Code Review By Arthur Rocha

This Document points out some errors in code, and give solutions and best practices to follow, also propose a new Architecture for the application and some frameworks to include and why.

Errors

- the project was missing some internet permissions in order to perform network calls
  - ```
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    ```

- `Getting the cover bitmap code was place on the UI thread, also the URL for cover was misplaced`

The UI is tightly coupled to the model and database objects

Although this problems are not and issue for the compiler or are not shown as RuntimeExceptions, they are a solid example on how a bad architecture is done, making the project less maintainable, less expandable and reusable. I give the examples I found in the project.

- SongListActivity make direct calls to the Database Object (DBManager) to fetch data, this will complicate thing when some unit test are created, also makes the code less maintainable (more difficult to grow in the future).
  - `Example: RetrieveSongList() and retrieveSongListFromDB()`

- It does the sorting of the songList inside the activity, it should be done into the business logic.
- On SongListActivy a progressDialog is shown each time it fetch data using the DownloadSongListTask, i would delegete the show/hide process of the progressDialog to a presenter using the Model-View-Presenter design pattern, Or it can use a ProgressBar along with the AsyncTask:onProgressUpdate() method for a better understanding of the ongoing process.
- ListArrayAdapter is not using the **ViewHolder** Pattern, although the ListView don not enforce the developer to implement it, it's a bad practice that lead to performance issues because the adaptor is looking up the view compents each time a row is shown
  - I suggest using a ReciclerView that implements the ViewHolder pattern.

- ListArrayAdapter also make calls to the database methos to save and fetch some data, this shouldn't be done in the UI layer. (an adapter is part of the presentation layer)

Performance Issues

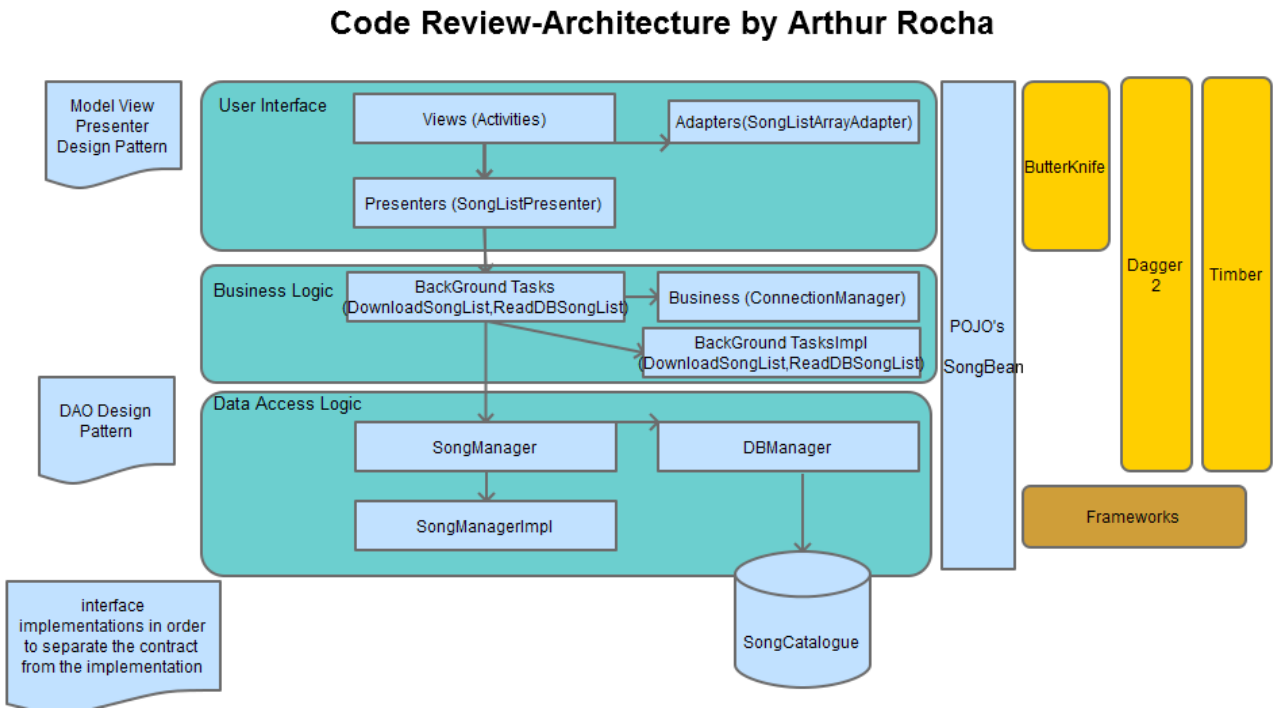
- on JsonParser class it uses while loops like if they where If statements (3 whiles nested in
    ```

parseSongList method), this makes the code less readable and can lead to unexpected results and performance decay. It increase the complexity of the algorithm unnecessarily.

Context-related Memory leaks

- On SongListActivity:displaySelectionDialog() creates a Dialog and shows it, this method is called in the overrided method OnResume(), this leads to memory leaks each time the activity change it's orientation (from portrait to landscape), because it will recreate the activity but the reference to the dialog will still exist, so can not be garbage collected.
  - Overriding onPause method to dismiss the dialog, so it will prevent this memory leak

- On the AsyncTasks (DownloadsonglistTask and ReadDBSongListTask) it uses the context of the SongListActivity, lets remaind that and activity context has a lifetime and a background process lifetime like the asyncTask is independent of any Activity. Doing this the background process may continue even if the activity is destroyed
  - It should be avoid holding a reference to a context in an AsyncTask to prevent memory leaks

Recommendations

- On Build.gradle there is only one flavour for release, I suggest creating other one for debugging purposes

- ```
  buildTypes {
        debug {
              debuggable true
              minifyEnabled false
        }
        release {
               minifyEnabled false
    proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
  }
  ```

- The code has not comments at all, this would make it difficult to read and maintain.
  ```

Architecture of the application

I propose the next architecture in order to have a maintainable, expandable and robust project, also to have a readable and reusable code.

The project is separated in tree layers:

- The presentation Layer, has the UI shown to the user, this layer implements the Model-View-Presenter design pattern, so we can separate the data representation from the Views that show them (Activities, dialogs, fragments), also the Adapter can implement the viewholder patter, wich become easier using butterknife framework.
- The Business Layer contains all the logic, sorting lists of data, perform heavy operations, parse data, etc.
- The Data Access Layer contains the connections to the database, and perform queries to insert, save, delete, update some data to the DataBase, or another DataSource (a datasource can be a webservice, a database etc.). This uses the DAO design pattern.



Framework recommendations

- Picasso API for loading images from an URL
- ButterKnife for binding views, listeners and implementing the view holder pattern easily.
- Dagger for dependency injection in order to have a good management of the instances created, also will simplified the implementation of an Architecture using Model-View-Presenter,

Abstract implementations, Data Access Objects (DAO's).

- some logger like Timber, LogCat, or crashlytics.