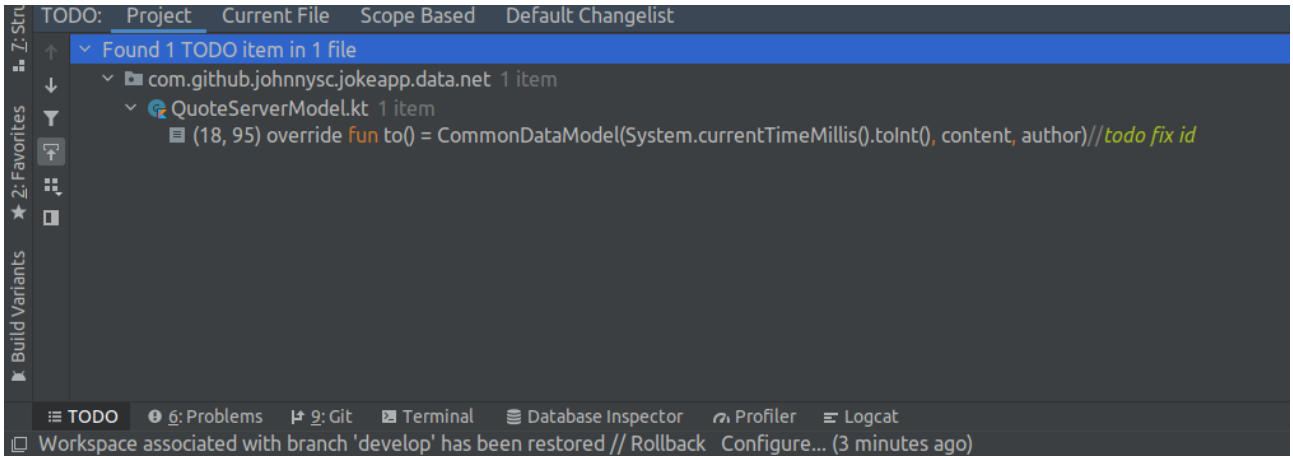
TODO

Как не забыть пофиксить код

В предыдущей лекции мы столкнулись с тем, что от серверов приходят разные типы для айди шутки и цитаты. В одном месте число, в другом строка. И как мы можем это пофиксить? В предыдущий раз я создал тудушку. Просто написал : //todo fix this later

И теперь остается вопрос : в каком классе я это написал? Как это найти? Во-первых есть глобальный поиск Ctrl+Shift+F , но специально для тудушек есть целый подраздел на дне AC



И теперь мы можем более эффективно искать тудушки в проекте или в текущем файле если он у вас открыт и в нем так много линий кода что вы не видите тудушку.

Ладно, давайте уже пофиксим эту тудушку.

Если помните, а если не помните посмотрите – мы работали с айди лишь в реалм моделях. И мы можем сделать дженерик тип для айди и просто прописать нужный тип в маперах к реалммоделям.

Поехали

```
interface CommonDataModelMapper<T, E> {
    fun map(id: E, first: String, second: String, cached: Boolean): T
}
```

Сначала в мапере делаем дженерик тип для айди и меняем инт на Е. Теперь можем пофиксить место вызова метода мап.

Я просто саму модель сделаю дженерик и передам тип айди внутрь. Далее мне нужно пофиксить реализации маперов и это самое просто. Начнем с мапера успеха, там где айди не используется вовсе. Если оно не используется то и неважно какого оно типа. Передам любой тип. Пусть будет Any.

```
class CommonDataModel<E>(
    private val id: E,
    private val firstText: String,
    private val secondText: String,
    private val cached: Boolean = false
) : ChangeCommonItem {

    fun <T> map(mapper: CommonDataModelMapper<T, E>): T {
        return mapper.map(id, firstText, secondText, cached)
    }
```

```
class CommonSuccessMapper : CommonDataModelMapper<CommonItem.Success, Any> {
    override fun map(id: Any, first: String, second: String, cached: Boolean) =
        CommonItem.Success(first, second, cached)
}
```

А для двух других просто передам нужные типы

```
class JokeRealmMapper : CommonDataModelMapper<JokeRealmModel, Int> {
    override fun map(id: Int, first: String, second: String, cached: Boolean) =
        JokeRealmModel().also { joke ->
```

Чтобы мапер цитат теперь работал со строковым айди я просто переделаю модель

```
open class QuoteRealmModel : RealmObject() {
    @PrimaryKey
    var id: String = ""
```

Вот и все. Теперь можно пофиксить мапер

```
class QuoteRealmMapper : CommonDataModelMapper<QuoteRealmModel, String> {
    override fun map(id: String, first: String, second: String, cached: Boolean) =
        QuoteRealmModel().also { quote ->
            quote.id = id
```

И вроде как все классно теперь, но у нас одно но. Вернемся в класс CommonDataModel.

```
override suspend fun change(changeStatus: ChangeStatus) = changeStatus.addOrRemove(id, this)
```

У нас метод был с измененим статуса и мы прокидывали туда айди. Значит ChangeStatus тоже должен быть дженериком. Давайте пофиксим и его

```
interface ChangeStatus<E> {
    suspend fun addOrRemove(id: E, model: CommonDataModel<E>): CommonDataModel<E>
}
```

И теперь посмотрим где оно используется и пофиксим там

```
interface ChangeCommonItem<E> {
    suspend fun change(changeStatus: ChangeStatus<E>): CommonDataModel<E>

    class Empty : ChangeCommonItem<Any> {
        override suspend fun change(changeStatus: ChangeStatus<Any>): CommonDataModel<Any> {
            throw IllegalStateException("empty change item called")
```

Идем дальше и ищем еще места использования или же нажимаем Ctrl и на имя интерфейса

```
interface CachedData<E> : ChangeCommonItem<E> {
    fun save(data: CommonDataModel<E>)
    fun clear()
}
```

Кажется у нас половина проекта будет параметризирована. Ну и пусть.

```
class BaseCachedData<E> : CachedData<E>{
    private var cached: ChangeCommonItem<E> = ChangeCommonItem.Empty()
    override fun save(data: CommonDataModel<E>) {
        cached = data
    }

    override fun clear() {
        cached = ChangeCommonItem.Empty()
    }

    override suspend fun change(changeStatus: ChangeStatus<E>): CommonDataModel<E> {
        return cached.change(changeStatus)
    }
}
```

Здесь у нас конфликт с Empty и потому давайте его перепишем с дженериком тоже

```
class Empty<E> : ChangeCommonItem<E> {
    override suspend fun change(changeStatus: ChangeStatus<E>): CommonDataModel<E> {
```

Идем дальше и ищем что еще пофиксить нужно. Например апликейшн класс

```
val jokeRepository = BaseRepository(cacheDataSource, cloudDataSource, BaseCachedData<Int>())
```

2 места где используется базовый кеш с дженериком

```
QuoteCloudDataSource(retro
BaseCachedData<String>()
```

И конечно же раз у нас базовый репозиторий, то и там нужен дженерик

```
interface CommonRepository<E> {
    suspend fun getCommonItem(): CommonDataModel<E>
    suspend fun changeStatus(): CommonDataModel<E>
    fun chooseDataSource(cached: Boolean)
}
```

Сначала интерфейс репозитория с дженериком и после уже реализация

```
interface CacheDataSource<E> : DataFetcher, ChangeStatus<E>
```

Но сначала конечно же датасорсы

```
abstract class BaseCachedDataSource<T : RealmObject, E>(
    private val realmProvider: RealmProvider,
    private val mapper: CommonDataModelMapper<T, E>,
    private val realmToCommonDataMapper: RealmToCommonDataMapper<T>
) : CacheDataSource<E> {

    protected abstract val dbClass: Class<T>

    override suspend fun getData(): CommonDataModel<E> {
        realmProvider.provide().use {  it: Realm
```

И этот класс будет самым проблемным, посмотрите на второй метод.

```
override suspend fun addOrRemove(id: E, model: CommonDataModel<E>): CommonDataModel<E> =
    withContext(Dispatchers.IO) {  this: CoroutineScope
        realmProvider.provide().use {  it: Realm
            val itemRealm =
                it.where(dbClass).equalTo( fieldName: "id", id).findFirst()
```

Мы упираемся в Реалм – он не знает какого типа айди. И мы должны явно сказать ему – строка или число. Ведь у него перегруженные методы equalTo. Как же нам это решить?

Конечно же наследованием! Пусть наследники сами скажут какой у них тип. Для этого напишем абстрактный метод в классе и пусть наследники укажут как им найти айди

```
protected abstract fun findRealmObject(realm: Realm, id: E): T?

override suspend fun addOrRemove(id: E, model: CommonDataModel<E>): Co
    withContext(Dispatchers.IO) {  this: CoroutineScope
        realmProvider.provide().use {  it: Realm
            val itemRealm = findRealmObject(it, id)
            return@withContext if (itemRealm == null) {
```

Итак, мы отдаем наследникам самим найти свой реалм обьект, а остальная логика все равно инкапсулирована в базовом датасорсе. Напишем же реализации методов наследника

```
class JokeCachedDataSource(
    realmProvider: RealmProvider,
    mapper: JokeRealmMapper,
    commonDataMapper: JokeRealmToCommonMapper
) : BaseCachedDataSource<JokeRealmModel, Int>(realmProvider, mapper, commonDataMapper) {
    override val dbClass = JokeRealmModel::class.java
    override fun findRealmObject(realm: Realm, id: Int) =
        realm.where(dbClass).equalTo( fieldName: "id", id).findFirst()
}
```

Теперь как видите из родительского класс пришел метод с инт типом и мы можем легко и просто написать квери реалма. Ровно так же поступим с цитатами

```
class QuoteCachedDataSource(
    realmProvider: RealmProvider,
    mapper: QuoteRealmMapper,
    commonDataMapper: QuoteRealmToCommonMapper
) : BaseCachedDataSource<QuoteRealmModel, String>(realmProvider, mapper, commonDataMapper) {
    override val dbClass = QuoteRealmModel::class.java
    override fun findRealmObject(realm: Realm, id: String) =
        realm.where(dbClass).equalTo( fieldName: "id", id).findFirst()
}
```

Вот и все. Проверим что все сделали. Просто запустим код и АС подскажет что забыли

CommonDataModel метод без дженерика остался

```
override suspend fun change(changeStatus: ChangeStatus<E>) = changeStatus.addOrRemove(id,  model: this)
```

И конечно же интерактоpы стали зависеть от дженерика из-за репозитория. Пофиксим интерактоpы!

```
class BaseInteractor<E>(
    private val repository: CommonRepository<E>,
    private val failureHandler: FailureHandler,
    private val mapper: CommonDataModelMapper<CommonItem.Success, E>
) : CommonInteractor {
```

Просто докинем дженерик и передадим 2 зависимостям и конечно же надо будет пофиксить апликейшн класс, но перед этим пофиксим мапер успеха там где написали Any

```
class CommonSuccessMapper<E> : CommonDataModelMapper<CommonItem.Success, E> {
    override fun map(id: E, first: String, second: String, cached: Boolean) =
        CommonItem.Success(first, second, cached)
}
```

И теперь апликейшн класс заиграл новыми красками. Посмотрите теперь на это

```
val quoteMapper = CommonSuccessMapper<String>()
quoteViewModel = BaseViewModel(
    BaseInteractor(quoteRepository, failureHandler, quoteMapper),
```

Нам пришлось написать отдельный мапер для цитат, хотя и там не используется дженерик.

Ладно, давайте уже запустим код, предлагаю вам разобраться самостоятельно с этим.

```
interface DataFetcher<E> {
    suspend fun getData(): CommonDataModel<E>
}
```

Мы не все параметризировали и поэтому надо дописать еще немного кода

```
interface CacheDataSource<E> : DataFetcher<E>, ChangeStatus<E>

interface CloudDataSource<E> : DataFetcher<E>

abstract class BaseCloudDataSource<T : Mapper<CommonDataModel<E>>,
E> : CloudDataSource<E> {
   protected abstract fun getServerModel(): Call<T>
   override suspend fun getData(): CommonDataModel<E> {

class JokeCloudDataSource(private val service: BaseJokeService) :
   BaseCloudDataSource<JokeServerModel, Int>() {

class QuoteCloudDataSource(private val service: QuoteService) :
   BaseCloudDataSource<QuoteServerModel, String>() {

class BaseRepository<E>(
   private val cacheDataSource: CacheDataSource<E>,
   private val cloudDataSource: CloudDataSource<E>,
   private val cached: CachedData<E>
) : CommonRepository<E> {
   private var currentDataSource: DataFetcher<E> = cloudDataSource

interface RealmToCommonDataMapper<T : RealmObject, E> {
   fun map(realmObject: T): CommonDataModel<E>
}
class JokeRealmToCommonMapper :
RealmToCommonDataMapper<JokeRealmModel, Int> {

class QuoteRealmToCommonMapper :
RealmToCommonDataMapper<QuoteRealmModel, String> {

abstract class BaseCachedDataSource<T : RealmObject, E>(
   private val realmProvider: RealmProvider,
   private val mapper: CommonDataModelMapper<T, E>,
   private val realmToCommonDataMapper: RealmToCommonDataMapper<T,
E>
) : CacheDataSource<E> {
```

```
data class JokeServerModel(
    @SerializedName("id")
    private val id: Int,
    @SerializedName("type")
    private val type: String,
    @SerializedName("setup")
    private val text: String,
    @SerializedName("punchline")
    private val punchline: String
) : Mapper<CommonDataModel<Int>> {
    override fun to() = CommonDataModel(id, text, punchline)
}
```

И наконец-то мы можем пофиксить серверную модель цитаты

```
class QuoteServerModel(
    @SerializedName("_id")
    private val id: String,
    @SerializedName("content")
    private val content: String,
    @SerializedName("author")
    private val author: String
) : Mapper<CommonDataModel<String>> {
    override fun to() = CommonDataModel(id, content, author)
}
```

И да, мы опять поменяли реалм обьект – у цитаты поменяли тип айди с числового на строковый, а не написав кода для миграции у нас будет краш. Поэтому удалите с девайса приложение и установите заново. В следующих лекциях мы рассмотрим вопрос миграции реалма подробно.
Неожиданно! Все работает как и прежде. Если у вас возникли проблемы при запуске проекта, то просто посмотрите на все изменения которые я запушил в гитхаб по ссылке

https://github.com/johnnysc/cleanarchexample

В следующих лекциях мы разделим фичи на экраны и покажем сразу все избранные цитаты и шутки