Aplicação ChatApp

# Descrição da Arquitetura

Para a realização desta app optei usar um pattern architecture pelo simples facto de organização de código e aproveitei também para finalmente aplicar os meus conhecimentos neste tema.

A arquitetura usada nesta aplicação foi a recomendada pela Google, ou seja a MVVM (Model View View-Model). A opção recaiu sobre esta arquitetura porque não tinha conhecimento suficiente sobre a MVC ou a MVP para poder desenvolver uma app. No entanto, mesmo se tivesse, acho que acabaria sempre por escolher a MVVM.

As três têm em comum o modulo de Model (que representa as interações com a rede, base de dados, etc) e o de View (que representa a parte gráfica). Os módulos C(controller) para MVC, P(Presenter) para MVP e VM(View Model) para MVVM trabalham com o Model para obter os dados necessários para apresentar na parte gráfica da app, conhecida como View.

Explicação da MVC

Explicação da MVP

Explicação da MVVM

# Descrição da estrutura do projecto

Eu dividi o projeto por classes tal como é aconselhado pela Google.

As pastas são as seguintes:

**ui** – onde coloquei todas as classes que dizem respeito à parte visual e gráfica. Também aqui está o ViewModel.

**api** – onde coloquei as chamadas à rede via Retrofit2.

**preferences** – onde coloquei todas as ações relacionadas com as shared preferences.

**repository** – repositório que é o que vai fazer a comunicação entre o ViewModel e o modulo Model (shared prefences, network e database).

**di** – aqui coloquei as classes relacionadas com a dependency injection.

**db** – tudo o que esta relacionado com a base de dados Room que foi a que foi utilizada neste projeto.

**utils** – pasta com classes utils para o porjecto

**vo** – a chamada pasta value objets usadas para as classes POJO usadas para o JSON.

Dividindo o projeto desta forma eu garanti que a parte View estivesse associada à pasta ui, que a parte do ViewModel estivesse associada às partes repository e di e a parte Model com db, preferences, api, vo.

# Livrarias usadas

As livrarias usadas durante a elaboração deste projeto foram as seguintes:

* Dragger2 (dependency injection)
* Glide (download de imagens)
* Retrofit2 (chamadas à rede)
* Gson (tratamento de dados JSON)

# Como pensei para desenvolver a app

# Classes usadas

Em termos de base de dados criei duas tabelas, Messages e User que se ligavam entre elas através do userId. Dentro da pasta db encontram-se os ficheiros que descrevo de seguida.

## Pasta db

### Message.java

Tabela Message tinha os seguintes campos:

* Long id
* String userId
* String content
* String attachments

### User.java

Tabel User tinha os seguintes

* Long id
* String name
* String avatarId

### AppDatabase.java

Descrição da Base de dados, onde especifico as tabelas, o tipo de base de dados que vou usar(RoomDatabase), onde indico o nome do ficheiro DAO e onde também refiro que vou usar TypeConverters.

### AppDbChatAppHelper.java

Classe onde “chamo” os procedimentos da minha classe DAO.

### Converter.java

Classe que contem os métodos para converter de ArrayList para String e vice versa e depois acabei por não usar (meti diretamente no código)

### DBChatAppHelper.java

Interface com a declaração dos métodos relativos à base de dados.

### DbsInfo

Classe usada para a Dependency Injection por parte da base de dados.

Classe onde é defenida a RetencionPolicy. (Existem três tipos possíveis de RetencionPolicy: Class, RunTime e Source. Por defeito é Classe. Neste caso usou-se o valor Runtime para referir que as anotações vão estar disponíveis durante o tempo de execução. Se fosse class queria dizer que gravaria as anotações no ficheiro .class, se fosse Source queria dizer que as anotações marcadas não estarão no código binário).

### ChatAppDAO

Classe que tem as query que se vão fazer à base de dados.

## Pasta api

### MainApiHelper.java

Interface com a chamada à rede utilizando Retrofit2

### ApiResponse.java

Classe para tratar das respostas devolvidas pelas chamadas feitas à rede. Classe tirada da net.

## Pasta di

Esta pasta é composta por 3 pastas: Componente, Module e Adapter.

Dentro da Componente temos a classe

### MainComponent.java

Interface. Esta classe é usada pela Dragger2 para gerar o código que utiliza os módulos que requerem dependência.

Dentro da pasta Module temos a classe

### ActivityModule.java

Dependency Injection

### AppModule.java

Usado para a dependency injection e diz q providencia uma instancia de cada classe para a dependecy injection.

### NetModule.java

Constrói a chamada à rede através do Retrofit2.

Dentro da paste Adapter temos as classes

## Pasta Preferences

Esta pasta tem todas as classes e métodos relacionados com as shared preferences.

### [AppPreferencesHelper.java](https://github.com/porthfind/ChatApp/blob/master/app/src/main/java/com/rdev/chatapp/preferences/AppPreferencesHelper.java)

Esta classe é constituída por dois métodos que o que fazer é ver se é a primeira vez que está a executar-se a app (isFirstRun()) e outro que o que faz é colocar o valor a TRUE quando é a primeira vez que se executa (setFirstRun()).

Utilizo o @Inject no construtor para injetar uma dependência.

### [PreferencesHelper.java](https://github.com/porthfind/ChatApp/blob/master/app/src/main/java/com/rdev/chatapp/preferences/PreferencesHelper.java)

Interface que contêm os métodos descritos na AppPreferencesHelper.java

### [PreferencesInfo.java](https://github.com/porthfind/ChatApp/blob/master/app/src/main/java/com/rdev/chatapp/preferences/PreferencesInfo.java)

Mesma coisa que a DbsInfo.

## Pasta ui

Esta pasta contém duas pastas base e main.

A pasta **Base** é constituída por duas classes:

### BaseActivity.java

Classe de apoio que tem métodos auxiliares e de DataBinding. Para além do DataBinding tem métodos de pedir permissão ao user e de esconder o teclado.

### BaseViewModel.java

Classe que não usada. No inicio criei-a a pensar que a usava e posteriormente vi que não a usava. Tem métodos de acesso ao repositório.

A pasta **Main** é constituída pelas seguintes classes:

### MainActivity.java

Nesta classe começo por ter um observer para ver se é a primeira vez que estou a entrar na app. Esse observer o que está a fazer é chamar o método que esta no ViewModel para ver se é a primeira vez que se executa a app.

Esta classe tem também outro observer para ir obter os dados da Base de Dados e para depois mostrar no ecrã. Os dados desse procedimento virão no formato CardViewItem que é um tipo de objeto criado que falarei mais adiante.

### MainActivityModule.java

Esta classe é usada para a dependency Injection e nela estou a dizer que existe uma dependência com o repositório através do @Provides e do @Module.

### MainNavigator.java

Interface.

A classe BaseViewModel é declarada como BaseViewModel<T> onde o T é a MainNavigator. Podia ser dispensada.

### MainViewModel.java

Classe que representa o ViewModel da MainActivity. Nesta classe tenho a chamada ao repositório dos métodos para ver se é a primeira vez que estamos a executar a app (firstTime()). Caso o firstTime() me devolver um true, quer dizer que é a primeira vez que estou a executar o método então vou chamar o repositório com o método saveValues() que vai guardar na base de dados os valores lidos da rede via Retrofit2.

O método isFristRun vai sempre devolver um true ou um false, o que acontece é que eu tinha de ter um else no código porque senão dava-me erro, então para tratar deste erro o que fiz foi “inventar” um procedimento, getCount(), apenas e só para ter um else e não ter erro. Antes tinha um return null, mas dava erro.

**public** LiveData<Boolean> firstTime(){  
 **if**(**mDataManager**.isFirstRun())  
 {  
 Timber.*d*(**"-------------- It is first time reading from database "**);  
 **return mDataManager**.saveValues();  
 }  
 **else** {  
 Timber.*d*(**"-------------- It is not first time reading from database "**);  
 **return mDataManager**.getCount();  
 }  
}

Nesta classe está também o método getValuesFromDb() que depois destas validações de ser o não ser a primeira vez que se executa vai buscar à base de dados as mensagens e os users. Depois de ter programado tudo e de ter chegado a este ponto dei-me conta que

**public** LiveData<List<CardViewItem>> getValuesFromDb()  
{  
 **user** = AbsentLiveData.*create*();  
 **user** = **mDataManager**.findAllUsers();  
 **mCardViewItem** = AbsentLiveData.*create*();  
 **mCardViewItem** = **mDataManager**.findAllMessages();*//get20Message* **return mCardViewItem**;  
}

### MessagesAdapter.java

Esta classe é usada para mostrar as mensagens no ecrã, faz um extends da RecyclerView. Para poder mostrar as mensagens na RecyclerView usei **android.support.v7.widget.CardView.**

Nesta classe, para mostrar as mensagens com diferentes layouts utilizamos o mesmo layout de mensagens para todas as mensagens (mensage\_list\_item.xml) e depois o que fiz foi “jogar” com a propriedade Visibility.

## Pasta vo

Pasta usada para tratar dos dados JSON. Com o URL fomos a um JSON Parser para nos criar as classes (POJO) que iriamos precisar para tratar esses dados. O link que usei foi:

<http://www.jsonschema2pojo.org/>

Com isto foram criadas as classes Attachment, Conversation, Messages e User.

A classe CardViewItem foi criada posteriormente para poder apresentar no ecrã os elementos que se obtêm da base de dados.

## Pasta utils

Pasta com classes úteis para o desenvolvimento do projecto.

Essas classes são:

### ViewModelProviderFactory.java

Esta classe é usada para criar o ViewModel e faz uso do Provider Factory que sabe exatamente como criar View Models. Desta maneira a criação de ViewModels transfere-se para uma classe separada que a nível de código é melhor.

### AppConstants.java

Classe usada para as constantes usadas no projecto.

### AbsentLiveData.java

Classe para inicializar o tipo de dados AbsentLive Data.

## Pasta repository

Esta pasta é onde estão os ficheiros que têm a lógica para comunicar com as shared preferences, a network e a base de dados.

### AppDataManager.java

É desde esta classe que se faz a chamada à Shared Preferences pelos métodos isFirstRun() e setFirstRun() que se faz a chamada à rede através do getConversation().

O método saveValues é utilizado para gravar na Base de Dados os valores obtidos na chamada à rede e utiliza uma classe de Java chamada Transformations para alterar o valor do objeto LiveData obtido nessa chamada à rede.

Para isso utiliza um tipo de valor MediatorLiveData que é uma subclasse do LiveData.

Nesta classe também está o método para ir buscar os dados à base de dados e transforma-los em objetos do tipo CardViewItem, utiliza tb a classe Transformations.

### DataManager.java

Interface

-Chamadas à rede

Descodificação de JSON

Inserção na bd

Leitura de bd

Mostrar no ecrã

Como trabalhei com as preferences

* Coisas que teria mudado se a tivesse feito agora
* MVVM – conceitos importantes e que é necessário fazer