An evaluation of modern Android Libraries and Frameworks

Author: Josep Rodríguez López
joseprl89@uoc.edu

TFM-Desenvolupament d'Aplicacions en Dispositius Mòbils
Consultants: Ignasi Lorente Puchades & Jordi Almirall López

Being the final paper submitted to the Open University of Catalonia, in partial fulfillment of the requirements for the award of M.Eng. degree in Computer Sciences.

January 2016

Acknowledgements

Firstly, I would like to express my sincere gratitude to my consultants Ignasi and Jordi for the support and guidance of this thesis.

My sincere thanks also goes to all my colleagues and seniors at work who allowed me to learn as much as I've been able to during my professional career.

Last but not the least, I would like to thank my family and friends for supporting me throughout writing this thesis and in my life in general.

Table of Contents

Abstract

"Don't reinvent the wheel", the old computer science saying goes. It is probably one of the most quoted sentences that students all around the world have to go through. And it is in most cases also correct, since we don't start all our projects by writing a new kernel.

However, committing to use something someone else has created has a few drawbacks. First of all, since the moment it becomes part of your project, you become responsible for it. If it breaks, you can't shift the blame, and will have to provide a fix on a piece of code you have not created. If the library is poorly maintained, updates to the ecosystem can easily break it and render it useless. What if the framework you imported forces you to change the way you code and adapt to it, and it eventually becomes obsolete?

Stepping now into a more specific sector, The mobile app ecosystem is in an ever evolving state, with new and better frameworks appearing frequently to cover gaps and improve features of an ecosystem that has appeared relatively recently (Android initial release was in 2007, only 8 years ago). This forces as a side effect the need for a mobile application developer to always keep his toolbox updated with the libraries more appropriate for the project he's working on.

With this context, relying on 3rd party libraries and frameworks feels like a necessity, however, selecting the appropriate libraries is key to the long term maintainability of your code.

This dissertation will make an evaluation of some of the libraries that have caught the attention of the author since they do a great job at improving the Android SDK or help the development of features uncovered by either the SDK or the Java JDK. The goal of it is to identify libraries in the Android ecosystem that will allow us to leverage them to better our code and the application we finally ship to the users.

Workplan

Context and value of the project

As expressed before, app development is a fast paced environment that requires constant learning of new techniques and tools not to fall behind it's new features and 3rd party libraries that ease the process of building a great app.

Lately, Android has gone through a major UX migration to the new "Material Design"[1], a plethora of new libraries appear continuously to help tackle several different shortcomings of the Android SDK and a major Andorid SDK version has been released.

This dissertation aims to build an app that uses aspects of Material design leveraging some of the most relevant Android libraries, with the aim to provide an engaging and delightful user experience.

Objectives

The main purpose of this work is to produce an evaluation of the used libraries to be considered on future development. The main goals are:

- Evaluate pros and cons of Dagger2[2], Mosby[3], RxAndroid[4], Realm[5], the Android support design libraries[6], Butterknife[7], Retrofit[8] and Glide[9], if it makes sense to use them during the development of the product.
- Produce a usable music app.

[1] "Introduction - Material design - Google design guidelines." 2014. 20 Sep. 2015 <https://www.google.com/design/spec/material-design/introduction.html>
[2] "Dagger 2 - Google." 2014. 20 Sep. 2015 <http://google.github.io/dagger/>
[3] "sockeqwe/mosby · GitHub." 2015. 20 Sep. 2015 <https://github.com/sockeqwe/mosby>
[4] "ReactiveX/RxAndroid · GitHub." 2014. 20 Sep. 2015 <https://github.com/ReactiveX/RxAndroid>
[5] "Realm is a mobile database: a replacement for SQLite ..." 2015. 20 Sep. 2015 <https://realm.io/>
[6] "Android Design Support Library | Android Developers Blog." 2015. 12 Dec. 2015 <http://android-developers.blogspot.com/2015/05/android-design-support-library.html>
[7] "Butter Knife." 2013. 12 Dec. 2015 <http://jakewharton.github.io/butterknife/>
[8] "Retrofit - Square." 2013. 12 Dec. 2015 <http://square.github.io/retrofit/>
[9] "bumptech/glide · GitHub." 2013. 26 Sep. 2015 <https://github.com/bumptech/glide>

Since the main goal of this work is educational, the most sensible approach is getting our hands dirty by starting an app from scratch, and perform the evaluation of the aforementioned libraries in an app development that will mimic a real world development.
Other approaches considered to perform this task were:

- Static analysis: Study the libraries in isolation. This would result in a lack of understanding of the inconveniences that can be found during the development of the app.
- Boilerplate code per library: Create a simple app for each of the libraries that we intend to evaluate. The drawback of this option is that keeping the apps simple might not expose some of the drawbacks you might encounter in a more extensive development.

Deliverables

The output of this work will be a functional music app that leverages a set of libraries that are trendy amongst the Android developer community at the moment. Furthermore, an analysis of benefits and drawbacks of each of the used libraries will be performed to share the insight obtained from the work.

Workplan

This project will be approached in an Agile manner, by using the Kanban[10] methodology. The tasks will be formulated in a user story[11] manner, and prioritised in a backlog.

The time commitment on the development of the app will be 0.5 hours per weekday, and 2 hours per weekend day. After the 14 week timeframe working on this project, this will add up to 35 hours during weekdays and 56 weekend hours.

In the following section the backlog is introduced, assigning ID's to each task. This section will refer to those ID's to specify the tasks that will be included in each milestone.

Milestones

Milestone 1 - 30/09/2015 - Work plan

Milestone 1 includes an initial plan of the work to be done and an initial setup of the project in Android.

Milestone 2 - 28/10/2015 - Analysis, design and prototype

The goal of this milestone is to deliver a navigational prototype, the app architecture and a functional analysis of the app. The scope of working stories for this milestone will be 1,2,3,4,5,6. As a result, a user should be able to, apart from navigating through the app, use a basic functionality of the app, kind of in a Minimum Viable Product (MVP) sense.

Milestone 3 - 15/12/2015 - Working app

The delivery of this milestone consists of the working app and the source code created for it. A video of the app usage will also be included in the release.

Milestone 4 - 08/01/2016 - Final delivery

The complete dissertation is delivered.

[10] "Kanban - Wikipedia, the free encyclopedia." 2011. 20 Sep. 2015 <https://en.wikipedia.org/wiki/Kanban>
[11] "User Stories: An Agile Introduction - Agile Modeling." 2003. 20 Sep. 2015 <http://www.agilemodeling.com/artifacts/userStory.htm>

Backlog

The following table enumerates the user stories and provides an identifier for each of them.

Id	Title	User story
1	List available songs	AS A User I WANT TO see all my songs SO THAT I know what songs are available to me
2	Listen to a song	AS A User I WANT TO listen to a song SO THAT I can enjoy music
3	Listen to all songs	AS A User I WANT TO listen to all my songs SO THAT I don’t need to play a song at a time
4	Create a playlist	AS A User I WANT TO Create a new playlist SO THAT I can create custom lists of songs
5	Add a song to a playlist	AS A User I WANT TO Add a song to a given playlist SO THAT I can populate a playlist
6	View a playlist	AS A User I WANT TO view the songs in a playlist SO THAT I know which songs are in there
7	Remove a song from a playlist	AS A User I WANT TO Remove a song from a playlist SO THAT I can undo adding a song by mistake
8	Delete a playlist	AS A User I WANT TO Delete a playlist SO THAT I can get rid of playlists I no longer want
9	Listen to a playlist	AS A User I WANT TO Listen to a playlist SO THAT I can limit the songs I’ll be listening to
10	Shuffle playing order	AS A User I WANT TO Shuffle play SO THAT I don’t know what song will play next
11	Repeat all songs	AS A User I WANT TO Repeat all songs

		SO THAT I don't need to press play again
12	Repeat one song	AS A User I WANT TO repeat a single song SO THAT I can listen to my favourite song all day
13	Download a youtube video sound	AS A User I WANT TO download the sound of a music video SO THAT I can populate my music library
14	Favourite playlist	AS A User I WANT TO Have a playlist with my favourite songs SO THAT I can have a list with my favourite music

The initial goal is to implement all this stories, however, code quality and getting a good insight at the libraries will be preferred to scope breadth.

Current Project status

The work proposed upfront has been delivered, however, there was not enough time to perform automated testing of the app using Calabash.

To make up for it manual testing has been performed to cover the whole of the app's functionality. This would not be desired if we were to maintain the project, since perform the manual testing is costly if you have to do it in each iteration or sprint of your product.

Apart from that, as is reflected in the repo, all tasks have been completed successfully, and every usage of the libraries has been documented in the Readme.md file of the root of the repo.

Proper use of git-flow has been done and all releases (navigational prototype, a 0.2 intermediate release, and the 1.0.0 release) are tagged and their code is downloadable from the downloads section.

We have decided to not release the app nor it's code publicly due to the fact that we break Youtube's Terms and Conditions by extracting the sound from a music video. The code is currently hosted privately in Bitbucket in the following URL:

https://bitbucket.org/joseprl/tfm

Development methodology

Code versioning system approach

Git flow[12] will be used to assure that the master and develop branch contain stable code. Each user story will be developed in its own feature branch, which won't be merged into develop until the story is completely done. The definition of done described in the next subsection defines when a feature is seen as complete and can be merged to develop.

Architectural patterns

A Model-View-Presenter (MVP) design will be implemented by leveraging the Mosby library, being evaluated in this dissertation.

Definition of ready

The stories are assumed to be ready, all dependencies will need to be resolved during the analysis phase, since there are no other parties involved in the development of the app that could block us from progressing.

Definition of done

To complete a user story, the definition of done must be met. In this project, this will include:

- Pull request created in BitBucket that allows to visualize the changes performed in the branch.
- If a library is used, there will be documentation of what issues were encountered while doing the implementation, if any, and what benefits did it brought to the project.
- The story will be documented in the dissertation.

[12] "Gitflow Workflow | Atlassian Git Tutorial." 2014. 20 Sep. 2015 <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow/>

Acceptance criteria

The acceptance criteria will be produced during the analysis phase due to the lack of a Product Owner (PO).

Testing approach

Before starting the development, a test script will be written for the given user story using the gherkin[13] language, which is easy to read and to compose into a full test script for an end-to-end system testing.

Before performing any release an end-to-end (E2E) testing will be performed manually using the test script generated while developing the user stories. If a bug were to be detected, it would be addressed, a code impact analysis performed to decide whether the E2E test is required again or a smoke test would be enough, and then, another set of tests would be performed.

Test devices

The app will be tested using a Motorola Moto G (2nd generation) device running Android 5.0.2.

Repository

The repository of this work is hosted in a private Bitbucket repo in the following URL: https://bitbucket.org/joseprl/tfm

[13] "Gherkin · cucumber/cucumber Wiki · GitHub." 2011. 3 Jan. 2016 <https://github.com/cucumber/cucumber/wiki/Gherkin>

User Centered Design

Analysis

In this section, an analysis of the potential users and use contexts that the app will target is provided.

Since the main goal of this project is to evaluate Android libraries and code techniques, only a lightweight approach at UCD has been performed, sufficient to create a usable app, but without the required depth to be able to obtain conclusions to extrapolate what the opinion would be of a wide spectrum of users regarding the delivered app.

Users

A music mobile app will have as user the intersection of people who have a smartphone (in this case an Android smartphone), and like to listen to music.

Thus, a research for existing data regarding the user demographics for those two sets of user has been performed.

Who uses a smartphone

Thanks to data provided by Google[14] in the demographic section of the Our Mobile Planet reports, we can tell that the average smartphone user in Spain is:

- Any gender
- Young (50% of them are below 34 years old, and only 19% are above 45)
- City dweller
- Has higher studies
- Has a work

However, smartphones are so ubiquitous nowadays that some of the statistics offered might be very similar to the demographics of the country. The demographics for music listeners should have more weighting when deciding what our target user should be.

[14] http://services.google.com/fh/files/misc/omp-2013-es-local.pdf

Who loves music

According to UK's art council[15], there's two demographics highly engaged to music.

- Urban arts eclectic
  - Young
  - Affluent
  - Highly educated
  - Lives in a city area
  - Slightly skewed towards a minority ethnic
- Traditional culture vultures
  - Woman
  - Mature, with good health (majority over 45 years)
  - White ethnic
  - Highly educated
  - Rural areas
  - Affluent

Target user

As a mixture of the two previous studies we could conclude that the demographics most likely to use a music app would be the Urban arts eclectic, as reported by the UK's art council.

This demographic is quite similar to the one reported by Google as smartphone user, thus the biggest overlap will be found in there.

Use context

Once we have defined the target user we intend our app for, we can identify some of the contexts in which he'll be using our app.

Commuting

As defined in the demographics, our user is an affluent professional living in a city with higher studies.

[15] http://www.artscouncil.org.uk/media/uploads/Arts_audiences_insight.pdf

This probably means that he'll have to commute to his work on a daily basis. In a city like London, the average commute[16] is measured to be 75 minutes.

On top of that, the London underground doesn't provide any kind of data connection on smartphones. This means that a very important use case will be listening to music offline while commuting.

Chilling at home

Another important use case of music in general is relaxing with it at home. After a 75 minutes commute, it's probably time to do so. Also, it's possible the user wants to cache some new songs to listen to offline.

Task analysis

In this section, the main tasks that the user will need to achieve his goals are listed.

Playing music offline

The user must be able to play music even when offline.

Downloading content

The user must be able to easily download more content to keep cached so he can later play it offline.

Managing playlists

In order to manage his music library, the user must be able to create, modify and delete playlists.

16 http://www.dailymail.co.uk/news/article-2232243/London-commute-Workers-spend-75-minutes-day-getting-work-worse-women.html

Design

Use scenarios

As explained before, the demographics of the users will probably generate two major use scenarios.

Commuting

In this scenario, the demographic we selected as user will be heading to work/back home for an average of 75 minutes, most of that time being out of connectivity.

His goal is to listen to music to disconnect from the tedious trip to work.

The task that will be performed by the user is playing music offline.

The information he needs is what playlists he can listen to, what's currently playing, and what song is playing at the moment.

The user should achieve the goals he has in this scenario in a "Fire and forget" approach. He'll probably start playing some songs, move the app to the background and start reading news, answering mails or other tasks he performs on his commute.

Chilling at home

In this case, the user will be just relaxing at home.

His goal is to listen to music to relax for a while, and maybe add songs into his music library.

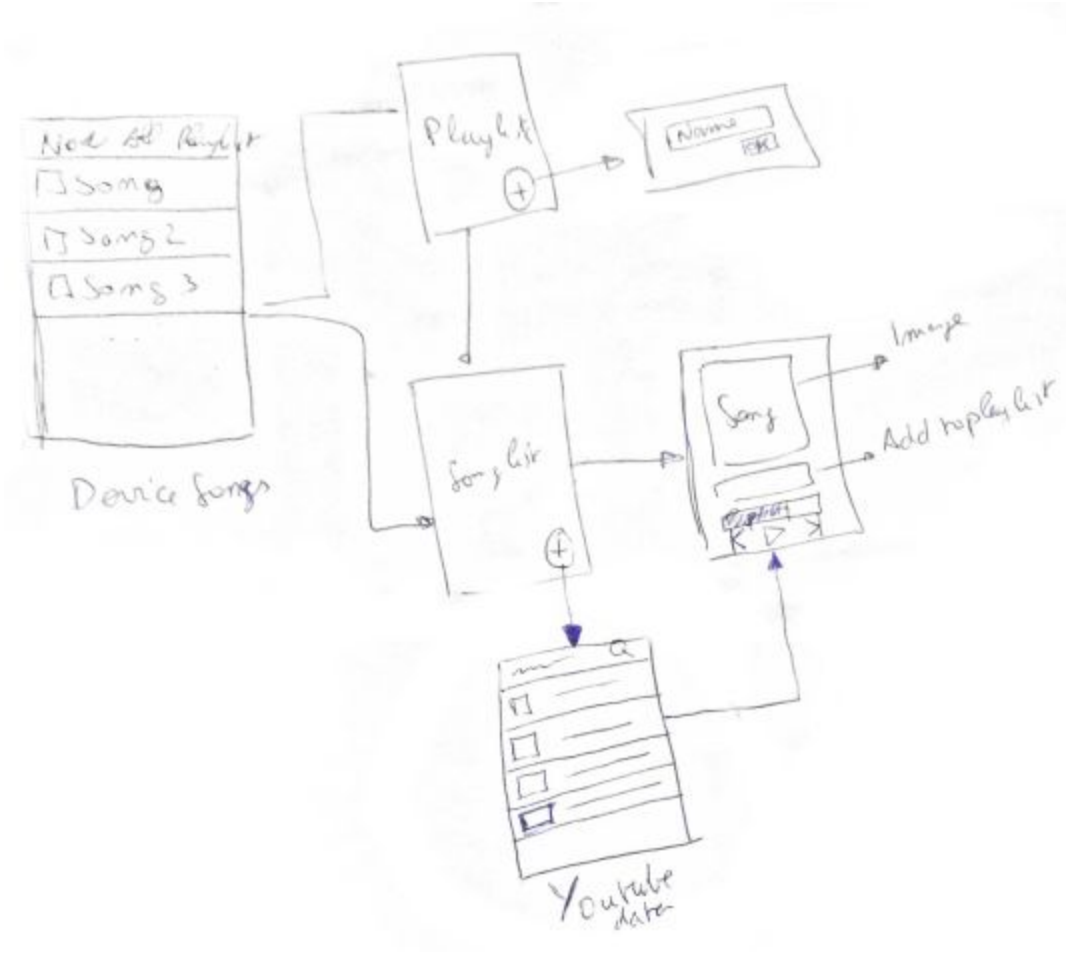
The task in this case is again to listen to music, but also managing playlists and downloading content.

The information he needs in this scenario is the same as before, but also, the content he has available to download.

As before, a “Fire and forget” approach will be used for the listening task. However, the other two tasks will require a more interactive approach and higher time spent on the app.

Navigational prototype

The first approach at creating a sketch of the app was this app map:



The initial idea for the app was a fairly simple one with the following screens:

- Main activity with 3 tabs, Now playing, All music, and Playlists.
- A Song activity, in which the user can start playing it, move to the next or previous one.
- A search song activity. Once the user tries to add a song, a search activity should appear allowing to obtain songs from the internet.
- Playlist creation. A simple dialog will suffice since it's only setting the name, the population of songs can be done via the Song activity.

Instead of creating a low level design of the app, we opted for creating a navigational prototype using the Android SDK.

The APK can be downloaded from:
https://bitbucket.org/joseprl/tfm/downloads

However, the prototype evolved and the final app has an extra tab and a new screen amongst other changes to the app sitemap.

Evaluation

In order to evaluate the result of the design, we'd run a batch of user testing on the navigational prototype we just created.

Background information form

The questions asked to the users would be:

- Do you regularly listen to music?
- Do you regularly use smartphone apps?
- How many apps have you installed on your phone?
- Do you have a favourite band?
- Do you listen to the radio?
- How much do you value having offline accessibility to music?

User testing tasks

The tasks the user would have to accomplish to measure the design quality would be:

- Listening to a song
- Listening to a playlist
- Creating a playlist
- Adding songs to a playlist
- Removing songs from a playlist
- Getting new songs

Post-test form

Once the user has performed the aforementioned tasks, the following form would need to be filled in order to collect feedback from the user.

- How do you rate the UI of the app?

- Did you have any problem while navigating it?
- Did you miss any functionality?
- Was there any interaction that you found to be tedious?
- In case you failed to accomplish a task, what impeded you? What would you expect to change in order to solve it?

Technical Architecture

Use cases

Instead of defining the use cases via UML, a user story approach has been deemed more appropriate, since it will allow a more agile approach in this project.

The user stories are reported in BitBucket's bug tracker, and can be filtered using the following query:
https://bitbucket.org/joseprl/tfm/issues?kind=task&sort=version

Architecture design

There's two main designs to describe for this project. The first one is the model layer, which will be covered in a UML diagram later on, the other one is the *Model-View-Presenter* (MVP) approach used to separate View logic from Business and Model logic.

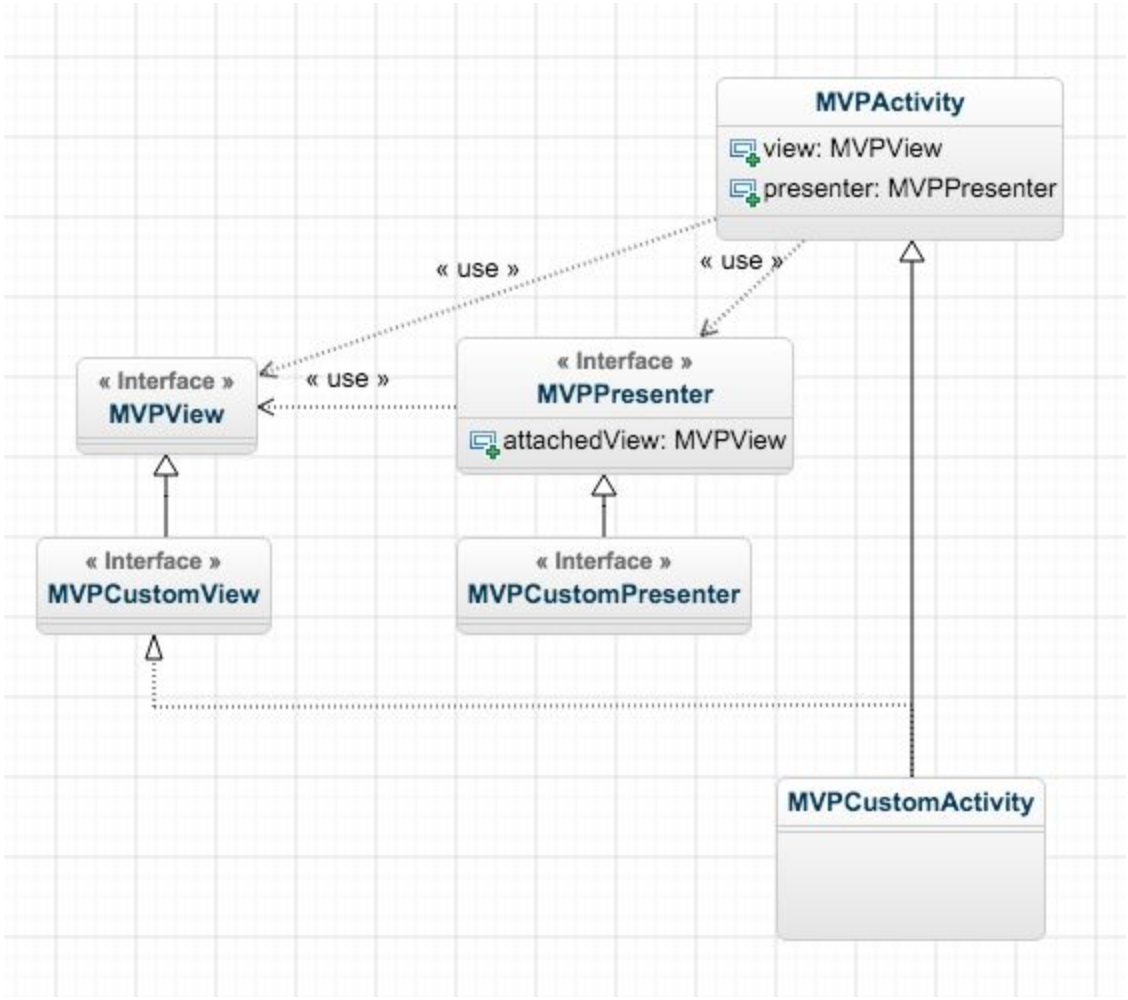
Model-View-Presenter (MVP)

One of the libraries that will be evaluated in this project is a library to ensure MVP approach is followed throughout your Android activities and fragments. Their website[17] defines its goal as:

> *The aim of this library is to help you build modern android apps with a clean Model-View-Presenter architecture. Furthermore, Mosby helps you to handle screen orientation changes by introducing ViewState and retaining Presenters.*

The library proposed architecture model is represented in the following UML diagram.

[17] http://hannesdorfmann.com/mosby/



In this diagram, the classes *MVPView*, *MVPPresenter* and *MVPActivity* (and *MVPFragment*) are provided by the library (among some helpful interfaces extending them). The usages in the diagram represent how generics tie the interfaces together.

An *MVPPresenter* keeps an *attachedView* to it, which is any interface extending *MVPView*. On the other hand an *MVPActivity* has references to interfaces overriding both *MVPView* and *MVPPresenter*.

The classes including Custom are basically a naming example, instead of Custom we'll be using SongList, or similar names instead.

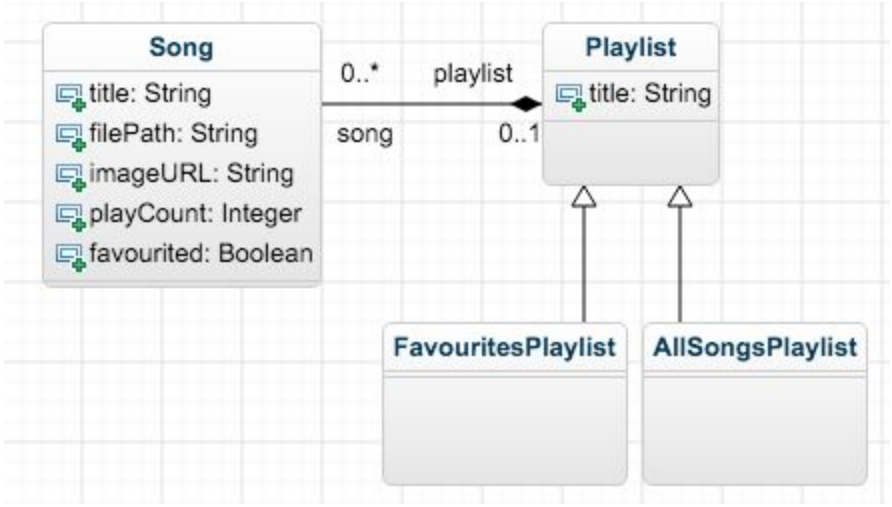
This architecture, when properly implemented, will ensure that all presenter code will be in a class implementing *MVPPresenter*, the view logic can be extracted easily from the

activity into any class implementing *MVPView*, thus leaving the activity with only the usual boilerplate code to transition to other activities and such (it can be argued if the activity should be implementing *MVPView*, since it will have to be notified somehow in order to perform the transitions).

Model

The model of the app is actually quite simple. There are only two entities, the Song and the Playlist. The playlist is basically a list of songs, with some metadata and convenience methods to help playing, shuffling the list and so on and so forth. There will also be convenience classes to represent the favourites playlist and the playlist for all the songs in the device.

The following diagram represents it:



APIs

The only task that involved an API for the app was the download of songs from Youtube. In order to do so, we required to implement clients for two different API endpoints, Youtube, and Youtube-mp3.org.

Youtube data API

The Youtube data API provides metadata of videos stored in Youtube and allows to perform several different kinds of queries, including searching by keywords.

In our case, the only requirement we had is to obtain the URL of the youtube video the user wants to get the music from, and later extract the audio out of it.

Notice that this contravenes the Terms & Conditions of Youtube, and because of it the code of this app won't be made publicly available and will remain just an educational research.

Regarding the implementation of the client, we used Google's client as detailed in:

https://developers.google.com/youtube/v3/?hl=en

The code that performs the query is the following:

```
1. /**
2. * Performs a search query on youtube.
3. *
4. * @param query The query to perform on Youtube
5. */
6. public static List<YoutubeVideo> searchYoutubeAPI(String query)
   throws IOException {
7.   // This object is used to make YouTube Data API requests. The last
8.   // argument is required, but since we don't need anything
```

```
9.  // initialized when the HttpRequest is initialized, we override
10. // the interface and provide a no-op function.
11. youtube = new YouTube.Builder(new NetHttpTransport(), new
    JacksonFactory(), new HttpRequestInitializer() {
12.     public void initialize(HttpRequest request) throws IOException {
13.     }
14. }).setApplicationName("TFM").build();
15.
16. // Define the API request for retrieving search results.
17. YouTube.Search.List search = youtube.search().list("id,snippet");
18.
19. // Set your developer key from the Google Developers Console for
20. // non-authenticated requests. See:
21. // https://console.developers.google.com/
22. search.setKey(API_KEY);
23. search.setQ(query);
24.
25. // Restrict the search results to only include videos. See:
26. // https://developers.google.com/youtube/v3/docs/search/list#type
27. search.setType("video");
28.
29. // To increase efficiency, only retrieve the fields that the
30. // application uses.
31.
    search.setFields("items(id/kind,id/videoId,snippet/title,snippet/thu
    mbnails/default/url)");
32. search.setMaxResults(NUMBER_OF_VIDEOS_RETURNED);
```

In the lines 11 to 32, we are preparing the request to the Youtube API with the query the user entered.

```
33.  // Call the API and print results.
34.  SearchListResponse searchResponse = search.execute();
35.
36.  List<SearchResult> searchResultList = searchResponse.getItems();
```

Lines 33 to 36 fetch the response Youtube yields

```
37.  // Convert it to our own format so we can implement required interfaces
     for the UI
38.  List<YoutubeVideo> videos = new
     ArrayList<>(searchResultList.size());
39.  for (SearchResult result : searchResultList ) {
40.      YoutubeVideo video = new YoutubeVideo();
41.      video.setTitle(result.getSnippet().getTitle());
42.
     video.setImageURL(result.getSnippet().getThumbnails().getDefault().ge
     tUrl());
43.      video.setVideoId(result.getId().getVideoId());
44.      videos.add(video);
45.  }
46.
47.  return videos;
48.}
```

From lines 40 to 48 we map the Youtube data structures to custom ones coded by us so we obtain more flexibility, and finally return it to the caller.

The model classes are in place just so we can implement protocols in them.

Youtube-mp3.org

Once we are able to search for videos in Youtube, we need for a way to download only the audio of it.

One way of doing that would be to find a way to get the video from Youtube and then use a library to rip out the audio, however, since we need some sort of API to help us get the video, we could already use one that gets the audio in MP3 format, and use that instead.

After researching a few alternatives, none were found that suited what we wanted to do, so we opted for performing reverse engineering on a website that offers to convert a Youtube video into an MP3 audio file.

The service selected was youtube-mp3.org[18], since it's a service that has been up and running for a long time without changes, which leads us to believe that it is unlikely for the service to modify now it's behaviour.

We studied the interactions between the browser and it's API in order to discover how it worked using Charles Proxy[19]. Luckily for us, it was using AJAX to get the content, and thus the javascript code was available by inspecting the code in the browser, so that also provided some of the answers we needed.

Upon inspection of the call we noticed that first of all, upon tapping on the convert button, it was sending a GET request to the URL:

http://www.youtube-mp3.org/a/pushItem/?item=https%3A//www.youtube.com/watch%3Fv%3DnXIdQ4uHd7s&el=na&bf=false&r=1449741223012&s=38482

This request was responded from the server with a String in the body.

[18] "Online YouTube to MP3 Converter." 2014. 12 Dec. 2015 <http://yt2mp3.org/>
[19] "Charles Web Debugging Proxy • HTTP Monitor / HTTP ..." 2015. 12 Dec. 2015 <https://www.charlesproxy.com/>

Upon inspecting its query parameters we noticed that most of them were fixed except for:

- item: It was the URL of the youtube video we pasted in the box.
- r: It was the timestamp in milliseconds of the request.
- s: A signature passed. There's further detail on the signature in the next section.

After this request succeeded, the site started polling the backend with requests looking like the next URL:

http://www.youtube-mp3.org/a/itemInfo/?video_id=nXIdQ4uHd7s&ac=www&t=grp&r=1449741228263&s=66385

Again, we noticed how most of the parameters were fixed except for:

- video_id: It was the string in the body of the response from the first call.
- r: The timestamp.
- s: The signature.

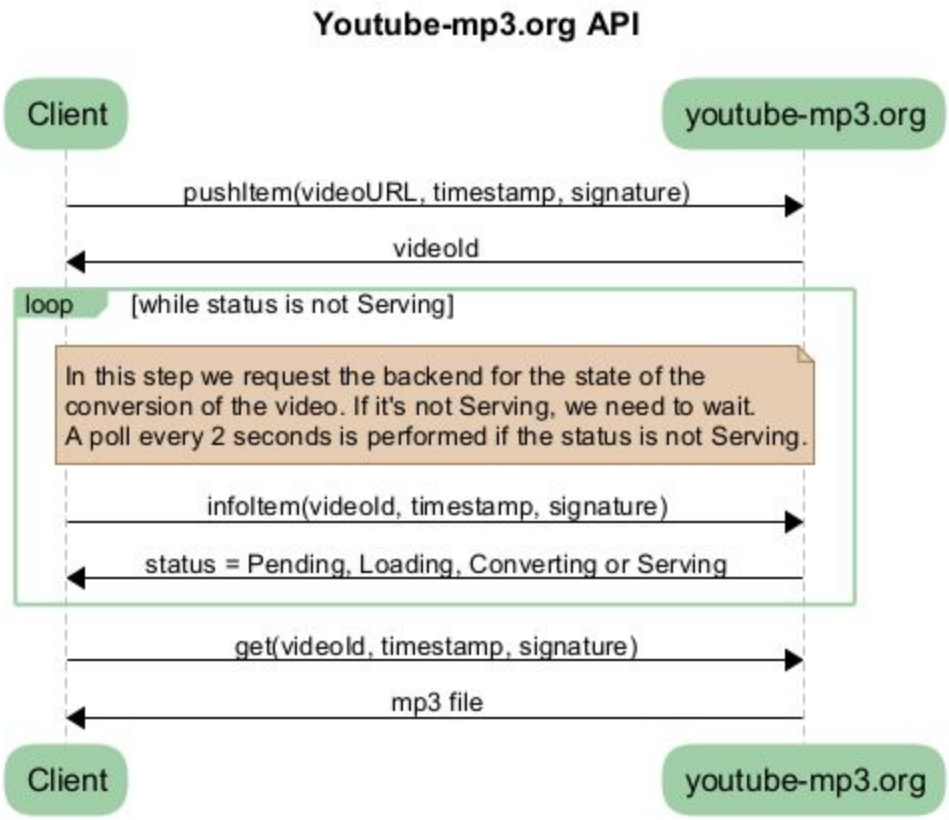
The important part of this call is that it was repeated until the status returned in the JSON body (broken by some extra javascript around it that needed to be trimmed) was equal to "Serving".

Once that status was met, we had a download button enabled and upon tapping it we saw a call to:

http://www.youtube-mp3.org/get?video_id=nXIdQ4uHd7s&ts_create=1449741273&r=ODguMTExLjY0LjY0&h2=ff9c9988f4379b7a65c20a17d4ed65ed&s=20987

Where a 302 HTTP status was given from the server and we were redirected to the file to be downloaded.

In summary, the client required to be downloaded should behave like the following sequence diagram:



The final code of the client is composed of a Retrofit service interface, a wrapper to handle the signature of calls, and an AsyncTask that leverages those methods and exposes the progress to the UI via the publishProgress method every time the item info is obtained.

Service interface

```
1. public interface YoutubeMP3Service {
2.
3.     @Headers({
4.             "Accept-Location : *",
```

```
5.          "User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_1)
            AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.80
            Safari/537.36"
6.      })
7.      @GET("/a/pushItem/")
8.      Call<String> pushItem(@Query("item") String youtubeURL,
9.                            @Query("el") String na,
10.                           @Query("bf") String falsy,
11.                           @Query("r") long currentMillis,
12.                           @Query("s") String signature);
```

Lines 3 to 12 declare the call used to push an item to the youtube-mp3 API. In it we define a few headers, the relative URL, the HTTP method to use and the query parameters to use.

```
13.     @Headers({
14.         "User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_1)
            AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.80
            Safari/537.36"
15.     })
16.     @GET("/a/itemInfo/")
17.     Call<String> itemInfo(@Query("video_id") String videoId,
18.                           @Query("ac") String www,
19.                           @Query("t") String grp,
20.                           @Query("r") long currentMillis,
21.                           @Query("s") String signature);
```

Lines 13 to 21 declare the request to obtain information regarding an item previously pushed. The declaration is pretty straightforward and similar to the push item method.

```
22.  @Headers({
23.          "User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_1)
     AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.80
     Safari/537.36"
24.  })
25.  @GET("/get")
26.  @Streaming
27.  Call<byte[]> getSong(@Query("video_id") String videoId,
28.                  @Query("ts_create") String ts_create,
29.                  @Query("r") String r,
30.                  @Query("h2") String h2,
31.                  @Query("s") String signature);
32.}
```

The rest of the class simply provides a method to obtain the mp3 file from the API. Compared to the other two methods, we can notice a new annotation *@Streaming*, which tells the Retrofit engine to treat the call as a streaming one, since we expect a file to be returned.

Wrapper Manager

```
1.  public class YoutubeMP3ServiceManager {
2.
3.    private static YoutubeMP3ServiceManager instance;
4.
5.    private YoutubeMP3Service mService, mServiceCustomParser;
6.
7.    public synchronized static YoutubeMP3ServiceManager
     getInstance() {
8.        if ( instance == null ) {
```

```
9.            instance = new YoutubeMP3ServiceManager();
10.       }
11.
12.       return instance;
13.   }
```

Lines 1 to 13 of the wrapper manager basically declare the required services and an static instance getter.

```
14.   private YoutubeMP3ServiceManager() {
15.       Retrofit retrofit = new Retrofit.Builder()
16.               .baseUrl("http://www.youtube-mp3.org")
17.               .addConverterFactory(GsonConverterFactory.create())
18.               .build();
19.       mService = retrofit.create(YoutubeMP3Service.class);
20.       mServiceCustomParser = new Retrofit.Builder()
21.               .baseUrl("http://www.youtube-mp3.org")
22.               .addConverterFactory(new RetrofitStringConverterFactory())
23.               .addConverterFactory(new
      RetrofitByteArrayConverterFactory())
24.               .build()
25.               .create(YoutubeMP3Service.class);
26.   }
```

Lines 14 to 26 declare the constructor of the wrapper class. In this creator, we initialize the Retrofit services we'll be later using. We have two different ones so we can use different converter factories.

```
27.  public String startDownloadVideo(String youtubeURL, Context
     context) throws IOException {
28.      // Instead of proper encoding we use this as the API rejects
     URLEncoder.encode output.
29.      // Since we know how the url looks like
     (https://www.youtube.com/watch?v=hpuLhzbf6Eo), we
30.      // can assume no strange characters will appear.
31.      String youtubeURLEscaped =
     youtubeURL.replaceAll(":","%3A").replaceAll("\\?","%3F").replace("=","
     %3D");
32.      long r = System.currentTimeMillis();
33.      String urlToSign =
     "http://www.youtube-mp3.org/a/pushItem/?item=" +
     youtubeURLEscaped + "&el=na&bf=false&r=" + r;
34.      Call<String> call = mService.pushItem(youtubeURL, "na", "false", r,
     SignatureGenerator.signURLRequest(urlToSign, context));
35.
36.      Response<String> response = call.execute();
37.      Log.i(YoutubeMP3ServiceManager.class.toString(), "pushItem
     response: " + response.body());
38.
39.      // Return the raw body response, which is the video Id.
40.      return response.body();
41.  }
```

Lines 27 to 41 declare a method to push a Youtube video to be parsed into an MP3 song. More precisely, we build the url to be signed from lines 31 to 33, and in line 34 we perform the call including the signature provided by the SignatureGenerator

class. Finally, we return the response body in lines 34 to 41, which contains the video Id extracted by Youtube-mp3.

```
42.  public ItemInfo getInfoVideo(String videoId, Context context) throws
     IOException {
43.      long r = System.currentTimeMillis();
44.      String urlToSign =
     "http://www.youtube-mp3.org/a/itemInfo/?video_id=" + videoId +
     "&ac=www&t=grp&r=" + r;
45.      Call<String> call = mServiceCustomParser.itemInfo(videoId,
     "www", "grp", r, SignatureGenerator.signURLRequest(urlToSign,
     context));
46.
47.      Response<String> response = call.execute();
48.      String cleanedJSON = response.body().substring(7,
     response.body().length() - 1);
49.
50.      // Return the raw body response, which is the video Id.
51.      ItemInfo info = new Gson().fromJson(cleanedJSON, ItemInfo.class);
52.
53.      return info;
54.  }
```

Lines 42 to 54 perform a very similar task for the get info video request. In this case though, we need to clean the JSON response, since it contains some extra Javascript code. That task is performed in line 47 and 48. Finally, the JSON obtained is parsed using Gson into a ItemInfo POJO we declared.

```
55.  public File getSong(String videoId, ItemInfo info, Context context)
     throws IOException {
56.      String urlToSign =
     "http://www.youtube-mp3.org/get?video_id=" + videoId
57.              + "&ts_create=" + info.getTs_create()
58.              + "&r=" + info.getR()
59.              + "&h2=" + info.getH2();
60.
61.      Call<byte[]> call = mServiceCustomParser.getSong(videoId,
62.              info.getTs_create(),
63.              info.getR(),
64.              info.getH2(),
65.              SignatureGenerator.signURLRequest(urlToSign, context));
66.      Response<byte[]> response = call.execute();
67.
68.      byte[] body = response.body();
```

Lines 55 to 68 perform the request for the mp3 file of the song similarly to how other requests where wrapped. However, so far we only obtain the byte array to be stored.

```
69.      File directory = new
     File(Environment.getExternalStoragePublicDirectory(Environment.DIRE
     CTORY_MUSIC), "TFM");
70.      if (!directory.exists() && !directory.mkdirs() ) {
71.          throw new IOException("Couldn't create the Directory to
     save music.");
72.      }
73.
```

```
74.     File saveTo = new File(directory, info.getTitle() + ".mp3");
```

Lines 69 to 74 define where the file should be stored, and creates the required directories if needed.

```
75.     FileOutputStream output = new FileOutputStream(saveTo);
76.     output.write(body, 0, body.length);
77.     output.close();
```

After that, lines 75 to 77 write the mp3 file we have received and writes it to the file.

```
78.     new SongImporter().saveFileToMediaStore(saveTo, context);
79.
80.     return saveTo;
81. }
82.}
```

Finally, we use the SongImporter class to force Android to scan the file and save it to its MediaStore.

AsyncTask

```
1. public class DownloadYoutubeSongAsyncTask extends
   AsyncTask<YoutubeVideo, ItemInfo, File> {
2.
3.   private final Context context;
4.
5.   private Throwable exception = null;
6.
7.   public DownloadYoutubeSongAsyncTask(Context context) {
8.     this.context = context;
```

```
9.    }
```

In this case, lines 1 to 8 provide a way to inject the context we need to use when downloading the mp3 file.

```
10.  @Override
11.  protected File doInBackground(YoutubeVideo... params) {
12.     if ( params.length != 1 ) {
13.        throw new UnsupportedOperationException("Only the
       download of one song at a time is supported");
14.     }
15.
16.     YoutubeVideo videoToDownload = params[0];
17.
18.     YoutubeMP3ServiceManager manager =
       YoutubeMP3ServiceManager.getInstance();
```

Lines 10 to 18 perform a check on the count of songs to download (only one is supported) and prepare the YoutubeMP3ServiceManager object.

```
19.     try {
20.        String videoId =
       manager.startDownloadVideo(videoToDownload.getVideoURL(),
       context);
21.
22.        ItemInfo info;
23.        do {
24.           // Sleep for 2 seconds between retries
25.           Thread.sleep(2000, 0);
```

```
26.
27.             info = manager.getInfoVideo(videoId, context);
28.
29.             if ( info == null ) {
30.                 throw new IOException("Couldn't retrieve the
    information from the video ID.");
31.             }
32.
33.             publishProgress(info);
34.
35.         } while ( !info.canStartDownload() );
36.
37.         return manager.getSong(videoId, info, context);
38.     }
```

Lines 19 to 38, there is a try clause that will push the item to download (lines 19 to 22), then periodically check its status every two seconds (lines 23 to 35) an finally return the song we want to obtain, in line 37.

```
39.     catch (Throwable exception) {
40.         Log.e(DownloadYoutubeSongAsyncTask.class.toString(),"An
    error occured while downloading the song", exception);
41.         this.exception = exception;
42.         return null;
43.     }
44. }
```

Lines 39 to 44 provide error handling whilst downloading the song. In case an error occurs, we keep an exception in the task and return no files back.

```
45.  public Throwable getException() {
46.      return exception;
47.  }
48.}
```

Signature

One of the challenges found while reverse engineering this API was to sign correctly the calls.

Upon studying the Javascript code used by the website, we detected and isolated the piece of code responsible to receive the initial url, and append to it the “s” parameter with the correct signature.

We modified the said code in order to not modify the url but return the signature parameter instead, so that it could work nicely with Retrofit.

However, the code we had was in Javascript, and had to run on Android. We had two different approaches to solve this:

- Write a unit test, translate the method to Java, and debug it until it signs as expected.
- Use a Javascript engine to execute the code we already had and return the response to Java.

We opted for the second one as a simpler solution for a short sighted project as this was. However, where this to be released and maintained the first solution would have been more appropriate.

In appendix A the Javascript code is available.

Library evaluations

Evaluation approach

Each evaluation performed will consist in the following sections:

- Summary
- Library author
- License
- Documentation
- Tests
- Github stats, if available
- Snippets of code
- Drawbacks detected, if any
- Problems that allowed to overcame, if any

With this, we plan to provide enough information to the user to be able to do an informed decision on whether a specific library is well suited for his project or not.

Dagger2

Summary

Dagger2 is the evolution of Square's dagger by Google. It's goal is to remove the burden of writing the boilerplate code required to perform dependency injection in your code.

In Dagger, the original approach, most checks were performed on runtime, however in Dagger2, Google opts for generating the boilerplate code in compile time, which gives better safety to the user.

Library author

The author of this library is Google.

License

Apache v2.0 license

Documentation

Well documented in http://google.github.io/dagger/ and https://github.com/google/dagger .

Tests

Extensively tested

Github stats

232 watchers, 2360 stars, and 974 forks. Seems to have an active community and it's pulse seems to be quite active:

https://github.com/google/dagger/pulse

Snippets of code

In order to pass parameters to another object, the developer needs to define:

A component, which represents the interface that can be called to obtain the required dependencies:

```
1. @Component(modules = ModelManagerModule.class)
2. public interface ModelManagerComponent {
3.
4.     PlaylistManager playlistManager();
```

```
5.  }
```

A module that implements a method that the automatically generated component will be calling in order to obtain the required dependency.

```
1.  @Module
2.  public class ModelManagerModule {
3.
4.    static PlaylistManager manager;
5.
6.    @Provides
7.    PlaylistManager providePlaylistManager(){
8.      if (manager == null) {
9.        manager = new PlaylistManager();
10.     }
11.
12.     return manager;
13.   }
14.
15. }
```

In this module, we basically keep a static instance, since we want only one PlaylistManager to exist.

Drawbacks detected

It is quite cumbersome to understand its working, and the amount of boilerplate code it saves you from writing is not that much.

Problems that allowed to overcame

None. We could use just basic dependency injection to provide the dependencies. However, it looks promising when the amount of dependencies of a class grow

considerably. In most cases in our app we just needed to pass one dependency or two, which made it easier to create a constructor with one or two parameters and pass them directly rather than creating an interface and a class.

Mosby

Summary

Quoting again it's website[20], it's goal is to:

> *... help you build modern android apps with a clean Model-View-Presenter architecture. Furthermore, Mosby helps you to handle screen orientation changes by introducing ViewState and retaining Presenters.*

We have found this library to have immense value to avoid our Activities and fragments from becoming bloated with extra responsibilities they should not have.

Library author

Hannes Dorfmann.

License

Apache v2.0 license

Documentation

The documentation is quite extensive in http://hannesdorfmann.com/mosby/ . Also, the code is small and simple enough to dive into it if required for a better understanding.

Tests

Unit tests extensively testing the source code.

Github stats

77 Watchers, 855 Stars and 133 forks. It's pulse is rather weak, however the author answered quickly when we raised a suggestion to change it's samples. (https://github.com/sockeqwe/mosby/pulse/monthly)

Snippets of code

The gist of the MVP stack is to split the concerns of the view and the presenter from the fragments and activities.

[20] "Mosby (MVP library) - Hannes Dorfmann." 2015. 12 Dec. 2015 <http://hannesdorfmann.com/mosby/>

To do so, the user must first declare two interfaces:

The view interface:

```
1. public interface ISearchSongsView extends MvpView {
2.
3.   void setSearchResults(List<YoutubeVideo> results);
4.
5.   void showError(String errorMessage);
6. }
```

And the presenter interface:

```
1. public interface ISearchSongsPresenter extends
   MvpPresenter<ISearchSongsView> {
2.
3.     void searchQueryChangedTo(String query);
4.
5.     void showSearchResultView(YoutubeVideo result);
6.
7. }
```

Notice how both interfaces override an interface offered by Mosby. With that given, the user must now implement them in different objects.

It is recommended to have your activity/fragment implementing the view since it will make it easier when you need to handle navigation (launching intents and so on).

Drawbacks detected

Sometimes it's not clear where the navigation code fits better, whether in the view or the presenter. Our opinion is that the navigation fits better the presenter, and therefore we launch intents from the presenter. However, the opposite approach could also be argued as valid.

This drawback, on the other hand, is inherent from mixing the MVC design pattern and Android's SDK, and if we were to implement a similar approach without this library we'd find ourselves having a similar discussion.

Problems that allowed to overcame

In short term none, however, it is quite noticeable how easy it is to extend an existing activity using this pattern as how it would be if we were to place all the code in the Activity.

This is due to the fact that Activities, if not properly maintained, tend to become bloated god objects tightly coupled to all the layers of a typical MVC design and really hard to refactor.

As explained in "Advocating against Android Fragments"[21]:

On Android, Context is a god object, and Activity is a context with extra lifecycle. A god with lifecycle? Kind of ironic. Fragments aren't gods, but they make up for it by having extremely complex lifecycle.

> *On Android, Context is a god object, and Activity is a context with extra lifecycle. A god with lifecycle? Kind of ironic.*

The inherent complexity of an Activity is given by the SDK, and therefore the simpler we keep our Activities and Fragments, the better.

[21] "Advocating Against Android Fragments." 2015. 3 Jan. 2016 <https://corner.squareup.com/2014/10/advocating-against-android-fragments.html>

RxAndroid

Summary

RxAndroid, supported with RxJava, is the Reactive extensions library port to Android. As such, most of the interfaces are already defined in several languages and one can expect little changes to its API.

It's benefits are transitioning your app into a rather more functional approach, since you work with streams of events being modified by different pipes on a pipeline.

If performed correctly, the pipes should avoid accessing any global state, to ensure the benefits of the functional programming are not lost due to Java's imperative nature.

Library author

ReactiveX, an organisation who has ported the reactive extensions to several languages and platforms.

License

Apache v2.0 license.

Documentation

Extensively documented, with several ports having consistent API's that you could check independently. There's also resources to graphically display the evolution of a stream depending on what functions you apply to it like http://rxmarbles.com/, which results in a really helpful tool to decide on what needs to be done in your stream.

Tests

Extensively tested.

Github stats

435 Watchers, 4025 Stars and 551 forks. Furthermore, it´s pulse appears to be quite active.

Snippets of code

In order to smooth the user search in the Youtube API we used the following snippet:

```
1. Observable.create(new Observable.OnSubscribe<String>() {
2.     @Override
3.     public void call(Subscriber<? super String> subscriber) {
```

```
4.        SearchSongsPresenter.this.subscriber = subscriber;
5.    }
6.  })
```

In lines 1 to 6 we create an Observable and keep the subscriber in the SearchSongsPresenter class.

```
7.        // Based on
      https://medium.com/@diolor/improving-ux-with-rxjava-4440a13b157f#.p
      mnmqbzau
8.        .subscribeOn(Schedulers.newThread())
9.              // Debounce the text events
10.       .debounce(200, TimeUnit.MILLISECONDS)
11.             // Perform a call to the youtube API
```

In line 8, we force subscriptions to occur on a background thread, so we don't block the UI thread.

In line 9, we debounce the events, which means that two (or more) events occurring within less than 200 events will be compacted into the last one.

```
12.       .map(new Func1<String, List<YoutubeVideo>>() {
13.          @Override
14.          public List<YoutubeVideo> call(String s) {
15.             try {
16.                // Don't look for strings shorter than 3 chars
17.                if (s != null && s.length() > 3) {
18.                   return YoutubeSearch.searchYoutubeAPI(s);
19.                } else {
20.                   return new ArrayList<YoutubeVideo>();
```

```
21.                 }
22.             } catch (IOException e) {
23.                 Log.e(SearchSongsPresenter.class.toString(), "Error while
    fetching youtube data", e);
24.                 // Null signals there's been an error to the subscribe method
25.                 return null;
26.             }
27.         }
28.     })
```

On lines 12 to 28 we map the result from a query string to a list of YoutubeVideo's so that we can later bind it into the view.

```
29.             // Migrate it to the main thread
30.     .observeOn(AndroidSchedulers.mainThread())
```

Line 30 forces the events to be observed on the Android main thread, so that we can update the UI once the network request has finished.

```
31.             // Handle data
32.     .subscribe(
33.             // On next
34.             new Action1<List<YoutubeVideo>>() {
35.                 @Override
36.                 public void call(List<YoutubeVideo> youtubeVideos) {
37.                     if (view != null) {
38.                         if ( youtubeVideos != null ) {
39.                             view.setSearchResults(youtubeVideos);
40.                         }
41.                         else {
```

```
42.                         view.showError(context.getString(
    R.string.search_song_error));
43.                     }
44.                 }
45.                 lastResults = youtubeVideos;
46.             }
47.         }
48.     );
```

Finally, lines 31 to 48 will set to the view the videos we fetched before or display an error.

It basically listens for events in the search query text, debounces them, so quick taps from the user won't trigger several calls to the API but just one once it's done, map's the string input to a *YoutubeVideo* object from the model using an API call, and finally displays it on the UI thread.

If we were to avoid RxAndroid, this 50 lines of code would probably extend to some 100-150, amongst different methods, with race conditions between all of them, which would increase considerably the complexity of our solution.

Drawbacks detected

When downloading a song from youtube-mp3.org, we found that AsyncTask was better suited for the task.

This in itself is not a limitation, but rather a confirmation that there is no silver bullet that will solve all tasks.

Problems that allowed to overcame

Apart from the example provided in the code snippet, it also allowed to overcame the fact that Realm has no method to insert an object only if it does not exist in the database. To do so, we executed a stream of observables in which the modifications where:

1. Emit all Media obtained from Android.
2. Map the media to a custom model object.
3. Filter the ones that already exist in the database.
4. Join them into a list with all the songs not yet in the database.
5. Save the new songs.
6. Map the new songs to a query that returns all the songs.

Again, this would require more complex code than the one we generated to perform the same task.

Realm

Summary

Realm is a DBMS focused on mobile platforms without being just a wrapper on top of SQLite.

It's main goals are usability, low latency, and being cross platform (has an iOS/OS X counterpart).

It has its core written in C and shared between all platforms, and a Java (Swift and Objective-C) API to encapsulate the core and offer an easy API to the user.

Library author

Realm

License

Apache v2.0 license

Documentation

Really well documented, with plenty of tutorials and online resources.

Tests

Well tested

Github stats

210 Watchers, 3082 stars, 337 forks. It's pulse is really active and since it has a company behind it is foreseeable that it is properly maintained.

Snippets of code

The first step in order to use Realm is to set up a model object. To do so, we must define a POJO that overrides *RealmObject* and provides setters and getters to whatever properties it must hold.

As an example, this is our Song model:

```
1. public class Song extends RealmObject implements
   RecyclerViewAdapter.IAdaptableObject {
2.
3.     @PrimaryKey
```

```
4.  private long id;
5.
6.  private String filePath;
7.
8.  private String title;
9.
10. private boolean isFavourited;
11.
12. public String getFilePath() {
13.     return new String(filePath);
14. }
15.
16. public void setFilePath(String filePath) {
17.     this.filePath = filePath;
18. }
19.
20. @Override
21. public String getTitle() {
22.     return title;
23. }
24.
25. public void setTitle(String songName) {
26.     this.title = songName;
27. }
28.
29. public long getId() {
30.     return id;
31. }
32.
```

```
33.  public void setId(long id) {
34.      this.id = id;
35.  }
36.
37.  public void setFavourited(boolean isFavourited) {
38.      this.isFavourited = isFavourited;
39.  }
40.
41.  public boolean isFavourited() {
42.      return isFavourited;
43.  }
44.}
```

As is seen, lines 1 to 9 define the instance variables of the class, including it's primary key. Finally, 10 to 44 define the getters and setters required to access them.

Once the model is defined, we can save objects to our database by initialising an object with all the data and then save it with:

```
1.  // Save the songs we have at this point and return a fetch of all of them
2.  realmDb.beginTransaction();
3.  realmDb.copyToRealmOrUpdate(songs);
4.  realmDb.commitTransaction();
```

This lines open a transaction, copies all the data from your object (or objects in a List in this case) into the database, updating the objects if the ID's match, and finally commits the transaction.

Transactions in Realm lock the entire database so it yields a better performance if you prepare the objects and then copy them into Realm rather than performing any task within the transaction.

Finally, to fetch objects from the database a simple and effective API is provided:

```
1. realmDb.where(Song.class).findAllSorted("title", true);
```

Drawbacks detected

All your model **must** override *RealmObject*, which forbids you from using a hierarchy in your models.

Also, your model will be overridden by Realm when you fetch objects from the database. This means that if you have a setter that performs side effects, a model created via your constructor will execute them, but the same “class” when fetched from Realm will not. Thus, the setters should be left without side effects for safety.

Even though this represents a limitation, it is also enforcing the developer to keep its model dumb, which tends to be a wise approach.

Problems that allowed to overcame

None apart from what any other ORM would cover. However, it has a really nice API, and an iOS counterpart, which would make it nicer when transitioning from one platform to another.

Android Support Design libraries

Summary

This library helps to use Material design components in your app.

It's rather annoying requiring to add libraries into your app to provide a native look & feel to it. When developing for iOS you could easily implement most of the components of Material design without much complexity.

The mere existence of this library reflects on how much artificial complexity the Android SDK contains.

Library author

Google

License

Apache v2.0 license

Documentation

Extensively documented.

Tests

Well tested

Github stats

N/A

Snippets of code

There are several examples of usages of different components throughout the app. In this section we demonstrate only the tab layout, which composes the main activity's layout.

The tab layout needs to be defined in the layout file with:

```
1. <android.support.design.widget.TabLayout
2.     android:id="@+id/tabs"
3.     android:layout_width="match_parent"
4.     android:layout_height="wrap_content" />
```

Once it’s defined, the activity needs to provide via a datasource the fragments to hold and the titles of each tab.
Once that’s done, the tab layout is already functional as any other tab layout.

Drawbacks detected

None.

Problems that allowed to overcame

None, just provided the Material design components.

Butterknife

Summary

Butterknife allows to easily inject the view's and events from your layout file into your code.

Library author

Jake Wharton, a developer from Square

License

Apache v2.0 license

Documentation

Well documented in its website[22].

Tests

Well tested.

Github stats

515 Watchers, 6471 stars, and 1371 forks. The pulse is slightly active.

Snippets of code

In order to inject the view into your code (you can do it into an Activity, fragment, or any kind of Object, even POJO's) you need to mark with annotations the properties that you want to inject, and the methods you want to handle events from the UI.

As an example, this is the code from a POJO (Implementing a Mosby View protocol) that uses Butterknife:

```
1. public class SongActivityView implements ISongView {
2.
3.   private ISongPresenter presenter;
4.
5.   private boolean isSongFavourited;
6.
7.   @Bind(R.id.song_title) protected TextView titleTextView;
8.   @Bind(R.id.seekBar) protected SeekBar songProgressBar;
```

[22] "Butter Knife." 2013. 12 Dec. 2015 <http://jakewharton.github.io/butterknife/>

```
9.    @Bind(R.id.love) protected ImageButton favouriteButton;
10.
11.   private final SeekBar.OnSeekBarChangeListener
      onSeekBarChangedListener = new
      SeekBar.OnSeekBarChangeListener() {
12.
13.       // ...
14.   };
15.
16.   public void bindToView(View view) {
17.       ButterKnife.bind(this, view);
18.       songProgressBar.setOnSeekBarChangeListener
      (onSeekBarChangedListener);
19.   }
20.
21.   /// ...
22.
23.   @OnClick(R.id.play) void onPlayButtonPressed(){
24.       presenter.playSong();
25.   }
26.
27.   @OnClick(R.id.pause) void onPauseButtonPressed(){
28.       presenter.pauseSong();
29.   }
30.
31.   @OnClick(R.id.love) void onFavouriteButtonPressed(){
32.       presenter.setSongIsFavourited(!isSongFavourited);
33.   }
34.
```

```
35.  @OnClick(R.id.add_to_playlist) void onAddToPlaylistPressed(){
36.      presenter.addSongToPlaylist();
37.  }
38.
39.  // ...
40.}
```

As is visible here, some of the methods are annotated with @OnClick and with an id of a view. Similarly, some of the views are annotated with @Bind. When the line Butterknife.bind is executed, the properties are filled with the given views, and the methods are referenced from an OnClickListener that will call it.

Notice however how the OnSeekBarChangeListener had to be implemented manually since it's not supported by Butterknife yet.

Drawbacks detected

Some listeners are not yet supported.

Problems that allowed to overcame

Removed plenty of boilerplate code from our app.

Retrofit

Summary

According to its website, it's a type safe HTTP client for Android and Java. However, the library goes much further and actually lifts you of all the burden of having to write any kind of network code.

Library author

Square.

License

Apache v2.0 license

Documentation

Well documented in the website.

Tests

Well tested

Github stats

661 Watchers, 8610 stars and 1539 forks. It's pulse is really active.

Snippets of code

In order to perform an HTTP request using Retrofit you need to first create the interface of the service. As an example, the interface for the API we used looks like this:

```
1. public interface YoutubeMP3Service {
2.
3.   @Headers({
4.         "Accept-Location : *",
5.         "User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_1)
   AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.80
   Safari/537.36"
6.   })
7.   @GET("/a/pushItem/")
8.   Call<String> pushItem(@Query("item") String youtubeURL,
9.                         @Query("el") String na,
```

```
10.             @Query("bf") String falsy,
11.             @Query("r") long currentMillis,
12.             @Query("s") String signature);
```

Lines 3 to 12 declare the call used to push an item to the youtube-mp3 API. In it we define a few headers, the relative URL, the HTTP method to use and the query parameters to use.

```
13. @Headers({
14.         "User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.80 Safari/537.36"
15. })
16. @GET("/a/itemInfo/")
17. Call<String> itemInfo(@Query("video_id") String videoId,
18.             @Query("ac") String www,
19.             @Query("t") String grp,
20.             @Query("r") long currentMillis,
21.             @Query("s") String signature);
```

Lines 13 to 21 declare the request to obtain information regarding an item previously pushed. The declaration is pretty straightforward and similar to the push item method.

```
22. @Headers({
23.         "User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.80 Safari/537.36"
24. })
25. @GET("/get")
26. @Streaming
27. Call<byte[]> getSong(@Query("video_id") String videoId,
28.             @Query("ts_create") String ts_create,
29.             @Query("r") String r,
```

```
30.             @Query("h2") String h2,
31.             @Query("s") String signature);
32.}
```

The rest of the class simply provides a method to obtain the mp3 file from the API. Compared to the other two methods, we can notice a new annotation @Streaming, which tells the Retrofit engine to treat the call as a streaming one, since we expect a file to be returned.

As it's visible, you define the calls in a static manner, and don't need to implement the interface at all, since Retrofit will do it for you. In order to obtain the implementation Retrofit offers, you have to run the following code:

```
1. Retrofit retrofit = new Retrofit.Builder()
2.         .baseUrl("http://www.youtube-mp3.org")
3.         .addConverterFactory(GsonConverterFactory.create())
4.         .build();
5. mService = retrofit.create(YoutubeMP3Service.class);
```

This snippet of code creates an object implementing *YoutubeMP3Service* pointing to the given baseUrl and using the provided converter factory.

Drawbacks detected

The detected drawbacks is that Retrofit makes it a bit awkward to customise the calls further if needed.

In order to sign the calls, for example, we had to build a string with the URL, then sign and pass it as parameter since intercepting the call and getting the URL had higher complexity than that.

Problems that allowed to overcame

None, however it made very easy to implement a client for a REST service, which tends to be a difficult and error prone task.

Glide

Summary

Glide provides an easy way of loading images from http URL's, including a customisable caching mechanism.

Library author

Bumptech, a company bought by Google.

License

Apache v2.0 license

Documentation

Well documented

Tests

Well tested

Github stats

424 Watchers, 5874 stars, and 1177 forks. It's pulse is quite active.

Snippets of code

The use of this library is quite simple and straightforward. You need to initialize Glide with a context, and then tell it to load a url and put it into an imageView. An example of it is like the following:

```
1. Glide.with(mImageView.getContext())
2.         .load(objectWithImage.getImageURL())
3.         .placeholder(R.drawable.placeholder_image)
4.         .into(mImageView);
```

Drawbacks detected

None

Problems that allowed to overcame

None

Conclusions

After this piece of work, we have come to the conclusions that, thanks to Gradle and the standards it sets, the Android ecosystem has matured to the point to easily allow the developer to include 3rd party libraries.

However, the developer needs to consider thoroughly which libraries are required in its project and which are not, in order to avoid any kind of maintainability issues it may bring.

The aim of this work has been to provide the developer with an evaluation of a group of libraries, in the hope that it will simplify the task to cherry pick which may be most beneficial to her work.

Out of this work, we have personally obtained a knowledge of what are the use cases each library is useful for, and when to (and not to) use them.

Future work

The scope breadth of this stream of work could be increased by extending the analysis performed on the Dagger2 library framework, which we found to be non efficient in such a small application, which didn't have enough dependencies to inject to make the library worthwhile.

Furthermore, the original dagger would also prove to be an interesting study to perform, to study the reasons why Google decided to fork it.

Finally, the ever-evolving Android ecosystem will continue its fast pace of evolution. With this, the only thing certain is that most of the libraries studied will eventually become obsolete, and newer, hopefully better, ones will take their place.

As time goes by, the work performed will need to be redone to keep up with cutting edge libraries and practices.

Objectives met

The objectives set when starting this stream of work were to evaluate the following libraries:

- Dagger2
- Mosby
- RxAndroid
- Realm
- Android support design libraries
- Butterknife
- Retrofit
- Glide

And also to produce a usable music app which would act as a coding playground for us to experiment with them.

All of the aforementioned objectives have been met, however the study of Dagger2 has been deemed inadequate due to the fact that the app remained small enough to not have many dependencies to inject from one component of the app to the others.

Furthermore, all deliverables have been delivered on time and comprising the whole scope we committed to.

Appendix

Appendix A: Javascript signature code

The following piece of obfuscated code is the Javascript code responsible to generate a signature query parameter for the calls to the youtube-mp3.org backend. In order to avoid potential mistakes we decided to include a Javascript engine called Rhino to execute the sign_url method from Java and avoid the risk of spending a considerable amount of time unit testing and debugging the conversion of this code into Java.

```
1.  var b0I = {
2.    'V': function(I, B, P) {
3.      return I * B * P;
4.    },
5.    'D': function(I, B) {
6.      return I < B;
7.    },
8.    'E': function(I, B) {
9.      return I == B;
10.   },
11.   'B3': function(I, B) {
12.     return I * B;
13.   },
14.   'G': function(I, B) {
15.     return I < B;
16.   },
17.   'v3': function(I, B) {
18.     return I * B;
19.   },
20.   'I3': function(I, B) {
```

```
21.     return l in B;
22.   },
23.   'C': function(l, B) {
24.     return l % B;
25.   },
26.   'R3': function(l, B) {
27.     return l * B;
28.   },
29.   'O': function(l, B) {
30.     return l % B;
31.   },
32.   'Z': function(l, B) {
33.     return l < B;
34.   },
35.   'K': function(l, B) {
36.     return l - B;
37.   }
38. };
39.
40. function _sig(H) {
41.  var U = "R3",
42.    m3 = "round",
43.    e3 = "B3",
44.    D3 = "v3",
45.    N3 = "l3",
46.    g3 = "V",
47.    K3 = "toLowerCase",
48.    n3 = "substr",
49.    z3 = "Z",
```

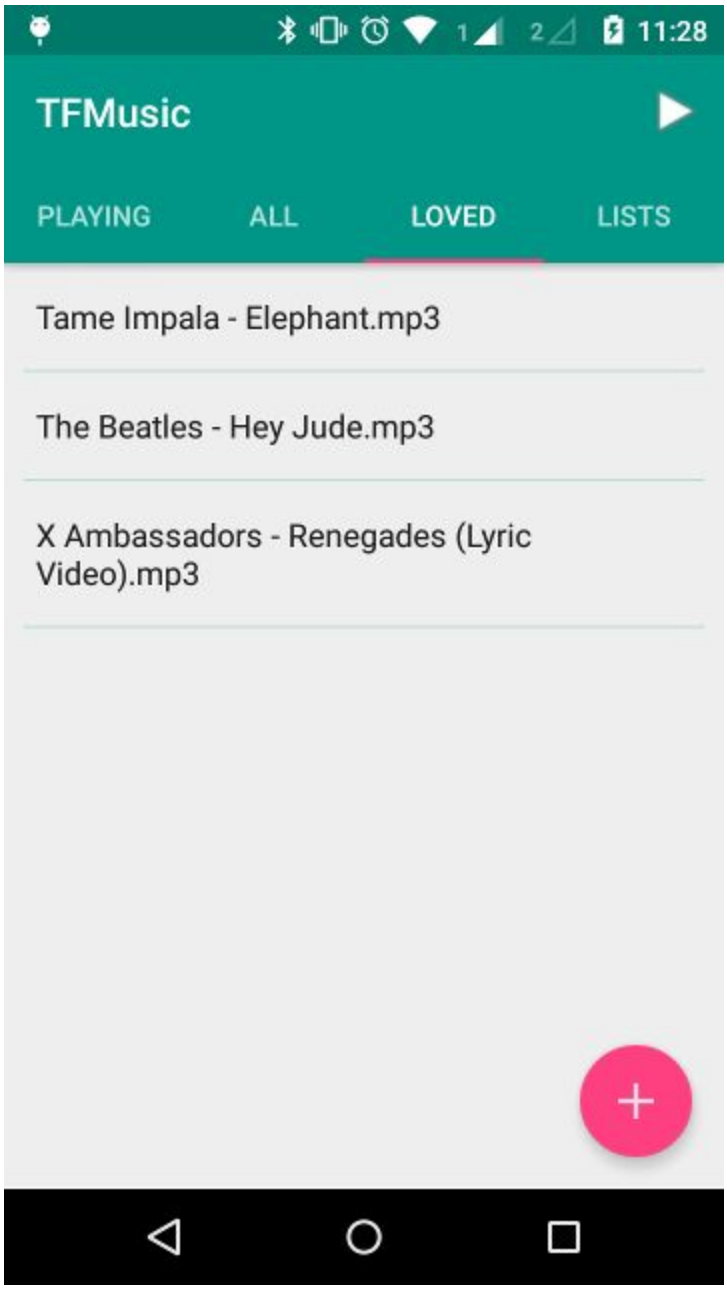
```
50.    d3 = "C",
51.    P3 = "O",
52.    x3 = ['a', 'c', 'e', 'i', 'h', 'm', 'l', 'o', 'n', 's', 't', '.'],
53.    G3 = [6, 7, 1, 0, 10, 3, 7, 8, 11, 4, 7, 9, 10, 8, 0, 5, 2],
54.     M = ['a', 'c', 'b', 'e', 'd', 'g', 'm', '-', 's', 'o', '.', 'p', '3', 'r', 'u', 't', 'v', 'y',
    'n'],
55.    X = [
56.       [17, 9, 14, 15, 14, 2, 3, 7, 6, 11, 12, 10, 9, 13, 5],
57.       [11, 6, 4, 1, 9, 18, 16, 10, 0, 11, 11, 8, 11, 9, 15, 10, 1, 9, 6]
58.    ],
59.    A = {
60.       "a": 870,
61.       "b": 906,
62.       "c": 167,
63.       "d": 119,
64.       "e": 130,
65.       "f": 899,
66.       "g": 248,
67.       "h": 123,
68.       "i": 627,
69.       "j": 706,
70.       "k": 694,
71.       "l": 421,
72.       "m": 214,
73.       "n": 561,
74.       "o": 819,
75.       "p": 925,
76.       "q": 857,
77.       "r": 539,
```

```
78.       "s": 898,
79.       "t": 866,
80.       "u": 433,
81.       "v": 299,
82.       "w": 137,
83.       "x": 285,
84.       "y": 613,
85.       "z": 635,
86.       "_": 638,
87.       "&": 639,
88.       "-": 880,
89.       "/": 687,
90.       "=": 721
91.     },
92.     r3 = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"];
93. gs = function(I, B) {
94.     var P = "D",
95.         J = "";
96.     for (var R = 0; b0I[P](R, I.length); R++) {
97.         J += B[I[R]];
98.     };
99.     return J;
100.    };
101.    ew = function(I, B) {
102.        var P = "K",
103.            J = "indexOf";
104.        return I[J](B, b0I[P](I.length, B.length)) !== -1;
105.    };
106.    gh = function() {
```

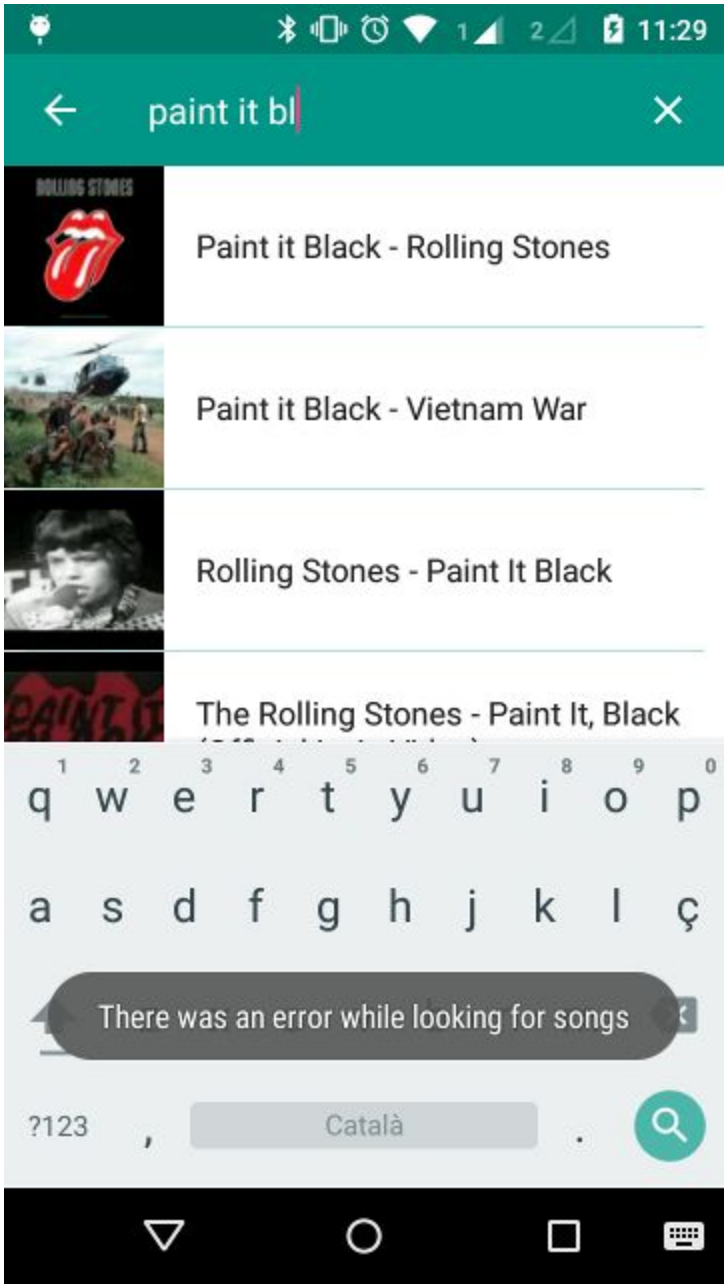
```
107.     var I = gs(G3, x3);
108.     return eval(I);
109.   };
110.   fn = function(I, B) {
111.     var P = "E",
112.       J = "G";
113.     for (var R = 0; b0I[J](R, I.length); R++) {
114.       if (b0I[P](I[R], B)) return R;
115.     }
116.     return -1;
117.   };
118.   var L = [1.23413, 1.51214, 1.9141741, 1.5123114, 1.51214, 1.2651],
119.     F = 1;
120.   try {
121.     F = L[b0I[P3](1, 2)];
122.     var W = gh(),
123.       S = gs(X[0], M),
124.       T = gs(X[1], M);
125.     if (ew(W, S) || ew(W, T)) {
126.       F = L[1];
127.     } else {
128.       F = L[b0I[d3](5, 3)];
129.     }
130.   } catch (I) {};
131.   var N = 3219;
132.   for (var Y = 0; b0I[z3](Y, H.length); Y++) {
133.     var Q = H[n3](Y, 1)[K3]();
134.     if (fn(r3, Q) > -1) {
135.       N = N + (b0I[g3](parseInt(Q), 121, F));
```

```
136.       } else {
137.           if (b0I[N3](Q, A)) {
138.               N = N + (b0I[D3](A[Q], F));
139.           }
140.       }
141.       N = b0I[e3](N, 0.1);
142.   }
143.   N = Math[m3](b0I[U](N, 1000));
144.   return N;
145.  };
146.
147.  function sig(a) {
148.    if ("function" == typeof _sig) {
149.      var b = "X";
150.      try {
151.          b = _sig(a)
152.      } catch (e) {
153.        console.log("Error sig:" + e)
154.      }
155.      if ("X" != b) return b
156.    }
157.    return "-1"
158.  };
159.
160.  function sign_url(a) {
161.    var b = sig(a);
162.    return escape(b)
163.  };
```

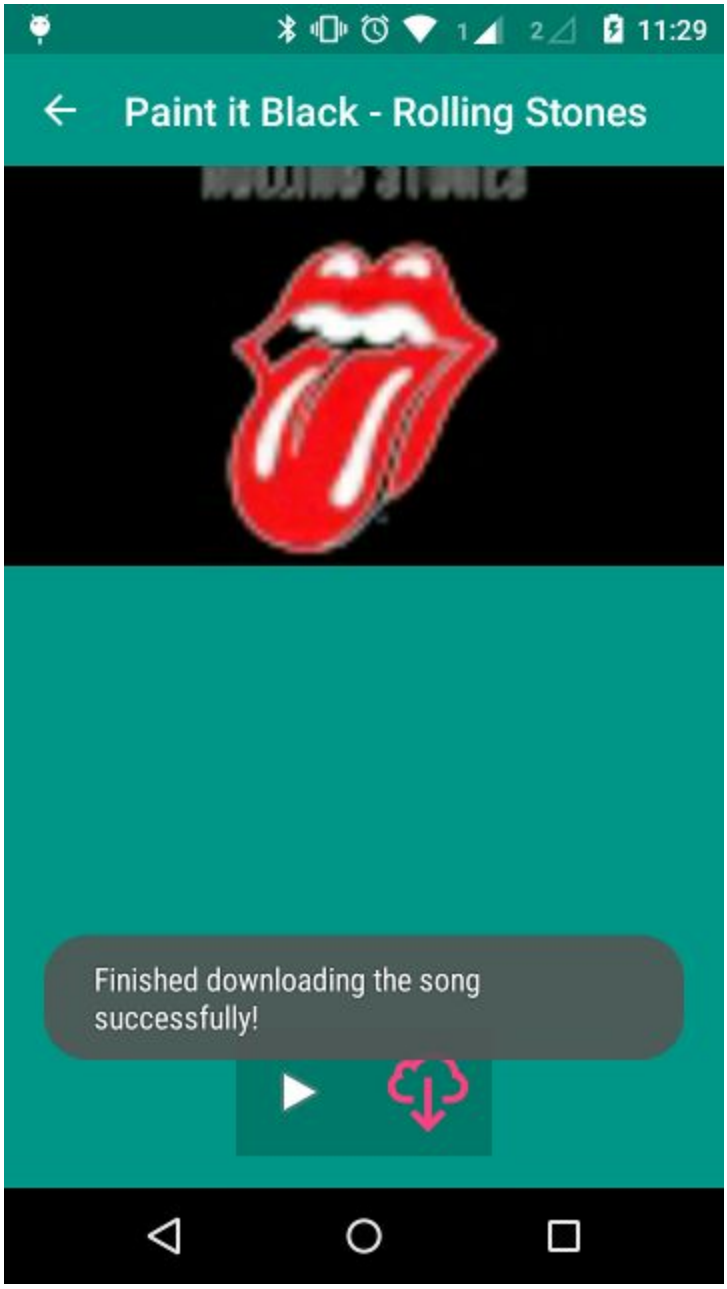
Appendix B: App screenshots



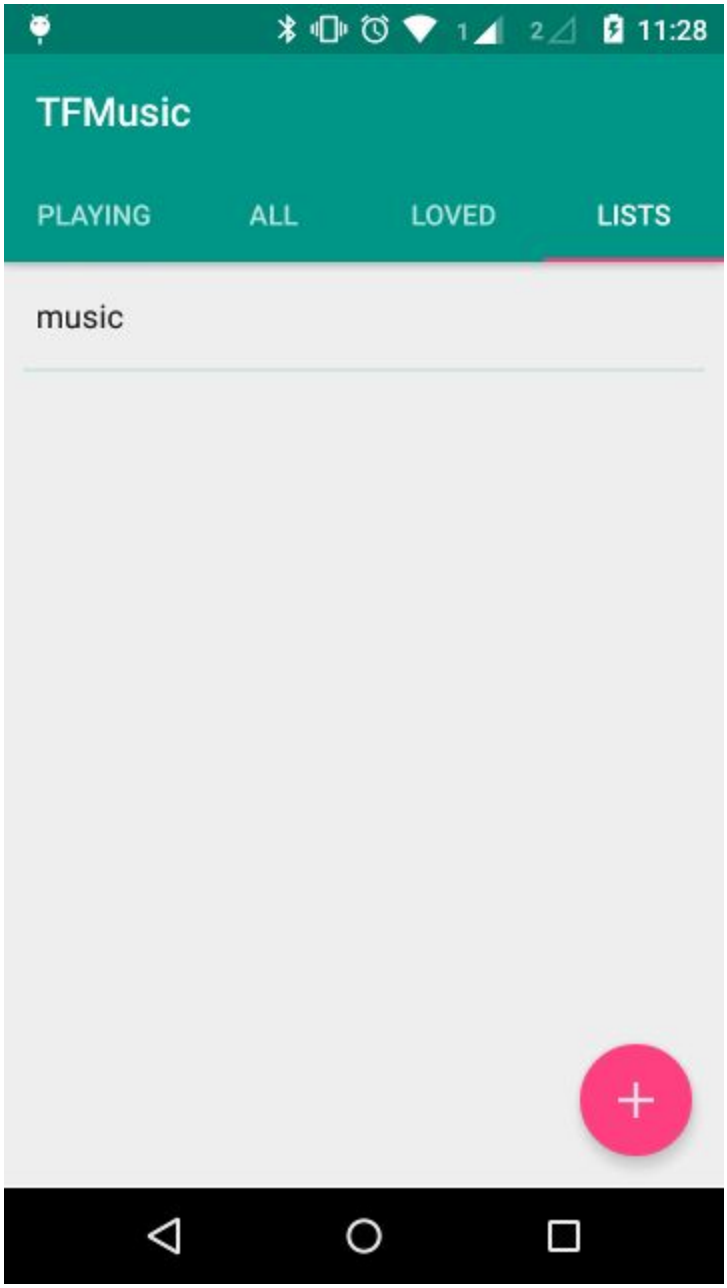
Favourited songs.



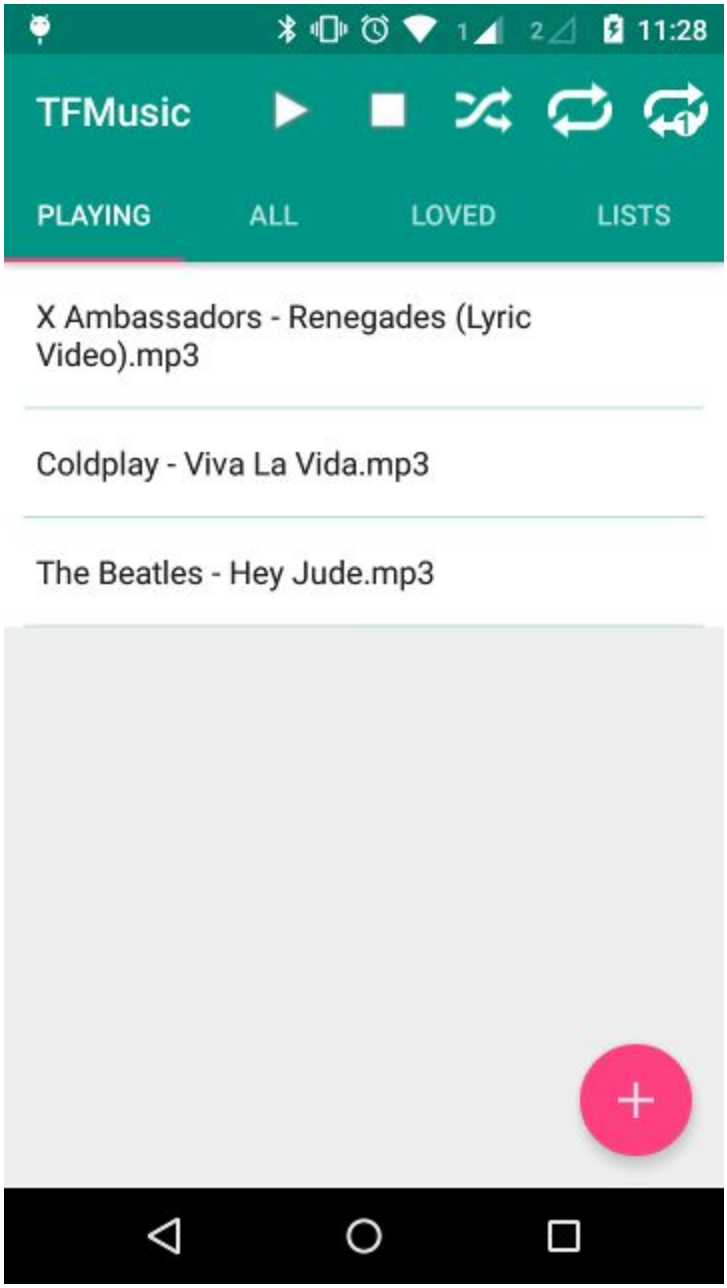
Searching for songs in Youtube and error reporting.



Downloading song.



Playlists available.



Currently playing songs (Paused state).