



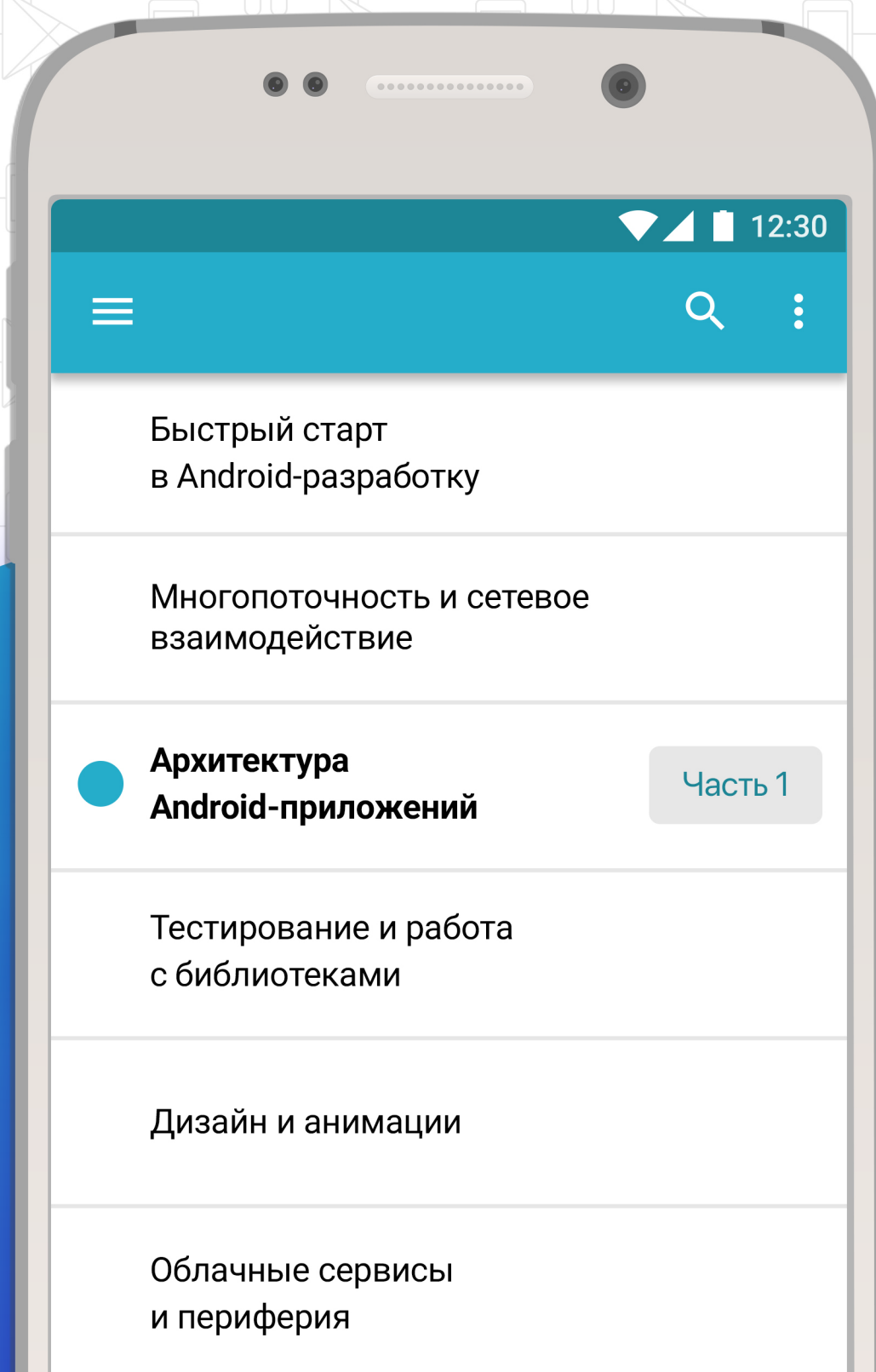
фонд развития
онлайн образования
eidf.ru

e·legion

academy.e-legion.com

Программа Android-разработчик

Конспект



Оглавление

1 НЕДЕЛЯ 1	2
1.1 Паттерн MVP	2
1.1.1 Знакомство с курсом	2
1.1.2 Для чего нужна архитектура	2
1.1.3 Принципы SOLID	5
1.1.4 Знакомство с приложением Behancer	7
1.1.5 Получение API key	9
1.1.6 Обзор Model/View/Presenter	10
1.1.7 Создание базовых классов MVP	12
1.1.8 MVP в ProjectsFragment	13
1.1.9 Знакомство с Моху	15
1.1.10 Добавляем Моху в Behancer	16

Глава 1

НЕДЕЛЯ 1

1.1. Паттерн MVP

1.1.1. Знакомство с курсом

Рад вас приветствовать в третьем блоке программы по Android-разработке. Вы уже знакомы с фреймворком и вполне себе в состоянии написать небольшое клиент-серверное Android-приложение. Но как насчет больших, крупных проектов? Что делать, если компонентов так много, что боишься что-либо трогать, а то вдруг сломается? В больших приложениях никуда без хорошей структуры и организации классов. Попросту говоря, никуда без архитектуры.

Поэтому в этом блоке мы и займемся изучением распространенных архитектурных подходов. Мы изучим MVP и MVVM. Сначала в своей реализации, а потом и с помощью специальных библиотек: Моху и Architecture Components. Изучим внедрение зависимостей, DI, что это такое, и зачем оно нужно. И снова разберем библиотеки, помогающие с DI. Это Dagger2 и Toothpick. В конце концов разберем парадигму Clean Architecture, она же «чистая архитектура», которая задает жесткие рамки для разработчиков, но в то же время награждает его абсолютным контролем над проектом. Давайте начнем.

1.1.2. Для чего нужна архитектура

Зачем мне архитектура в моем приложении? Все ведь и так нормально работает. Возможно, вас посещали такие мысли, и, возможно, вы с какой-то стороны правы. Но давайте разбираться.

Во-первых, есть существенная разница между подходом к разработке приложений в команде и в одиночку. Если вы разрабатываете приложение самостоятельно, то код вам знаком всегда, вы ведь сами его написали или скопировали со Stack Overflow. Вы в курсе, как приложение работает, где что инициализируется, где какое поведение, как устроена логика и так далее. Но в команде дела обстоят по-другому. Я не знаю, что написал мой коллега, и он точно так же не знает, что написал я. Только если не проводить постоянно review.

Во-вторых, создание приложений – это долгий процесс. Чем дольше вы над ним работаете, тем меньше вам хочется менять старый, уже написанный, и главное рабочий код. Тем меньше у вас контроля над приложением. Это только полбеды.

Прежде чем идти дальше, давайте сначала определимся с термином «бизнес-логика» или «бизнес-правила». Бизнес-логика – это то, что делает приложение. Бизнес-логику приложения вы можете описать человеку, при этом не вдаваясь в технические подробности реализации на какой-либо платформе. Описание бизнес-логики чата может быть таким: «У нас есть список контактов, из которых мы можем выбирать кому написать, можем отправить ему сообщение, картинку или файл, можем заблокировать контакт и тогда он уже не сможет нам написать, и так далее». Подумайте, если бы писали чат не для Android, а для Web или iOS, то его работа, его бизнес-логика поменялась бы? Нет, поменялись бы отображения и детали реализации того или иного механизма, но чат остался бы чатом.

А теперь посмотрим на слайд. Здесь у нас в Activity происходит сетевой запрос. Activity – это компонент, который отвечает за отображение интерфейса и обработку пользовательского ввода (нажатия, жесты). Хорошо, вспомнили, закрепили. Сетевой запрос возвращает нам данные, которые нужно отобразить, плюс, если произошла ошибка, то нужно дать пользователю знать об этом. Что я только что произнес? Бизнес-правила. А на слайде произошло смешивание логики приложения с ее отображением. Если я захочу вместо Activity использовать фрагмент, мне придется переместить этот код. Если я захочу добавить новый запрос для этого экрана, мне придется изменить этот код. Если я захочу обработать несколько вариантов ошибки, мне тоже придется менять весь код. То есть изменение отображения нельзя сделать без изменения бизнес-логики. И наоборот. Так как они находятся в одном месте, они связаны. Это плохо.

```

@Override
public void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    mRecyclerView.setAdapter(mProjectsAdapter);

    ApiUtils.getApiService().getProjects(BuildConfig.API_QUERY)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(
            response -> {
                mErrorView.setVisibility(View.GONE);
                mRecyclerView.setVisibility(View.VISIBLE);
                mProjectsAdapter.addData(response.getProjects(), true);
            },
            throwable -> {
                mErrorView.setVisibility(View.VISIBLE);
                mRecyclerView.setVisibility(View.GONE);
            }
        );
}

```

Рис. 1.1: Слайд

Еще подумайте вот над чем. Activity – поистине громоздкий компонент. По сути это god object – он может все. Фреймворк Android так устроен, и мы не можем изменить Activity. В наших силах не нагружать Activity еще и бизнес-логикой. Пусть ей занимаются отдельные специальные классы. Перефразирую последнее предложение. Бизнес-логика должна быть отделена от отображения. Если они отделены, то как их использовать вместе? Тут нам на помощь приходят MVP-паттерны. В этом курсе мы рассмотрим MVP и MVVM. Нельзя не упомянуть предка этих паттернов – MVC. MVC впервые был предложен в 78-79 гг. Трюгве Реенскаугом, норвежским специалистом, тогда работавшим над языком Smalltalk. MVC означает Model View Controller. Это ни классы, ни пакеты в структуре проектов. Это слои, то есть группа классов, решающих однотипные задачи. Model – то есть «модель» – это и есть бизнес-логика, это данные приложения, а также способы их получения, хранения, обработки и передачи. View – не путать с андроидным view - это представление, отображение, то есть то, что видит пользователь. Контроллер обрабатывает пользовательский ввод. Опять же, не забывайте про время, конец семидесятых. Пользовательский ввод – это не нажатие на сенсорный экран, это клавиатура и мышь. Сейчас для каждого языка найдется библиотека UI-компонентов, всяких кнопок, полей ввода, контейнеров для изображений и текста с уже предустановленным и понятным поведением. Тогда интерфейс нужно было рисовать вручную, и нужно понимать, что тогда идея разделять код по ответственности была революционной. Возможность спокойно менять код, не затрагивая остальные слои,

была встречена на ура. Оно и понятно.

Что касается нас, Android-разработчиков. В первоначально задуманном виде MVC реализовать не получится, так как взаимодействие происходит через экран, а это у нас View, но мы можем использовать некоторые разновидности MVC, коих за прошедшее время появилось предостаточно. Мы разберем MVC и MVVM, как самые распространенные и укладывающиеся с фреймворком Android паттерны. Да, отвечая на тот самый вопрос, архитектура, то есть правильное структурирование классов и логики в приложении нужна для большего понимания и контроля над происходящим, легкого ввода разработчиков в проект, для простоты тестирования бизнес-логики и для переиспользования отдельных компонентов в других модулях и приложениях. Я думаю, это того стоит.

1.1.3. Принципы SOLID

Привет! От вопросов глобальных к вопросам более мелким. В этом видео мы познакомимся с принципами грамотного проектирования классов, а именно с SOLID. SOLID – это акроним, введенный Робертом Мартином для пяти основных принципов программирования. Мы рассмотрим каждый принцип и выясним, почему эти принципы нужно соблюдать. Если вкратце, то для того, чтобы код был лаконичным, понятным, с ожидаемым поведением, без лишних зависимостей и ненужной логики. Давайте начнем.

S – Single Responsibility. Каноническое трактование этого принципа следующее: каждый объект должен иметь одну ответственность и только эта ответственность должна быть полностью инкапсулирована в класс. Все его поведения должны быть направлены исключительно на обеспечение этой ответственности. Тут должно быть все понятно. Класс должен заниматься чем-то одним. Давайте приведем пример из реального мира. Возьмем швейцарский нож. В принципе отличная вещь. Множество вариантов использования и все такое. Но кто из нас будет пользоваться швейцарским ножом, если под рукой есть нормальный специализированный инструмент? Резать колбасу кухонным ножом удобнее, пилить доски пилой удобнее. Вы меня поняли, надеюсь. Пример из программирования. Рассмотрим класс карточки работника. Все стандартно: поля, методы, все такое. Давайте подробнее к методам. Изначально реализация методов для расчета налогов и повышения в должности находится внутри класса. Это неправильно, это прямое нарушение принципа ответственности, поэтому мы выносим эти методы в отдельные классы. Пусть они этой логикой и занимаются.

Дальше. O – Open Closed. Принцип открытости/закрытости. Опять же каноническое трактование из википедии: программные сущности (классы, модули, функции и т.п.) должны быть

открыты для расширения, но закрыты для изменения. Простыми словами: новое поведение должно быть добавлено в наследника класса, а не в сам класс. Этот принцип можно поддерживать, используя абстрактные классы, оставляя в них всю функциональность, которая у них будет меняться. На слайде можете наблюдать, как создаются абстракции для простейшей операции сохранения сущности. Мы видим `AbstractEntity` и двух его наследников: `AccountEntity` и `RoleEntity`. Также видим интерфейс-репозиторий с generic-параметром `AbstractEntity`. Две реализации этого репозитория для каждого из конкретных entity. Если бы мы не создали такие абстракции, нам пришлось бы постоянно проверять, какая реализация нам нужна, используя `if/else` или оператор `instance of`. Напротив, в нашем подходе класс не надо будет изменять. Вместо этого все изменения в логике будут содержаться в конкретных наследниках. Стоит также отметить, что нарушение данного принципа также автоматически нарушает и принцип единственной ответственности.

Дальше. L – Liskov Substitution. Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом. Простой пример. Мы написали класс-родитель со строковым и численными полями. Строка у нас приватная, поэтому используется сеттер для изменения значения. Число `protected`, доступно из потомков. Мы решили отнаследоваться от этого класса и переопределили метод `setText` так, что он вдобавок устанавливает в поле `value` нулевое значение. Если не знать про этот нюанс, то можно впоследствии словить непонятный баг, как в приведенном коде. На лицо нарушение принципа.

Дальше. I – Interface Segregation. Принцип разделения интерфейса. Клиенты не должны зависеть от методов, которые они не используют. Это наверное самый простой и понятный принцип. Принцип разделения интерфейса говорит о том, что слишком толстые интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о тех методах, которые им необходимы в работе. В итоге при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют. Яркой демонстрацией этого принципа является создание интерфейсов с `callback`'ами вместо одного большого сложного.

Наконец, D – Dependency Inversion. Принцип инверсии зависимостей. Он состоит из двух постулатов. Первое: модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций. Второе: абстракции не должны зависеть от деталей. Детали должны зависеть от абстракции. Давайте рассмотрим пример. Он очень простой. `Reporter` просит `ReportBuilder` создать отчеты, затем `ReportSender` их по очереди отправляет. Получается, внутри класса `Reporter` две зависимости, причем конкретных. Если нужно будет добавить еще логику, то придется все поменять. Что нужно сделать с точки зрения инверсии зависимостей? Нам нужно создать базовые интерфейсы для создания и отправки отчетов, передавать их реализации на вход конструктора класса `Reporter`. Мы создали интерфейсы и передали их в конструктор `Reporter`'у. Этот процесс называется внедрением зависимостей и ему

посвящена целая неделя блока. В целом мы решили проблему, Reporter работает только с интерфейсами, то есть с абстракциями.

Есть еще парочка принципов, о которых нельзя не упомянуть. DRY. Don't repeat yourself. Не повторяйся. Цель этого принципа – это избегание копирования и вставки. Вносите одинаковую логику в методы и классы и используйте уже их.

KISS. Keep it simple, stupid. Не усложняй, дурачок. Не надо использовать сложное решение, если есть простое. Разрастись вы всегда успеете.

YAGNI. You aren't gonna need it. Тебе это не понадобится. Простой принцип, который сложно соблюдать. Нужно писать только тот код, который вам нужен здесь и сейчас. Если же есть подозрения об изменении функциональности в будущем, то нужно спросить себя: «А что в моем текущем коде может помешать добавить эту функциональность?», и поправить что надо.

В конце концов, соблюдение принципов SOLID дает нам следующие плюсы. Мы всегда знаем, чем занимается класс. Мы можем переиспользовать класс в другом модуле или даже проекте. Мы можем легко написать тесты к таким классам. Мы можем спокойно менять код, не боясь сломать что-то в неожиданном месте. Будьте SOLIDнее – это того стоит.

1.1.4. Знакомство с приложением Behancer

Привет! В этом видео мы познакомимся с нашим приложением, которое мы будем использовать на протяжении всего курса для того, чтобы тестировать на нем разные подходы к архитектуре, внедрять DI, а потом вообще Clean Architecture.

Что из себя представляет это приложение? Оно называется Behancer, потому что работает с API сайта Behance. Behance – это сайт для дизайнеров, для того чтобы они могли выкладывать какие-то свои наработки, свое видение дизайна. Соответственно, приложение может получать кое-какие данные с этого сайта. Давайте посмотрим. Вообще я попросил своего коллегу посмотреть первые два курса нашей программы и написать приложение, опираясь только на эти знания.

Давайте зайдем и посмотрим, что из себя Behancer представляет. Мы видим папку Common, давайте откроем. Видим SingleFragmentActivity – это Activity с fragment container'ом, в котором будут меняться фрагменты. Вот он, этот код, changeFragment. Также в нем находится SwipeRefreshLayout, значит, мы можем дергать обновления. В чем фишка? Обновления находятся в activity, обновить нужно фрагмент, поэтому мы реализовали два интерфейса, Refreshable

и RefreshOwner. SingleFragmentActivity – это RefreshOwner, то есть у него есть логика для показа процесса обновления. Фрагменты, которые будут находиться в нем, будут реализовывать Refreshable, если им нужно будет обновиться. Видим, что в методе onRefresh мы находим фрагмент, который находится в контейнере; если он Refreshable, то вызываем у него метод onRefreshData. Refreshable и RefreshOwner – это обычные интерфейсы, ничего такого. Зайдем в data-api-database-model. model – это project, user и еще Storage. В api у нас ApiKeyInterceptor, кажется, тоже с проекта Марата. BehanceApi – ничего сложного, это обычный ретрофитовский интерфейс. В базе данных у нас BehanceDao. Dao для работы с таблицами, тоже ничего необычного. BehanceDatabase, стандартно все.

Заходим в модель, в проекты. Получается, ProjectResponse, в котором находятся проекты, в котором находятся id, имя, дата публикации, обложка и хозяева, owners. Также я вижу, что Влад менял идею Марата насчет того, что класс одновременно является и POJO-объектом для десериализации (мы видим SerialisedName), и в то же время entity-сущностью для базы данных room, то есть этот класс можно использовать одновременно и в gson'е и в room. В маленьких проектах это оправданно, в больших лучше так не делать.

Дальше тоже ничего необычного, обычное клиент-серверное приложение, обычная логика. Storage – это класс, который работает с Dao, он записывает данные в него и выдергивает данные из него, видим insertProjects, getProjects – обычный кэш. Обычный класс для работы с кэшем.

Откроем ui, открываем проекты. ProjectsActivity – это наследник SingleFragmentActivity, в котором у нас находится ProjectsFragment. ProjectsFragment – это обычный экран со списком, то есть в нем находится RecyclerView, а также адаптеры, ссылка на RefreshOwner, ссылка на Storage. Storage он, кстати, получает из SingleFragmentActivity, так как в нем также реализован механизм Storage. Adapter самый обычный. Holder самый обычный. При нажатии вызывается метод onItemClick, который определен во фрагменте, при нажатии на элемент списка открывается экран профиля. Открываем профиль, в него передается username, и во фрагменте с помощью этого username мы выдергиваем данные по этому профилю. Мне кажется, ничего необычного, запрос самый обыкновенный.

utils. DataUtils преобразовывает миллисекунды в строковый формат, понятный для чтения, то есть дата-месяц-год. ApiUtils, тоже, мне кажется, из проекта Марата – обычная настройка сетевого слоя. У нас тут gson, OkHttp, клиент и Retrofit. AppDelegate – это application, в котором инициализируются, настраиваются базы данных, и также ссылка, чтобы мы могли ее достать, точнее, мы можем достать storage, основываясь на базе данных.

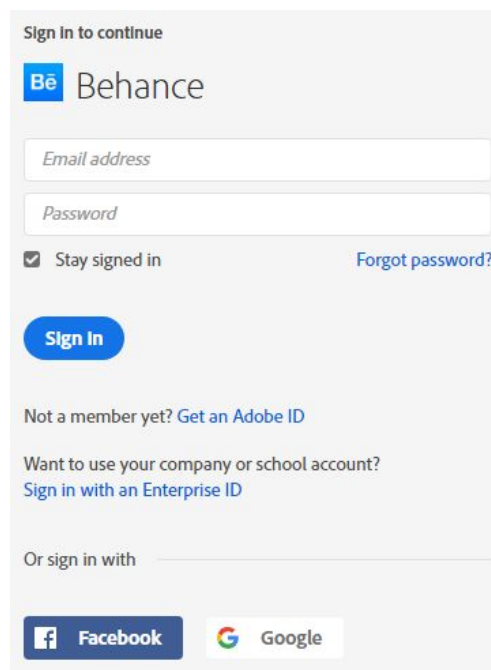
Давайте запустим и посмотрим, как оно работает. Приложение запустилось, обновляется, получили список проектов, щелкаем на любой из проектов, видим информацию о профиле, о хозяине

этого проекта. В принципе на этом все. Давайте теперь экспериментировать и прикручивать различные архитектурные подходы к этому приложению.

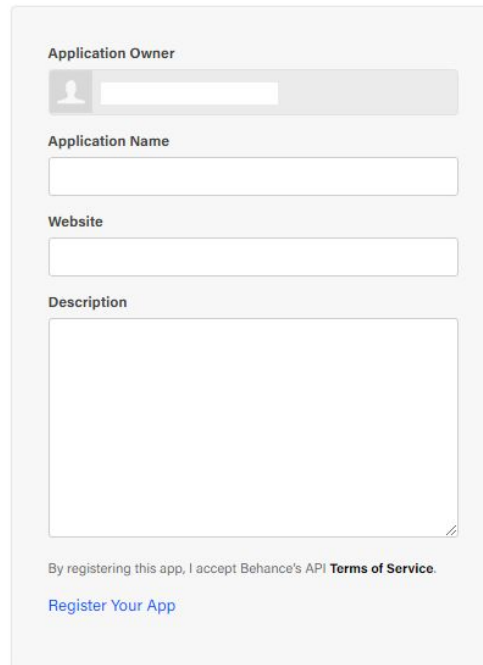
1.1.5. Получение API key

Для того, чтобы мы могли обращаться к сервису Behancer, необходимо иметь собственный API-ключ. Чтобы получить его, необходимо зарегистрировать свое приложение на сайте. Последовательность действий:

- Зайти на сайт <https://www.behance.net/dev/register>, зарегистрироваться или авторизоваться:

The image shows a Behance sign-in form. At the top, it says "Sign in to continue" with the Behance logo. Below the logo are two input fields: "Email address" and "Password". There is a checkbox labeled "Stay signed in" and a link "Forgot password?". A blue "Sign in" button is below the password field. Further down, it says "Not a member yet? Get an Adobe ID". Below that, it asks "Want to use your company or school account?" with a link "Sign in with an Enterprise ID". At the bottom, it says "Or sign in with" followed by two buttons: "Facebook" and "Google".

- Ввести информацию о своем приложении и нажать кнопку «Register Your App»:



The image shows a registration form for the Behance API. It includes fields for 'Application Owner' (with a profile icon placeholder), 'Application Name', 'Website', and 'Description' (a larger text area). At the bottom, there is a checkbox for 'By registering this app, I accept Behance's API Terms of Service' and a blue link 'Register Your App'.

- Скопировать ключ из поля API KEY / CLIENT ID. Открыть наш проект, файл build.gradle уровня модуля. В секции defaultConfig найти строку

```
1      buildConfigField "String" , "API_KEY" , '"YOUR_KEY"'
```

и заменить значение последнего параметра на свое.

1.1.6. Обзор Model/View/Presenter

Привет! В этом видео мы изучим первый MV-паттерн нашего курса Model-View-Presenter. MVP имеет три основных компонента. Model отвечает за данные, а также за методы получения, сохранения и обработки этих данных. View отвечает за визуальное представление данных. Presenter – посредник между View и Model. Он получает команды с View, обрабатывает данные, при необходимости обращаясь к Model, и передает результат обратно во View.

Presenter контролирует взаимодействие между Model и View, накладывая следующие ограничения. View не имеет доступа к Model. Presenter привязан к одному View, View полностью пассивен,

его задача заключается в отображении данных и передаче команд в Presenter. Приведем пример. Предположим, что у нас есть экран с кнопкой, при нажатии на которую с сервера загружается список с элементами и выводится на экран. Данный код не разбит на слои, то есть и загрузка данных, и их отображение на экране происходит в рамках одной activity. Давайте применим паттерн MVP. Очевидно, что отобразить данные – это работа view-компонента. Соответственно, вынесем данную часть кода в отдельный интерфейс. Его будет реализовывать наша activity, в которой будут отображаться данные. В свою очередь загрузка данных с сервера – это задача Presenter'а. Создадим соответствующий класс и перенесем туда этот метод. Также Presenter'у понадобится ссылка на view-компонент, чтобы передать данные для отображения. View он может получить, например, в конструкторе, как показано на слайде. Обратите внимание, что сейчас мы загружаем данные в Presenter'е для простоты и понимания. В конце курса этим будет заниматься другой объект, репозиторий.

Теперь взглянем на нашу обновленную activity. Она реализует интерфейс View, метод setData, а также имеет ссылку на Presenter. И теперь при нажатии на кнопку Activity вызывает метод Presenter'а и уже он занимается получением данных. Также заметьте, что activity ничего не знает о другом слое, то есть о процессе получения данных, она лишь передает ui события от пользователя к Presenter'у. В свою очередь Presenter ничего не знает о процессе отображения данных на экран, он просто передает данные обратно во View. Как видите, ничего сложного.

Рассмотрим плюсы и минусы данного паттерна. Основные преимущества исходят из того, что компоненты ничего не знают друг о друге. Первый плюс заключается в том, что слои можно независимо тестировать. На практике это выражается в том, что для Presenter'а и для модели пишутся относительно дешевые в плане трудозатрат unit-тесты, тогда как для activity написать unit-тесты возможности нет. Второй плюс в том, что наш код становится читабельным, понятным, его легко поддерживать и расширять.

Что касается минусов MVP – это с непривычки значительное увеличение объема кода, если сравнивать с проектом без какой-либо архитектуры. Как было показано в примере, один класс, activity, превратился в три отдельных класса, каждый из которых отвечает за свою часть работы, но этот минус, избыток кода, с лихвой перекрывается плюсами подхода. Рекомендуются все связанные классы в структуре проекта хранить в одном пакете.

Давайте подведем итоги. MVP хорошо для первого знакомства с MV-паттернами. Он является удобным, понятным, простым, как следствие, самым популярным среди разработчиков.

1.1.7. Создание базовых классов MVP

В данном занятии мы создадим три класса для MVP. Это BasePresenter, BaseView и PresenterFragment. Начнем с BasePresenter'а. Перейдем в behancer-проект. Package common, сделаем New, Java Class, BasePresenter, нажмем Enter. Далее, добавим protected-переменную CompositeDisposable.

```
1  protected CompositeDisposable mCompositeDisposable = new CompositeDisposable();
```

Эта переменная нужна для того, чтобы мы могли отписываться абсолютно от всех подписок, что у нас есть. Создадим метод

```
1  public void disposeAll() {
2      mCompositeDisposable.clear();
3  }
```

Далее создадим BaseView. Java Class, Interface, BaseView. Далее добавим три простых метода

```
1  public interface BaseView {
2      void showLoading();
3      void hideLoading();
4      void showError();
5  }
```

Далее создадим PresenterFragment, который унаследуем от Fragment(android.support.v4.app). Переопределим метод onDetach. Добавим protected abstract P getPresenter.

```
1  public abstract class PresenterFragment<P extends BasePresenter> extends Fragment {
2
3      protected abstract P getPresenter();
4
5      @Override
6      public void onDetach() {
7          if (getPresenter() != null) {
8              getPresenter().disposeAll();
9          }
10         super.onDetach();
11     }
12 }
```

Кстати говоря, в Base Presenter'е нужно добавить abstract. Вот и все. В этом занятии мы успешно создали три базовых класса для MVP, которые нам помогут в будущем.

1.1.8. MVP в ProjectsFragment

В данном занятии мы с вами добавим MVP в ProjectsFragment. Для этого создадим ProjectsView и ProjectsPresenter. Начнем с ProjectsView. Перейдем в behancer-проект. ui, projects, New, Java Class, Interface, ProjectsView. Enter. Унаследуем ее от BaseView, далее добавим два метода: showProjects, в который будем передавать список наших полученных проектов, далее добавим метод openProfileFragment, в который будем передавать String username.

```
1  public interface ProjectsView extends BaseView {
2
3      void showProjects(@NonNull List<Project> projects);
4
5      void openProfileFragment(@NonNull String username);
6  }
```

Теперь в папке projects создаем ProjectsPresenter, который будет наследоваться от BasePresenter. В этом presenter'е у нас будет две переменные. Это ProjectsView mView и private Storage mStorage. Далее добавим конструктор и добавим сюда два метода.

```
1  public class ProjectsPresenter extends BasePresenter {
2
3      private final ProjectsView mView;
4      private final Storage mStorage;
5
6      public ProjectsPresenter(ProjectsView view, Storage storage) {
7          mView = view;
8          mStorage = storage;
9      }
10
11     public void getProjects() {
12     }
13
14     public void openProfileFragment(String username) {
15     }
```

Далее переходим в `ProjectsFragment`. Теперь он наследуется не от фрагмента, а от `PresenterFragment`, в который мы передаем ему тип нашего `Presenter`'а. В нашем случае это `ProjectsPresenter`. Далее имплементируем его также от `ProjectsView`.

Где-нибудь внизу сочетанием клавиш `Ctrl+Alt` реализуем абсолютно все методы, которые у нас есть. `Disposable` нам теперь не нужен, теперь у нас это `ProjectsPresenter mPresenter`, который инициализируется в `onActivityCreated`. `mPresenter` равняется `new ProjectsPresenter`, в который мы передаем `this` и передаем наш `storage`. Далее избавляемся от `mDisposable`, он нам не нужен. Для начала изменим `onRefreshData`, теперь у нас не метод `getProjects`, а `mPresenter.getProjects`. Давайте посмотрим, где еще этот метод используется. Больше нигде. Логика из этого метода мы переместим в `ProjectsPresenter`, метод `getProjects`.

Теперь мы делаем `mCompositeDisposable.add` и будем добавлять наш `disposable` соответственно. Далее в методе `doOnSubscribe` мы делаем `mView.showRefresh`. Соответственно в `ProjectsFragment`'е в `showRefresh` будет вот этот код. Далее в `doFinally` мы делаем `RefreshOwner.setRefreshState false`. Соответственно в `hideRefresh` будет вот этот код. В методе `showError` будет вот этот код. `doFinally` мы вызываем `mView.hideRefresh`. Если у нас возникла какая-либо ошибка, мы делаем `mView.showError`. Далее, если пришел успешный `response`, то мы делаем `mView.showProjects` и передаем туда список наших проектов, то есть в нашем случае это будет наш `response`.

Наведем красоту. Далее перейдем в `ProjectsFragment`. В методе `getPresenter`, он будет возвращать наш `presenter`. В методе `showProjects` добавляем следующий код. Сюда будет приходить сразу список проектов, поэтому делаем вот так. В методе `openProfileFragment` копируем код из метода `onItemClick` и вставляем его сюда.

Кстати, метод `getProjects` нам больше не нужен. Теперь в методе `onItemClick` мы вызываем `mPresenter.openProfileFragment`. Мы забыли туда передать переменные. Давайте исправим это. Соответственно в методе `mView`, в методе `openProfileFragmentPresenter` мы будем вызывать `mView.openProfileFragment` у `view`. `Username` передаем также как и здесь. Это нужно для большего разделения логики, чтобы у нас был практически чистый MVP.

Вот и все. Теперь давайте посмотрим, как это работает в эмуляторе. Нажмем для проверки, обновим на всякий случай. Как вы видите все работает успешно и без ошибок. В данном занятии мы добавили MVP в `ProjectsFragment`.

1.1.9. Знакомство с Моху

Привет! В этом в этом видео мы познакомимся с библиотекой Моху, которая реализует паттерн MVP. В прошлых видео мы подробно рассмотрели возможности и удобства архитектуры MVP.

Немного повторим. View реагирует на поведение пользователя, передает команды в Presenter. Presenter их обрабатывает, используя или не используя модель, и говорит View, как она должна измениться. View – это интерфейс, чьей реализацией занимается фрагмент или activity. Основная проблема, которая стоит перед Android-разработчиками, которые используют MVP, это как сделать так, чтобы Presenter переживал смену конфигурации. В нашей прошлой реализации Presenter умирает вместе с компонентом, к которому он относится. Моху элегантно решает эту проблему, внося понятие View State.

View State – это еще одна прослойка, и находится она между View и Presenter'ом. Давайте рассмотрим эту замечательную анимацию, которую приготовили авторы библиотеки. Самый интересный момент в ней в том, что команды с Presenter'a во View сохраняются во View State, и когда к Presenter'у подключается новая View, например после переворота экрана, сохраненные команды передаются в нее. И состояния старой view и новой view становятся одинаковыми, потому что к ним были применены одни и те же команды. Логично.

Давайте рассмотрим, как использовать Моху. Предположим, что мы работаем с фрагментом, тогда нам нужно наследоваться от местного базового фрагмента, это либо MvpFragment, либо MvpAppCompatActivity. Presenter'у добавляем аннотацию @InjectPresenter, тогда будет использован его конструктор без параметров. Если же в Presenter нужно передать зависимости, то необходимо создать метод, который возвращает сформированный Presenter, и добавить к этому методу аннотацию @ProvidePresenter. View наследуется от MvpView, и первое, что бросается в глаза, это аннотация @StateStrategyType, которая указывает с какой стратегией – чуть попозже – эти команды будут запоминаться во View State. Стратегии можно указывать как для каждого метода отдельно, так и для всей View. Давайте познакомимся с ними.

В библиотеке Моху существует несколько стандартных стратегий, но никто не мешает вам создавать свои собственные, если хотите. По умолчанию стоит AddToEndStrategy. Авторы библиотеки говорят, что несмотря на то, что это не самое часто используемое поведение, оно самое понятное с первого взгляда, так как все команды в пересозданном view выполняются в том же порядке, в котором они были выполнены до переворота. Описание стратегии вы видите на слайде, но лучше увидеть все самим в приложении. В Presenter'е мы добавляем аннотацию @InjectViewState, и Моху сам создаст объект View State и даст нам к нему доступ через метод getViewState. Каждый раз, когда нам нужно обращаться к View в Presenter'е, мы используем именно метод getViewState.

Что из себя представляет View State? Вскользь я упоминал, что это некая прослойка между View и Presenter'ом. Если мы посмотрим, что у нее внутри, то увидим ViewCommands, а внутри ViewCommands непосредственно сам список команд и стратегии, которые будут применены.

Что из себя представляет ViewCommand? Как понятно из названия класса, это команда, некое действие, которое делает View, и которое нужно запомнить и повторить при восстановлении или при добавлении новой View. Сам по себе класс абстрактный, а конкретная реализация появляется путем кодогенерации, когда в его абстрактный метод apply добавляются именно те методы, которые мы уже определили в соответствующем View и в соответствующей стратегии. Надеюсь, понятно. Вот сгенерированный класс команды. Надо признать это действительно впечатляющая работа.

Раз уж начали, давайте продолжим раскрывать секреты Моху. Посмотрим, что внутри у MvpFragment'a, видим, что основная фишка базового фрагмента это MvpDelegate. Он следит за тем, чтобы были правильно инициализированы все экземпляры Presenter'a, и для этого нам нужна аннотация @InjectPresenter.

Вам, наверное, интересно, кто написал эту замечательную, мощную библиотеку. Приготовьтесь испытать приступ гордости, ибо создатели – наши соотечественники. Это Александр Блинов, руководитель Android-направления в группе компаний HeadHunter и Юрий Шмаков, скромно представляющийся как Android-разработчик. Плюс надо иметь в виду, что Моху дико популярна. Сообщество постоянно растет, помогает с power quest'ами, да и просто советами. Информации о Моху в интернете очень много: видео с конференций, статьи от создателей библиотеки и от тех, кто попробовал ее в производстве. Я рекомендую присоединиться к Telegram-каналу, а также почитать wiki на GitHub'e.

Скажем пару слов про идейного предшественника Моху – библиотеку Mosby, написанную Ханнесом Дорфманом. Mosby реализует MVP со встроенной поддержкой жизненного цикла activity и фрагмента. Основное отличие заключается в том, что Mosby сохраняет состояния, а не команды, как мы уже разбирали раньше.

Давайте теперь посмотрим на Моху в деле. Я думаю, это того стоит.

1.1.10. Добавляем Моху в Behancer

В данном занятии мы с вами добавим Моху в наш behancer-проект, который основывался на MVP и будем использовать функционал Моху presenter'a внутри ProjectsFragment'a.

Давайте перейдем в проект. Откроем Gradle Scripts, build.gradle (Module: app), далее добавим три зависимости, это сам Моку, annotation processor для moxy compiler'a и moxy app compat. Нажмем «проинициализироваться», синхронизация прошла успешно, теперь давайте обновим наши базовые классы.

Теперь BaseView у нас наследуется от MvpView. BasePresenter наследуется от MvpPresenter, в который мы должны передать ему тип view, соответственно в BasePresenter у нас добавляется генериковый тип V, который наследуется от BaseView, в mPresenter мы передаем V. Далее, можно перейти в PresenterFragment. Теперь здесь нам больше не нужен генериковый тип, он у нас может просто возвращать BasePresenter. PresenterFragment у нас наследуется от MvpAppCompatActivity.

Далее, перейдем в ProjectsFragment. У mPresenter'a сотрем идентификатор private и добавим ему аннотацию InjectPresenter. Далее найдем, где он у нас инициализировался. Вырежем эту строчку, она нам в будущем пригодится, и также перенесем метод getPresenter чуть повыше, чтоб нам было проще ориентироваться и мы видели все места, где у нас идет инициализация presenter'a. Добавим аннотацию ProvidePresenter, которая будет возвращать ProjectsPresenter, метод, который будет называться providePresenter. Соответственно, он будет возвращать просто new ProjectsPresenter.

Теперь давайте перейдем в сам presenter. Над классом добавим аннотацию InjectViewState. Введем ProjectsFragment, сотрем здесь генериковый тип.

Теперь давайте запустим эмулятор и посмотрим, ничего ли мы не сломали. You cannot use InjectPresenter in classes that are not View, which is typified target Presenter. Давайте перейдем в ProjectsPresenter и увидим, что мы в Base Presenter забыли добавить тип view, то есть в нашем случае это ProjectsView.

Давайте запустим еще раз. Приложение успешно запустилось. Как вы видите, все работает, все переходы есть. В данном занятии мы успешно добавили Моку в наш проект.

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