

# Сохраняем много данных

## Введение в базы данных

### Содержание

1. Сохраняем шутку и показываем статус
2. Добавляем поSQL

### 1. Сохраняем шутку и показываем статус

В предыдущей лекции мы получали шутку по АПИ и показывали ее на экране. Теперь же давайте рассмотрим такую задачу. А что если мне настолько понравилась шутка, что я хочу ее сохранить и перечитывать потом без интернета? Ок, давайте попробуем это реализовать. Но юай я не хочу пока что сильно менять, меня больше интересует сейчас остальной код. Пусть у нас будет просто чекбокс – брать сохраненные шутки и чтобы добавить шутку в сохраненные нам нужна 1 кнопка, пусть это будет сердечко для примера.

Итак, добавим 2 выю в наш юай

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:orientation="vertical">

    <CheckBox
        android:id="@+id/checkBox"
        android:padding="@dimen/padding"
        android:text="@string/show_favorite_joke"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <TextView
            android:padding="@dimen/padding"
            android:id="@+id/textView"
            android:layout_width="0dp"
            android:layout_weight="1"
            android:layout_height="wrap_content"
            android:gravity="center"
            tools:text="Joke or error message" />

        <ImageView
            android:layout_gravity="center_vertical"
            android:id="@+id/iconView"
            tools:src="@android:drawable/ic_input_add"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>

    </LinearLayout>

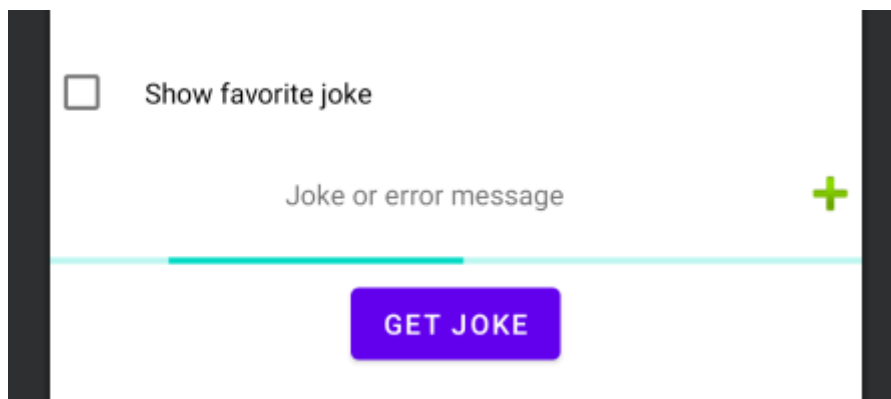
</LinearLayout>
```

```
</LinearLayout>
```

```
<ProgressBar  
    android:indeterminate="true"  
    android:id="@+id/progressBar"  
    style="?android:attr/progressBarStyleHorizontal"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content" />
```

```
<Button  
    android:id="@+id/actionButton"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/get_joke" />
```

```
</LinearLayout>
```



Пусть будет чекбокс над шуткой и иконка рядом с самой шуткой чтобы добавить. Возьмем из интернета 2 иконки сердечек – одно пустое другое полное. <https://fonts.google.com/icons?selected=Material+Icons>

```
baseline_favorite_24.xml (v24)  
baseline_favorite_border_24.xml (v24)
```

Если вы не помните как добавлять изображения в проект посмотрите лекцию 2. Итак, идем дальше. Как пишут плохой код – делают изменяемо поле у класса Joke – var favorite:Boolean и меняют его и пишут if else. Как мы сделаем – напишем 2 класса и исправим уже наконец тот факт, что наш класс шутки не наследует ничего

```
class BaseJoke(text: String, punchline: String) : Joke(text, punchline) {  
    override fun getIconResId() = R.drawable.baseline_favorite_border_24  
}  
  
class FavoriteJoke(text: String, punchline: String) : Joke(text, punchline) {  
    override fun getIconResId() = R.drawable.baseline_favorite_24  
}  
  
class FailedJoke(text: String) : Joke(text, punchline: "") {  
    override fun getIconResId() = 0  
}  
  
abstract class Joke(private val text: String, private val punchline: String) {  
  
    fun getJokeUi() = "$text\n$punchline"  
  
    @DrawableRes  
    abstract fun getIconResId() : Int  
}
```

У нас 3 состояния – шутка в сохраненных, шутка не в сохраненных и ошибка. Значит нам нужно 3 класса наследующихся от абстрактного. У него 2 метода – как показать текст и какую иконку. Как видите для сохраненной шутки будет полное сердце, а для обычного пустое. А для ошибки передаем 0 чтобы иконки вообще не было. Я решил не удалять классы для ошибок, пусть они останутся пока (JokeFailure). Если нужно будет поменять далее. Ладно, давайте тогда напишем код в активности.

```
val checkBox = findViewById<CheckBox>(R.id.checkBox)
checkBox.setOnCheckedChangeListener { _, isChecked ->
    viewModel.chooseFavorites(isChecked)
}
```

Расскажу логику – юзер жмет на чекбокс и выбирает источник данных – сохраненные или нет. По изменению чекбокса ничего не будет делать на юай. Когда юзер нажмет кнопку get joke мы исходя из выбора будем отдавать из нужного датасorsa. Я не хочу передавать в методе кнопки второй аргумент и переделывать его. Лучше написать независимый метод. Если помните мы передавали колбеком только текст TextCallback давайте добавим еще один метод для иконки и переименуем его.

```
interface DataCallback {
    fun provideText(text: String)
    fun provideIconRes(@DrawableRes id: Int)
}
```

В активности находим иконку и сетим данные

```
viewModel.init(object : DataCallback {
    override fun provideText(text: String) = runOnUiThread {
        button.isEnabled = true
        progressBar.visibility = View.INVISIBLE
        textView.text = text
    }
    override fun provideIconRes(id: Int) = iconView.setImageResource(id)
})
```

Теперь пофиксим нашу вьюмодель. В принципе наша активность не должна знать о такой вещи как Joke. Поэтому отдельными методами отдаем данные в вью.

Как видите нам нужно переписать наш ResultCallback потому что создавать FailedJoke и передавать туда сообщение и потом писать такой же код как и для успеха это нарушение DRY. Давайте перепишем колбек с 1 метом чтобы он работал.

```
interface ResultCallback {
    fun provideJoke(joke: Joke)
}
```

И теперь посмотрите как изящно будет выглядеть наш код в вьюмодели.

```

fun init(callback: DataCallback) {
    dataCallback = callback
    model.init(object : ResultCallback {
        override fun provideSuccess(data: Joke) {
            dataCallback?.provideText(data.getJokeUi())
            dataCallback?.provideIconRes(data.getIconResId())
        }
        override fun provideError(error: JokeFailure) {
            val joke = FailedJoke(error.getMessage())
            dataCallback?.provideText(joke.getJokeUi())
            dataCallback?.provideIconRes(joke.getIconResId())
        }
    })
}

```

Стало лучше, но не намного, да?

```

model.init(object : ResultCallback {
    override fun provideJoke(joke: Joke) {
        dataCallback?.run { this: DataCallback
            provideText(joke.getJokeUi())
            provideIconRes(joke.getIconResId())
        }
    }
})

```

Просто проблема здесь в том, что человек может забыть вызвать второй метод. Как это решить? Давайте немного перепишем класс шутки чтобы он работал с колбеком.

```

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

    protected fun getJokeUi() = "$text\n$punchline"

    @DrawableRes
    protected abstract fun getIconResId(): Int

    fun map(callback: DataCallback) = callback.run { this: DataCallback
        provideText(getJokeUi())
        provideIconRes(getIconResId())
    }
}

```

Теперь никто снаружи класса шутки не знает о структуре класса. Никто не может вызвать методы отдельно и получить данные. Только колбек может работать с классом. Заметьте что даже в абстрактном классе у нас методы не видны снаружи класса ибо можно написать protected. И теперь наша выюмодель стала изящнее.

```
fun init(callback: DataCallback) {
    dataCallback = callback
    model.init(object : JokeCallback {
        override fun provide(joke: Joke) {
            dataCallback?.let { it: DataCallback
                joke.map(it)
            }
        }
    })
}
```

Мы бы могли написать метод тир принимающий нулабл колбек и тогда бы было еще красивее. Но я против передавать в аргументы методам нулабл. Поэтому так.

Ладно, давайте уже попробуем проверить наш код. Если помните у нас была тестовая модель. Давайте ее отредактируем таким образом, чтобы она отдавала обычную шутку, после сохраненную и после ошибку.

```
when (count) {
    0 -> callback?.provide(BaseJoke( text: "testText",  punchline: "testPunchline"))
    1 -> callback?.provide(FavoriteJoke( text: "favoriteJokeText",  punchline: "favorite joke punchline"))
    2 -> callback?.provide(FailedJoke(serviceUnavailable.getMessage()))
}
```

Не забудем поменять инстанс в апликейшн классе и запустим код

```
viewModel = ViewModel(TestModel(BaseResourceManager( context: this)))
```

И мы забыли поменять код в JokeDTO, кстати плохое название, поменяем чуть позже когда напишем логику

```
fun toJoke() = BaseJoke(text, punchline)
```

Да, я забыл написать runOnUiThread в активити

```
override fun provideIconRes(id: Int) = runOnUiThread {
    iconView.setImageResource(id)
}
```

Запускаем уже проект и проверяем



Обычная шутка выглядит правильно, нажимаем еще раз и должна быть сохраненная.

Да, все так, сердечко заполнено как видите, и нажмем еще раз оно должно исчезнуть

☐ Show favorite joke

favoriteJokeText  
favorite joke punchline



GET JOKE

И все правильно отображается

Service is unavailable

GET JOKE

Но мы забыли самое главное! Чтобы иконка сердечка работала! Да, надо поменять `ImageView` на `ImageButton` и поставить кликиснер. Опять же, как делают плохой код – с юай отправляют данные о том, что нужно сделать – добавить в сохраненные или убрать оттуда. Мы поступим иначе. Когда от сервера придет шутка мы проверим есть ли она в нашей базе данных, и тогда отдадим на юай `FavoriteJoke` и сохраним это в локальный кеш. Когда юзер нажмет на сердечко, то мы просто у этого сохраненного объекта вызовим метод изменения статуса. Давайте для начала напишем некий `CacheDataSource`. Я сделаю 1 метод – аргументом пойдет айди и вернет нам шутку нужного вида `BaseJoke` или `FavoriteJoke`. Но так как у меня айди в классе `JokeDTO` имеет приватный модификатор доступа я напишу метод в классе шутки для вызова метода интерфейса. Смотрим

```
interface CacheDataSource {  
  
    fun addOrRemove(id: Int, joke: JokeServerModel): Joke  
  
}
```

И пишем метод в самом классе `JokeDTO` и давайте переименуем в `JokeServerModel`

```
fun change(cacheDataSource: CacheDataSource) = cacheDataSource.addOrRemove(id, joke: this)
```

Теперь, нам нужно выделить еще и `CloudDataSource` чтобы получать ответ от сервера отдельно, а не в модели. И если помните мы в одной лекции написали `Repository` и в нем 2 датасорса. У `CloudDataSource` будет 1 метод получения шутки и класс будет принимать сервис.

```
interface CloudDataSource {  
  
    fun getJoke(callback: JokeCallback)  
  
}
```

Мы передадим аргументом колбек чтобы вернулся результат. Напишем базовую реализацию но чтобы она не мапила данные к FailedJoke и так далее, а просто отдавала или сам JokeServerModel или же ошибку. Для этого напомним другой колбек. Назовем его JokeCloudCallback.

```
interface JokeCloudCallback {  
    fun provide(joke: JokeServerModel)  
    fun fail(error: ErrorType)  
}  
  
enum class ErrorType {  
    NO_CONNECTION,  
    SERVICE_UNAVAILABLE  
}
```

```
interface CloudDataSource {  
    fun getJoke(callback: JokeCloudCallback)  
}
```

Теперь вернем просто тип ошибки если нет ответа от сервера. Далее пусть модель мапит к чему нужно.

```
class BaseCloudDataSource(private val service: JokeService) : CloudDataSource {  
    override fun getJoke(callback: JokeCloudCallback) {  
        service.getJoke().enqueue(object : retrofit2.Callback<JokeServerModel> {  
            override fun onResponse(  
                call: Call<JokeServerModel>,  
                response: Response<JokeServerModel>  
            ) {  
                if (response.isSuccessful) {  
                    callback.provide(response.body()!!)  
                } else {  
                    callback.fail(ErrorType.SERVICE_UNAVAILABLE)  
                }  
            }  
        })  
    }  
  
    override fun onFailure(call: Call<JokeServerModel>, t: Throwable) {  
        if (t is UnknownHostException)  
            callback.fail(ErrorType.NO_CONNECTION)  
        else  
            callback.fail(ErrorType.SERVICE_UNAVAILABLE)  
    }  
}
```

Мы не будем зачищать колбек потому что это будет модель и она умрет в тот же момент что и датасорс. Теперь напишем тестовый кешдатасорс где просто будет мапа. Но нам нужно спаять серверную модель к базовой шутке и к сохраненной, значит нам нужно 2 метода. Если в мапе есть айди, то просто убираем оттуда и отдаем базовую шутку, если же в мапе нет айди, значит кладем туда и отдаем сохраненный. Все просто.

```
class TestCacheDataSource : CacheDataSource {  
  
    private val map = HashMap<Int, JokeServerModel>()  
  
    override fun addOrRemove(id: Int, jokeServerModel: JokeServerModel): Joke {  
        return if (map.containsKey(id)) {  
            val joke = map[id]!!.toBaseJoke()  
            map.remove(id)  
            joke  
        } else {  
            map[id] = jokeServerModel  
            jokeServerModel.toFavoriteJoke()  
        }  
    }  
}
```

Ну и наконец-то можем написать нашу модель которая работает с датасорсами

```
class BaseModel(  
    private val cacheDataSource: CacheDataSource,  
    private val cloudDataSource: CloudDataSource,  
    private val resourceManager: ResourceManager  
) : Model {  
    private val noConnection by lazy { NoConnection(resourceManager) }  
    private val serviceUnavailable by lazy { ServiceUnavailable(resourceManager) }  
  
    private var jokeCallback: JokeCallback? = null  
  
    private var cachedJokeServerModel: JokeServerModel? = null  
  
    override fun getJoke() {  
        cloudDataSource.getJoke(object : JokeCloudCallback {  
            override fun provide(joke: JokeServerModel) {  
                cachedJokeServerModel = joke  
                jokeCallback?.provide(joke.toBaseJoke())  
            }  
  
            override fun fail(error: ErrorType) {  
                cachedJokeServerModel = null  
                val failure = if (error == ErrorType.NO_CONNECTION) noConnection else serviceUnavailable  
                jokeCallback?.provide(FailedJoke(failure.getMessage()))  
            }  
        })  
    }  
  
    override fun init(callback: JokeCallback) {  
        this.jokeCallback = callback  
    }  
  
    override fun clear() {  
        jokeCallback = null  
    }  
}
```

Итак, когда нам нужна шутка, мы берем из CloudDataSource и кешируем если пришел нормальный ответ от сервера. Если пришла ошибка, то мы зачищаем закешированное. Вроде все нормально выглядит, идем дальше



Теперь нам нужно поменять юай чтобы мы могли добавлять или удалять нашу шутку в кешдатасорс. Поменяем хмл

```
<ImageButton
    android:background="?selectableItemBackground"
    android:layout_gravity="center_vertical"
    android:id="@+id/changeButton"
    tools:src="@android:drawable/ic_input_add"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

Теперь в активити добавим кликлиснер

```
val changeButton = findViewById<ImageView>(R.id.changeButton)
changeButton.setOnClickListener { it: View!
    viewModel.changeJokeStatus()
}
```

И напишем метод у вьюмодели

```
private val jokeCallback = object : JokeCallback {
    override fun provide(joke: Joke) {
        dataCallback?.let { it: DataCallback
            joke.map(it)
        }
    }
}

fun init(callback: DataCallback) {
    dataCallback = callback
    model.init(jokeCallback)
}

fun changeJokeStatus() {
    model.changeJokeStatus(jokeCallback)
}
```

У нас получилось дублирование ибо надо в метод дать колбек и он был в инит методе, поэтому вынесли полем. И напишем метод в модели

```

override fun changeJokeStatus(jokeCallback: JokeCallback) {
    cachedJokeServerModel?.change(cacheDataSource)?.let { it: Joke
        jokeCallback.provide(it)
    }
}

```

Ну что ж, время проверить все что мы написали до сих пор.

У нас из тестового клаудДатасорса должны приходить например данные и мы нажимая на сердечко должны менять их статус.

```

class TestCloudDataSource : CloudDataSource {
    override fun getJoke(callback: JokeCloudCallback) {
        callback.provide(JokeServerModel( id: 0, type: "testType", text: "TestText", punchline: "TestPunchline"))
    }
}

```

Собираем нашу модель в аппликейшн классе

```

viewModel = ViewModel(
    BaseModel(TestCacheDataSource(), TestCloudDataSource(), BaseResourceManager( context: this))
)

```

И запустим код

TestText  
TestPunchline



Получаем тестовый текст, нажмем теперь на сердечко

TestText  
TestPunchline



Все работает. Теперь мы должны нажать еще раз и оно должно вернуться в тот статус в котором было. Можете подебажить и проверить что все работает правильно.

Давайте отследим вызовы

Activity → viewModel#changeJokeStatus → cacheDataSource#addOrRemove вернуло шутку в нужном статусе и после уже вернуло колбеком в модель и на активити. Все классно. И мы теперь можем дойти до нашего чекбокса. Что оно должно делать: просто менять активный датасорс. Напишем же код в мод в vm

```

fun chooseFavorites(favorites: Boolean) {
    model.chooseDataSource(favorites)
}

```

Добавим метод в модель

```
fun chooseDataSource(cached: Boolean)
```

И напишем if else внутри метода getJoke, но перед этим нам нужен JokeCacheCallback

```
interface JokeCachedCallback {  
  
    fun provide(jokeServerModel: JokeServerModel)  
  
    fun fail()  
}
```

А еще нам нужен новый наследник ошибки если нет сохраненных шуток

```
<string name="no_cached_jokes">"No favorite jokes! Add one by heart icon"
```

```
class NoCachedJokes(private val resourceManager: ResourceManager) : JokeFailure {  
    override fun getMessage() = resourceManager.getString(R.string.no_cached_jokes)  
}
```

Теперь можем написать наш метод в кешдатасорсе

```
interface CacheDataSource {  
    fun getJoke(jokeCachedCallback: JokeCachedCallback)
```

Напишем имплементацию метода в тестовом классе

```
override fun getJoke(jokeCachedCallback: JokeCachedCallback) {  
    if (map.isEmpty())  
        jokeCachedCallback.fail()  
    else  
        jokeCachedCallback.provide(map[0]!!)  
}
```

И наконец-то мы можем дописать метод getJoke в модели

```

private var getJokeFromCache = false

override fun chooseDataSource(cached: Boolean) {
    getJokeFromCache = cached
}

override fun getJoke() {
    if (getJokeFromCache) {
        cacheDataSource.getJoke(object : JokeCachedCallback {
            override fun provide(jokeServerModel: JokeServerModel) {
                jokeCallback?.provide(jokeServerModel.toFavoriteJoke())
            }
            override fun fail() {
                jokeCallback?.provide(FailedJoke(noCachedJokes.getMessage()))
            }
        })
    } else {
        ccloudDataSource.getJoke(object : JokeCloudCallback {

```

Создаем 1 булеан переменную, меняем его легко и просто. Когда юзер нажмет на кнопку то он исходя из выбора будет брать или из кеша или из сети. Давайте потестируем : сначала не будем класть в кеш ничего и увидим ошибку, после поменяем обратно чекбокс и получим из кеша. Вернем чекбокс и увидим сохраненное



Show favorite joke

No favorite jokes! Add one by heart icon

Да, нет сохраненных шуток, Теперь убираем чекбокс и жмем кнопку



Show favorite joke

TestText  
TestPunchline



Жмем сердечко и жмем на чекбокс и потом еще раз на кнопку get joke



Show favorite joke

TestText  
TestPunchline

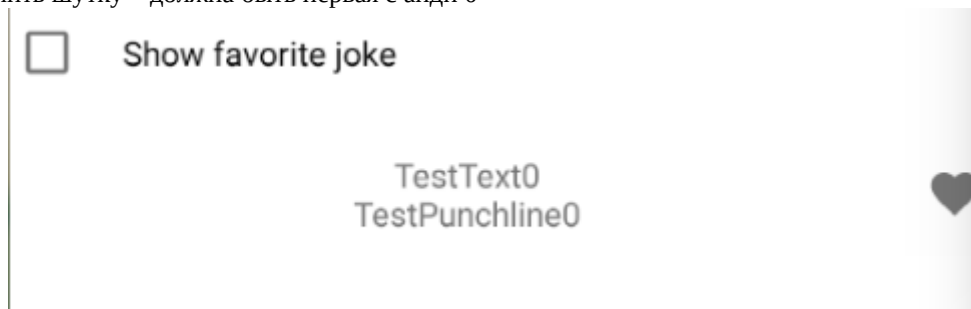


GET JOKE

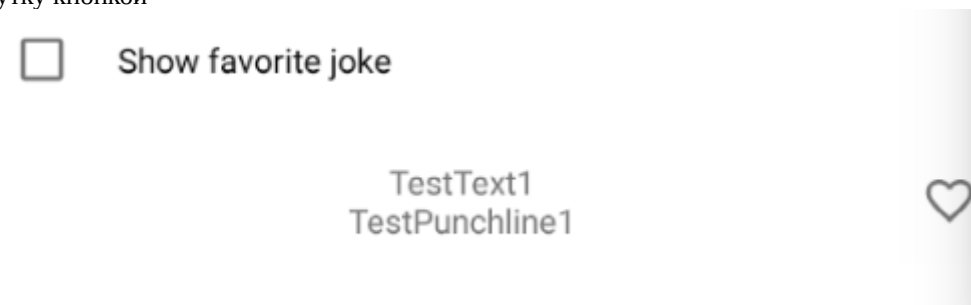
Чтобы проверить по-настоящему нам нужно чтобы из сети приходило хотя бы 2 разные шутки. Допишем тест клауд датасорс

```
class TestCloudDataSource : CloudDataSource {  
    private var count = 0  
    override fun getJoke(callback: JokeCloudCallback) {  
        val joke = JokeServerModel(count, type: "testType", text: "TestText$count", punchline: "TestPunchline$count")  
        callback.provide(joke)  
        count++  
    }  
}
```

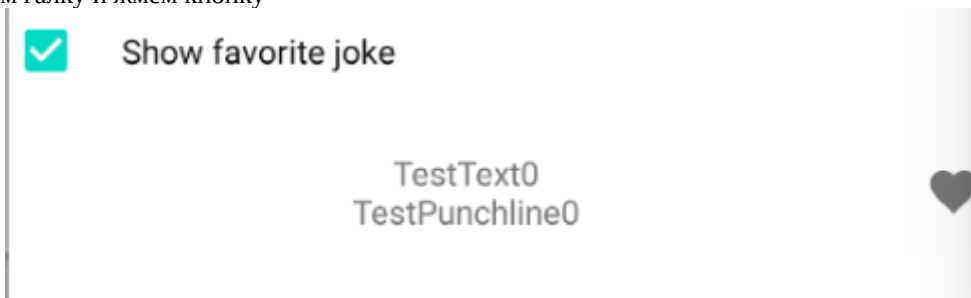
Запустим код. Первую шутку кладем в кеш, после получаем другую шутку. В чекбокс ставим галку и жмем кнопку получить шутку – должна быть первая с айди 0



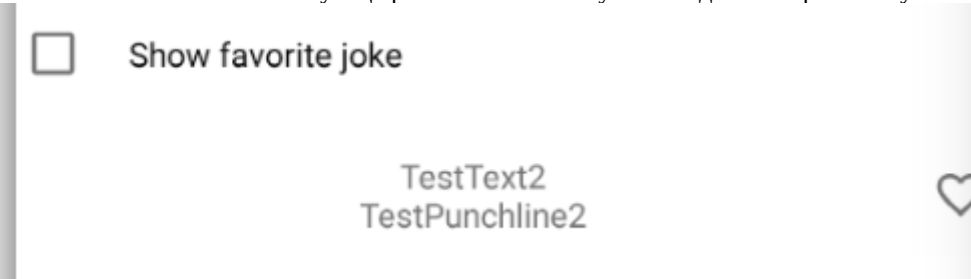
Поменяем шутку кнопкой



Теперь ставим галку и жмем кнопку



Для достоверности можете нажать кнопку еще раз. Там всего 1 шутка с айди 0. Уберем галку и нажмем еще раз



Да, все работает как мы и планировали. Единственное что даже если вы добавите 2 шутки, то все равно из кеша сможете посмотреть лишь первую. Там тоже можете поставить счетчик и сделать по аналогии с клауд. Ладно, все это прекрасно. Давайте использовать базовый клауд и тестовый перепишем немного

```
viewModel = ViewModel(
    BaseModel(
        TestCacheDataSource(),
        BaseCloudDataSource(retrofit.create(JokeService::class.java)),
        BaseResourceManager(context: this)
```

Я переписал на список, но это не совсем верно, хотя для тестового сойдет

```
class TestCacheDataSource : CacheDataSource {

    private val list = ArrayList<Pair<Int, JokeServerModel>>()

    override fun getJoke(jokeCachedCallback: JokeCachedCallback) {
        if (list.isEmpty()) {
            jokeCachedCallback.fail()
        } else {
            jokeCachedCallback.provide(list.random().second)
        }
    }

    override fun addOrRemove(id: Int, jokeServerModel: JokeServerModel): Joke {
        val found = list.find { it.first == id }
        return if (found != null) {
            val joke = found.second.toBaseJoke()
            list.remove(found)
            joke
        } else {
            list.add(Pair(id, jokeServerModel))
            jokeServerModel.toFavoriteJoke()
        }
    }
}
```

Давайте запустим уже код с базовым клауд дата сорсом и посмотрим на результат  
Я последовательно получу из сети несколько шуток и все добавлю в кеш, после чего  
выключу интернет и поменяю источник на сохраненные шутки



Show favorite joke

What do you call a fake noodle?  
An impasta.



GET JOKE

И давайте уберем из избранных эту шутку и последовательно все из кеша.  
Итак, у нас бага. Мы меняли статус у закешированной шутки, но когда получили шутку из  
кешдатарсorsa то не клали ее в кеш переменную в модели. Нужно пофиксить

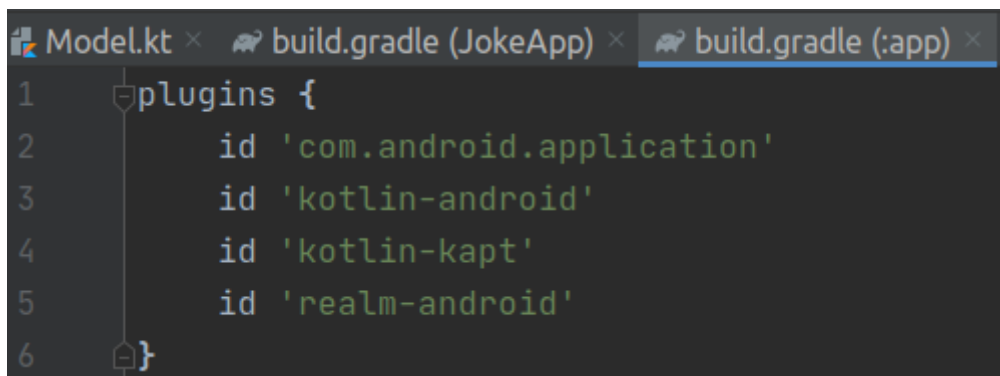
```
cacheDataSource.getJoke(object : JokeCachedCallback {
    override fun provide(jokeServerModel: JokeServerModel) {
        cachedJokeServerModel = jokeServerModel
        jokeCallback?.provide(jokeServerModel.toFavoriteJoke())
    }
    override fun fail() {
        cachedJokeServerModel = null
    }
})
```

Запустим код еще раз и проверим – добавить из сети 3 шутки в кеш, после убрать их. Да, теперь все верно. Кстати, мы сделали хорошо то, что при удалении из сохраненных мы не меняем сами по себе шутку, может человек решил все же вернуть шутку, мы должны дать такую возможность. Не нужно всплывающих снэкбаров с отменой действия. Пусть юзер добавляет в кеш так же как и всегда.

Вы можете проверить кейс, когда убираем из кеша и добавляем обратно.

## 2. Добавляем noSQL

Все хорошо, вроде как, но у нас одна проблема. Все наши шутки которые мы хотим сохранить не будут жить после смерти приложения. И кто-то скажет – положим в SharedPreferences! Но нет, туда мы клали такие простые вещи как число или строка. Здесь же у нас нечто сложнее. Да, айди и текст, но все равно, как их сохранять? У нас же не 1 шутка может быть, а много. Поэтому для более сложных структур данных нам нужна полноценная база данных на андроид. Я если честно люблю noSQL потому что с ними легче работать. И поэтому мы сейчас внедрим мою любимую библиотеку Realm.



```
1 plugins {
2     id 'com.android.application'
3     id 'kotlin-android'
4     id 'kotlin-kapt'
5     id 'realm-android'
6 }
```

Добавим 2 плагина и во втором градл файле добавим зависимость

```
dependencies {
    classpath "com.android.tools.build:gradle:4.2.1"
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    classpath "io.realm:realm-gradle-plugin:10.0.1"
```

инструкции можно найти в гугле или по ссылке

<https://docs.mongodb.com/realm/sdk/android/install/>

Теперь мы должны написать наш базовый кешдатасорс, который будет хранить шутки

Инициализируем реалм в апликайшн классе

```
override fun onCreate() {  
    super.onCreate()  
    Realm.init(context: this)  
}
```

И уже спокойно пишем наш класс. Но погодите-ка. Мы не можем класть в реалм любые классы. Нам нужны классы наследники RealmObject. Проблема в том, что сам класс должен быть наследуемым и все поля не финальными, да, такое вот ограничение. Можете сами прочитать документацию почему так

```
open class JokeRealm : RealmObject() {  
    @PrimaryKey  
    var id: Int = -1  
    var text: String = ""  
    var punchLine: String = ""  
    var type: String = ""  
}
```

Мы будем класть в реалм по айди и потому мы пишем аннотацию PrimaryKey что означает ключ. Суть в том, что в реалм вы не сможете добавить 2 объекта с одинаковыми значениями в поле помеченном этой аннотацией. Итак, модель данных готова и можно приступить к самому классу. Создадим BaseCachedDataSource. Я не указываю Realm в имени потому что возможно в будущем мы будем использовать другую либу (например Room).

```
class BaseCachedDataSource(private val realm: Realm) : CacheDataSource {  
  
    override fun getJoke(jokeCachedCallback: JokeCachedCallback) {  
        realm.use { it: Realm  
            val jokes = it.where(JokeRealm::class.java).findAll()  
            if (jokes.isEmpty())  
                jokeCachedCallback.fail()  
            else  
                jokes.random().let { joke ->  
                    jokeCachedCallback.provide(  
                        JokeServerModel(  
                            joke.id,  
                            joke.type,  
                            joke.text,  
                            joke.punchLine  
                        )  
                    )  
                }  
        }  
    }  
}
```

Мы получаем в конструктор реалм и используем его потому что это база данных которую нужно открывать и закрывать после использования так же как и стримы данных. Если не



закроете реалм, то у вас при каждом обращении будет открыт инстанс. Итак, что же мы делаем : находим все объекты по типу шутки и проверяем список на пустоту или же получаем рандомную и создаем серверную модель. Здесь конечно же неправильно что мы из модели базы данных мапим к серверной. В идеале должна быть независимая модель к которой мапим из серверной и из бдшной. Но вы сами можете это сделать. Если я не забуду то перепишу это в конце лекции.

С чтением разобрались, идем дальше. Как класть в реалм и удалять

```
override fun addOrRemove(id: Int, jokeServerModel: JokeServerModel): Joke {
    realm.use { it: Realm
        val jokeRealm = it.where(JokeRealm::class.java).equalTo(fieldName: "id", id).findFirst()
        return if (jokeRealm == null) {
            val newJoke = jokeServerModel.toJokeRealm()
            it.executeTransaction { transaction ->
                transaction.insert(newJoke)
            }
            jokeServerModel.toFavoriteJoke()
        } else {
            it.executeTransaction { it: Realm
                jokeRealm.deleteFromRealm()
            }
            jokeServerModel.toBaseJoke()
        }
    }
}
```

Сначала находим наш объект по айди, и здесь нам нужно указать по какому полю ищем. Если в реалме нет такого объекта – то мы создаем его и кладем через транзакцию методом insert.

Иначе же если такая модель есть – удаляем из реалма и в конце вернем то, что от нас ожидает метод. Вот и все. Но так как поля класса JokeServerModel приватные мы написали метод

```
fun toFavoriteJoke() = FavoriteJoke(text, punchline)
fun toJokeRealm(): JokeRealm {
    return JokeRealm().also { it: JokeRealm
        it.id = id
        it.type = type
        it.text = text
        it.punchLine = punchline
    }
}
```

Теперь фиксим класс аплишейшн и можно запускать

```
viewModel = ViewModel(
    BaseModel(
        BaseCachedDataSource(Realm.getDefaultInstance()),
```

Итак, сохраним из сети несколько шуток и убьем приложение. Откроем и выберем сохраненные шутки и посмотрим правильно ли все работает или нет

Running transactions on the UI thread has been disabled.

Да, как ни странно, но для записи чтения и удаления из реалма нам опять нужен новый поток.

Но можно просто делать это асинхронно с помощью метода с суффиксом асинк

```
it.executeTransactionAsync { transaction ->
    transaction.insert(newJoke)
}
```

И теперь все должно работать как надо

Но нет. Опять краш и на этот раз потому что мы передали инстанс в конструктор в апликайшне и после первого использования закрыли его. Закрытый реалм нельзя снова использовать. Значит убираем use и используем без него. Замените use на let .

Вот теперь все работает как нужно. Но вопрос. Мы же использовали инстанс все время и не закрыли его, как же так? Предлагаю вам самостоятельно изучить этот вопрос.

Так же самостоятельно напишите отдельный класс для шуток в которую получают данные из серверной модели и из реалм объекта, так же мапинг из нее в серверную и в реалм.

Чтобы закрепить работу с 2 датасорсами возьмите опен апи которые вы могли сделать в прошлой лекции как задание и кладите в реалм самостоятельно.