# Архитектура Android: Hilt, MVVM, Kotlin Coroutines, Live Data, Room and Retrofit (ft. Рик и Морти)

[](https://medium.com/@sberoch?source=post_page-----8b746cab4a06--------------------------------)

[Сантьяго Бероч](https://medium.com/@sberoch?source=post_page-----8b746cab4a06--------------------------------)

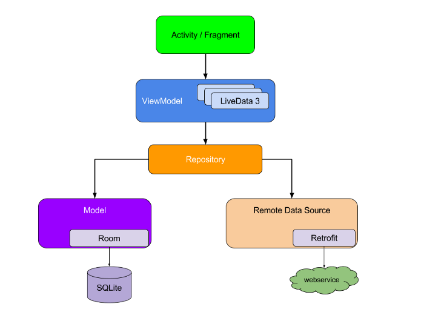
[Мы в соцсетях](https://medium.com/m/signin?actionUrl=%2F_%2Fsubscribe%2Fuser%2Fdf8696b300da&operation=register&redirect=https%3A%2F%2Fitnext.io%2Fandroid-architecture-hilt-mvvm-kotlin-coroutines-live-data-room-and-retrofit-ft-8b746cab4a06&source=post_page-df8696b300da----8b746cab4a06---------------------follow_byline-----------)

[14 июля](https://itnext.io/android-architecture-hilt-mvvm-kotlin-coroutines-live-data-room-and-retrofit-ft-8b746cab4a06?source=post_page-----8b746cab4a06--------------------------------) · 6 мин читала

При разработке приложения для Android важно спланировать архитектуру проекта. Это позволит нам создавать сложные, надежные, качественные, простые в обслуживании приложения.

Сегодня я хочу показать вам свой подход к использованию рекомендуемых практик в разработке Android.

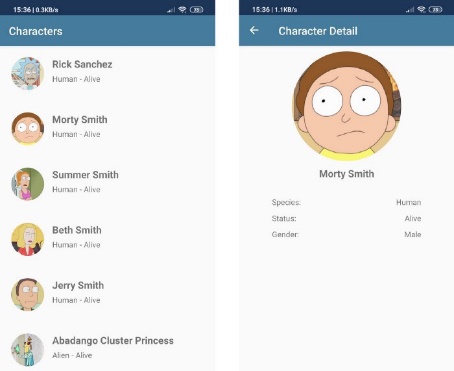
Как вы знаете, написание всего кода нашего приложения в виде действия или фрагмента является распространенной ошибкой. Мы должны разделить различные проблемы наших компонентов приложения. Следующая диаграмма покажет вам модули и их взаимодействия, которые мы будем использовать:



Компоненты Архитектуры Android

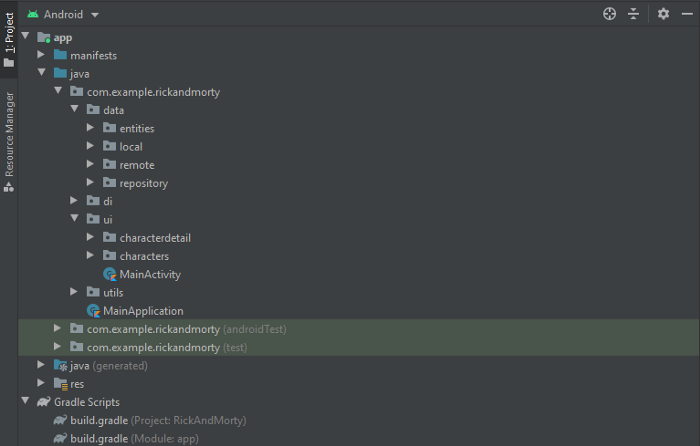
# Проект

Мы будем использовать [The Rick and Morty API,](https://rickandmortyapi.com/) оттуда мы будем извлекать информацию о символах, чтобы показать их в RecyclerView, и, нажав на один из его элементов, приложение отобразит детали одного символа. Простой.



Приложение

И это структура папок проекта:



структура папок

Это может показаться слишком большим. Вот вам некоторое объяснение:

**данные:**Наша м (модель) в МВВМ. Где мы выполняем операции с данными.

**di:**Инъекция зависимостей с помощью Эфеса.

**пользовательский интерфейс**: Наши фрагменты и ViewModels помогают отображать данные пользователю.

**utils**: Вспомогательные классы и функции.

Я рекомендую вам взглянуть на репозиторий GitHub, пока вы следуете этому учебнику.

И последнее, прежде чем продолжить. API возвращает этот ответ при запросе списка символов

|  |  |
| --- | --- |
|  | { |
|  | "info":{ |
|  | "count": 591, |
|  | "pages": 20, |
|  | "next": "https://rickandmortyapi.com/api/character/?page=2", |
|  | "prev": null |
|  | }, |
|  | "results":[ |
|  | // ... |
|  | ] |
|  | } |

[**view raw**](https://gist.github.com/sberoch/92e3983fc0a9d84e153004cc3563190a/raw/834c5c23d781f727981d5315756f6eb27a2162b7/rickmortyapiexample.json5)[**rickmortyapiexample.json5**](https://gist.github.com/sberoch/92e3983fc0a9d84e153004cc3563190a#file-rickmortyapiexample-json5) hosted with ❤ by [**GitHub**](https://github.com/)

Мы получаем разбитый на страницы Ответ. Фактические символы возвращаются в разделе "Результаты". Вот почему вы найдете три класса под папкой сущности. Но самое главное — это характер.

Сказав это, мы переходим к Хилту и Ди.

# Рукоять

Это не учебник DI, поэтому я настоятельно рекомендую немного больше ознакомиться с этой концепцией, прежде чем продолжать. Я все равно вкратце объясню.

Это очень распространено для приложений, где проблемы разделены, что некоторые компоненты имеют зависимости. **ViewModel**зависит от **репозитория**для получения данных. Но репозиторий также зависит от **CharacterRemoteDataSource** и **CharacterDao**. И это простое приложение. Это может довольно быстро превратиться в ад зависимости.

На помощь приходит Хилт. Нам просто нужно сообщить, где нам нужно ввести эти зависимости и откуда они должны быть получены.

Мы начинаем с создания в корневой папке класса, который наследуется от приложения, чтобы аннотировать его, сообщая, что мы будем использовать Hilt в приложении.

@HiltAndroidApp

class MainApplication : Application()

Затем нам нужно аннотировать каждую активность и фрагмент как AndroidEntryPoints:

@AndroidEntryPoint

class MainActivity : AppCompatActivity() {

//...

}

@AndroidEntryPoint

class CharacterDetailFragment : Fragment() {

//...

}

Наш следующий шаг - теперь создать нашмодуль внутри папки di. Если вы не так хорошо знакомы с концепциями di, думайте о модуле как о "мешке", из которого мы будем получать наши зависимости. (Я покажу много зависимостей из следующих разделов этого урока, так что не пугайтесь, если вы не понимаете их все сейчас).

@Module

@InstallIn(ApplicationComponent::class)

object AppModule {

@Singleton

@Provides

fun provideRetrofit(gson: Gson) : Retrofit = Retrofit.Builder()

.baseUrl("https://rickandmortyapi.com/api/")

.addConverterFactory(GsonConverterFactory.create(gson))

.build()

@Provides

fun provideGson(): Gson = GsonBuilder().create()

@Provides

fun provideCharacterService(retrofit: Retrofit): CharacterService = retrofit.create(CharacterService::class.java)

@Singleton

@Provides

fun provideCharacterRemoteDataSource(characterService: CharacterService) = CharacterRemoteDataSource(characterService)

@Singleton

@Provides

fun provideDatabase(@ApplicationContext appContext: Context) = AppDatabase.getDatabase(appContext)

@Singleton

@Provides

fun provideCharacterDao(db: AppDatabase) = db.characterDao()

@Singleton

@Provides

fun provideRepository(remoteDataSource: CharacterRemoteDataSource,

localDataSource: CharacterDao) =

CharacterRepository(remoteDataSource, localDataSource)

}

AppModule.уз

**@Module**аннотирует объект, чтобы указать, что мы получим наши зависимости отсюда. Мы будем использовать только один в этом простом приложении, но более масштабные проекты обычно имеют много модулей.

**@Singleton**это приведет к тому, что только один экземпляр зависимости будет создан и использован во всем приложении.

**@Provides**указывает, что предстоящая функция обеспечит зависимость.

Так как же мне сделать инъекцию?

class CharacterRepository @Inject constructor(

private val remoteDataSource: CharacterRemoteDataSource,

private val localDataSource: CharacterDao

) {

//...

}

@Впрыскивать

# ViewModel, LiveData и отображение данных.

Благодаря использованию LiveData и Kotlin Coroutines мы можем избежать обратных вызовов и иметь действительно простой способ представить наши данные нашим представлениям. Вот как это делает эта ViewModel:

class CharactersViewModel @ViewModelInject constructor(

private val repository: CharacterRepository

) : ViewModel() {

val characters: LiveData<Resource<List<Character>>> = repository.getCharacters()

}

CharactersViewModel.уз

Возможно, вас интересует **ресурс.**Именно этот вспомогательный класс

data class Resource<out T>(val status: Status, val data: T?, val message: String?) {

enum class Status {

SUCCESS,

ERROR,

LOADING

}

companion object {

fun <T> success(data: T): Resource<T> {

return Resource(Status.SUCCESS, data, null)

}

fun <T> error(message: String, data: T? = null): Resource<T> {

return Resource(Status.ERROR, data, message)

}

fun <T> loading(data: T? = null): Resource<T> {

return Resource(Status.LOADING, data, null)

}

}

}

Ресурс.уз

Это помогает нам инкапсулировать наши ответы репозитория в соответствии с их состоянием, что облегчает нашим представлениям отображение соответствующей информации. Именно так CharactersFragment наблюдает за значением LiveData и соответствующим образом обновляет себя.

viewModel.characters.observe(viewLifecycleOwner, Observer {

when (it.status) {

Resource.Status.SUCCESS -> {

progress\_bar.visibility = View.GONE

if (!it.data.isNullOrEmpty()) adapter.setItems(ArrayList(it.data))

}

Resource.Status.ERROR ->

Toast.makeText(activity, it.message, Toast.LENGTH\_SHORT).show()

Resource.Status.LOADING ->

progress\_bar.visibility = View.VISIBLE

}

})

Фрагментация символов.kt (символы наблюдателя)

У нас есть свой презентационный слой. Как я могу получить свои данные?

# Retrofit and remote data.

Retrofit помогает нам получить доступ к удаленным данным из нашего приложения. Вы можете увидеть, как построить объект Retrofit в **AppModule.КТ**. Нам также необходимо кодировать интерфейсы с операциями HTTP, которые мы будем выполнять для связи с API.

interface CharacterService {

@GET("character")

suspend fun getAllCharacters() : Response<CharacterList>

@GET("character/{id}")

suspend fun getCharacter(@Path("id") id: Int): Response<Character>

}

CharacterService.уз

Модификатор **suspend** приходит к нам от сопрограмм Kotlin, он указывает, что следующая функция будет выполняться в сопрограмме (аналогично потоку), позволяя нам держать поток пользовательского интерфейса разблокированным во время выполнения длительных операций, таких как получение наших данных из интернета.

Этот CharacterService используется CharacterRemoteDataSource

class CharacterRemoteDataSource @Inject constructor(

private val characterService: CharacterService

): BaseDataSource() {

suspend fun getCharacters() = getResult { characterService.getAllCharacters() }

suspend fun getCharacter(id: Int) = getResult { characterService.getCharacter(id) }

}

CharacterRemoteDataSource.уз

Зачем мы это делаем? Справедливый вопрос. **getResult**инкапсулирует ответ Retrofit в **ресурс**, так что мы можем ловить ошибки красиво.

abstract class BaseDataSource {

protected suspend fun <T> getResult(call: suspend () -> Response<T>): Resource<T> {

try {

val response = call()

if (response.isSuccessful) {

val body = response.body()

if (body != null) return Resource.success(body)

}

return error(" ${response.code()} ${response.message()}")

} catch (e: Exception) {

return error(e.message ?: e.toString())

}

}

private fun <T> error(message: String): Resource<T> {

Log.e("remoteDataSource", message)

return Resource.error("Network call has failed for a following reason: $message")

}

}

BaseDataSource.уз

# Помещение и локальное хранилище данных

С другой стороны, мы также заинтересованы в хранении и получении данных из локальной базы данных. Нам это нужно для того, чтобы показать хотя бы что-то нашим пользователям, когда они находятся вне связи. Если вы знакомы с этой концепцией, мы собираемся использовать ее в качестве кэша для нашей системы.

Комната приносит нам приятный интерфейс для работы с локальными базами данных. Вот мы его и построим:

@Database(entities = [Character::class], version = 1, exportSchema = false)

abstract class AppDatabase : RoomDatabase() {

abstract fun characterDao(): CharacterDao

companion object {

@Volatile private var instance: AppDatabase? = null

fun getDatabase(context: Context): AppDatabase =

instance ?: synchronized(this) { instance ?: buildDatabase(context).also { instance = it } }

private fun buildDatabase(appContext: Context) =

Room.databaseBuilder(appContext, AppDatabase::class.java, "characters")

.fallbackToDestructiveMigration()

.build()

}

}

AppDatabase.уз

Не забудьте включить все объекты, которые вы хотите сохранить в аннотации @Database. Для того, чтобы комната могла видеть наши сущности, нам нужно будет также аннотировать их.

@Entity(tableName = "characters")

data class Character(

val created: String,

val gender: String,

@PrimaryKey

val id: Int,

val image: String,

val name: String,

val species: String,

val status: String,

val type: String,

val url: String

)

Характер.уз

Наконец, нам нужно указать операцию, которую мы будем выполнять с базой данных. Вот тут-то и включаются наши DAOs (объект доступа к данным)

@Dao

interface CharacterDao {

@Query("SELECT \* FROM characters")

fun getAllCharacters() : LiveData<List<Character>>

@Query("SELECT \* FROM characters WHERE id = :id")

fun getCharacter(id: Int): LiveData<Character>

@Insert(onConflict = OnConflictStrategy.REPLACE)

suspend fun insertAll(characters: List<Character>)

@Insert(onConflict = OnConflictStrategy.REPLACE)

suspend fun insert(character: Character)

}

Характер такой.уз

Обратите внимание, что Room поддерживает LiveData, поэтому вы можете получить наблюдаемые значения из базы данных. А Kotlin Coroutines делает это еще проще, позволяя нам приостановить наши функции комнаты.

# Репозиторий и наша стратегия кэширования

Давайте объединим оба наших источника данных. Это время, когда нам нужно планировать стратегию доступа к данным. Подумайте, например, в процессе получения персонажа.

* Сначала нам нужно сообщить нашим LiveData, что мы ищем персонажа, чтобы он имел состояние загрузки.
* Затем мы хотели бы получить этот символ из локального источника данных, потому что это быстрее, чем получить его из интернета. Если он найдет его, мы изменим состояние на успешное
* Независимо от результата работы локальной базы данных, мы хотели бы сохранить наше приложение синхронизированным, поэтому мы также извлекаем этот символ из интернета (но помните, что поток пользовательского интерфейса не будет заблокирован, и пользователю уже может быть показана правильная информация о символе из-за того, что он находится в базе данных!).
* Наконец, нам нужно сохранить наш результат от удаленного вызова в базе данных, чтобы он постоянно обновлялся.

Все это мы будем делать в нашем репозитории:

class CharacterRepository @Inject constructor(

private val remoteDataSource: CharacterRemoteDataSource,

private val localDataSource: CharacterDao

) {

fun getCharacter(id: Int) = performGetOperation(

databaseQuery = { localDataSource.getCharacter(id) },

networkCall = { remoteDataSource.getCharacter(id) },

saveCallResult = { localDataSource.insert(it) }

)

fun getCharacters() = performGetOperation(

databaseQuery = { localDataSource.getAllCharacters() },

networkCall = { remoteDataSource.getCharacters() },

saveCallResult = { localDataSource.insertAll(it.results) }

)

}

Характер восстановительный.уз

Он **performGetOperation**определяется в соответствии со стратегией, которую мы обсуждали:

fun <T, A> performGetOperation(databaseQuery: () -> LiveData<T>,

networkCall: suspend () -> Resource<A>,

saveCallResult: suspend (A) -> Unit): LiveData<Resource<T>> =

liveData(Dispatchers.IO) {

emit(Resource.loading())

val source = databaseQuery.invoke().map { Resource.success(it) }

emitSource(source)

val responseStatus = networkCall.invoke()

if (responseStatus.status == SUCCESS) {

saveCallResult(responseStatus.data!!)

} else if (responseStatus.status == ERROR) {

emit(Resource.error(responseStatus.message!!))

emitSource(source)

}

}

DataAccessStrategy.уз

Я хочу выделить первую строку, где мы это делаем **liveData(Dispatchers.IO)**

Благодаря Сопрограммам Kotlin и LiveData, когда мы это делаем, мы запускаем новую сопрограмму ввода-вывода, поэтому позволяя нам использовать наши функции приостановки, а также храним наши результаты в держателе LiveData, который наблюдается ViewModel.

И это все!

# дальнейшие меры

Мы дошли до конца. Надеюсь, я ясно выразился на протяжении всего урока.

У нас есть некоторые TODOs, которые я опустил, чтобы держать вещи короткими:

* Мы все время синхронизируемся со стратегией, которую я представил, и ваше приложение может в этом не нуждаться. Помните о своих потребностях и разработайте разумную стратегию.
* Кроме того, мы выполняем только операции get. Что делать, если нам нужно загрузить что-то на сервер? В этом случае стратегия также будет иной.
* API разбивает свои символы на страницы. А почему бы и нет? Взгляните на библиотеку подкачки с помощью Jetpack.
* Тестирование еще предстоит провести.

Это все. Помните, что вы можете найти весь код, который я использовал в репозитории GitHub

## [sberoch/RickAndMorty-AndroidArchitectureSample](https://github.com/sberoch/RickAndMorty-AndroidArchitectureSample" \t "_blank)

### [Простое приложение Rick And Morty, чтобы показать один подход к использованию некоторых лучших практик в разработке Android. …](https://github.com/sberoch/RickAndMorty-AndroidArchitectureSample" \t "_blank)

#### [github.com](https://github.com/sberoch/RickAndMorty-AndroidArchitectureSample" \t "_blank)

И если что-то из этого вам пригодится, **хлопайте**в ладоши ! Я был бы очень благодарен и мотивирован для следующего урока.