляем новый layout resource file. Назовем его activity container и root'ом у него будет linear layout. Щелкаем OK, добавляем.

Приходим сюда. FrameLayout, match parent, match parent, id такой же, как у старого frame layout'a. Fragment container, Ctrl+C, Ctrl+M. Закрываем.

Теперь single fragment activity у нас возвращает return R, layout, activity, container. Странно,

почему не распарсился. RefreshActivity, его get layout возвращает R layout swipe container.

Мне кажется, RefreshActivity готов, можно закрывать. Может, здесь что-то не так? Дальше наш ProjectsActivity extends SingleFragmentActivity, хорошо. А ProfileActivity теперь extend'ит RefreshActivity, то есть у ProfileActivity мы оставляем все как было раньше. Также он должен теперь реализовывать storage owner. Переходим в SingleFragmentActivity и копируем этот код. Дальше ProjectsActivity должен реализовывать storage owner, снова копируем. Теперь это можно удалить. Во фрагменте должен быть SwipeRefreshLayout, забыл совсем.

Что мы делаем дальше? fr_projects оборачиваем в SwipeRefreshLayout. Кстати, linear layout можно заменить на frame layout, ничего страшного не произойдет.

Теперь переходим во фрагмент проектов и refreshable нам больше не нужен. SwipeRefreshLayout.on RefreshListener, Alt+Enter, Implement methods, OK и просто копируем getProjects в onRefresh, удаляем onRefreshData, удаляем RefreshOwner. Он нам больше не нужен. Вместо onRefreshData onRefresh. Теперь нужно сообразить listener, точнее SwipeRefreshLayout mSwipeRefereshLayout.

```
1  mSwipeRefershLayout = view.FindViewById(R.id.refresher).
```

И в onActivityCreated mSwipeRefreshLayout.setOnRefreshListener, this. mSwipeRefreshLayout, on RefreshOwner мы ссылку удалим. mSwipeRefreshLayout, set refreshing, true. И тут set refreshing, false.

Запускаем. Как видите, рефакторинг произошел с некоторыми трудностями, но все-таки успешно. Хорошо. На этом этот урок заканчивается, в следующем мы начнем строить ViewModel для экрана с проектами и применим MVVM.

### 1.1.7. Создание ProjectsViewModel

Привет! Продолжаем внедрять MVVM в наше приложение. В этом уроке мы займемся добавлением ViewModel для экрана projects fragment. Давайте начнем.

Projects, New, Java class, ProjectsViewModel, OK, добавляем Git, правой кнопкой, Split Vertically, сворачиваем, переходим во фрагмент. Слева у нас фрагмент, справа ViewModel для этого фрагмента. Чего я хочу добиться? Я хочу, чтобы во фрагменте не было никаких переменных, полей, кроме самого ViewModel. Получается, все, что у нас есть во фрагменте, нужно перенести во ViewModel, чтобы фрагмент, он же View, просто смотрел на ViewModel и относительно

ViewModel строил свое состояние.

Давайте начнем. Самый простой способ начать избавляться от логики, от данных во фрагменте – это перенести метод getProjects, то есть наш сетевой запрос. Ctrl+X, Ctrl+V. Что мы видим? Давайте решать проблемы по мере их поступления. Нам нужен Disposable – переносим его, нам нужен Storage – переносим его. Дальше SwipeRefreshLayout, ErrorView, RecyclerView и адаптер. Адаптер закомментируем, SwipeRefreshLayout комментируем, RecyclerView комментируем, ErrorView комментируем. Их здесь не должно быть. Storage нужно перенести во ViewModel; мы можем передать его в конструкторе ViewModel, так что создадим конструктор. Открываем конструктор и передаем storage.

```
1  mStorage = storage;
```

Получается, на onAttach'е фрагмента нужно создавать экземпляр ProjectsViewModel. Давайте сделаем поверх переменной.

```
1  private ProjectsViewModel mProjectsViewModel;
```

```
1  mProjectsViewModel = new ProjectsViewModel(storage);
```

Хорошо. Что дальше? Дальше у нас есть SwipeRefreshLayout, который должен либо показывать индикатор загрузки, либо не показывать его. Соответственно, если у нас ошибка, то мы показываем ErrorView, если нет ошибки, если данные пришли нормально, то мы показываем RecyclerView. Получается, что показ ошибки или данных мы можем обработать с помощью одной boolean переменной. Давайте создадим ее. Это будет не простая boolean переменная, а observable boolean, то есть boolean, за изменениями состояния которого мы сможем наблюдать.

```
1  private ObservableBoolean mIsErrorVisible = new ObservableBoolean(value: false);
```

Что мы получим? Если IsErrorVisible у нас true, то мы показываем ошибку, а RecyclerView прячем. Переходим в response. mIsErrorVisible, в этом случае set false, то есть данные пришли, ошибки нет. Переходим в throwable. mIsErrorVisible set true. Произошла ошибка, данных нет. Эти две строки мы можем удалить. Эти две строки тоже можем удалить.

Дальше ProjectsAdapter. Я не могу создать адаптер во ViewModel, но я могу создать список проектов, который мы можем закинуть в сам адаптер. Этот список точно так же, как и observable boolean, будет обозреваемым. private ObservableArrayList, тип указываем project, называем его projects, new ObservableArrayList. Соответственно в mProjects add all, response, getProjects. Добавили проекты.

Теперь SwipeRefreshLayout. Точно так же создаем новую переменную типа observable boolean mIsLoading, равную new observable boolean, и тоже значение в начале будет false. Что мы делаем дальше? При запросе показываем индикатор загрузки, то есть mIsLoading set true, по окончанию запроса mIsLoading set false. Alt+Enter. Ну вот как-то так.

Что мы можем сделать дальше? Мы можем передать во ViewModel обработку нажатий на элементы списка. Сейчас обработка нажатий происходит во фрагменте, и сам фрагмент реализует интерфейс ProjectsAdapter, OnItemClickListener. Для того, чтобы мы могли передать этот listener во ViewModel, нам нужно вынести его в поле. private OnItemClickListener mOnItemClickListener, равная new OnItemClickListener. Внутрь onItemClick мы просто копируем и вставляем этот код.

Дальше удаляем реализацию интерфейса, точнее, сам интерфейс и передаем OnItemClickListener в конструкторе. OnItemClikListener, Alt+Enter, Create field точно так же.

Хорошо. Что мы делаем дальше? Дальше нам нужно создать геттеры для вот этих полей. Давайте сначала передадим dispatch. dispatch переводится как «передать». Реализация SwipeRefreshLayout.OnRefreshListener нам здесь тоже не нужна. Она будет в самой разметке. Удаляем. Вместо onRefresh пишем mProjectsViewModel. Это будет у нас public. getProjects. get не самое лучшее название, пускай будет loadProjects.

Дальше что мне нужно сделать? Геттеры для полей, точно. Ctrl+N, геттеры, генерируем геттеры для OnItemClickListener'a, IsLoading'a, IsErrorVisible и mProjects. Щелкаем OK.

Метод onViewCreated мне будет не нужен. Удаляю его спокойненько. Настройка RecyclerView и адаптера – я не хочу, чтобы она была в коде фрагмента. Настройка SwipeRefreshLayout'a тоже. Получается, нам нужно перенести их в layout, но просто так это сделать не получится. Layout можно задать, но все остальное сделать не получится. Нам нужно создать кастомные аттрибуты, этим мы сейчас займемся. onCreateView тоже скоро поменяется. Закрываем. Переходим в проект. Открываем CustomBindingAdapter.

### 1.1.8. Добавление кастомных атрибутов

Нам нужно два метода. Первый – для конфигурации RecyclerView, другой – для конфигурации SwipeRefreshLayout'а. Каждый из этих методов будет содержать два новых атрибута.

Давайте начнем. public static void configureRecyclerView, Ctrl+D, configureSwipeRefreshLayout. Соответственно метод, который настраивает RecyclerView на вход получает сам RecyclerView. Дальше во ViewModel у нас список проектов, метод получает его, List<Project> projects. Метод получает также обработку нажатий на элементы списка. OnItemClickListener.

Создаем адаптер:

```
1  adapter = new ProjectsAdapter;
```

Передаем туда listener. Нам нужно также передать туда проекты. Projects, listener. Alt+Enter, Add list, переносимся в адаптер, здесь убираем проекты. Точнее, убираем начальное значение new array list.

```
1  mProjects = projects;
```

Метод addData нам тоже не нужен, удаляем его, давайте сначала закомментируем, потом, если что, вернем. CustomBindingAdapter, передали, в RecyclerView задаем адаптер adapter. Создаем аннотацию BindingAdapter и первая аннотация будет называться bind:data, второй атрибут bind:clickHandler.

Переходим к следующему методу configureSwipeRefreshLayout. На вход он принимает непосредственно SwipeRefreshLayout, состояние isLoading, это у нас булево состояние boolean isLoading и OnRefreshListener, то есть что будет при обновлении экрана. Layout, SetOnRefreshListener, listener. Layout, setRefreshing, точнее layout.post, setRefreshing, isLoading, закрываем скобку. Хорошо. Задаем атрибуты. BindingAdapter, фигурные скобки, bind:refreshState, bind:onRefresh.

### 1.1.9. Настройка Databinding

Что мы делаем дальше? Мы переходим во фрагмент и открываем разметку fr_projects. Добавляем DataBinding в разметку. Рут нода у нас теперь layout. Переносим SwipeRefreshLayout внутрь layout. Добавляем ноду data. Добавляем ноду variable. Хорошо. Теперь xmlns переносим a layout.

В data можем задать название будущего сгенерированного класса. Класс равен ProjectsBinding. Переменная, название пускай будет vm – ViewModel – и тип непосредственно ProjectsViewModel.

Дальше что? Нам нужно связать разметку с ViewModel. ViewModel я добавил, теперь давайте сделаем опять split screen. Что у нас тут происходит? Во-первых, IsErrorVisible для RecyclerView и разметки с ошибкой. Получается, атрибут visibility зависит от этой переменной. vm.isErrorVisible. По идее сейчас должно быть gone, двоеточие, visible. Чтобы мы могли это добавить, нужно импортировать класс View. Зададим alias большой буквой V. Если видна ошибка, то RecyclerView должен быть невидимым. Большая V.GONE, иначе V.VISIBLE. То же самое касается слоя с ошибкой, только наоборот. Если ошибка true, то она должна быть видима, если она false, то она должна быть gone.

Давайте соберем. Вроде все собралось, вроде все нормально. Теперь настроим SwipeRefreshLayout. Добавляем наш новый атрибут refreshState. Добавили новый namespace. Ctrl+L, собачка, фигурные скобки, ViewModel is loading, все просто. Теперь нам нужно добавить onRefreshListener. Его также нужно добавить во ViewModel. private OnRefreshListener. OnRefreshListener равен new on refresh listener. onRefresh у нас будет вызывать метод loadProjects.

Возвращаемся к SwipeRefreshLayout'у. Так, bind, onRefresh, собачка, фигурные скобки, ViewModel, точка, геттер для onRefreshListener'а, onRefreshListener. Хорошо. Задаем данные для RecyclerView. Это у нас bind data, собачка, фигурные скобки, ViewModel, точка, projects. Задаем clickHandler, собачка, фигурные скобки, ViewModel, точка, onItemClickListener.

Теперь нам нужно задать менеджер для нашего RecyclerView. Мы можем это сделать как в разметке, так и в самом коде CustomBindingAdapter'а. recyclerView, setLayoutManager, new LinearLayoutManager, recyclerView, getContext. Закрываем все. Последнее, что нам нужно сделать с нашим кодом – это добавить связывание между разметкой и фрагментом. Переходим во фрагмент, метод onCreateView. Добавляем связывание. ProjectsBinding, сгенерированный класс. ProjectsBinding, точка. inflater, container, false. Это у нас возвращает ProjectsBinding binding. Binding. setVm, ProjectsViewModel.

```
1  return binding.getRoot();
```

Давайте запустим, OK. Вроде как все работает, только почему-то я вижу одновременно и ошибку, и список. Давайте все таки вернем LinearLayout обратно. Меня терзают смутные сомнения. Orientation vertical. Ctrl+Alt+L. Запускаем, видим, что все работает. Щелкаем на элемент списка, видим, что функциональность не повреждена. Обновляем и обновление проходит.

Что мы сделали? Мы вынесли всю логику, связанную с ui, во ViewModel и в разметку. То есть

разметка связана с ViewModel, что приводит к тому, что она постоянно сразу же обновляется. Это непривычно на первый взгляд. Я бы не сказал, что это самый лучший способ программировать. Мы видим, что такая возможность есть. Если это не будет вас пугать, то почему бы не воспользоваться таким способом? Дальше давайте перевернем экран и увидим стандартную проблему, что state при повороте не сохраняется, понятно. Впоследствии мы добавим сюда life components гугловские, архитектурные компоненты и посмотрим, как изменится наш проект.

### 1.1.10. Обзор Android Architecture Components

Привет. В этом видео будем знакомиться с архитектурными компонентами, которые Google представила на I/O 2017. И, кстати говоря, Room, с которым мы уже работали, тоже входит в их число.

Для начала обратимся к истории. Первый Google I/O прошел в мае 2008. Тогда же и была представлена первая версия Android, которая затем, в сентябре, вышла на рынок. Разработчики получили платформу, но не получили инструкцию к ней. Это привело к тому, что появилось много способов того, как делать приложение под Android, как выполнять те или иные задачи – сетевые запросы, обработку поворота экрана, сохранение состояния... Множество архитектур и их вариаций, множество библиотек и фреймворков, множество вариантов дизайна приложений. Что из всего этого было правильным, решительно невозможно было сказать. Общество просило Google разобраться, и Google начал разбираться. Появились паттерны ABC, которые задавали порядок обработки сетевого запроса – в двух словах, результаты запроса должны сохраняться в базу данных, а затем появляться на экране. Появилось некоторое видение дизайна – Holo, на замену которому пришел Material Design, который растет и развивается по сей день. И в 2017 были представлены архитектурные компоненты. И гайд о том, как Google рекомендует писать приложения.

Однако нужно оговориться о том, что Google понимает, что они слегка опоздали, и советует использовать компоненты, только если у разработчиков нет готовой архитектуры под рукой. Если же у команды есть сформированная рабочая архитектура, то переходить не нужно.

Обратите внимание на рисунок. Там представлена рекомендуемая Google архитектура. View берет данные из ViewModel, которая берет данные из репозитория. Репозиторий инкапсулирует работу с сетью и БД, выдавая актуальные данные. Зависимость линейная, и весь этот концепт похож на Clean Architecture, подробнее о которой вы узнаете далее по курсу.

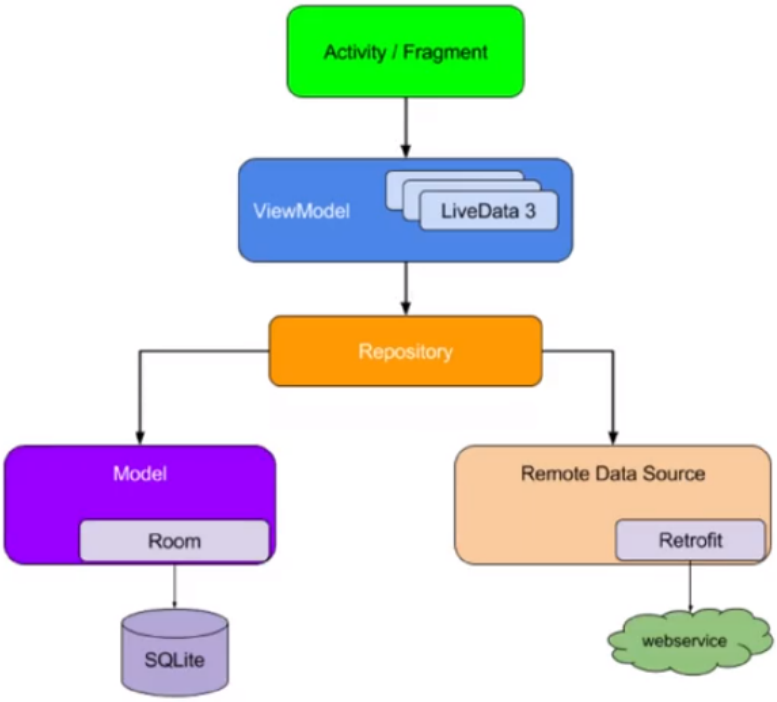



Рис. 1.6: Рекомендуемая архитектура

Вряд ли можно сказать что-то лучше самого первоисточника. Я рекомендую вам ознакомиться с гугловским гайдом по архитектуре, а сейчас давайте разберем его основные тезисы.

- Нельзя хранить контент или состояние в компонентах и компоненты не должны зависеть друг от друга;
- Интерфейс и контент приложения должны заполняться, опираясь на модель. Модель не должна зависеть от компонентов андроид.

Архитектурные компоненты предоставляют нам класс ViewModel, на который может подписаться View. И главное преимущество ViewModel в том, что при пересоздании Activity или фрагмента ViewModel не пересоздается. Она привязывается к новому компоненту. Достигается это тем, что мы не создаем экземпляров ViewModel вручную. Нам его дает фабрика. Соотношение жизненного цикла Activity и ViewModel показано на рисунке. Видно, что ViewModel очищается или

уничтожается, когда Activity окончательно уничтожено и не собирается воссоздаваться. Аналогично с фрагментом.

Что касается ограничений, во ViewModel не должно быть ссылок на объекты с жизненным циклом. Если вам нужен контекст, то можно наследоваться от Android ViewModel, который на вход принимает context application.

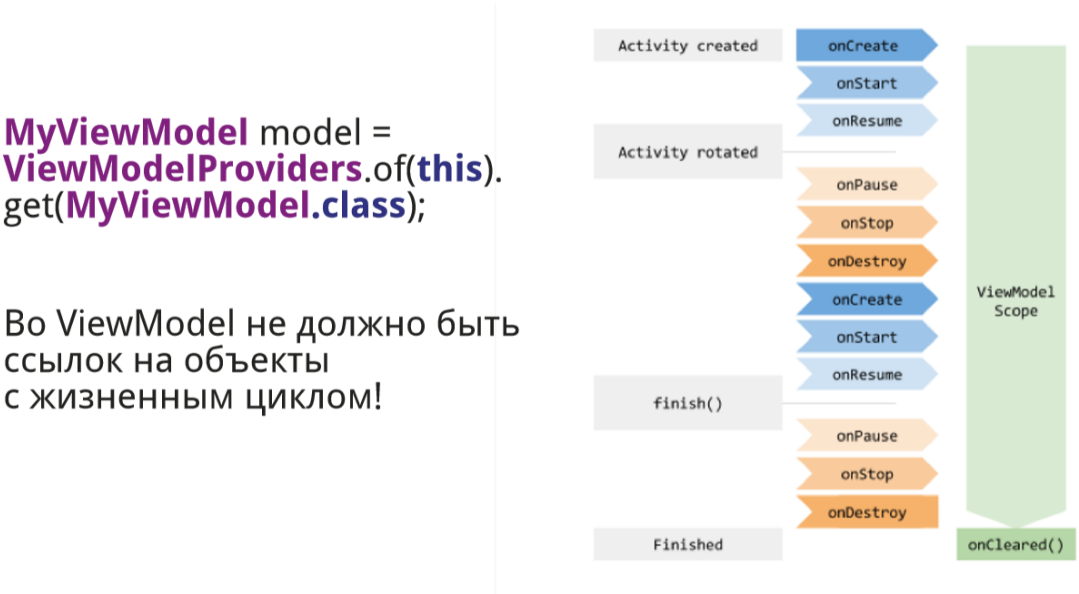



Рис. 1.7: ViewModel

Кстати о жизненном цикле. В компонентах он обрел почти что физическую форму – класс Lifecycle, и вместе с этим появилась возможность наблюдать за ним. На самом деле классов интерфейсов, так или иначе связанных с жизненным циклом, много. Целый пакет Android Arch Lifecycle. ViewModel, с которым мы уже знакомы, тоже входит в этот пакет. Давайте рассмотрим некоторые из этих классов, и начнем с самого Lifecycle.

Этот класс содержит информацию о жизненном цикле компонента и позволяет наблюдать за ним. Для этого используется 2 набора обозначений – events и states, события и состояния.

Далее, LifecycleObserver, который используется для наблюдения за событиями и состояниями – ничего необычного.

Далее, LifecycleOwner – это интерфейс, который показывает, что у класса есть жизненный цикл.

В библиотеке поддержки 26.1.0 активити и фрагменты уже реализуют интерфейс LifecycleOwner. Если же возникла необходимость снабдить свой класс жизненным циклом, можно воспользоваться классом LifecycleRegistry, с помощью которого и помечаются состояния. Правда, в конструктор LifecycleRegistry нужно передать настоящий LifecycleOwner. На практике же гораздо чаще реализовывают LifecycleObserver.

И мой любимый класс LiveData. LiveData – это контейнер для данных, на который можно подписаться. Подписываются на него компоненты с жизненным циклом, и LiveData уведомляет эти компоненты только тогда, когда компонент в активном состоянии, то есть в started или resumed. Какие еще преимущества дает использование LiveData? LiveData очищает наблюдателей, если их жизненный цикл перешел в состояние destroyed, то есть нет лишних объектов – нет утечки памяти. Если компонент был в фоне, а LiveData, на которую он подписан, получила новые данные, то компонент получит эти данные, как только перейдет на передний план, то есть данные не потеряются и всегда будут актуальными. То же самое касается и смены конфигурации, как, например, поворот экрана. Что касается обновления данных, есть класс MutableLiveData с методами setValue и postValue для использования в главном и фоновых потоках соответственно. Также LiveData поддерживает трансформации с помощью класса Transformations и слияние нескольких LiveData в одну с помощью MediatorLiveData. Стоит отметить, что в LiveData можно обернуть результаты запроса в Room.

И последнее, с чем мы познакомимся в этом видео – Paging Library. На самом деле эта библиотека решает одну из самых распространенных задач в Android-разработке – реализацию догружаемого списка. Суть в чем – вы листаете список, доходите до конца, загружается еще немного данных, опять листаете до конца, и опять загрузка. Деление на отдельные, скажем так, страницы нужно, чтобы не захламлять память лишний раз. Не имеет смысла загружать 500 элементов на экран, если пользователь даже до десятого не пролистает. Подробнее в реализацию углубляться не будем, разберем ее на практическом занятии.

Архитектурные компоненты, несмотря на некоторую запоздалость, – это огонь. Это действительно те инструменты, с которыми хочется работать. За это огромная благодарность Google. Что я рекомендую сделать? Во-первых, конечно же, прочитать гайд. Это даже обязательно. Во-вторых, с момента выхода архитектурных компонентов прошло достаточно много времени, проведено много конференций, и на каждой из них были доклады про компоненты. Посмотрите их. Третье – статьи на Хабре и статьи на Medium тоже читайте на досуге. И самое-самое главное – напишите тестовый проект, в котором попробуете использовать компоненты. Ну или хотя бы скачайте чей-нибудь с GitHub. А мы дальше займемся внедрением архитектурных компонентов, а точнее, ViewModel, в наш проект, потому что это того стоит.

### 1.1.11. Добавление архитектурных компонентов

Привет, мы познакомились с архитектурными компонентами, давайте попробуем вписать их в наш проект. Первое, что нужно сделать, это добавить их в gradle, я их просто скопировал и просто оставлю. Щелкаем на Sync now, переходим на ProjectsViewModel. Что нам нужно сделать? Нам нужно, чтобы наша ViewModel наследовалась от гугловской ViewModel. Пишем extends ViewModel, android arch lifecycle ViewModel.

Теперь давайте думать, какие изменения могут у нас произойти. Во-первых, observable boolean'ы и ObservableArrayList теперь можно убрать, использовать вместо них MutableLiveData. Давайте начнем. MutableLiveData. boolean равен new MutableLiveData. Ctrl+D, Ctrl+C. Ctrl+V, Ctrl+X. private MutableLiveData, List Projects, project равен new, Ctrl+X, Ctrl+V, new MutableLiveData, Ctrl+X.

Дальше что мы можем сделать? Эта ViewModel создается один раз, поэтому первую загрузку можем добавить просто в конструктор. В проекты желательно добавить какой-нибудь контейнер. Projects, setValue, new ArrayList. В этот ArrayList будут добавляться наши новые проекты. Дальше. isLoading.postValue, postValue, postValue, postValue, postValue. Вот это мы можем удалить и сгенерировать заново, геттер, вот так. dispatchDetach можем переименовать в onCleared. Override.

Хорошо. Переходим во фрагмент, удаляем метод onDetach, удаляем метод loadProjects. Теперь в ProjectsViewModel нам нужно брать из фабрики. Так как наша ViewModel имеет нестандартный конструктор, то нам нужно создать фабрику, которая будет нам этот ViewModel поставлять. Переходим в utils и созадем new CustomFactory. Щелкаем OK, добавляем Git. Этот класс будет у нас наследоваться от ViewModelProvide.NewInstanceFactory. private Storage mStorage, private ProjectsAdapter.OnItemClickListener mOnItemClickListener. Те же поля, что нам нужны для ViewModel. И конструктор public CustomFactory, на вход у нас Storage storage, OnItemClickListener onItemClickListener. mStorage равен storage. onItemClickListener равен onItemClickListener. И override'им метод create. Возвращаем ProjectsViewModel(mStorage, mOnItemClickListener).

Alt+L, cast to T. Да, unchecked cast, но что поделать. Переходим во фрагмент и в onAttach создаем фабрику. Она должна быть по идее после storage. CustomFactory factory = new CustomFactory (storage, mOnItemClickListener). ViewModel это теперь у нас ViewModelProviders.of this, custom factory, .get(ProjectsViewModel.class); Ctrl+Alt+L.

Хорошо, мы добавили гугловскую ViewModel и теперь у нас возникает следующий вопрос, следующая сложность, как связать сгенерированный binding-класс с ViewModel. Как это все должно работать вместе? На самом деле это не очень сложно делается. У нас есть binding, у нас есть

ViewModel и мы должны указать, что у binding'а есть Life Cycle. setLifecycleOwner, this. В принципе все, давайте теперь запустим и посмотрим, что при повороте устройства состояние списка остается тем же, то есть список не перегружается. Запустилось. Поворачиваем. Бац. Список тот же самый, поворачиваем обратно, список тот же самый. По сравнению с тем, что было раньше, когда после поворота у нас заново загружался список. Надо сказать, что гугловская ViewModel со своей задачей справляется.

Что дальше? Мы можем убрать title вообще в манифест. Так, ProjectsActivity, label, string/projects. Возвращаемся во фрагмент. Вообще мы, конечно, можем использовать LiveData и ViewModel без привязки к DataBindingLibrary, вот таким образом. Какой-нибудь код, мы могли бы тут обновить адаптер, если бы он у нас был. Так как мы все вырезали и у нас MVVM на всю катушку, мы используем все сразу, этот код нам не нужен. Эта ручная обработка: взять модель, взять проекты, наблюдать за ней – это все делается уже внутри DataBindigLibrary, удобно. Таким коротким, маленьким у нас выглядит фрагмент. У нас один класс для создания ViewModel, второй класс для конфигурирования сгенерированного binding файла.

Хорошо. В следующем уроке мы посмотрим, что можно сделать с нашей ViewModel, чтобы еще сильнее, еще лучше, полнее, использовать LiveData.

### 1.1.12. Создание RichProject

Привет. Продолжаем знакомить с Live-компонентами. Давайте посмотрим на наш ProjectsViewModel. В принципе он сейчас работает так, как работал раньше. Логично же, я же его просто поменял. Суть в том, что в текущем состоянии мы не используем всей мощи LiveData. Room может нам выдавать данные в виде LiveData и как только табличка обновляется, то данные тоже будут обновляться и вместе с этим обновятся ViewModel, переменные и ui, которое связано с этой переменной, с этим полем.

Давайте теперь немножко пошебуршим в data-слое и попробуем изменить базу данных нашей модели таким образом, чтобы мы могли использовать именно LiveData. Перейдем в BehanceDao и смотрим, что здесь все хорошо, лучше в storage. Видим, что getProjects у нас на самом деле собирает данные из двух таблиц, компонует их, отправляет нам в виде ответа, который потом обрабатывается, как если бы с сети пришло. Мне кажется, это слегка не очень. Это правильно с одной стороны, с другой стороны не очень.

Давайте мы попробуем добавить, посмотрим на проект, видим Ignore owners, Ignore cover. Давайте попробуем создать новую сущность, точнее новый класс, который будет называться project

версия 2 или Rich Project, который будет целиком и полностью собираться в том виде, который нам нужен для отображения на ui. Это позволит нам обернуть этот проект в LiveData. Project, New, Java Class, RichProject. Щелкаем OK, добавляем. Это не сущность, это обычный Java Class, внутри которого находится обычный project. Project. В проекте у нас есть owners, тогда owners еще. private List<Owner> mOwners. Ctrl+N, геттеры. Вообще пускай это будут публичные поля, ничего страшного. Добавим аннотацию embedded, а здесь аннотацию Relation. Тем самым связываем список owners, табличку из owner'ов с этим классом. entity = owner.class, entityColumn в owner по-моему это project_id. Да. parentColumn это просто id.

Дальше перейдем в project и сделаем то же самое с cover. Cover – на самом деле не список, это один элемент, так что можем сделать его embedded. Нам не нужна ссылка на project id, потому что он уже находится внутри таблички с project'ами, то есть отношение один к одному. PrimaryKey у нас будет cover_id. foreignKeys и все такое удаляем.

Хорошо. Теперь. Project добавили, cover добавили. Я полагаю, что приложение ругнется, скажет, что мы изменили базу данных, но не поменяли версию. Давайте мы просто удалим наш проект, щелкаем OK, запустим заново. Теперь у нас будет старый проект, но с новой базой данных, без увеличения версий. Что у нас в холдере, адаптере, кстати этот код можно удалить.

Теперь переходим в Dao и удаляем метод getCoverFromProject, потому что он уже внутри проекта. То есть обложка уже внутри списка проектов. Давайте создадим новый метод, назовем его getProjectsLive. Возвращать он будет нам LiveData от списка RichProject. Запрос к базе данных будет выглядеть как query, то же самое, что и раньше. select * from project. Оно пишет, что он никогда не используется, давайте в storage добавим новый метод. public void getProjectsLive, return, тут не void, а LiveData от List от Project. RichProject. Метод assemble меняем, этого больше не будет. getOwner, вместо пары просто возвращаем список owner'ов. Return owners точнее. Тут у нас list. Это удаляем. List<Owner> owners = assemble(projects); Тут owners. Assemble переименуем в getOwners. Этот метод удаляем. Что можем сделать дальше. Dao у нас готово. Переходим в сборку. Собралось без ошибок.

Хорошо. На этом текущий урок завершается, в следующем мы добавим новую функциональность, а именно LiveData из Room в нашу ViewModel.

## 1.1.13. Получение Live данных из БД

Продолжаем работать над Behancer'ом. Давайте откроем нашу ViewModel, ProjectsViewModel, и попробуем реализовать новую функциональность. Первым делом меняем MutableLiveData на

LiveData, List<Project> на List<RichProject>. Вместо new LiveData в конструкторе обращаемся к нашему Storage и пишем getProjectsLive. Теперь у нас данные из Room берутся прямо в виде LiveData. Это означает, что как только данные в таблице обновятся, обновится и список проектов. Нам нужно будет немножко переделать запрос на LoadProjects, потому что в текущем виде он обновляет проекты, это нам не катит. Нам нужно обновлять саму базу данных. Это у нас тут тоже есть, но мы сделаем чуть-чуть по-другому. Давайте создадим новый метод, назовем его public void updateProjects. Сначала все скопируем. Ctrl+C, Ctrl+V. Получили данные, давайте замапим их в Projects, получается getProjects.

Дальше работаем со списком проектов. doOnSuccess у нас не будет, потому что запись в Storage у нас будет в subscribe. Пишем mStorage.insertProjects(response). В doOnSuccess перенесем это значение. Убираем точку с запятой. onErrorReturn тоже перенесется в throwable. subsrcibeOn, observeOn остается. Projects.postValue на response у нас больше нет. В throwable пишем следующий код. Если список проектов, которые мы получили с базы данных, пустой, то показываем ошибку.

```
1  mIsErrorVisible.postValue(mProjects.getValue().size()==0);
2  mProjects.getValue() == null;
```

Либо null, либо ноль. Storage.insertProjects выдает ошибку. Закомментируем этот код, переносим все сюда. public void insertProjects(List<Project> projects). Ctrl+X, Ctrl+V, insertProjects, getProjects. Исправили ошибку. Здесь у нас не MutableLiveData, а LiveData<List<RichProject». Вместо loadProjects у нас updateProjects. Здесь тоже updateProjects. Это удаляем, здесь у нас private и дальше по цепочке нам нужно пройтись по всем классам, которые использовали наши данные.

Первым делом это CustomAdapter, который теперь должен принимать List<RichProject>. Projects Adapter тоже должен получать <RichProject> и тут <RichProject>. Проекты в getItemCount нужно проверить на не null, потому что мы написали, что Projects теперь это данные из базы данных. Но данные из базы данных нельзя забрать моментально, в какой-то момент они могут быть null, поэтому мы пишем, что

```
1  mProjects == null 0 size;
```

Все, у нас студия ругнется, ошибка там все-таки может быть. Это у нас не Project, а RichProject. На bind тоже передаем RichProject. Дальше сюда продолжаем RichProject. Точка, mProject, getCover. Shift+F6. item. Owners у нас тоже, relation может не быть доступным сразу, поэтому мы ее тоже обернем в if/else. Совсем забыл. observeOn убираем, потому что весь этот запрос

должен работать в фоновом потоке, мы не можем добавлять в базу данных в главном потоке. Видим рабочий проект.

В принципе все. Теперь данные из базы данных мы получаем в виде LiveData, то есть как только они изменятся, изменится и интерфейс. Давайте, кстати, проверим это. Можем поступить следующим образом. Я здесь вижу 12, 12, 15, давайте это отсортируем. order by published_on desc. Запустим. Самые поздние проекты будут в начале списка. Теперь сделаем следующее. Зайдем в ProjectsAdapter и закомментируем метод updateProjects. Также в getProjects напишем новый запрос, например, на dog какой-нибудь.

Что сейчас происходит? При запуске из базы данных потянулись находящиеся в этой базе данные. Это выполнился метод Storage getProjectsLive. Метод updateProjects не выполнялся. Сейчас я запущу обновление экрана и посмотрю, что произойдет со списком. Список обновился сразу же. Хотя мы в запросе не работаем со списком, мы работаем с базой данных. Мы просто вносим данные в таблицу, точнее, так как мы подписаны на изменение этой таблицы, мы сразу же увидели изменение, как-то так. В следующем видео мы разберемся с paging library, то есть мы будем получать данные из таблицы небольшими порциями, так называемыми страницами. До встречи.

## 1.1.14. Получение данных страницами из БД

Приветствую еще раз. В этом видео мы собираемся – я собираюсь, вы смотрите – добавить paging library. Это архитектурный компонент от Google. Для чего он нужен? Он реализует докручиваемый список, как только мы добираемся до конца списка, не до самого конца, а до определенного элемента перед концом, то срабатывает callback и в этот список догружаются новые данные, новые элементы. В этом уроке мы сделаем такую реализацию, которая будет брать данные из базы данных. Непосредственно сам запрос на updateProjects или что-то там еще мы пока рассматривать не будем. Давайте начнем.

build.gradle, добавляем paging library, Ctrl+V. Я опять же его скопировал откуда-то. Нет, пока оставим. Sync now, синхронизировались. Начнем, пожалуй, с Dao. Мы можем получать данные из Room, обернутые в страницы. Для этого напишем новый запрос, причем сам query-запрос будет таким же, как у getProjectsLive, только вот возвращаемое значение будет другим. Вместо LiveData пишем DataSource.Factory <Integer, RichProject>. Названием метода getProjectsPaged. Теперь открываем наш Storage, давайте использовать метод getProjectsPaged. Ctrl+C, Ctrl+V. Название метода getProjectsPaged, возвращает он теперь не List, а PagedList. PagedList. Внутри у нас new LivePagedListBuilder. Key value можем не писать. getProjectsPaged, размер страницы

30 элементов. Точка build. Page size, то есть сколько элементов мы будем получать на загрузке можно вынести в константу.

Command+Alt+C, PAGE_SIZE. Хорошо, теперь идем во ViewModel. Вместо getProjectsLive используем getProjectsPaged. Alt+Enter. Change field type to, допустим. Здесь что? Здесь тоже Alt+Enter. Хорошо. ViewModel у нас теперь работает с paged-листом. Только вот адаптер с ним работать не умеет. Чтобы он умел это делать, его нужно отнаследовать от специального paged-адаптера. Пишем ProjectsAdapter extend, удаляем это все. PagedListAdapter, T RichProject, ViewHolder такой же, как и раньше. ProjectsHolder. List<Projects> убираем из конструктора и из полей. Ctrl+Alt+L. Здесь нам нужно добавить super. Super принимает на вход diff Callback. diff callback нам тоже нужно создать. Diff callback – это специальный callback, который проверяет, что два списка одинаковые либо разные, сравнивает два списка. Давайте создадим его.

```
1  private static final DiffUtil.ItemCallback<RichProject> CALLBACK = callback;
```

Дальше пишем areItemsTheSame, это у нас

```
1  oldItem.mProject.getId() == newItem.mProject.getId();
```

areContentsTheSame. oldItem.equals(newItem). Соответственно мы проверяем, что изменился сам объект либо изменилась начинка объекта. Callback скармливаем в super. Callback, точка с запятой. getItemCount нам больше не нужен, меняем onBindViewHolder. У нас нет project'ов, поэтому мы просто вызываем метод getItem. Если этот item не null, то bind'им его, Ctrl+Alt+L. Идем в CustomBindingAdapter и меняем конструктор. Адаптер, submitList и скармливаем ему первую порцию projects, вместо list'а показываем теперь, что это paged-лист.

Хорошо, мы закончили изменять наш проект, теперь посмотрим, как это все работает. Для этого нам нужно будет чуть-чуть пошаманить, зайдем в Storage и изменим page size с 30 на 10, чтобы нам проще было уловить момент загрузки. Переходим в ProjectsAdapter и ставим точку останова на onBindViewHolder. Щелкаем на дебаг. Сработала точка останова, мы в адаптере. Открываем ProjectsAdapter, то есть this. Открываем Differ, в нем хранятся наши элементы. Видим PagedList, size 96, то есть в списке должно быть 96 элементов. Прокручиваем вниз, видим 29. PagedList не стал брать сразу всю таблицу, все 96 элементов. Вместо этого он взял ту пачку, которая сейчас ему необходима. Возобновим программу. Нужно убрать точку останова. Возобновляем, запускаем. Давайте чуть-чуть прокрутим и снова поставим точку останова, видим, что весь список целиком загружен. Чем дальше вы прокручиваете список, тем дальше порционно, постранично данные берутся из вашей таблицы. В этом и суть PagedAdapter.

Есть еще одна возможность, это дополнительный callback. В storage в builder'е мы можем добавить BoundaryCallback. Это тот callback, который выполнится, если список дойдет до конца и ему окажется нечего загружать. Тогда он пойдет в сеть и оттуда попытается скачать новые данные.

Хорошо. На этой неделе мы познакомились с MVVM подходом, с MVVM-паттерном presentation слоя, то есть слоя, который работает с интерфейсом. Мы изучили Databinding библиотеку и архитектурные компоненты. Можно использовать их по отдельности, можно использовать вместе. Как видите, они друг друга прекрасно дополняют. Я рекомендую вам самим попробовать использовать у себя на проекте или где-либо еще эти библиотеки. Удачи в обучении.

**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 “Архитектура Android-приложений”

### Блок 1. Быстрый старт в Android-разработку

- Описание платформы Android
- Знакомство с IDE — Android Studio и системой сборки — Gradle
- Дебаг и логгирование
- Знакомство с основными сущностями Android-приложения
- Работа с Activity и Fragment
- Знакомство с элементами интерфейса — View, ViewGroup

### Блок 2. Многопоточность и сетевое взаимодействие

- Работа со списками: RecyclerView
- Средства для обеспечения многопоточности в Android
- Работа с сетью с помощью Retrofit2/Okhttp3
- Базовое знакомство с реактивным программированием: RxJava2
- Работа с уведомлениями
- Работа с базами данных через Room

### Блок 3. Архитектура Android-приложений

- MVP- и MVVM-паттерны
- Android Architecture Components
- Dependency Injection через Dagger2
- Clean Architecture

### Блок 4. Тестирование и работа с картами

- Google Maps

- Оптимизация фоновых работ
- БД Realm
- WebView, ChromeCustomTabs
- Настройки приложений
- Picasso и Glide
- Unit- и UI-тестирование: Mockito, PowerMock, Espresso, Robolectric

### Блок 5. Дизайн и анимации

- Стили и Темы
- Material Design Components
- Анимации
- Кастомные элементы интерфейса: Custom View

### Блок 6. Облачные сервисы и периферия

- Google Firebase
- Google Analytics
- Push-уведомления
- Работа с сенсорами и камерой