Клиент-серверное приложение

Получаем данные из сети

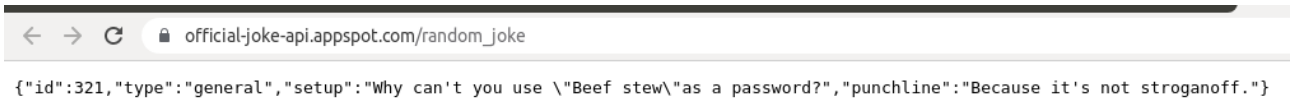
Содержание

1. Пишем юай слой без сервера

Итак, в предыдущей лекции мы обсудили принципы SOLID – что мы можем заменять реализации и получать зависимости извне. И в предыдущей лекции я показал зачем это делать : чтобы протестировать логику вашего класса без необходимости запускать все приложение. Поверьте, в реальных приложениях ваш функционал во-первых может зависеть от сотни всяких условий: начиная с наличия интернет соединения и заканчивая тем, что до вашего функционала нужно добраться еще (например залогиниться, перейти во вкладку настройки, там выбрать меню, перейти на другой экран и там уже проверить что код работает верно). Так что тесты спасают вас и ваше драгоценное время.

Но подождите, вы же не хотите сказать что мы постоянно интерфейсы и всего 2 реализации : базовую и тестовую? Конечно же нет. Когда мы будем проходить чистую архитектуру я покажу вам применение принципов на примере премиум юзеров и обычных на слое бизнес-логики. А сейчас мы рассмотрим еще один плюс заменяемости реализаций.

Проект который мы сейчас напишем достаточно прост. Есть API (далее апи) по которому мы можем получить случайную (рандом) шутку на английском языке. Что такое апи?

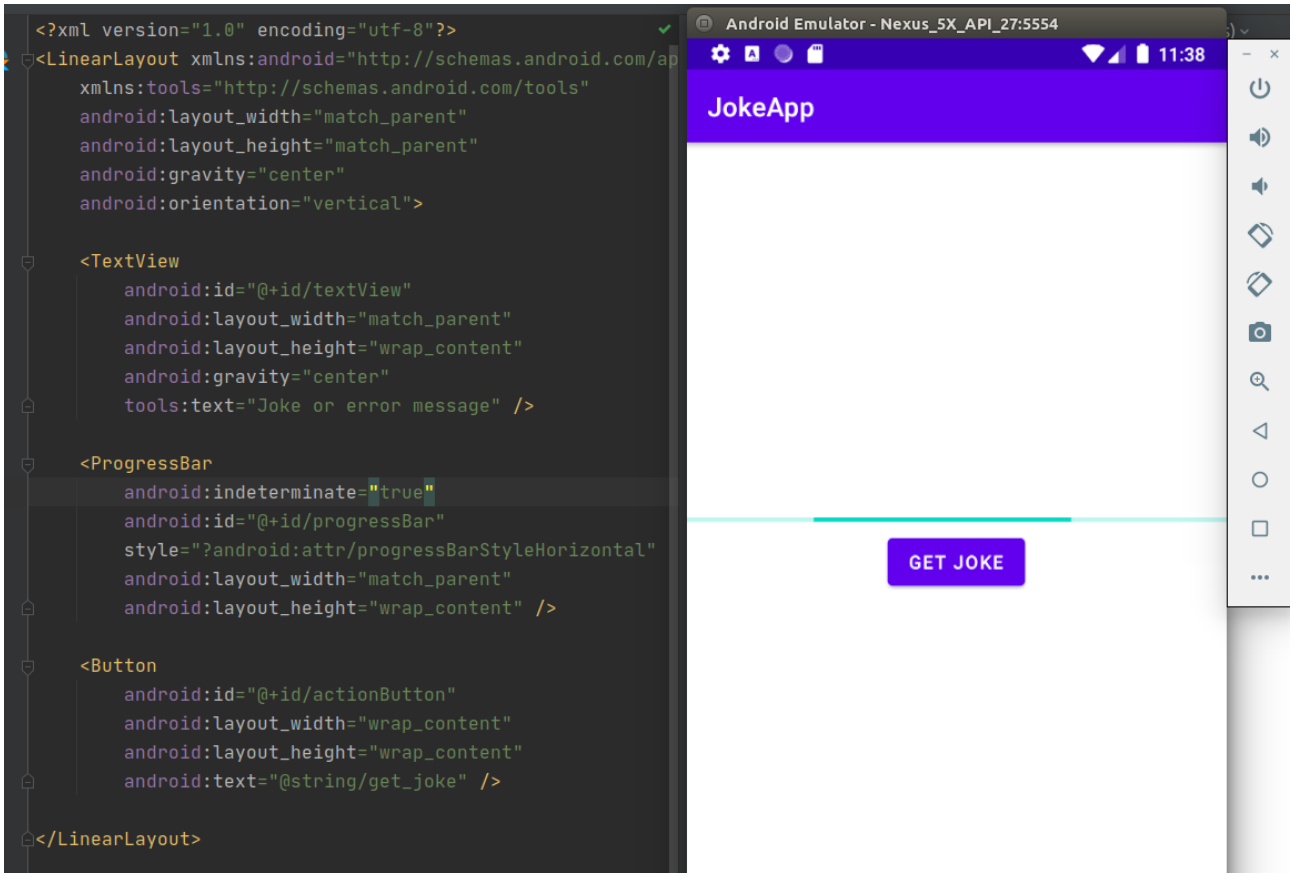


Говоря очень простым языком апи это тот адрес на сервере по которому доступна информация. Апи описывается в документации : как с ним работать. Т.е. какой адрес нужно использовать для получения информации, какие нужно передать параметры если ответ от сервера зависит от чего-то. Мы для простоты взяли очень простое открытое (бесплатное) апи. Как видите можно просто в браузере вбить ссылку и получить ответ. Но подождите, что это за непонятная строка? {“id”:321, “type… это JSON-format данных. Мы в Андроид пишем на котлин, ответ от сервера в данном случае приходит строкой но в формате JSON. Говоря простым языком – по ссылке мы получаем строку, которая имеет конкретную структуру. В нашем случае это id некий идентификатор шутки, type тип шутки, setup основной текст шутки и punchline кульминация панчлайн. Ок, что мы будем делать в андроид приложении?

Мы попробуем получить рандомную шутку от сервера и покажем ее на экране. Сделаем это с помощью кнопки, чтобы можно было читать разные шутки, а не одну за каждый вход в приложение. Хотя вы можете сделать как хотите. Но я предпочитаю кнопку.

Итак, вообще разработка начинается на самом деле с юай слоя – т.е. с написания хмл, дизайна вашего экрана и потом уже вьюмодель и модель. Итак, мы будем получать строку от сервера, т.е. она не лежит в наших строковых ресурсах и иногда при попытке получить ее мы будем сталкиваться с проблемами. Проблема номер 1 : нет интернета. Проблема номер 2: сервер не отвечает или ответил почему-то не то, что мы хотели. Давайте у нас будет 2 типа ошибок : нет интернета и сервер недоступен. Назовем это одним способом, ведь юзеру неважно на самом деле у вас сервер лег (не отвечает на ваши запросы) или в самом серверном коде есть проблемы (да, если вы андроид разработчик, то где-то есть серверный разработчик, который написал код, который по просьбе получить шутку берет из некой базы данных рандомную шутку, формирует json и отправляет вам на андроид).

Дизайн юай – у нас должна быть кнопка на которую будем нажимать и получать шутку, так же нам нужен загрузчик пока получаем ответ от сервера и нам нужен текст шутки и отобразить ошибку. Мы можем упростить нам жизнь тем, что переиспользуем одну текстовку и 1 кнопку. Кнопка у нас всегда будем с текстом : загрузить шутку, а текст будет или самой шуткой или ошибкой. На старте у нас не будет текста вообще, только кнопка. Так же сделаем невозможность нажать на кнопку пока показывается загрузчик и все. Вы можете дополнить дизайн своей логикой – показывать ошибку одним способом и менять текст кнопки если ошибка и так далее. В текущей лекции мы хотим показать как работать с серверной частью и не акцентируем внимание на юай слое и потому делаем его максимально простым.



Давайте напишем активити и хмл и перейдем к вьюмодели (создаем новый проект)

Я решил сделать горизонтальный прогрессбар и даже с отступами не решил заморачиваться. Как видите просто и легко быстро накидать 3 вью в линеар с гравити центр и вуаля. Запустил на эмуляторе и сразу работает. Вот и все. Напишем нашу вьюмодель и активити.

```
private lateinit var viewModel: ViewModel

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    viewModel = (application as JokeApp).viewModel
    val button = findViewById<Button>(R.id.actionButton)
    val progressBar = findViewById<View>(R.id.progressBar)
    val textView = findViewById<TextView>(R.id.textView)
    progressBar.visibility = View.INVISIBLE

    button.setOnClickListener {  it: View!
        button.isEnabled = false
        progressBar.visibility = View.VISIBLE
        viewModel.getJoke()
    }

    viewModel.init(object : TextCallback {
        override fun provideText(text: String) {
            button.isEnabled = true
            progressBar.visibility = View.INVISIBLE
            textView.text = text
        }
    })
}

override fun onDestroy() {
    viewModel.clear()
    super.onDestroy()
}
```

Инициализируем вьюшки, сразу убираем прогресбар, ставим обработчик нажатия на кнопку – при клике не даем повторно кликнуть, показываем прогресбар и получаем шутку. После инициализируем вьюмодель колбек – когда получаем текст то просто даем нажать кнопку

повторно, убираем прогресбар и показываем текст. Все просто – не забываем подчищать перед смертью активити вьюмодель. Теперь посмотрим на саму вьюмодель

```
class ViewModel(private val model: Model<Any, Any>) {

    private var callback: TextCallback? = null

    fun init(callback: TextCallback) {
        this.callback = callback
        model.init(object : ResultCallback<Any, Any> {
            override fun provideSuccess(data: Any) {
                callback.provideText( text: "success")
            }

            override fun provideError(error: Any) {
                callback.provideText( text: "error")
            }
        })
    }

    fun getJoke() {
        model.getJoke()
    }

    fun clear() {
        callback = null
        model.clear()
    }
}

interface TextCallback {

    fun provideText(text: String)
}
```

Храним колбек, когда инициализируем колбек от активити, то инициализируем колбек от модели (далее покажу код). Когда юзер нажал получить шутку то получаем от модели ее. Результат асинхронный, т.е. не получаем сразу шутку потому что надо попросить сервер дать нам строку и потом обработать ее. Метод подчистки убирает колбек и подчищает модель.

Если вы заметили то от модели может прийти 2 вида ответа – успешный и ошибка. Потому колбек от модели имеет 2 метода. Я написал тип Any потому что мы пока не написали обработку сети. Давайте уже посмотрим на модель. Я написал интерфейс чтобы подменить тестовой реализацией.

```
interface Model<S, E> {

    fun getJoke()

    fun init(callback: ResultCallback<S, E>)

    fun clear()
}

interface ResultCallback<S, E> {

    fun provideSuccess(data: S)

    fun provideError(error: E)
}
```

У интерфейса модели простые методы – получить шутку, инициализация через колбек и очистка. Сам интерфейс результата имеет 2 типа S-success, E-error. Мы не знаем какой будет результат от модели – успех и ошибка и я сделал типизацию для подстановки тестовой реализации. Не беспокойтесь, сейчас я покажу тестовую реализацию ниже. Здесь я отдаю строки в колбеке. Сам метод получения шутки просто засыпает на секунду чтобы имитировать процесс загрузки с сети и потом отдает поочередно то ошибку то успех, чтобы мы смогли поочередно проверить и то и другое. Хотя для юай слоя у нас нет разницы между ошибкой и успехом, мы и там и там показываем текст и кнопку. Но по крайней мере мы сможем проверить что все работает как мы заложили. Ладно, теперь соберем вьюмодель в апликейшне.

```
class JokeApp : Application() {

    lateinit var viewModel: ViewModel

    override fun onCreate() {
        super.onCreate()
        viewModel = ViewModel(TestModel())
    }
}
```

```
class TestModel : Model<Any, Any> {

    private var callback: ResultCallback<Any, Any>? = null

    private var count = 1

    override fun getJoke() {
        Thread.sleep( millis: 1000)
        if (count % 2 == 0) {
            callback?.provideSuccess( data: "success")
        } else {
            callback?.provideError( error: "error")
        }
        count++
    }

    override fun init(callback: ResultCallback<Any, Any>) {
        this.callback = callback
    }

    override fun clear() {
        callback = null
    }
}
```

Итак, мы можем запускать код.

Как видите мы написали весь юай и логику и нам не нужен сервер чтобы проверить все это. В реальной разработке такое часто бывает – серверный разработчик не написал код, а вам уже надо начинать делать экран. Вы пишете тестовые модели и проверяете свою юай логику. Позже когда серверный разработчик закончит свой код то вы напишете базовые реализации и подмените один класс другим. И все. Или же вы начинаете писать код одновременно с серверным. И пока он не закончил вы опять же можете протестировать на тестовых моделях.

Ну что ж, запустим проект и проверим? Через раз должно быть успех и ошибка с задержкой.

Но у нас маленькая проблема. У нас никакой асинхронности даже в тестовой модели. Мы написали Thread.sleep и он усыпит текущий тред который мейн. Посмотрите что у вас в эмуляторе – лаг! Ок, нам надо создать поток? Ну да. Или через таймер можно. Хотя и так и так сложно на самом деле точнее долго. Ну что ж, я выберу поток.

И запустим наш код, посмотрим что будет. Если вы забыли написать runOnUiThread то краш

```
override fun getJoke() {
    Thread{
        Thread.sleep( millis: 1000)
        if (count % 2 == 0) {
            callback?.provideSuccess( data: "success")
        } else {
            callback?.provideError( error: "error")
        }
        count++
    }.start()
}
```

Добавим переключение потоков – наш результат пришел из другого потока, а мы хотим отобразить строку на мейн потоке, нам нужно поменять потоки

```
viewModel.init(object : TextCallback {
    override fun provideText(text: String) = runOnUiThread {
        button.isEnabled = true
```

Вот так это выглядит. Теперь все работает исправно.

Ладно. С юай разобрались. Но мы все же хотели бы получать конкретные классы от модели, а не Any. Давайте тогда напишем 2 класса для успеха и ошибки. Это будут сущности бизнес-логики. Ведь что приходит при попытке получить шутку? Или сама шутка (с 2 текстовками) или же ошибка (2 вида – нет интернета и сервер недоступен). Давайте начнем с успеха.

```
class Joke(private val text: String, private val punchline: String) {

}
```

Исходя из инкапсуляции мы должны хранить поля класса внутри, т.е. сделать их приватными. Ок. Но у нас на юай 1 текст. Как нам преобразовать 2 текстовки к 1 притом что они не видны снаружи?

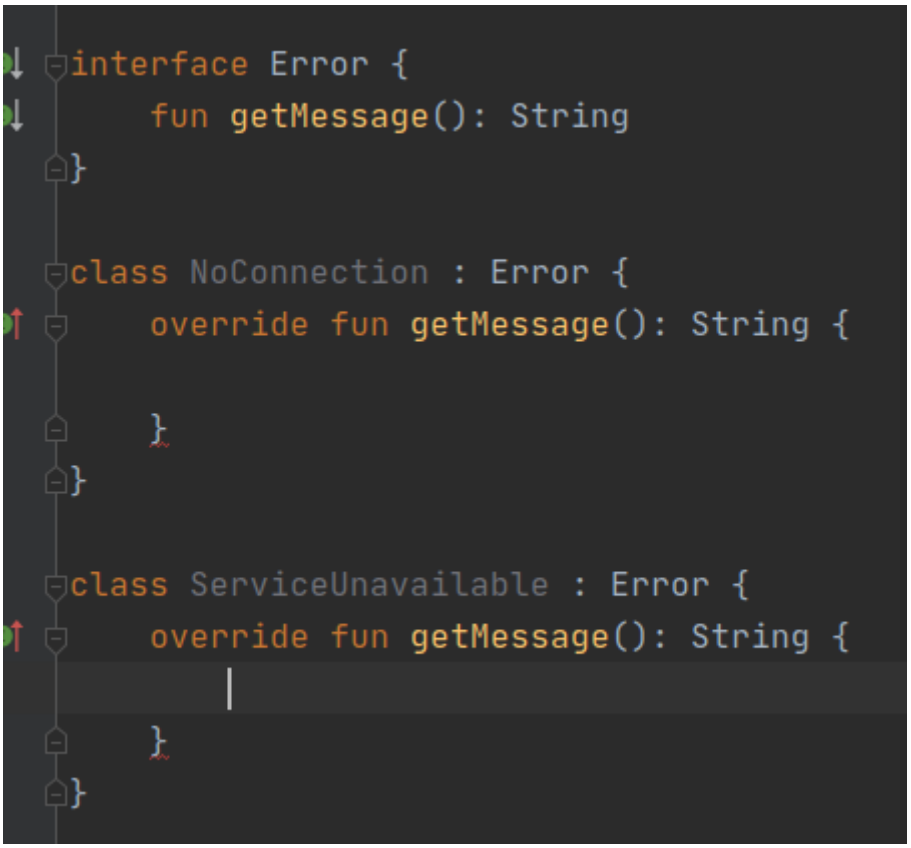
Запомните раз и навсегда, нарушать принцип инкапсуляции нельзя никогда ни при каких обстоятельствах.

Многие грешат тем, что делают паблик поля у класса, легко положить и легко получить. Но тогда мы сталкиваемся с тем, что можно отдельно получить одно из полей и делать с ним что угодно. Но с другой стороны написать метод, который преобразует 2 текста к 1 для юай слоя тоже неверно, значит нам нужно написать преобразователь и работать с ним. Тема текущей лекции работа с сервером, поэтому давайте пока сделаем так.

```
class Joke(private val text: String, private val punchline: String) {

    fun getJokeUi() = "$text\n$punchline"
}
```

Давайте пока что остановимся на этом плохом решении. В отдельной лекции про чистую архитектуру я подробно остановлюсь на маперах.

Теперь давайте напишем 2 класса для 2 видов ошибок. Мы не будем использовать Throwable или Exception из джавы, а сделаем свой интерфейс и 2 класса.



И здесь мы сталкиваемся с новой проблемой. Нам нужны тексты для этих ошибок. Ведь юзеру нужно сказать не просто : что-то пошло не так, а что конкретно – нет интернета или нет ответа от сервера. Все тексты мы храним в strings.xml но как к ним получить доступ? Через контекст. И мы знаем как решить эту задачу. Нужно написать обертку. Давайте!

Нам не нужны 250 методов контекста, а лишь 1. Для этого напишем 1 интерфейс и базовую реализацию с контекстом как аргумент конструктора. Теперь мы можем дописать наши классы ошибок.

```kotlin
interface ResourceManager {

    fun getString(@StringRes stringResId: Int) : String
}

class BaseResourceManager(private val context: Context) : ResourceManager {

    override fun getString(stringResId: Int) = context.getString(stringResId)
}
```

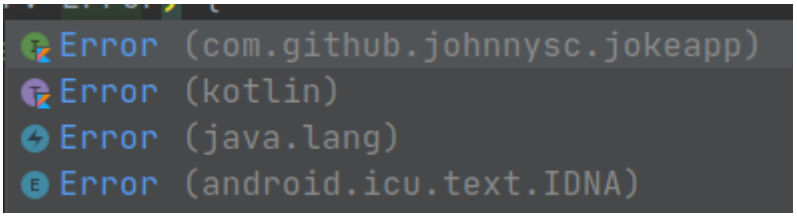
Передаем интерфейс в классы ошибок и вуаля

```kotlin
class NoConnection(private val resourceManager: ResourceManager) : Error {
    override fun getMessage()= resourceManager.getString(R.string.no_connection)
}

class ServiceUnavailable(private val resourceManager: ResourceManager) : Error {
    override fun getMessage() = resourceManager.getString(R.string.service_unavailable)
}
```

В строковых ресурсах положим текстовки

```xml
<string name="get_joke">Get joke</string>
<string name="no_connection">No internet connection</string>
<string name="service_unavailable">Service is unavailable</string>
```

Итак, все правильно сделали, теперь можем переписать наши Any на конкретные классы.



Неприятный момент – мы назвали класс ошибки так же как и в котлине уже есть название и где-то еще. Давайте переименуем в JokeFailure Alt+Enter → refactor.

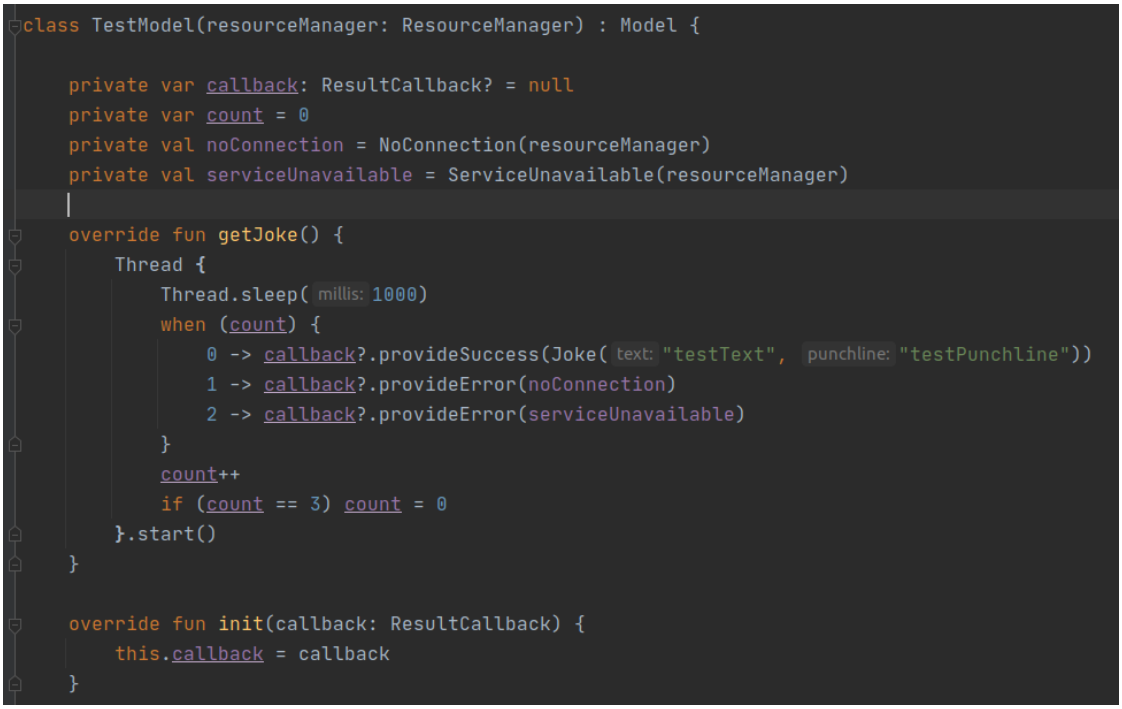
```kotlin
class ViewModel(private val model: Model<Joke, JokeFailure>) {

    private var callback: TextCallback? = null

    fun init(callback: TextCallback) {
        this.callback = callback
        model.init(object : ResultCallback<Joke, JokeFailure> {
            override fun provideSuccess(data: Joke) = callback.provideText(data.getJokeUi())
            override fun provideError(error: JokeFailure) = callback.provideText(error.getMessage())
        })
```

Теперь все более менее понятно – модель отдает или шутку или ошибку.

Если успех то берем юай вид шутки, если ошибка, то сообщение ошибки. Мы допустили 1 ошибку в классе Joke – паблик метод без оверайд. И я обьяснил это тем, что позже напишем мапер когда пройдем чистую архитектуру. Вы можете для простоты сейчас написать нечто типа интерфейса юаймапер и там метод toUiText. А мы пойдем дальше.

Давайте перепишем наш тестовый модель класс чтобы он отдавал сначала интернета нет, потом проблема с сервисом и потом уже успех. Проверим все кейсы.

В нашем проекте лишь 1 функциональность – шутки. Если бы их было больше, то я бы оставил интерфейс с дженериком. А пока просто убрал их и поставил конкретные классы.



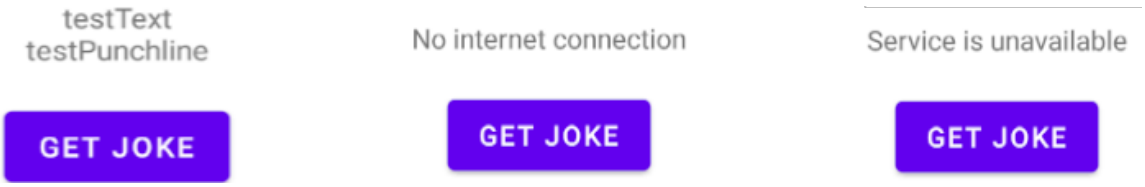
Я буду показывать сначала успех, потом нет интернета и потом нет ответа от сервера. Исправим уже класс инициализацию вьюмодели и запустим код!

```
override fun onCreate() {
    super.onCreate()
    viewModel = ViewModel(TestModel(BaseResourceManager( context: this)))
}
```

Мы не зря написали интерейс и базовую реализацию. Вы можете написать юнит-тест на тестовую модель и проверить все сами. Оставлю это вам как задание самостоятельное.

```
class ViewModel(private val model: Model) {

    private var callback: TextCallback? = null

    fun init(callback: TextCallback) {
        this.callback = callback
        model.init(object : ResultCallback {
```

Да, не забудьте пофиксить вм, мы же убрали дженерики.



Вуаля. Все работает!

2. Добавляем слой данных

Ладно, у нас все работает на уровне юай слоя. Логика юай безупречно отрабатывает.

Теперь пора уже написать базовую модель для работы с интернет. И для начала давайте добавим пермишен интернет в манифест.

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

<application
    android:allowBackup="true"
    android:name=".JokeApp"
```

Теперь нам нужно написать код, который бы получал данные из сети. Подождите. Мы же писали такое во 2-ой лекции, разве нет?

Я специально покажу вам что было во второй лекции. Там мы получали картинку

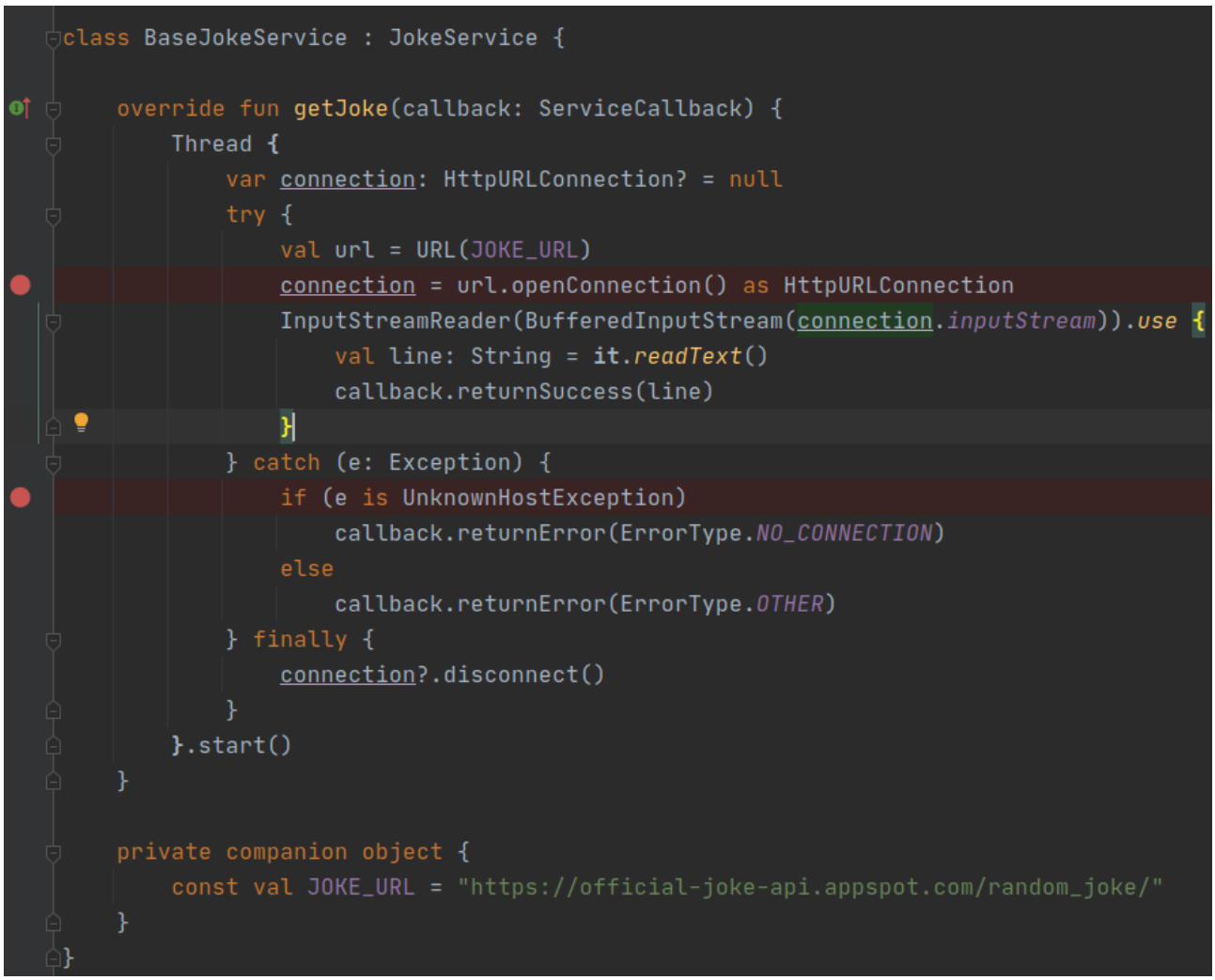
```
try {
    val connection = URL(url).openConnection()
    connection.doInput = true
    connection.connect()
    connection.inputStream.use { it: InputStream!
        callback.success(BitmapFactory.decodeStream(it))
    }
} catch (e: Exception) {
    callback.failed()
}
```

Таким способом. Но у нас все проще – нам не нужен стрим чтобы парсить в картинку.

Мы можем сразу получить строку. Давайте попробуем. Напишем отдельный класс для этого. Он просто будет получать строку по заданному адресу.

```
interface JokeService {

    fun getJoke(callback: ServiceCallback)
}

interface ServiceCallback {

    fun returnSuccess(data: String)

    fun returnError(type: ErrorType)
}

enum class ErrorType {
    NO_CONNECTION,
    OTHER
}
```

Мы вернем или строку в каком виде она пришла от сервера или же вернем ошибку, здесь у нас будет просто: или нет интернета или другой вид. Неважно какой. Далее эта ошибка будет мапиться к юай с помощью JokeFailure. Итак, пишем реализацию и давайте в ней же создадим поток и стартанем.



```
class BaseJokeService : JokeService {

    override fun getJoke(callback: ServiceCallback) {
        Thread {
            var connection: HttpURLConnection? = null
            try {
                val url = URL(JOKE_URL)
                connection = url.openConnection() as HttpURLConnection
                InputStreamReader(BufferedInputStream(connection.inputStream)).use {
                    val line: String = it.readText()
                    callback.returnSuccess(line)
                }
            } catch (e: Exception) {
                if (e is UnknownHostException)
                    callback.returnError(ErrorType.NO_CONNECTION)
                else
                    callback.returnError(ErrorType.OTHER)
            } finally {
                connection?.disconnect()
            }
        }.start()
    }

    private companion object {
        const val JOKE_URL = "https://official-joke-api.appspot.com/random_joke/"
    }
}
```

Примерно так если без либ выглядит получение строки из сети по адресу. Кто-то спросит, почему мы не смогли испольовать use для HttpUrlConnection – а потому что оно не Closable.

Здесь все просто на самом деле. Создаем поток, в нем открываем соединение и читаем через стрим данные. У нас максимум 1 линия, потому я не написал BufferedReader. И мы можем проверить что оно работает. Напишем базовую реализацию модели.

Мы передаем сервис в конструктор нашей модели, также передаем ресурсменеджер для мапинга ошибки. И здесь все просто на самом деле. Я написал ошибки через lazy потому что не факт что они нам могут понадобиться. Поэтому инициализируем по требованию.

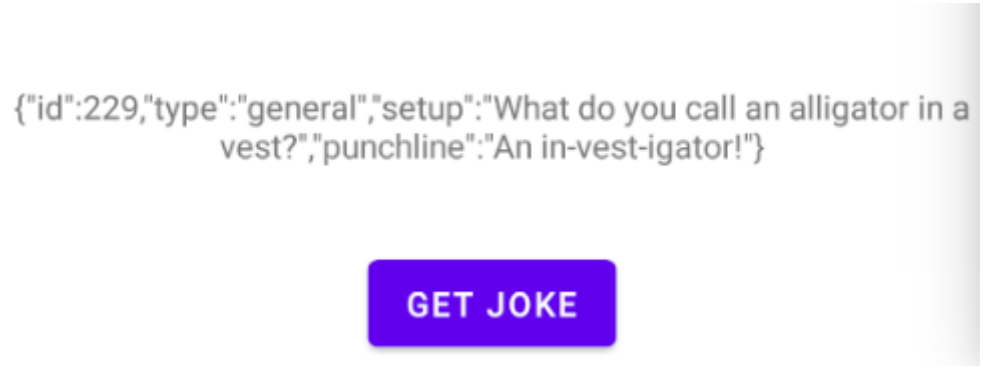
Самый плюс интерфейса и принципа Лисков в том, что я могу написать тестовый сервис и отдать сразу нужный результат и не дергать реальный сервер. Но сейчас я не буду этого делать. Вы можете заняться этим если хочется. Я заменю в вьюмодели нашу модель и проверю так что все работает.

```
viewModel = ViewModel(BaseModel(BaseJokeService(), BaseResourceManager( context: this)))
```

Давайте запускать код! Если вы заметили, то я получаю сырой ответ от сервера и просто показываю его на экране через Joke(data, “”). Запустим на эмуляторе уже!

```
class BaseModel(
    private val service: JokeService,
    private val resourceManager: ResourceManager
) : Model {
    private var callback: ResultCallback? = null
    private val noConnection by lazy { NoConnection(resourceManager) }
    private val serviceUnavailable by lazy { ServiceUnavailable(resourceManager) }

    override fun getJoke() {
        service.getJoke(object : ServiceCallback {
            override fun returnSuccess(data: String) {
                callback?.provideSuccess(Joke(data, punchline: ""))
            }

            override fun returnError(type: ErrorType) {
                when (type) {
                    ErrorType.NO_CONNECTION -> callback?.provideError(noConnection)
                    ErrorType.OTHER -> callback?.provideError(serviceUnavailable)
                }
            }
        })
    }

    override fun init(callback: ResultCallback) {
        this.callback = callback
    }

    override fun clear() {
        callback = null
    }
}
```

Итак, как видите все работает!



Но это не то, что мы хотели бы видеть на экране смартфона. Нам нужно отделить саму шутку от всего остального. Как же это сделать? Давайте поговорим о сериализации.

Сервер может возвращать вам что угодно, но в любом случае это будет текст. String. Теперь, вам нужно каким-то образом данные на сервере преобразовать (сериализовать) в строку

чтобы передать на андроид. А если серверный разработчик сериализовал в строку, то нам нужно сделать обратное – из строки получить обьект (класс со структурой). Как это сделать?

Для этого уже есть готовые механизмы. Давайте добавим либу gson в градл файл.

```
implementation 'androidx.constraintlayout:constraintlayout:2.0.4'
implementation 'com.google.code.gson:gson:2.8.7'
testImplementation 'junit:junit:4.13.2'
```

Если помните мы во второй лекции добавляли пикассо. Откройте build.gradle и найдите место где добавлены либы. Теперь нам нужен класс для десериализации – чтобы из строки преобразовать в конкретный обьект.

```
data class JokeDTO(
    @SerializedName( value: "id")
    private val id: Int,
    @SerializedName( value: "type")
    private val type: String,
    @SerializedName( value: "setup")
    private val text: String,
    @SerializedName( value: "punchline")
    private val punchline:String
)
```

Аннотация SerializedName нужна чтобы дать понять гсону какой элемент к чему нужно мапить. Что к чему грубо говоря. Ведь мы завязываемся не на порядок частей внутри строки, а не ключи. Да, Json это ключ значение но в одной строке.

{"id":229,"type":"general","setup":"What do you call an alligator in a vest?","punchline":"An in-vest-igator!"}

Здесь id это ключ, а 229 значение. Как видите у 229 нет кавычек с 2 сторон, значит это не String, а Int. Все остальное строки. Теперь мы можем поменять наш сервис таким образом, чтобы он отдавал не строку, а обьект. Для этого нужно десериализовать ее с помощью Gson().

И да, вы заметили наверно что я назвал класс JokeDTO – Data Transfer Object. Такой суффикс нужен чтобы отличать те классы, которые пришли от сервера от тех, которые используем в других слоях – в бизнес-логике или юай. И не забудьте поменять метод у сервисколбека.

```
interface ServiceCallback {

    fun returnSuccess(data: JokeDTO)
```

```
class BaseJokeService(private val gson: Gson) : JokeService {

    override fun getJoke(callback: ServiceCallback) {
        Thread {
            var connection: HttpURLConnection? = null
            try {
                val url = URL(JOKE_URL)
                connection = url.openConnection() as HttpURLConnection
                InputStreamReader(BufferedInputStream(connection.inputStream)).use {
                    val line: String = it.readText()
                    val dto = gson.fromJson(line, JokeDTO::class.java)
                    callback.returnSuccess(dto)
                }
```

Теперь наша строка будет превращаться в обьект. Я получу обьект гсон в конструктор класса просто чтобы он был у меня в единственном числе и не порождался каждый раз.

Теперь нам нужно исправить модель. Ведь там теперь приходит не строка, а обьект.

Но мы написали все поля класса приватными. Как же нам преобразовать DTO к Joke?

Давайте просто напишем 1 метод в классе DTO.

```
    @SerializedName( value: "punchline")
    private val punchline: String
) {
    fun toJoke() = Joke(text, punchline)
}
```

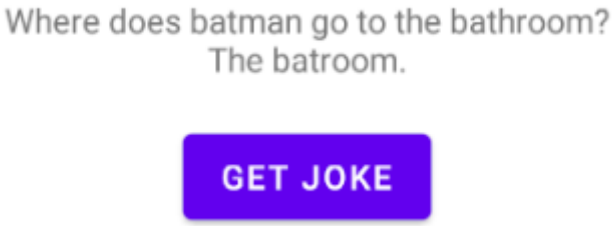
Это не самое лучшее решение, но для начала нормально. Теперь можем исправить модель.

```
override fun returnSuccess(data: JokeDTO) {
    callback?.provideSuccess(data.toJoke())
}
```

Запустим код? Нам нужно пофиксить зависимости в апликейшн классе и можем смотреть.

```
super.onCreate()
viewModel = ViewModel(BaseModel(BaseJokeService(Gson()), BaseResourceManager( context: this)))
```

Ну что ж, проверим? Да, все работает!



Но подождите, кто-то скажет : для десериализации строки в обьект мы подключили либу, а для работы с сетью нет готового решения? Посмотрите на наш сервис BaseJokeService. Он ужасен! Создаем поток, открываем соединение и так далее. Если для картинок мы переписали такой же код на пикассо и 1 линией получили изображение, то не может так же сделать сейчас?

Конечно же можем и сделаем. Либа называется Retrofit.

Подключим ее в build.gradle

```
implementation 'com.squareup.retrofit2:retrofit:2.6.0'
implementation 'com.squareup.retrofit2:converter-gson:2.6.0'
```

И сразу же подключаем фабрику конвертеров чтобы легко десериализовать данные.

Теперь можем написать наш сервис в 1 линию!

```
interface JokeService {

    @GET( value: "https://official-joke-api.appspot.com/random_joke/")
    fun getJoke() : Call<JokeDTO>
}
```

Метод GET означает что мы получаем данные и ничего не отправляем с нашей стороны чтобы изменить значения на сервере (почиайте про REST). В нашем случае весь урл я передаю в методе. Заметьте, нам не нужен теперь колбек внутрь, потому что метод вернет колбеком данные. Удалим базовую реализацию сервиса и пофиксим теперь модель.

```
override fun getJoke() {
    service.getJoke().enqueue(object : retrofit2.Callback<JokeDTO> {
        override fun onResponse(call: Call<JokeDTO>, response: Response<JokeDTO>) {
            if (response.isSuccessful) {
                callback?.provideSuccess(response.body()!!.toJoke())
            } else {
                callback?.provideError(serviceUnavailable)
            }
        }

        override fun onFailure(call: Call<JokeDTO>, t: Throwable) {
            if (t is UnknownHostException)
                callback?.provideError(noConnection)
            else
                callback?.provideError(serviceUnavailable)
        }
    })
}
```

Согласен, код стал немного ужасен, но вы всегда можете написать обертку, или подождать пока мы перейдем к корутинам в других лекциях. Пусть пока так остается.

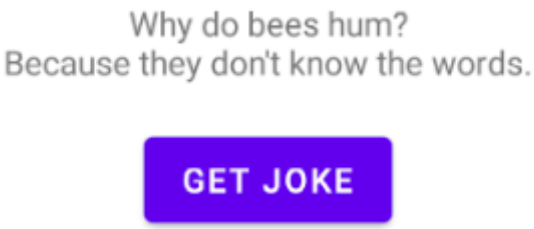
Теперь надо написать создание сервиса и дать его внутрь модели в апликейшн классе.

```
override fun onCreate() {
    super.onCreate()
    val retrofit = Retrofit.Builder()
        .baseUrl( baseUrl: "https://www.google.com")
        .addConverterFactory(GsonConverterFactory.create())
        .build()
    viewModel = ViewModel(
        BaseModel(
            retrofit.create(JokeService::class.java),
            BaseResourceManager( context: this)
        )
    )
}
```

Да, здесь стало много кода. Но это нестрашно, в последующих лекциях мы решим эту проблему. Здесь мы создаем ретрофит инстанс и ему нужен базовый урл. В нормальной вселенной у вас есть некий базовый урл типа myWebSite/api/ который вы переиспользуете для всех методов и просто добавляете суффиксы в конец. А в нашем случае мы используем весь урл сразу в методе и потому напишем в базовом что угодно. Также указываем какая фабрика конвертера, у нас Gson (кстати, можете удалить импорт из градла). И в модель просто передаем сервис создавая его через ретрофит и указывая класс.

В чем плюс ретрофита – он упрощает нам жизнь тем, что нам не нужно писать класс, в нем поток запускать, получать данные и десериализовать. Все это делается намного проще.

Запустим же проект и проверим!



Все работает по-прежнему. Более подробно мы обсудим ретрофит в других лекциях. А если вам хочется погуглите как логировать вызовы, ставить тайм-аут содинения и многое другое что предлагает эта либа.

Чтобы закрепить знания найдите open api и попробуйте написать самостоятельно клиент-серверное приложение которое просто получит данные и отобразит их на экране.

Почти на любой работе вы будете писать клиент-серверное приложение, именно поэтому мы еще не раз будем писать взаимодействие с сервером. Перепишем на корутины и так далее.