# Навигация

**com.goodgame.goodgameapp.navigation**

В navigation.kt мы указываем все экраны, которые присутствуют в приложении. Список существующих на данный момент:

object SplashScreen : Screen("SplashScreen")

    object LoginScreenNew : Screen("LoginScreenNew")

    object RegistrationScreen : Screen("RegistrationScreen")

    object LoginCodeConfirmScreen : Screen("LoginCodeConfirmScreen")

    object MainScreen : Screen("MainScreen")

    object IntroScreen : Screen("IntroScreen")

    object CharacterCreationScreen: Screen("CharacterCreationScreen")

    object PlanningCenterScreen: Screen("PlanningCenterScreen")

    object DiagnosticsScreen: Screen("DiagnosticsScreen")

    object SupplyScreen: Screen("SupplyScreen")

    object ExpeditionScreen: Screen("ExpeditionScreen")

**SplashScreen:** Экран появляющийся при запуске приложения.

**LoginScreenNew:** Экран логина.

**RegistrationScreen:** Экран с регистрацией

**LoginCodeConfirmScreen**: Экран на котором вводится код подтверждения для входа

**MainScreen**: Основной экран, который пользователь видит при входе. Отсюда можно перейти в центр планирования, магазин, на экран прокачки

**IntroScreen**: Экран с каруселькой карточек с предысторией

**CharacterCreationScreen**: Экран с выбором персонажа при регистрации

**PlanningCenterScreen**: Центр планирования. Отсюда переход к экспедиции, можно посмотреть историю экспедиций

**DiagnosticsScreen**: Центр диагностики. Здесь прокачиваем персонажа

**SupplyScreen**: Магазин бонусов, которые можно получить в клубе

**ExpeditionScreen**: Экран экспедиции

В каждый экран передаётся навигационный контроллер (через него осуществляется переход на другие экраны) и вью модель. Вью модель выполняет всякие разные бэк взаимодействия, которые не относятся к интерфейсу, например, отправка запросов на сервер, работа с памятью телефона.

# LoginViewModel

**com.goodgame.goodgameapp.viewmodel**

В данной вью модели мы посылаем запросы на сервер связанные с логином и регистрацией. После первого успешного логина мы получаем от сервера токен, который далее используем для входа. Храним его в app preferences потому что это удобно и безопасно. В целом, названия методов отражают их функционал.

# GameViewModel

**com.goodgame.goodgameapp.viewmodel**

Основная вью модель, с которой мы работаем во всём, кроме логина. Здесь мы храним статы персонажа:

val heroInfo = MutableLiveData(HeroInfo(

        hasHero = false,

        heroClass = "",

        expeditions = emptyList(),

        total\_progress = 0,

        level = 0,

        lvl\_exp = 0,

        next\_lvl\_need = 0,

        has\_expeditions = 0,

        stats\_points = 0,

        stats = StatsModel(0,0,0,0),

        coins = 0,

        new\_level = false,

        expedition\_passed = false

    ))

Во вью модели есть функционал для сохранения инфо о персонаже в памяти телефона, но это не используется.

**Запросы**

Остановлюсь здесь чуть более подробно. Если вы добавляете какую-то работу с интернетом после меня, то разобраться в этом важно. Сам запрос указывается в

com.goodgame.goodgameapp.retrofit

Параметры отправляемые на сервер указываются в параметрах функций в ApiInterface. Если будет нужен новый запрос, то можно сделать просто по аналогии с уже существующими. Возвщаемый результат автоматически декодируется из json в модель. Например:

@FormUrlEncoded

    @POST("hero/info")

suspend fun getHeroInfo(@Field("private\_key") private\_key : String) : HeroInfoResponse

Здесь мы отправляем на сервер private\_key, т.е. токен по адресу:

https://crmgoodgame.ru/telegram/api/hero/info

, а получаем от него HeroInfoResponse(уже декодированный из json):

data class HeroInfoResponse(

    val status : Boolean,

    val level: Int,

    val type : String,

    val expeditions : List<ExpeditionStoryModel>,

    val planet\_status : Int?,

    val lvl\_exp: Int,

    val rsp: Int,

    val next\_lvl\_need : Int,

    val has\_expeditions: Int,

    val stats\_points: Int,

    val stats: StatsModel)

Модель описывается как data class.

Далее разберём как интерфейс обрабатывает запрос

viewModel.getHeroInfo().observe(lifecycleOwner, Observer {

                when (it.status) {

                    Status.SUCCESS -> {

                        splashScreenState.value = SplashScreenState.GOT\_HERO\_INFO

                    }

                    Status.ERROR   -> { splashScreenState.value = SplashScreenState.INTERNET\_ERROR }

                    Status.LOADING -> { }

                }

            })

На первый взгляд всё немного запутано, разберём. Во вью модели GameViewModel есть функция отправки запроса на получения инфо о персонаже. Так как возвращаемый ей объект имеет тип LiveData, то мы можем подписаться на обновление её состояния. Для этого используем метод observe. LiveData является дженериком, его подтипом является Response. У него есть три статуса:

enum class Status {

    SUCCESS,

    ERROR,

    LOADING,

}

При каждом обновлении LiveData, будет запускаться указанная выше лямбда функция, а мы в свою очередь будем обрабатывать обновления состояния. Например, залочим интерфейс пока идёт загрузка, в случае если выпадет ошибка откинем плашку с ошибкой, но если всё прошло хорошо, то всё хорошо)

**Получение heroInfo:**

fun getHeroInfo(initial: Boolean = true) : LiveData<Response<Nothing?>> {

параметр initial это костыль. С если он false, то мы сравниваем лвл в текущем heroInfo и тем, который мы получили от сервера. Если на сервере больше, то значит персонаж лвлапнулся. Аналогично expedition\_passed. Необходимо для запуска миниокна с поздравительной анимации на MainScreen.

**Отправка обновленных скилов персонажа:**

fun setHeroSkill(skill\_type: String) : LiveData<Response<SkillResponse>> {

Просто указываем название: Сила -> power, Интеллект -> intellect, Харизма -> charisma, Удача -> fortune

**Список товаров из магазина:**

fun getShopList() : LiveData<Response<List<ShopItem>>> {

**Покупка предмета:**

fun buyShopItem(shopItem: ShopItem) : LiveData<Response<ShopBuyResponse>> {

**Список купленных плюшек:**

fun getRewardList() : LiveData<Response<List<Reward>>> {

**Получение изображения для экспедиции:**

fun getImage(url: String, loadingLiveData: MutableLiveData<Response<Int>>) {

Сперва мы скачиваем изображение, далее мы сохраняем его в кэш, а в самой экспедиции просто достаём изображение из кэша и его показываем. Всё просто, но выглядит страшно

**Получение информации о экспедиции:**

fun getExpedition() : LiveData<Response<Expedition>> {

Более подробно о содержании запроса о экспедиции смотри в модели Expedition

**Получение результата экспедии:**

fun getExpeditionResult(choice: String) : LiveData<Response<ExpeditionResult>> {

Аналогично getExpedition(). Choice это выбор между пропуском экспедии и попыткой успешного завершения

# Models

package com.goodgame.goodgameapp.models

В целом, не буду останавливаться на этой папке, здесь хранятся data классы моделек разных запросов или просто какой-нибудь инфы. Названия моделей вполне понятны, чтобы разобрать что конкретно они хранят.

# Экраны

package com.goodgame.goodgameapp.screens

## Состояния

На каждом экране подробно останавливаться не буду, слишком много всего. Опишу в целом фишки jetpack compose.

Основной фишкой композа являются состояния, которые имеют тип mutableStateOf. Пример:

var showExpeditionSuccess by remember { mutableStateOf(false)}

Это состояние не пропадает при повороте экрана, перерисовке экрана. В целом, жизненный цикл мутаблестейтов можно нагуглить.

При изменинии стейта, происходит перерисовка экрана, но только той части, в которой этот стейт изменился. Удобненько. Например:

if (showExpeditionSuccess) {

        ExpeditionCompletedView(

            expedition = heroInfo?.expeditions?.last()

                ?: ExpeditionStoryModel(exp = 0, rsp = 0, name = null, 0, result = "exp", dateTime = null)) {

            heroInfo?.expedition\_passed = false

            showExpeditionSuccess = false

        }

    }

Если showExpeditionSuccess == true, то показываем анимированную плашку о завршении экспедиции, если сбрасываем в false, то она просто пропадает. Не надо вызывать никаких функций, экран просто перерисуется с новым состоянием.

Подобным образом мы управляем состоянием текущего экрана, а так же в стэйты мы сохраняем данные которые вводит пользователь с клавиатуры.

## Вёрстка

Используется происходит при помощи кода, без xml. Если понадобится что-то верстать, проще посмотреть на ютубе школьников. Ну или разбираться в моих каракулях.