Developing a Music Player Mobile Application with Cloud Server

by

Ying Chen

B.S., Tianjin University, 2008
M.S., Nankai University, 2011
M.S., Kansas State University, 2014

A REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computer Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2016

Approved by:

Major Professor
Daniel A. Andresen

Abstract

A music player mobile application for Android is developed along with cloud server using Google's App Engine and Firebase. This music player application provides various ways of navigating to an audio file and different music visualizer options. What's more, the application also provides three major features: 1 user sign in and sign out, 2 display the most popular songs based on input, 3 users can submit comments and suggestions. These features are implemented by utilizing cloud services of Google's App Engine and Firebase. Specifically, an application running on App Engine plays as a server's role to verify user sign in. It also runs App Engine MapReduce jobs to consume large data stored in Google Cloud Storage and serves relatively small result about popular songs for the app. In addition, user's comments and suggestions are automatically synchronized with Firebase which makes modifying and analyzing synchronized data really convenient.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would first like to thank my advisor Professor Daniel A. Andresen of Computer Science at Kansas State University. His door is always open whenever I need suggestions when I was doing this project. With his support, friendly advice, and aspiring guidance. I enjoyed working on this project and learnt a lot new knowledge. Without his supervision and constant help, this project would not have been successfully conducted.

I would also like to express my warm thanks to my committee members, Professor Torben Amtoft and Professor Mitch Neilsen of Computer Science at Kansas State University. Their personal and professional guidance taught me a lot both academically and personally. Their support, insightful comments and encouragement have been very helpful during the project.

Furthermore, I would also like to thank my parents and friends, who helped me a lot during the project. Many thanks go to Computer Science department at Kansas State University for supporting Master students.

Chapter 1 - Introduction

The purpose of this mobile application is to build an easy-to-use music player with a server hosting on the cloud which facilities data routing and authentication. Mobile application design and development should take into account the important limitations of mobile devices and build applications that satisfy users specific habits. Some of the limitations include CPU performance characteristic, memory and storage space, battery life, screen size, mobility of the device, etc. This project takes care of each of the specific characteristics of mobile application in the design, develop and testing phases.

UI design is an essential part as it is the interface that user directly interacts with the application The front end provides clean, easy to use user interface and music visualization experience, simple color scheme and multiple ways of navigation. Each page holds only one or two colors, and major color is light green. User can navigation from the home page to the song that is about to play, through albums or artists. This multiple way of finding a song provides different navigation path which could much easier when there are many audio files on the device. Another important feature is music visualization. Three different visualizations are available to be added in the visualizer. User can add whichever they want during the playback.

Material Design is used to design user friendly UI features. It is a new design language developed by Google. It can be used in Android version 2.1 and up via the v7 appcompat library. According to the website for Android developers [8],

Material design is a comprehensive guide for visual, motion, and interaction design across platforms and devices.

There are lots of new design concepts about the look of layouts, animations, colors, paddings, etc. Many Google products have incorporated these design concepts, such as, Google

Docs, Google Hangouts, etc. One good example that this music player application uses elements from Material Design is its UI color scheme. Color in material design comprises primary and accent colors. Primary color is used at the base color of the application. It covers most the UI including application bar and menu bar. While accent colors are used to decorate other parts of UI, such as small icons, or buttons.

One advantage that this music player application over the other similar applications is its multiple choices of visualizations. The visualizations have beautiful colors and shapes changing with the song's rhythm. They can be added anytime when the song is playing. According to Android Central website [2], among the top five best music player, only one of them provided visualizer option. Google Play Music has a very tiny visualizer as an icon to indicating the song is playing. But it does not provide large screen of music visualizations.

Besides, this music reflects current data in the device without delay. If one song is deleted, it will not be show in any of the song list. Some of the other music player such as Google Play Music will still keep the deleted songs. If user tries to play the song, nothing happens. The deleted songs will only invisible when the application is restarted or user intentionally refresh it.

As one of the design purposes of this application, it will save memory space by not holding any of the redundant "hidden" functionalities, such as a radio, music equalizer, lots of setting on customizations, etc. As a consequence, this will help expand user base in the sense that most of people tend to like simple things with less unknown features to dig in.

Last but not the least, the music player mobile application is supported by a cloud server using cloud services Google's App Engine and Firebase. The server provides user authentication and is responsible for the heavy computation tasks. Specifically, an application running in App Engine plays as a server's role to verify user sign in. It also runs App Engine MapReduce jobs to consume large data stored in Google Cloud Storage and serve relatively small result about most popular songs for the app. In addition, user's comments and suggestions is automatically synchronized with Firebase which makes modifying and analyzing synchronized data really convenient.

The rest of this paper will discusses how the application is designed and implemented in Chapter 2. Chapter 3 describes how various testing is performed and presents some of the evaluation statistics. Chapter 4 presents some reflections on how the project could be better designed and implemented. It also states some of the possible directions to extend current application to be more appealing to users and takes its quality to the next level.

Chapter 2 - Implementation details

This chapter will cover the implementation details of both front end —Android application, and back end — App Engine application. Specifically, it presents what design pattern is used, and why, how classes are organized to complete a specific UI feature, how back end is designed to verify user and provide data, etc.

The front end is designed under Model View Presenter (MVP) architecture. Traditionally, most of Android application work is done by activities and fragments. However, when more functionality is introduced, activity and fragment classes become larger and larger, which makes testing and debugging difficult and hard to maintain and reuse the code. MVP pattern solves this issue by separating the logic of presenting data into different presenter classes. The View (often activities and fragments) talks to presenters through its interfaces. The new layer added leads to a decreases of coupling within the code and an increase of testability and reliability. Section 3.1 describe one example of applying MVP pattern in terms of UML diagram and how MVP makes designing unit tests easier.

Specifically, Section 2.1 – 2.3 describes how the major functionality associated with navigating and playing music is implemented. Section 2.4 shows how front end communicates with backend server and fetch data. Section 2.5 explains how the app talks to Firebase and submit synchronized data instantly. Section 2.6 involves how user sign in is implemented behind the scene and why server identifies user authentication in a secure manner, Finally, backend implementation which is primarily about design of MapReduce jobs is presented in Section 2.7.

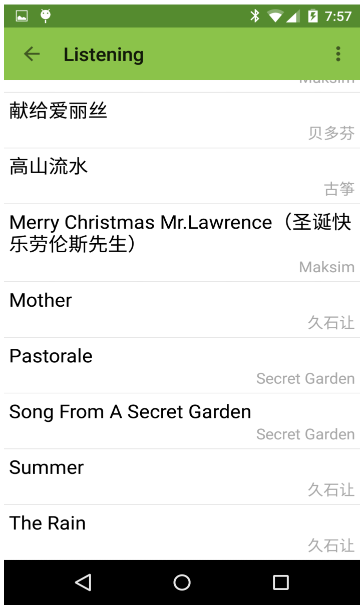
As basic UI features for this music player, navigating and playing audio files is implemented by three parts: 1. main activity with navigation drawer, which is also the home page of the application; 2. middle layer fragments that can navigate between artists, albums, and songs;

3. a separate activity, PlaySongActivity, to play a song with a music visualizer. Home page is connected with middle fragments through navigation drawer clicking, while the song playing activity is started whenever a song is clicked in any of the middle fragments that display songs' list. The following sections will cover implementation details including how they are designed, class diagrams, important pieces of code and screenshots.

2.1 Main Activity and navigation drawer

Home page (Figure 1) is loaded when the application is started. It displays all of the songs (including ringtones) in the device by default. If there are no music files, the home page will be empty. At the same time, navigation drawer (Figure 2) on the left will display four categories for the user to click. Depending on which one the user clicks, the main page will function accordingly. For example, if Albums is clicked, all the albums on the device will be listed.

Design for home page and navigation drawer is straightforward. When the application is started, queries of database will be executed, and all the audio files that are supported by Android [3] in the external storage will be listed as "songs" in the ListView. If any of these songs is clicked by the user, a new activity will be started directly to play the song. On the other hand, navigation fragment (Figure 2) is attached to the main page when the app is started. RecyclerView from Material Design is used as a container for displaying the four options that user can navigate to.



7:57

Listening

献给爱丽丝
贝多芬

高山流水
古筝

Merry Christmas Mr.Lawrence（圣诞快乐劳伦斯先生）
Maksim

Mother
久石让

Pastorale
Secret Garden

Song From A Secret Garden
Secret Garden

Summer
久石让

The Rain
久石让

Figure 1 Music Player Home page.

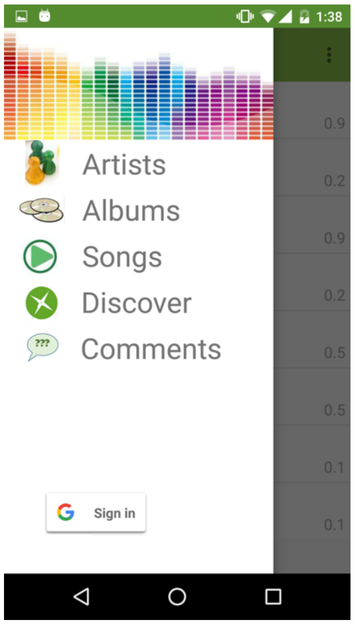


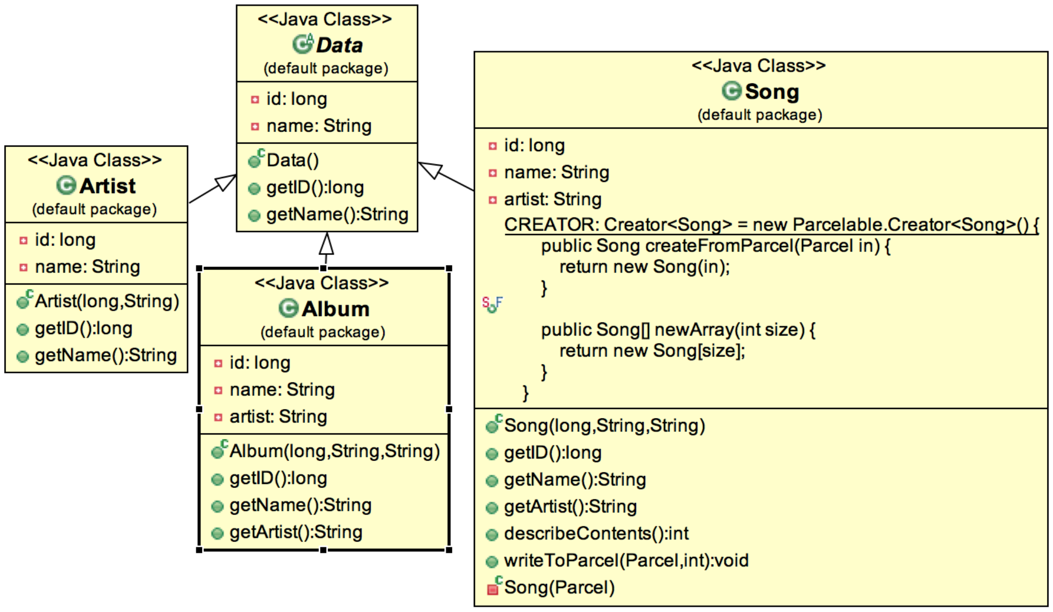
Figure 2 Navigation drawer

2.2 Middle layer fragments navigation

This section consists of two sub-sections. The first section will talk about the design of some important data structures, what their relationships with each other and how they are used in the middle layer navigations of the application. Class diagram and partial code will be presented. The second part will cover how the fragments connected to the caller activity and newly created activity, and how they connect with each other. Screenshots and important piece of code will be presented.

2.2.1 Design of data structures for various kind of data

Figure 3 shows the class diagram of the data set that is used and passed between fragments. Data is an abstract class that all the other three classes, Artist, Album, and Song, inherit from. All of these three kinds of data is essentially of type Data. They all have id, name and corresponding getters. Other than that, Song and Album classes also have a



```
CREATOR: Creator<Song> = new Parcelable.Creator<Song>() {
    public Song createFromParcel(Parcel in) {
        return new Song(in);
    }

    public Song[] newArray(int size) {
        return new Song[size];
    }
}
```

Figure 3 Class diagram for media data sets.

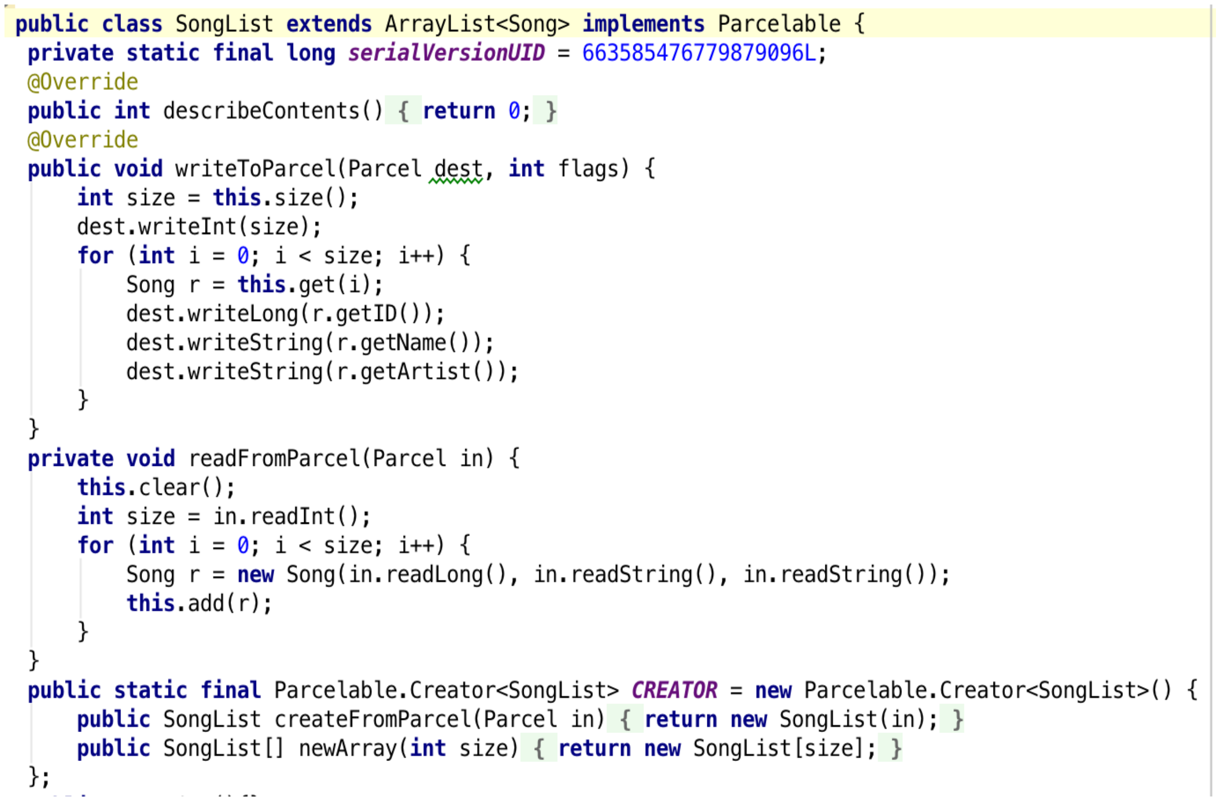
artist field to store the information that which artist owns the song or the album, while Artist class only need the default fields .

Song class also implement Parcelable interface from android.os.Parcelable which make passing instance object of Song class possible (Actually songs are passed in the form of ArrayList<Song>). The reason why Parcelable is used instead of Serialization in Standard java API, according to this website [4], is that it is well documented, faster, and creates less garbage objects. This inheritance allows the application to use dynamic type of the data structures and program with more flexibility and efficiency.

However, usually there is not only one song in the device, most of the time, an ArrayList of Songs need to be parceled and passed around. So this application use another data structure called SongList to store the song list when necessary. In order to satisfy the parcelable and passable requirements, SongList class is a subclass of ArrayList<Song>, and it also implements Parcelable (Figure 4).

Since SongList extends ArrayList<Song>, it has to save the size of the array (which gets written to the Parcel inside writeToParcel method), and write each of the Song to the parcel one by one. Similarly, in the readFromParcel method, read the size of the array first, and then read each of the song and add it to the current SongList instance object.

How are these data structures used? All of the middle layers use one fragment called SongFragment. Whenever one of the three items in navigation drawer (except "Folders", it's currently not functioning) is clicked, a new SongFragment is created. But depending on which one gets clicked, this new fragment will be fed with different data set (Figure 3) that is queried from the database on the device. The new SongFragment is then added to the main activity, or replace the old one (if there is any). Since the Artist objects only need to display one field, which is the name of the artist, the second textview in the View will be omitted.



```
public class SongList extends ArrayList<Song> implements Parcelable {
 private static final long serialVersionUID = 663585476779879096L;
 @Override
 public int describeContents() { return 0; }
 @Override
 public void writeToParcel(Parcel dest, int flags) {
     int size = this.size();
     dest.writeInt(size);
     for (int i = 0; i < size; i++) {
         Song r = this.get(i);
         dest.writeLong(r.getID());
         dest.writeString(r.getName());
         dest.writeString(r.getArtist());
     }
 }
 private void readFromParcel(Parcel in) {
     this.clear();
     int size = in.readInt();
     for (int i = 0; i < size; i++) {
         Song r = new Song(in.readLong(), in.readString(), in.readString());
         this.add(r);
     }
 }
 public static final Parcelable.Creator<SongList> CREATOR = new Parcelable.Creator<SongList>() {
     public SongList createFromParcel(Parcel in) { return new SongList(in); }
     public SongList[] newArray(int size) { return new SongList[size]; }
 };
```

Figure 4 Part of the code in SongList class.

2.2.2 Navigating between artists, albums, and songs

Figure 5 shows one of the navigating path, which is from Artist in navigation drawer to artist list in the device. Then if one of the artists is clicked, all of the albums belong to that artist is display. Therefore, user can click whichever album they like, then in the last screenshot of Figure 5, they can choose from the song list in that album, and the application will take them to the play song activity which I will talk about in the next section.

One noticeable feature of this navigating path is that the last three screenshots use the same activity which is main activity, and the same fragment which is SongFragment. One big difference is the data sets that were fed in. In the SongFragment class, there is a variable used to keep track of which situation it currently is so that different methods gets call and different queries gets executed. The data could be any of the three subclasses in Figure 3. Then they are stored in a bundle along with other tags indicating the state of fragment. used to set up a new SongFragment. This design ensures most reusable of code in SongFragment class and MainActivity class. Another difference is that for Artist list, only artist name is displayed, while in other cases, two lines of information are needed, such as an album's title and its author, a song's name and it's author.

Using back button, user can also navigate backwards. In order to navigate back from one SongFragment to previous SongFragment, two methods, onKeyUp and onBackPressed have to be overridden. Because the default behaviour won't work as expected due to there is always a blank activity without SongFragment gets push into Android back stack between SongFragments in Figure 5. So the major responsibility of these two methods is to pop one activity from the top of the back stack when the back button is clicked. In this way, there will not be blank main activity between those pages that user concerns about.

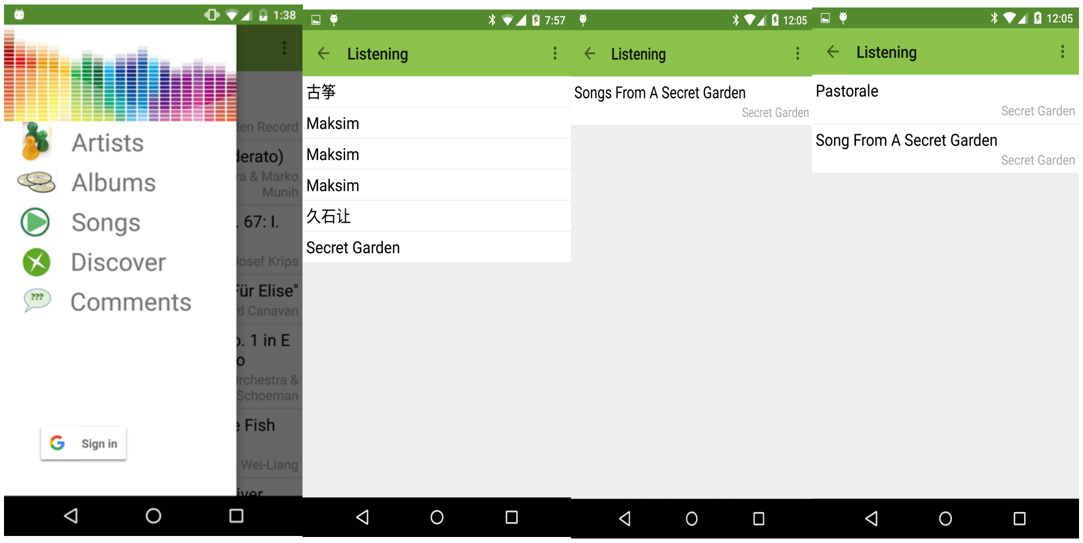


Figure 5 Navigating path when Artists is clicked in navigation drawer.

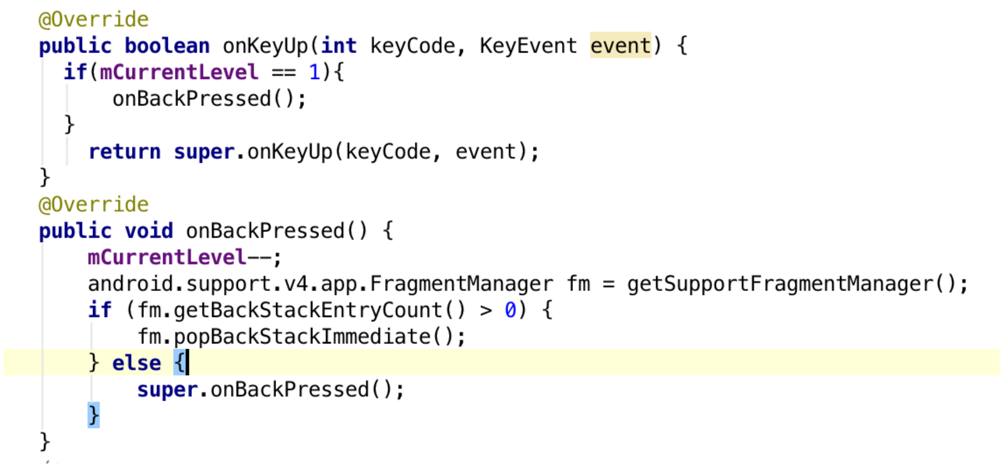
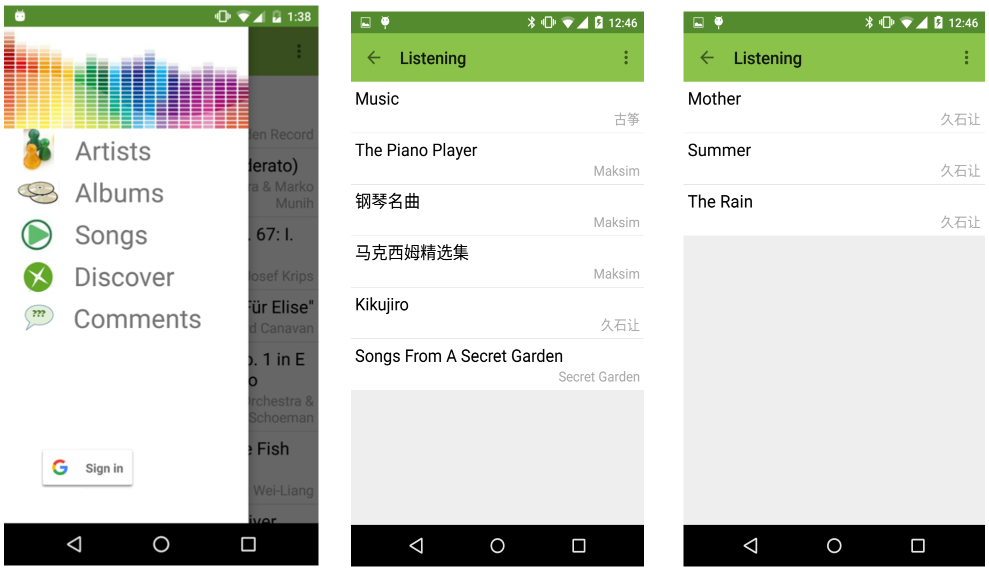

```
@Override
public boolean onKeyUp(int keyCode, KeyEvent event) {
  if(mCurrentLevel == 1){
      onBackPressed();
  }
    return super.onKeyUp(keyCode, event);
}
@Override
public void onBackPressed() {
    mCurrentLevel--;
    android.support.v4.app.FragmentManager fm = getSupportFragmentManager();
    if (fm.getBackStackEntryCount() > 0) {
        fm.popBackStackImmediate();
    } else {
        super.onBackPressed();
    }
}
```

Figure 6 Code in two methods onKeyUp and onBackPressed.

Before I articulate in detail how a song is played after it is clicked in the last screenshot of Figure 5, I want to talk about the other two navigating paths (Figure 7 and Figure 8) which are very similar as the above navigating path.

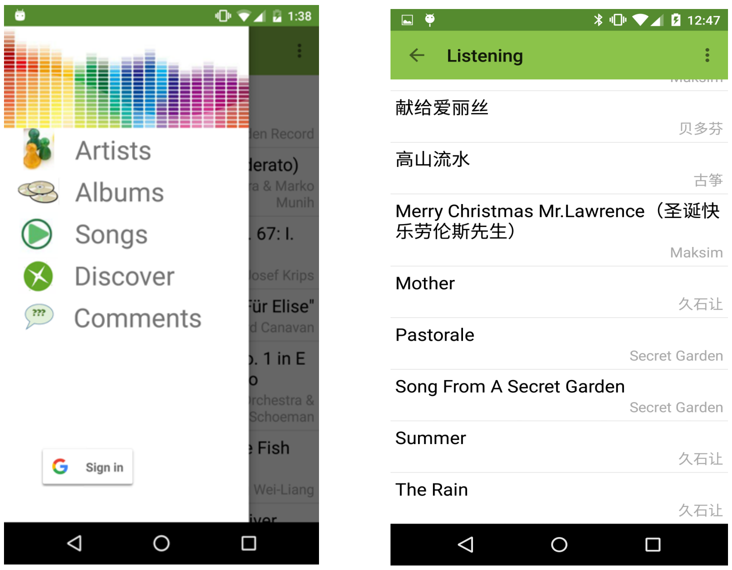
The path, Albums—album list—song list in an album—play a song (Figure 7), and path Songs—song list—play a song (Figure 8), they both use the same UI design and similar implementation as the above path. One difference is that they require executing different queries to perform different tasks: the former asks albums from the database, which means subtle difference in tags to distinguish this situation from all the others, which the later asks all of the songs from the database to display them (which looks the same as the home page when the application gets started.) Another difference depending on which case it is, the navigation will be one step less (Figure 7) or two steps less (Figure 8) to the activity that plays a song.



Music	古筝
The Piano Player	Maksim
钢琴名曲	Maksim
马克西姆精选集	Maksim
Kikujiro	久石让
Songs From A Secret Garden	Secret Garden

Mother	久石让
Summer	久石让
The Rain	久石让

Figure 7 Navigating path when Albums is clicked in navigation drawer.



献给爱丽丝	贝多芬
高山流水	古筝
Merry Christmas Mr.Lawrence（圣诞快乐劳伦斯先生）	Maksim
Mother	久石让
Pastorale	Secret Garden
Song From A Secret Garden	Secret Garden
Summer	久石让
The Rain	久石让

Figure 8 Navigating path when Songs is clicked in navigation drawer.

2.3 Playing and visualizing a song within PlaySongActivity

In the end of the above three navigating paths, if a song is clicked, a new activity—PlaySongActivity will be triggered to play and visualize the song. As the new activity is started, the song list that the trigger activity maintains gets passed to this PlaySongActivity, displayed in the listView, and be ready to play (Figure 9). As I have mentioned in the above section, the two classes SongList and Song have implemented the Parcelable interface which allows any instance object of this class can be parceled and passed between activities.

This section provides data structure design of audio data , overview of visualization implementation, as well as how PlaySongActivity works with MusicService class and MediaController class from Android. This activity is the key part of the whole application. It plays the song clicked by user, provides the music controller and the song list, and more importantly, displays the line visualizer (it’s up to the user to decide which visualization to add). Class diagram of the audio data, important piece of code and screenshots will be presented to illustrate how this activity works.

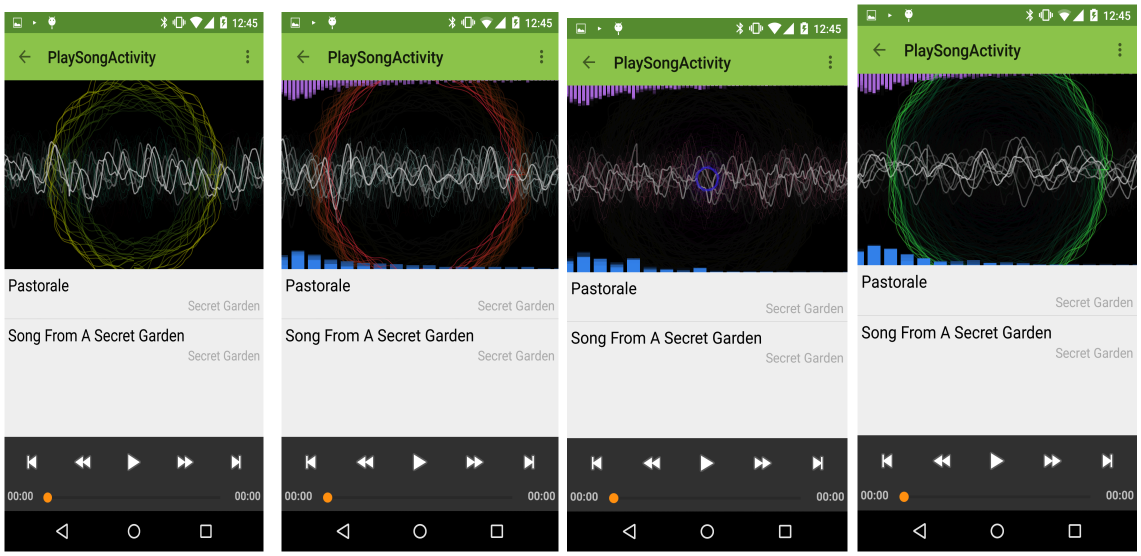


Figure 9 A song is playing with different visualization graph added.

Data structure design of audio data

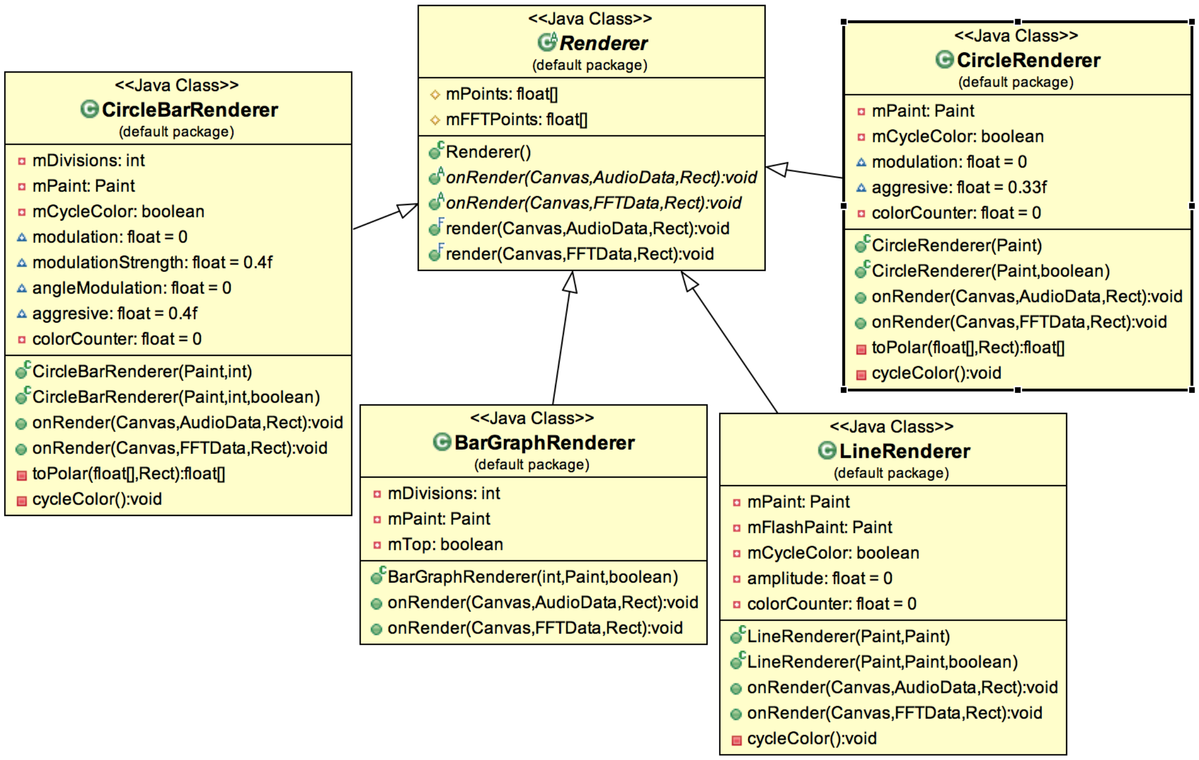
Code that is responsible for visualization comes from Felix Palmer and licensed under the MIT license [5]. The following class diagram (Figure 10) shows how they design the data structure for display audio data. Renderer is an abstract class defining onRenderer method which should be overridden by the four subclasses. Each of the four subclasses inherits Renderer class and overrides the onRenderer method. Inside each of the four onRenderer methods, either audio data (array of bytes containing the waveform data) or FFT data (array of bytes containing the frequency data) is used to draw on the canvas parameter. So after one of the four classes is constructed when necessary, they are passed to VisualizerView and inside of this View class (it extends View from Android) proper onRenderer method is called to draw visualizations on canvas.

Overview of VisualizerView class implementation

VisualizerView class is the middle class communicating between user class and the above four data classes. It captures the waveform data and FFT data through Visualizer listeners from Android, and stores the data. When user class is triggers to add a specific one visualization of the four in runtime, proper instance object of that subclass is constructed and added to visualizer view, which will be drawn using the onDraw method from Android View.

How PlaySongActivity works

PlaySongActivity is the activity that unmarshals the song list data that is passed in, starts music service, sets up media controller, constructs audio data structures and passes it to visualizer view to draw it. Firstly, when it gets created, it gets the song list and the position indicating which song is clicked, and after that starts music service. Then when the service is connected, three things must be done, 1. setting up media controller to control the media player (including show the controller on screen), 2. feeding views with song list to display them on screen, 3. adding a default visualization to visualizer view, which is line format of visualization (further visualizations are up to user to choose in the settings menu)



<<Java Class>>
Renderer
(default package)
mPoints: float[]
mFFTPoints: float[]
Renderer()
onRender(Canvas,AudioData,Rect):void
onRender(Canvas,FFTData,Rect):void
render(Canvas,AudioData,Rect):void
render(Canvas,FFTData,Rect):void
<<Java Class>>
CircleBarRenderer
(default package)
mDivisions: int
mPaint: Paint
mCycleColor: boolean
modulation: float = 0
modulationStrength: float = 0.4f
angleModulation: float = 0
aggresive: float = 0.4f
colorCounter: float = 0
CircleBarRenderer(Paint,int)
CircleBarRenderer(Paint,int,boolean)
onRender(Canvas,AudioData,Rect):void
onRender(Canvas,FFTData,Rect):void
toPolar(float[],Rect):float[]
cycleColor():void
<<Java Class>>
CircleRenderer
(default package)
mPaint: Paint
mCycleColor: boolean
modulation: float = 0
aggresive: float = 0.33f
colorCounter: float = 0
CircleRenderer(Paint)
CircleRenderer(Paint,boolean)
onRender(Canvas,AudioData,Rect):void
onRender(Canvas,FFTData,Rect):void
toPolar(float[],Rect):float[]
cycleColor():void
<<Java Class>>
BarGraphRenderer
(default package)
mDivisions: int
mPaint: Paint
mTop: boolean
BarGraphRenderer(int,Paint,boolean)
onRender(Canvas,AudioData,Rect):void
onRender(Canvas,FFTData,Rect):void
<<Java Class>>
LineRenderer
(default package)
mPaint: Paint
mFlashPaint: Paint
mCycleColor: boolean
amplitude: float = 0
colorCounter: float = 0
LineRenderer(Paint,Paint)
LineRenderer(Paint,Paint,boolean)
onRender(Canvas,AudioData,Rect):void
onRender(Canvas,FFTData,Rect):void
cycleColor():void

Figure 10 Class diagram for different kinds of visualization class.

As Figure 9 has shown, the four kinds of visualization reflects changes of audio data during song playback. However, when rendering these visualizations, colorful and beautiful shapes are predefined. In other words, they do not reflects the actual waveform or FFT audio data. For Bar visualization, only the height of the bars matters, while for Circle and Line, they are the radius of the circle and the vertical distance between the points in the line and the horizontal base line.

2.4 Implementation of Discover

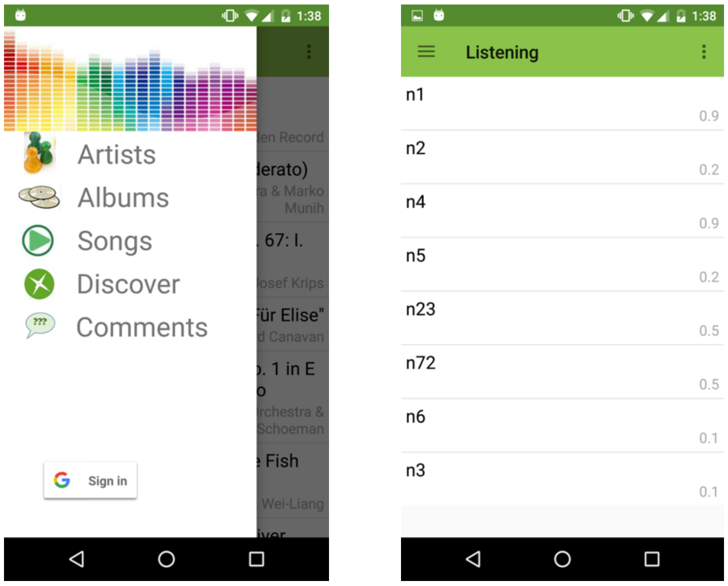
This section elaborates how Discover in navigation drawer is implemented. By clicking Discover (Figure 11), recommendation data read from the backend server is displayed in the screen. This functionality does not require authorization; all clients can get this public data through application API key used by the application. (According to google cloud platform documentation, each android application has a API key. This key uniquely identifies this application. By sharing the key with server, it can communicate with the application securely.)

The underline process that the system go through includes the following two steps,

(1) create new thread to handle new asynchronous task using FetchData class with AsyncTask as a helper class. It performs background operations and publish results on the UI thread with minimum handling of threads.

(2) send HTTP requests to server, and wait for results. UI thread is the listener who listens to the result of the request. If result returned, UI will be updated. If any exceptions happen, error will be handled and UI section to be updated will not be empty. One of the possible errors that might happen is caused bad networking connection. If this happens, error message will be shown.

As backend server runs in the cloud, App Engine applications makes it easy to deal with scaling. App Engine applications run in a secure, reliable environment that is independent of the hardware and operating system. When the traffic demands are heavy, for example, the problem is easily solved by changing the configuration file and redeploying the applications. While on the other hand, if automatic scaling is set up, then the server will automatically handle the scaling problem without any changing of the existing code. Also, asynchronous task queues are created automatically to perform work outside the scope of a request.



n1	0.9
n2	0.2
n4	0.9
n5	0.2
n23	0.5
n72	0.5
n6	0.1
n3	0.1

Figure 11 Navigation of Discover

2.5 Implementation of Comment

User submitting and deleting comments is another important feature of this project. It was implemented using Firebase with App Engine standard environment, which make real-time data synchronization easy and automatic.

Besides google cloud storage, Firebase is used as another backend service for backing up and modify synchronized data. It is a powerful platform for building applications offering real-time data storage and synchronization and user authentication. Firebase is not part of google cloud platform. It makes development easy by providing a web UI where specification of authentication and validation could be done without writing code. It also allows other backend services, such as App Engine, run periodically to process or analyze the stored data programmatically.

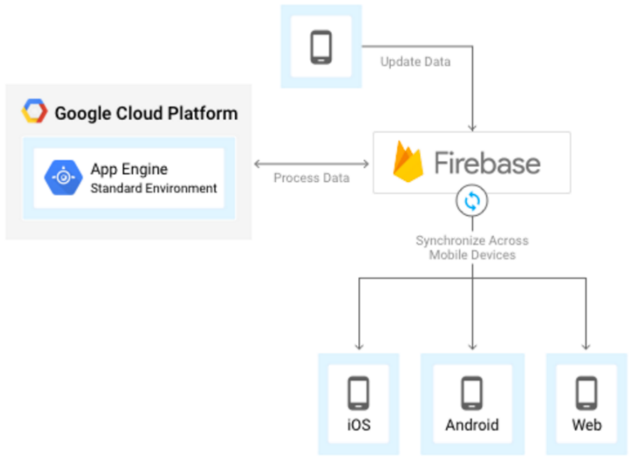
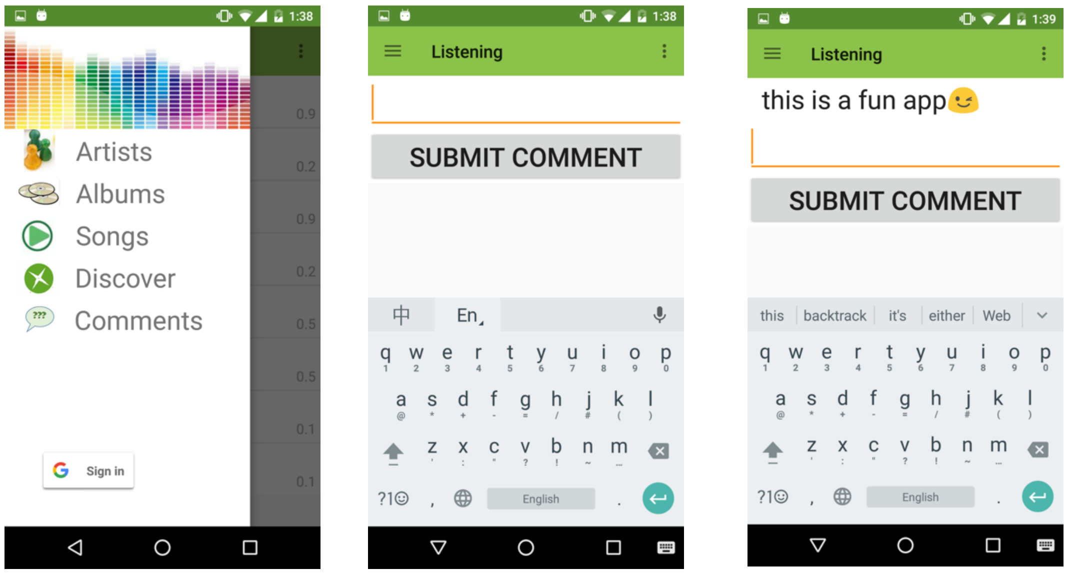


Figure 12 Firebase & App Engine standard environment.

In this project, Firebase is used as backend service to host user’s comments and suggestions. As shown in Figure12, Firebase is connected to both Android app and App Engine. So data is synchronized between the app and Firebase based on which App Engine retrieve data from Firebase and do its data analysis work. By clicking Suggestions in Navigation drawer (Figure 13), users will be directed to a page where they can submit messages (by clicking the submit button) and delete submitted messages (by long clicking a submitted comment). The action of submission and deletion will be reflected on both Firebase and Android app immediately. The automatic synchronization is handled by Firebase API.



this is a fun app😉

Figure 13 Navigation of Comments

2.6 Implementation of User Authentication

This application only provides google default sign in. In order to sign in, user has to have a google account beforehand. Figure 14 shows what UI looks like when user signs in and signs out. The first screenshot in Figure 14 shows the navigation drawer. After sign in button is clicked, a dialogue is shown as in the second screenshot. Once user approves, user's name, email address and sign out button are shown as in the third screenshot. Finally, it returns to the original state as before if sign out is clicked as in the last screenshot.

Android applications is required to provide an OAuth 2.0 token with the HTTP request for google default sign in. The process of authorizing requests with OAuth 2.0 is as follows,

(1) when the Android application is created, it has a client ID and a client secret with it.

(2) When user try to sign in, the application asks Google for a particular scope of access (such as google drive, google plus).

(3) user is directed to a chooser screen, asking s/he to choose one of the existing accounts or add a new account.

(4) if the user approves the sign in, then the application can get an access token.

(5) the application sends requests with the access token

(6) the server uses google sign in API to validating the request. If it determines that they are valid, that requested data is returned.

Each app engine application has an OAuth 2.0 client ID that is created automatically. This client ID is shared between Android application and the app engine application. In step (4), after the user has successful signed in, the client ID token is sent to the server through HTTP requests. Then in the app engine server side, it verifies the integrity of the ID token to ensure the secure communication between front end application and back end server.

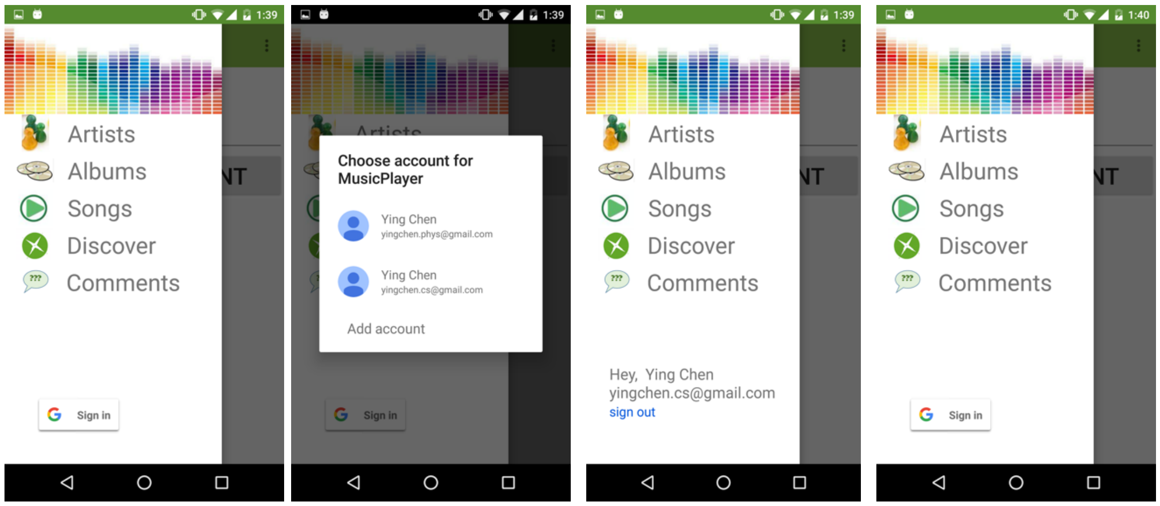


Figure 14 User sign in and sign out

2.7 Backend implementations

This project utilizes two backend services, one is Firebase which is explained in Section 2.4, the other one is App Engine, which will be discussed in this section.

App Engine is an environment for running application code in the cloud [9]. It is a Platform-as-a-Service cloud serving designed for distributed web applications. It manages the implementation details about maintaining virtual servers and operating systems which allows developers focus on coding and away from system administration.

This project uses App Engine standard environment that monitors, updates, and scales the hosting environment based on the customized settings. The App Engine application serves data for clients by running MapReduce jobs. All input, intermediate and output data are stored in cloud storage.

Besides the default bucket that App Engine applications created in cloud storage while running, all the input and output files are also hosted in cloud storage. Google cloud storage makes retrieval of any amount of data at any time easier. It is used as a service with reliable data storage and large data objects distribution.

2.7.1 Introducing App Engine MapReduce framework

MapReduce is a programming model used to processing and generating large data in a parallel and distributed manner. App Engine MapReduce libraries [10] allows running automatic scaling MapReduce jobs without developing worrying about partitioning input, scheduling execution on machines, handling failures, etc.

A MapReduce job consists of three stages as shown in Figure 15, map, shuffle, and reduce. They are run sequentially. Temp Results shown in the picture is intermediate data. It's temporarily stored between stages.

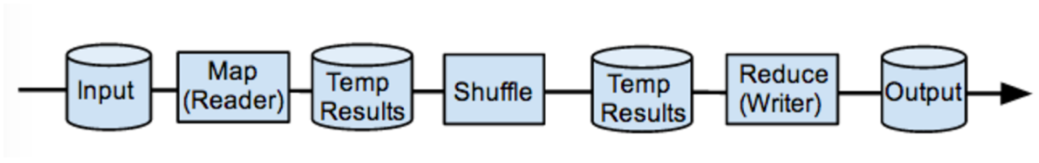


Figure 15 App Engine MapReduce Stages (Picture comes from [8])

The map stage reads input using an input reader one records at a time. A map functions is needed to be implemented, and the map function will be applied to each of the records. While in the reduce stage, reduce function receives a key and a list of values associated with that key, override the reduce function to do customized job, and then emit the output to the output writer.

Figure 16 shows that how sharding works during the whole process of MapReduce job. Firstly, large input files will be divided into multiple data sets (Shards). Each shard then is processed in parallel. For a particular shard, it is handled by a separate instance of Mapper class and processed sequentially. The shard is then divided into slices, and each slice is consumed with a map function call. Similarly, sharding and slicing happens for intermediate data for preparation for reducer function call.

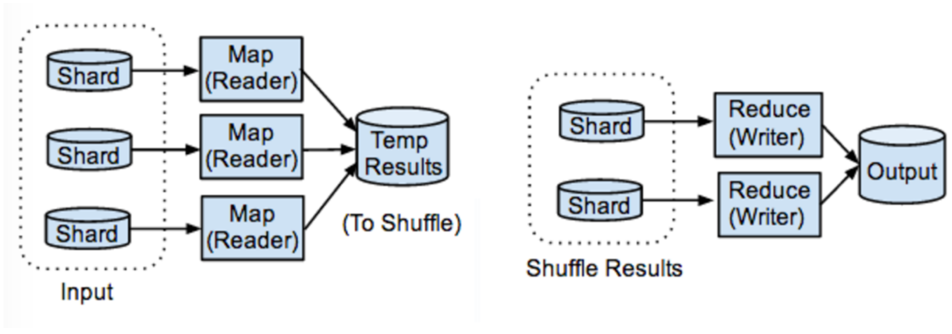


Figure 16 Sharding Input and intermediate data (picture comes from [8])

In this project, the input and output class of MapReduce jobs allowing user defined number of shards and instances running in the cloud. The combination of the input data size, number of shards, and how many instances used can be tricky. It determines directly whether the job can be process or not, and influences efficiency dramatically. The details with data analysis of performance will be discussed in Section 4.

2.7.2 MapReduce chaining jobs

Section 3.3 introduces that Android app can requests recommendation data from backend and displayed it. In this section, I will present in detail how the chaining MapReduce jobs are designed to obtain desired data while taking advantage of the implementation of the framework and making jobs run efficiently.

The App Engine application runs two MapReduce jobs when triggered by a button click event. It was not designed to run frequently as the MapReduce jobs take long time to finish when input files are in Gigabytes scale or even larger. It is suggested to run periodically when new data comes in a regular basis.

Before explaining how the two MapReduce jobs are designed, it is a good preparation to see how the input are formatted. Input is fake data generated for testing purposes. Input files are plain text files with each record in a separated line. A record has 15 fields separated by comma. Each field is a key value pair. The following is an example of a record, (it is one line in the input file.)

Id 60 , namenamename reinvent revolutionary web services , loudness 0.26 , artist Lillian Peterson , duration 445 , year 1979 , tempo 319.166 , trackID 3542266826238533 , hotnesshotnesshotness 48.95 , producer Goodwin and Sons , description1 “some description”, description2 "some description" , description3 “some description”

The records are sorted by unique Ids, but the value associated with a specific Id could be duplicated many times. The purpose of the two MapReduce jobs is to add up the hotness value of the same song with the same name, sort the records by the sums of hotness, and save all of the records in sorted order.

In order to get top hottest songs in the input files, the first MapReduce job is designed to get the sum for each song (For simplicity, assume that all of the song names are different with each other), and the second MapReduce job simply read the output of the first one, sort records by the hotness sum, collect all songs with the same hotness, and write result in the output file.

To be more specific, the input, Mapper, Reducer and output for each of the two MapReduce jobs are presented below.

MapReduce job 1:

Input: file in google cloud storage. GoogleCloudStorageLineInput is used as the input reader. It is provided by the App Engine team. With the specification of the bucket name, file name, line separator, and number of shards, it reads the file in cloud storage and do the sharding and slicing job automatically.

Mapper: GetHotnessSumMapper is the mapper class. It extends the mapper class provided by App Engine and overrides the map method. The map method reads one line at a time, parse it, emit the name of song and its hotness rate if the line contains the information.

Reducer: GetHotnessSumReducer is the reducer class which extends the reducer class provided by the developer and overrides the reduce method. The reduce method gets all hotness rates for one particular song, it adds them up, and emit the sum as key, and the name of the song as value.

Output: output of the first MapReduce job is a file in google cloud storage for later accessing and analyzing. GoogleCloudStorageLineOutput is used in this project as it satisfies the purpose of this specific scenario.

MapReduce job 2:

Input: the input file of the second job is the output of the first one, and the input reader class is GoogleCloudStorageLineInput.

Mapper: The mapper class of the second job is GetHotnessRankMapper. The map method inside it reads a line containing the hotness rate of the song and its name. It simply parses the line and emit hotness rate as key and song name as value. Since the default implementation of Mapper class in App Engine MapReduce framework sorts its output by key. So the output of this mapper will be sorted in ascending order.

Reducer: The reducer class of the second job is GetHotnessRankReducer. The reduce method inside it get the hotness rate and a list of songs with the same hotness rate. It simply emits the hotness rate as key and the names as value.

Output: file in google cloud storage containing list of songs sorted by rate of hotness. The output class used is GoogleCloudStorageLineOutput. It takes care of accessing cloud storage and writing files in it.

These classes (customized classes) and the MapReduce job specifications (defined in GetRankChainingMapReduceJob) are written for specific formatted input data. It may not work perfectly on other formatted data. The testing and performance analysis of the chaining jobs are discussed in the next section.

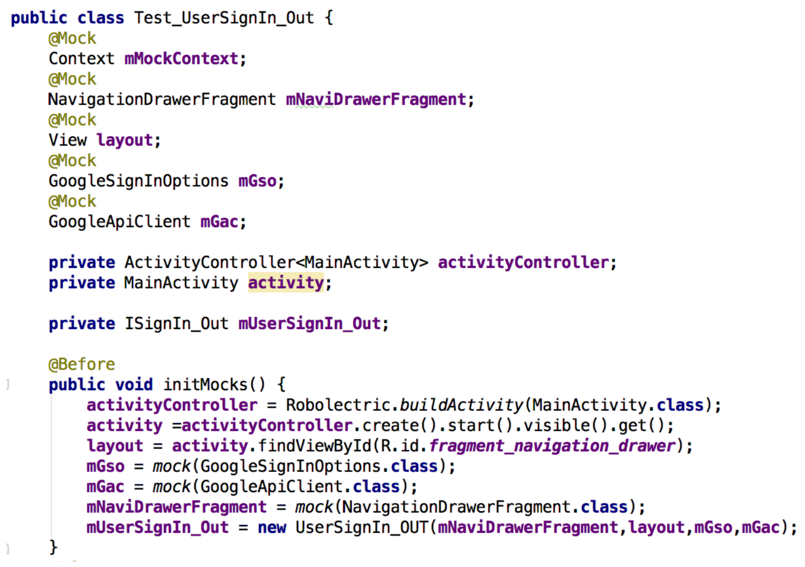
Chapter 3 - Testing and Performance

This chapter illustrates testing of frontend and backend and the performance of the whole system. Specifically, local unit test that runs on JVM and instrumented unit test that run on device/emulator are used to unit test the frontend. Based on that, system level test is also performed to ensure that all the components work well together. In this project, both local unit tests and instrumented unit tests are used to unit test front end features. While the system level test is easily performed by utilizing instrumented test and existing UI test tools provided by Android. Then the backend section is mainly about how MapReduce jobs are unit tested, and how it performs when being put together and fed relatively large-scale data.

Android tests [11] are based on JUnit. They can be run either as local unit tests on JVM or on Android device/emulator. Local unit tests are JUnit test classes that run on regular JVM and have no Android dependencies. The methods inside those test classes are supposed to test small pieces of the system in an isolated manner. They are fast because there is no overhead of loading code onto devices or emulator. Instrumented tests, on the other hand, are used to test UI features by simulating user interactions. As built into Android APK, they can invoke Android API methods while running on devices or emulators.

3.1 Unit testing

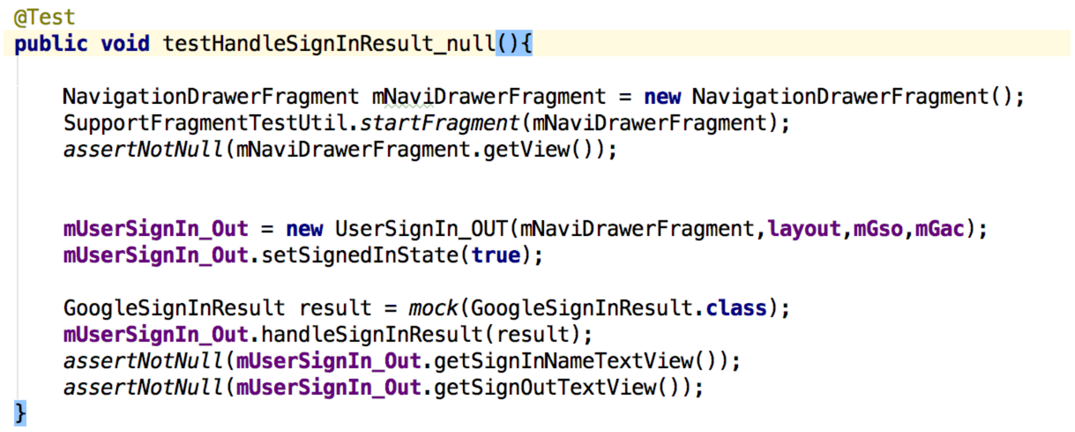
Local unit test is easy and fast when using Android testing libraries JUnit, along with Mockito [12] and Robolectric [13]. Mockito allows mocking Android SDK class objects with default values when necessary. Whilst Robolectric rewrites Android SDK classes as they are being loaded. By using these testing framework to simulate different system state, unit tests can be fast and cover various testing cases. Figure 17 shows an example of how to use Mockito and Robolectric API to mock Android classes and test the class UserSignIn_Out. In the example, in order to build an instance of UserSignIn_Out and an instance of activity for later usage, a GoogleSignInOption object, a GoogleApiClient object and a NavigationDrawerFragment object are mocked, then passed into the constructor of UserSignIn_Out to construct a new instance.



```
public class Test_UserSignIn_Out {
    @Mock
    Context mMockContext;
    @Mock
    NavigationDrawerFragment mNaviDrawerFragment;
    @Mock
    View layout;
    @Mock
    GoogleSignInOptions mGso;
    @Mock
    GoogleApiClient mGac;

    private ActivityController<MainActivity> activityController;
    private MainActivity activity;

    private ISignIn_Out mUserSignIn_Out;

    @Before
    public void initMocks() {
        activityController = Robolectric.buildActivity(MainActivity.class);
        activity =activityController.create().start().visible().get();
        layout = activity.findViewById(R.id.fragment_navigation_drawer);
        mGso = mock(GoogleSignInOptions.class);
        mGac = mock(GoogleApiClient.class);
        mNaviDrawerFragment = mock(NavigationDrawerFragment.class);
        mUserSignIn_Out = new UserSignIn_OUT(mNaviDrawerFragment,layout,mGso,mGac);
    }
```

Figure 17 Example code showing the initiation of particular test class

While in the following test methods which is targeting to test the method handleSignInResult in UserSignIn_Out class (Figure 18). In order to call this method on the object, the mocked NavigationDrawerFragment object is not enough as (some of the basic methods are not mocked in Mockito). So using Robolectric API to build a new instance becomes necessary. More unit tests organized in Table 1 will be discussed in the next section.



```
@Test
public void testHandleSignInResult_null(){

    NavigationDrawerFragment mNaviDrawerFragment = new NavigationDrawerFragment();
    SupportFragmentTestUtil.startFragment(mNaviDrawerFragment);
    assertNotNull(mNaviDrawerFragment.getView());


    mUserSignIn_Out = new UserSignIn_OUT(mNaviDrawerFragment,layout,mGso,mGac);
    mUserSignIn_Out.setSignedInState(true);

    GoogleSignInResult result = mock(GoogleSignInResult.class);
    mUserSignIn_Out.handleSignInResult(result);
    assertNotNull(mUserSignIn_Out.getSignInNameTextView());
    assertNotNull(mUserSignIn_Out.getSignOutTextView());
}
```

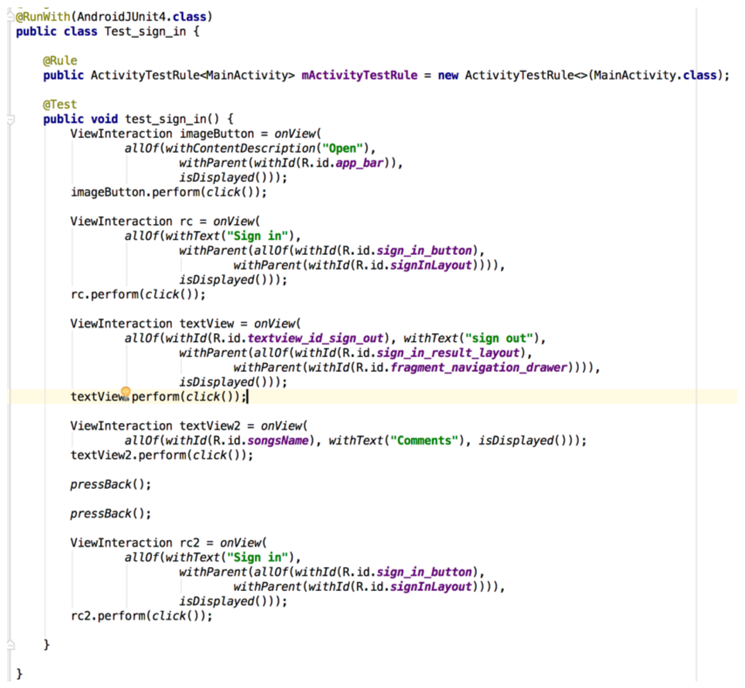
Figure 18 Example code showing test of particular test class (UserSignIn_Out)

3.2 System tests and performance analysis

System tests are tests that aim to figure out whether the system as a whole fulfill the functionality as expected. It's a black box testing technique so that the assumption of no knowledge of the underlining implementation is made. It usually is conducted to evaluate how the system behaves in some common scenarios as well as some uncommon cases where the system might be broken. For the system level testing in this project, UI testing and Usability testing are performed to figure out the defects of the system. UI testing is discussed in section 3.2.1 with reasons why some existing testing frameworks are suitable for Android UI testing and why some unit tests along with UI tests could ensure a specific functionality work correctly in most of the use cases. Section 3.2.2 presents performance analysis based on Android Studio tool Traceview and small-scale Usability testing.

3.2.1 UI testing

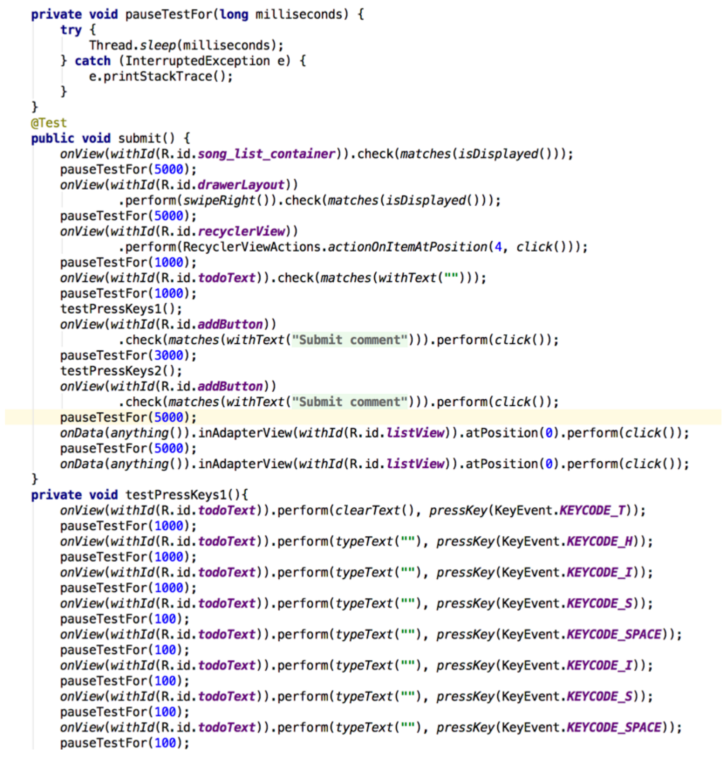
UI tests are used to verify correct UI output after a sequence of user interactions on a device. It is an end to end user perspective testing executed for real life scenarios. In Android, UI testing framework Espresso [14] allows programmatically simulating user actions and testing intra-app user interactions. Espresso also provide automatic testing recording function that could generate testing code as navigating UI features, which makes UI testing really easy. Specially, the new feature from Espresso Test recorder allows developers to click through the app UI, and the recorder will generate the testing code accordingly. On the other hand, UI Automator [15] supports cross-app interactions which is really useful when testing the Google account chooser in this project. UI Automator allows developers to write tests to perform interactions on user apps and system apps. This is used when testing user sign in where a chooser dialogue is displayed.



```
@RunWith(AndroidJUnit4.class)
public class Test_sign_in {

    @Rule
    public ActivityTestRule<MainActivity> mActivityTestRule = new ActivityTestRule<>(MainActivity.class);

    @Test
    public void test_sign_in() {
        ViewInteraction imageButton = onView(
                allOf(withContentDescription("Open"),
                        withParent(withId(R.id.app_bar)),
                        isDisplayed()));
        imageButton.perform(click());

        ViewInteraction rc = onView(
                allOf(withText("Sign in"),
                        withParent(allOf(withId(R.id.sign_in_button),
                                withParent(withId(R.id.signInLayout)))),
                        isDisplayed()));
        rc.perform(click());

        ViewInteraction textView = onView(
                allOf(withId(R.id.textview_id_sign_out), withText("sign out"),
                        withParent(allOf(withId(R.id.sign_in_result_layout),
                                withParent(withId(R.id.fragment_navigation_drawer)))),
                        isDisplayed()));
        textView.perform(click());

        ViewInteraction textView2 = onView(
                allOf(withId(R.id.songsName), withText("Comments"), isDisplayed()));
        textView2.perform(click());

        pressBack();

        pressBack();

        ViewInteraction rc2 = onView(
                allOf(withText("Sign in"),
                        withParent(allOf(withId(R.id.sign_in_button),
                                withParent(withId(R.id.signInLayout)))),
                        isDisplayed()));
        rc2.perform(click());

    }

}
```

Figure 19 Example code showing the generated code by Espresso Test Recorder.

Figure 19 shows the automatically generated code using Espresso Test Recorder. However, auto-generated test code does not always satisfy specific testing purposes. In that case, modifying or writing customized testing code allows more flexibility. One example is shown in Figure 20.



```java
private void pauseTestFor(long milliseconds) {
    try {
        Thread.sleep(milliseconds);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
@Test
public void submit() {
    onView(withId(R.id.song_list_container)).check(matches(isDisplayed()));
    pauseTestFor(5000);
    onView(withId(R.id.drawerLayout))
            .perform(swipeRight()).check(matches(isDisplayed()));
    pauseTestFor(5000);
    onView(withId(R.id.recyclerView))
            .perform(RecyclerViewActions.actionOnItemAtPosition(4, click()));
    pauseTestFor(1000);
    onView(withId(R.id.todoText)).check(matches(withText("")));
    pauseTestFor(1000);
    testPressKeys1();
    onView(withId(R.id.addButton))
            .check(matches(withText("Submit comment"))).perform(click());
    pauseTestFor(3000);
    testPressKeys2();
    onView(withId(R.id.addButton))
            .check(matches(withText("Submit comment"))).perform(click());
    pauseTestFor(5000);
    onData(anything()).inAdapterView(withId(R.id.listView)).atPosition(0).perform(click());
    pauseTestFor(5000);
    onData(anything()).inAdapterView(withId(R.id.listView)).atPosition(0).perform(click());
}
private void testPressKeys1(){
    onView(withId(R.id.todoText)).perform(clearText(), pressKey(KeyEvent.KEYCODE_T));
    pauseTestFor(1000);
    onView(withId(R.id.todoText)).perform(typeText(""), pressKey(KeyEvent.KEYCODE_H));
    pauseTestFor(1000);
    onView(withId(R.id.todoText)).perform(typeText(""), pressKey(KeyEvent.KEYCODE_I));
    pauseTestFor(1000);
    onView(withId(R.id.todoText)).perform(typeText(""), pressKey(KeyEvent.KEYCODE_S));
    pauseTestFor(100);
    onView(withId(R.id.todoText)).perform(typeText(""), pressKey(KeyEvent.KEYCODE_SPACE));
    pauseTestFor(100);
    onView(withId(R.id.todoText)).perform(typeText(""), pressKey(KeyEvent.KEYCODE_I));
    pauseTestFor(100);
    onView(withId(R.id.todoText)).perform(typeText(""), pressKey(KeyEvent.KEYCODE_S));
    pauseTestFor(100);
    onView(withId(R.id.todoText)).perform(typeText(""), pressKey(KeyEvent.KEYCODE_SPACE));
    pauseTestFor(100);
```

Figure 20 Example code showing modifying auto-generated test code using Espresso.

Table 1 displays three directions that testing three kinds of primary functionality in system level. For each of the test in a row, it shows the test steps and corresponding UI outcome, along with what UI tests and unit tests have been performed to ensure the function works as expected.

First, when testing clicking Discover in navigation drawer, user clicks Discover, then what is expected in UI is that, navigation drawer closes, data retrieved from server appears in screen. During this process, what happens behind the scene is that a new AsynTask is created to run HTTP requests and update UI thread when it’s done. The UI test class is Test_Discover_in_Nava_Drawer, and all the related unit tests are in Test_Discover. In Test_Discover, some example unit tests include, testing whether the correct view is displayed after clicking Discover, whether AsynTask is called, whether UI updating method is call after returning from server, etc.

Scenarios	Test case Steps	Outcome Expected	UI test	Unit test
Discover	User clicks Discover in the Navigation drawer	Navigation drawer closes after the click. Data retrieved from server is displayed on screen.	Test_Discover_in_Navi_Drawer	Test_Discover: shouldReturnRequestsByRule_MatchingMethod testItemClicked testOnResume
Suggestion	User clicks Suggestion in the Navigation drawer. Navigation drawer closes after the click. User type in TextView, click submit button User long clicks a submitted comment in the ListView	List of existing comments appears on the screen (if any) TextView displays. Submit button appears. Content in the TextView appears in the ListView Content in the TextView is saved in Firebase. Item long clicked is deleted in the ListView and Firebase	Test_Comment_In_Navi_Drawer	Test_Comment: testItemClicked testOnResume test_firebaseInit_1_listview_MainActivity test_firebaseInit_2_buttondelete_MainActivity test_initSong_MainActivity test_initSong_songCliked_MainActivity test_initSong_default_no_bundle_MainActivity testSuggestion
User authentication	User clicks sign in button in the navigation drawer. User click/add a google account User click sign out	A chooser is displayed for choose existing account(s) or add a new one If approved, sign in button disappear, sign out is displayed, display user name and email in navigation drawer If denied, show toast of unsuccessfully sign in message Sign out disappears, user name and email disappears, sign in button is displayed.	test_sign_in Test_Sign_In_Button Test_sign_out Test_user_signIn_signOut	Test_UserSignIn_Out: testHandleSignInResult_null() testHandleSignInResult_nonNull testCleanUp testNavigationDrawerFragment testResolvingError testSignInLayout testSignInResultLayout other getter, setter tests

Table 1 Summary of System level test on three primary features.

Secondly, when testing Suggestion function, user clicks Suggestion in navigation drawer, drawer closes, then a ListView with existing comments (if any) should be displayed, with an empty TextView and a submit button. Cursor should be placed in the TextView. Then user can perform different sequences of actions, such as enter comments and submit, long click on a submitted comment to delete it. UI outcome follows the first action should be, a new item is added to the ListView and a new object is added to Firebase instantly. Accordingly, UI outcome for the second action is expected to be, both ListView and Firebase delete the selected comment. Unit tests include testing of the related pieces of code are called, and corresponding views are displayed. For Firebase testing, the outcome should be checked by going to the connected Firebase online account and check if the objects are added or deleted as expected.

Lastly, user authentication testing involves the consistency of the sign in or sign out states. When user sign in approves, tests should include testing the sign in states remains for all of the following user actions until sign out is clicked. UI tests should focus on testing whether sign in state is consistent for different navigations. And unit test makes sure handleSignInResult method should be called for different input cases, and the view, such as the TextView, ListView, should be displayed or hidden as expected. Similarly, after user signs out, the sign out state should be consistent afterwards, and beside the similar tests as signing in, testing of clean up methods are necessary.

3.2.2 Performance Test

Performance tests focuses on all of the possible navigations by running on Nexus 5 device (Android version 5.1 and 5.1.1), testing in both cases of no data and small amounts of data, two other people test basic functionalities and user-interface. This section will illustrate in details how each of these tests are designed and performed.

Navigation paths tests aim to test the navigations between fragments in Figure 21, Figure 22, Figure 23 and Figure 24, both from left to right and from right to left. Testing of this part involves testing all of the possible ways that user can go, such as navigate from left to right, halfway to the right and then navigate back, etc.

This application is not meant to run large-scale data (anything bigger than gigabyte-size). When there are no audio files in the test device, no item will be listed on the home page, or any other pages that user can go from navigation drawer. Small data less than one gigabyte or at the gigabyte level would work smoothly. One reason is that this mobile application is designed to work on mobile phones with gigabit-size space. Space is limited if no remote databases or any other external storage are involved. Another reason is that this application use SQLite API provided by Android, which works well on traditional data management.

In terms of speed, usually it is fast enough to do local database operations [6]. Currently, only read operations are involved in this application, no write operations (Further development may use write operations), which makes it even faster. So when dealing with data which could be stored locally in the mobile device, this application performs well. But when data becomes big, it might be a bit slow (say, user waiting time approaches 5s). In that case, it is always safe to use other methods to speed up SQLite queries. [7]

Other than speed testing, the memory space is also an important part in testing phase. As an example, Figure 21- 24 show how memory space changes when the application is running on Nexus 5 for the newest Android API (megabytes' data is tested). These pictures are captured using Android Studio's Memory Monitor when the application is running. Figure 21 shows that memory usage roughly remains the same after application gets running until a song is playing in a newly started activity, which is shown in Figure 22. As the song is playing, there is no big change with memory usage. But if new visualizations are added, then memory usage goes up about 15 MB in a short time, and then drops back to previous level (Figure 23). As the song continues playing and the visualizations continue showing the audio data, memory usage has little ups and downs along with time, but overall remain at a certain level (Figure 24).

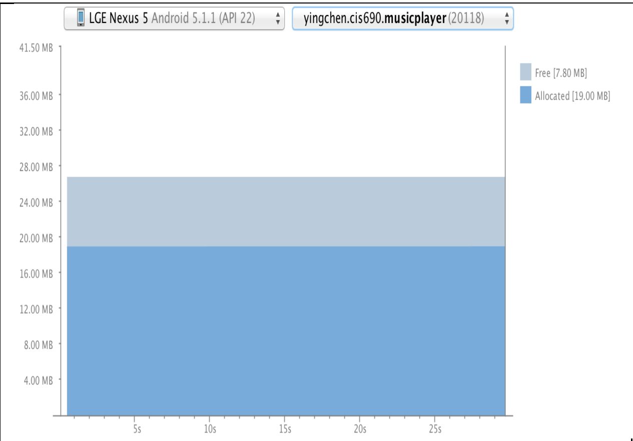


Figure 21 Memory used when only home page is shown.

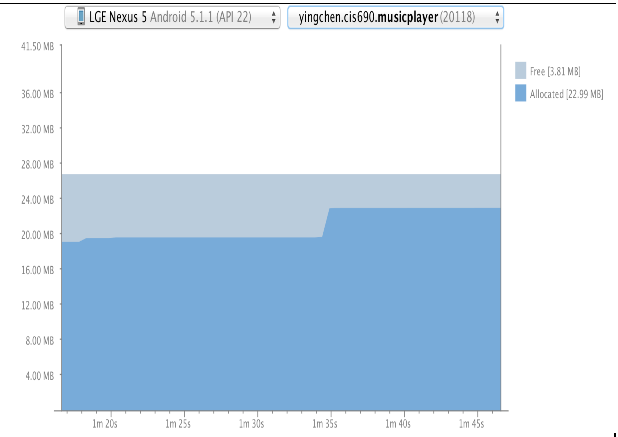


Figure 22 How memory used changes when a song is playing with only default line visualization.

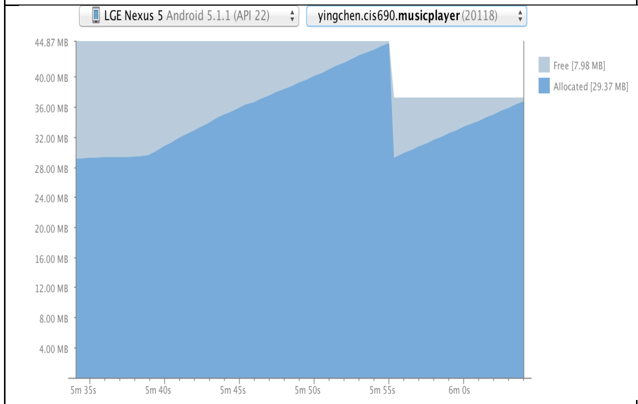


Figure 23 Memory used changes when circle bar visualization is added to screen at the first time.

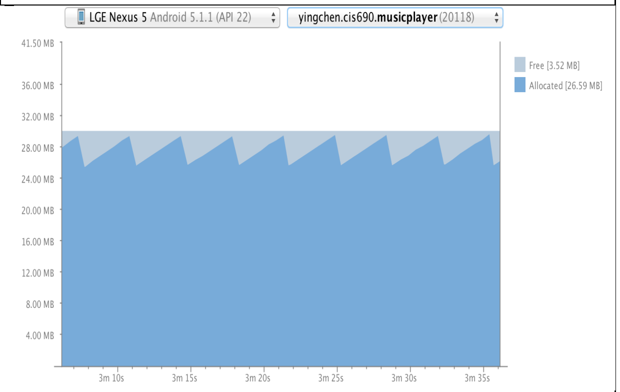
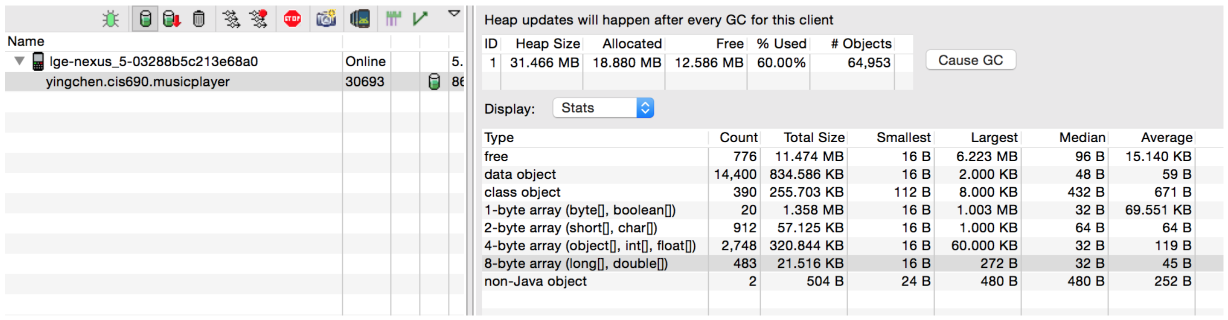


Figure 24 Memory used changes after circle bar visualization is added.



Name			
lge-nexus_5-03288b5c213e68a0	Online		5.
yingchen.cis690.musicplayer	30693		86

Heap updates will happen after every GC for this client

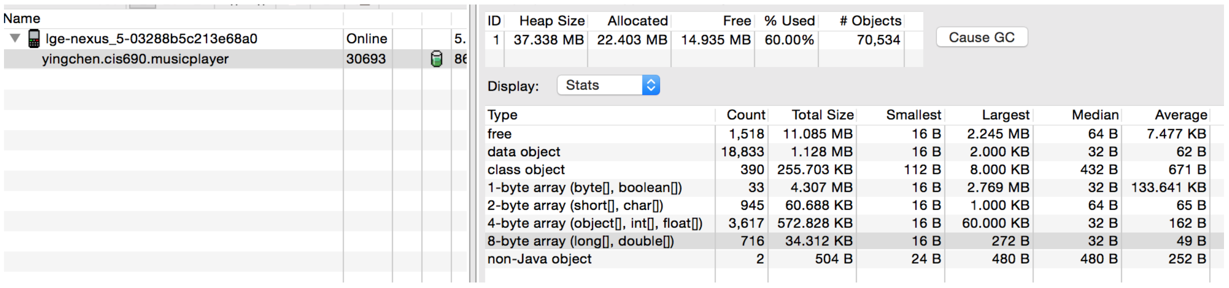
ID	Heap Size	Allocated	Free	% Used	# Objects
1	31.466 MB	18.880 MB	12.586 MB	60.00%	64,953

Cause GC

Display: Stats

Type	Count	Total Size	Smallest	Largest	Median	Average
free	776	11.474 MB	16 B	6.223 MB	96 B	15.140 KB
data object	14,400	834.586 KB	16 B	2.000 KB	48 B	59 B
class object	390	255.703 KB	112 B	8.000 KB	432 B	671 B
1-byte array (byte[], boolean[])	20	1.358 MB	16 B	1.003 MB	32 B	69.551 KB
2-byte array (short[], char[])	912	57.125 KB	16 B	1.000 KB	64 B	64 B
4-byte array (object[], int[], float[])	2,748	320.844 KB	16 B	60.000 KB	32 B	119 B
8-byte array (long[], double[])	483	21.516 KB	16 B	272 B	32 B	45 B
non-Java object	2	504 B	24 B	480 B	480 B	252 B

Figure 25 Heap size and memory allocated table before a song is played.



Name			
lge-nexus_5-03288b5c213e68a0	Online		5.
yingchen.cis690.musicplayer	30693		86

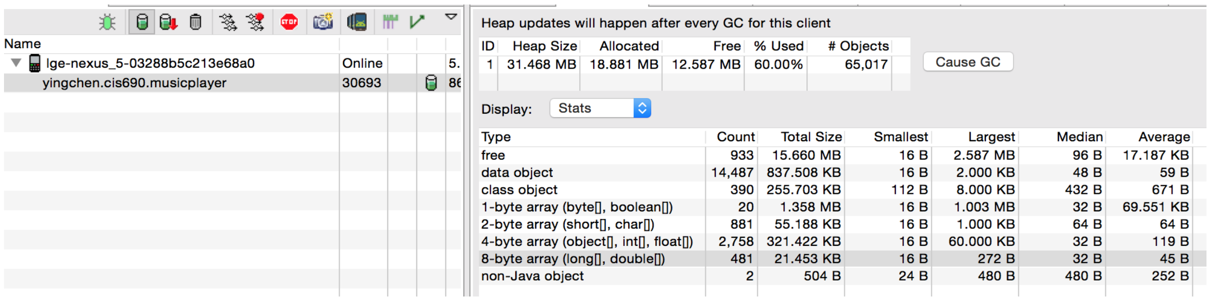
ID	Heap Size	Allocated	Free	% Used	# Objects
1	37.338 MB	22.403 MB	14.935 MB	60.00%	70,534

Cause GC

Display: Stats

Type	Count	Total Size	Smallest	Largest	Median	Average
free	1,518	11.085 MB	16 B	2.245 MB	64 B	7.477 KB
data object	18,833	1.128 MB	16 B	2.000 KB	32 B	62 B
class object	390	255.703 KB	112 B	8.000 KB	432 B	671 B
1-byte array (byte[], boolean[])	33	4.307 MB	16 B	2.769 MB	32 B	133.641 KB
2-byte array (short[], char[])	945	60.688 KB	16 B	1.000 KB	64 B	65 B
4-byte array (object[], int[], float[])	3,617	572.828 KB	16 B	60.000 KB	32 B	162 B
8-byte array (long[], double[])	716	34.312 KB	16 B	272 B	32 B	49 B
non-Java object	2	504 B	24 B	480 B	480 B	252 B

Figure 26 Heap size and memory allocated table during the playback.



Name			
lge-nexus_5-03288b5c213e68a0	Online		5.
yingchen.cis690.musicplayer	30693		86

Heap updates will happen after every GC for this client

ID	Heap Size	Allocated	Free	% Used	# Objects
1	31.468 MB	18.881 MB	12.587 MB	60.00%	65,017

Cause GC

Display: Stats

Type	Count	Total Size	Smallest	Largest	Median	Average
free	933	15.660 MB	16 B	2.587 MB	96 B	17.187 KB
data object	14,487	837.508 KB	16 B	2.000 KB	48 B	59 B
class object	390	255.703 KB	112 B	8.000 KB	432 B	671 B
1-byte array (byte[], boolean[])	20	1.358 MB	16 B	1.003 MB	32 B	69.551 KB
2-byte array (short[], char[])	881	55.188 KB	16 B	1.000 KB	64 B	64 B
4-byte array (object[], int[], float[])	2,758	321.422 KB	16 B	60.000 KB	32 B	119 B
8-byte array (long[], double[])	481	21.453 KB	16 B	272 B	32 B	45 B
non-Java object	2	504 B	24 B	480 B	480 B	252 B

Figure 27 Heap size and memory allocated table after back button is clicked.

As a part of testing memory leak through the way of keeping tracking of heap size and the amount of memory allocated, Android Device Monitor is used frequently to monitor the changes. The screenshots in Figure 25-27 are captured before a song is playing inside PlaySongActivity (Figure 25), during the playback (Figure 26), and after back button is clicked (Figure 27). We can see that the amount of memory allocated change from 18.880 MB to 22.403 MB when the new activity is started, and go back to 18.881 MB when the activity is killed after back button is clicked. Multiple tests have been done in order to ensure there is no unknown memory leak during run time, including the cases such as navigation fragment adding, music service connection, song playback, visualization adding, etc.

Small-scale usability test is performed to evaluate the application as a whole. Comments and suggestions have been reapplied to the later development phase. For instance, some of the helpful suggestions includes the color schemes of the UI, fonts' sizes, layout organization in the playing song page, and some of the texts and images that can be used as icons, such as title of the app bar, the icons in the navigation drawer.

Besides, the comments also involve user experience on UI design flavor and corner case functionalities. For example, most users are used to clicking back button to go back to home screen. When user navigates to a specific artist' list of albums, and decides to go back to the homepage instead of going to previous page. As a typical corner case example, the application should deal with screen lock and orientation changes properly. All of the comments are taken into account at later development phase. They are extremely valuable for making the music player a user friendly app.

Although the above tests on UI and local tests cover most of the application's most important features and performance, there are some limitations as well. For instance, some missing but should be added tests includes, tests on audio file formats except mp3, tests on real devices other than Nexus 5 and Nexus 7 tablet, and other Android versions other than 5.1.1 and 5.0.2. It is expected that cell phones and tablets with Android version between Android 4.1 Jelly Bean and Android 5.0 Lollipop will work, but this need to be confirmed with further testing.

3.3 MapReduce classes Testing and Performance analysis

In this project, testing for App Engine MapReduce consists of two parts, unit testing and integration testing. Unit testing focuses on testing of each map reduce class individually. They include tests on each of the mapper and reducer methods. Integration testing targets to test whether the two chaining MapReduce jobs are chained, works as expected and produces desired output. They are performed locally as App Engine SDK provides a local development server for testing purpose. The local server simulates App Engine Java runtime and all of its services, which is convenient for testing without deploying the application to the cloud.

3.3.1 MapReduce Testing

Unit Tests for MapReduce jobs cover the following,

(1) test on Java servlet, which ensure that the servlet deals with HTTP requests correctly. Test on doget methods makes sure that the doGet method in MyServlet works correctly when front end requests is received (see section 2.4). doPost method in MyServlet is supposed to be triggered only by the administrator of the back end server. When triggered, it leads to execution of the MapReduce jobs. The tests on this part are performed after the App Engine application is deployed, and various size of input files (from a couple of lines to ~100 MB) are stored in Cloud Storage for testing and performance analysis purpose (see section 3.3.2 for performance analysis).

(2) test on connection to Cloud Storage, which ensure that the connection can be built assuming good internet connection.

(3) test on read/write files from/to Cloud Storage, which ensure that the read and write methods are correct, and the settings for Cloud Storage is proper.

(4) tests on mapper and reducer methods for both MapReduce jobs. They ensure that mappers can parse and consume specific formatted input files, and reducers accept the output from mapper and output correct results to Cloud Storage.

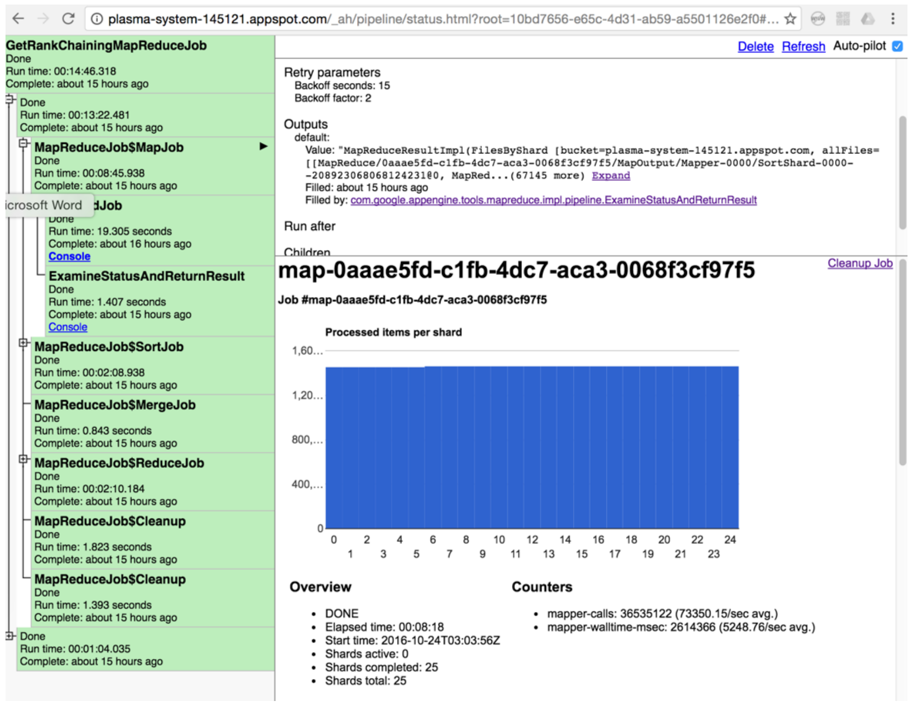
All of the unit tests together, if pass, ensure that the combination of input reader and output writer classes, which are GoogleCloudStorageFileOutput, GoogleCloudStorageLineInput, and the customized mapper and reducer classes works well together.

However, the configuration of the MapReduce jobs, such as the shard number (how many shards should the input be divided), the reducer number, cannot be tested if not running the whole application on App Engine. The next section, section 3.3.2 presents how the MapReduce jobs perform with different sizes of input files when running on App Engine.

3.3.2 Performance Analysis

Continued from the last section 3.3.1, tiny input files are used to test basic functionality and connection of MyServlet and MapReduce jobs. When running MapReduce jobs with small data, it is important to take a look at monitor of the job. It is a URL to display the information about the job. Figure 28 and Figure 29 show what successful and failing jobs look like. It tells the detail information about each stage of the MapReduce job, including how much time it takes, how the stage goes, and how many mapper/reducer calls, how many shards are completed and how many are still active, etc. The URL is useful for debugging when the jobs are failed. By clicking on the failed stages on the left, it tells the reason why it fails.

After making sure that the whole system works correctly for tiny data, larger files in Gigabytes are used to test the performance of the MapReduce jobs. Table 2 and Table 3 show how long it takes for 1.3 GB (8.7 million records) and 5.19GB (36.5 million records) input files in the situation of different sharding number.


plasma-system-145121.appspot.com/_ah/pipeline/status.html?root=10bd7656-e65c-4d31-ab59-a5501126e2f0#...

GetRankChainingMapReduceJob
Done
Run time: 00:14:46.318
Complete: about 15 hours ago

Done
Run time: 00:13:22.481
Complete: about 15 hours ago

MapReduceJob$MapJob
Done
Run time: 00:08:45.938
Complete: about 15 hours ago

dJob
Done
Run time: 19.305 seconds
Complete: about 16 hours ago
Console

ExamineStatusAndReturnResult
Done
Run time: 1.407 seconds
Complete: about 15 hours ago
Console

MapReduceJob$SortJob
Done
Run time: 00:02:08.938
Complete: about 15 hours ago

MapReduceJob$MergeJob
Done
Run time: 0.843 seconds
Complete: about 15 hours ago

MapReduceJob$ReduceJob
Done
Run time: 00:02:10.184
Complete: about 15 hours ago

MapReduceJob$Cleanup
Done
Run time: 1.823 seconds
Complete: about 15 hours ago

MapReduceJob$Cleanup
Done
Run time: 1.393 seconds
Complete: about 15 hours ago

Done
Run time: 00:01:04.035
Complete: about 15 hours ago

Delete Refresh Auto-pilot

Retry parameters
Backoff seconds: 15
Backoff factor: 2

Outputs
default:
```
Value: "MapReduceResultImpl(FilesByShard [bucket=plasma-system-145121.appspot.com, allFiles=
[[MapReduce/0aaae5fd-c1fb-4dc7-aca3-0068f3cf97f5/MapOutput/Mapper-0000/SortShard-0000-
-208923068068124231@0, MapRed...(67145 more) Expand
```
Filled: about 15 hours ago
Filled by: com.google.appengine.tools.mapreduce.impl.pipeline.ExamineStatusAndReturnResult

Run after

Children

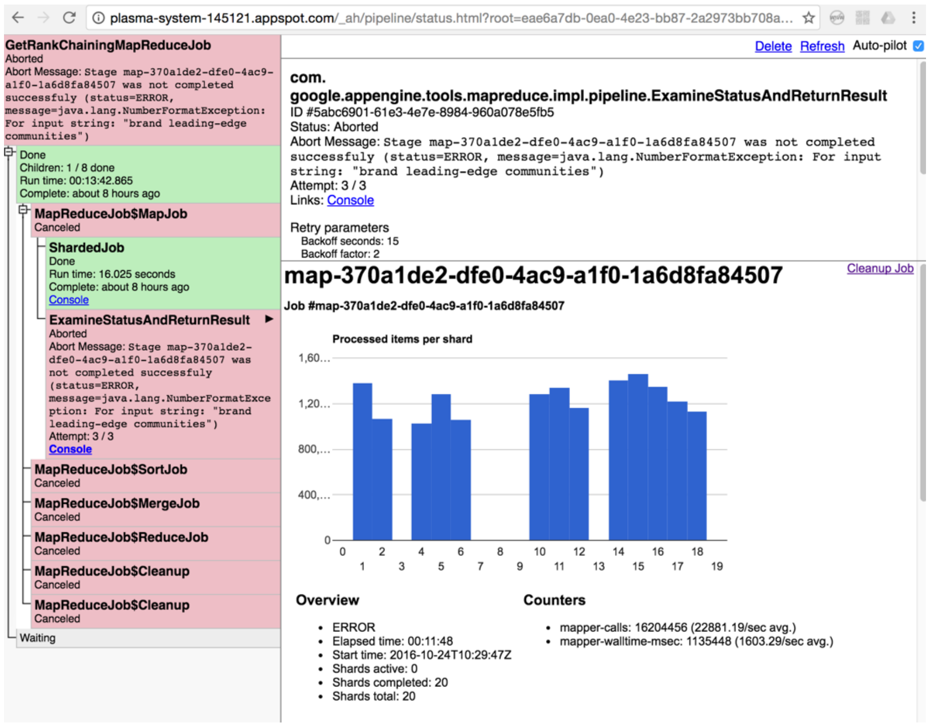
map-0aaae5fd-c1fb-4dc7-aca3-0068f3cf97f5 Cleanup Job

Job #map-0aaae5fd-c1fb-4dc7-aca3-0068f3cf97f5

Overview
- DONE
- Elapsed time: 00:08:18
- Start time: 2016-10-24T03:03:56Z
- Shards active: 0
- Shards completed: 25
- Shards total: 25

Counters
- mapper-calls: 36535122 (73350.15/sec avg.)
- mapper-walltime-msec: 2614366 (5248.76/sec avg.)

Figure 28 A successful MapReduce job


plasma-system-145121.appspot.com/_ah/pipeline/status.html?root=eae6a7db-0ea0-4e23-bb87-2a2973bb708a...

GetRankChainingMapReduceJob
Aborted
Abort Message: `Stage map-370a1de2-dfe0-4ac9-a1f0-1a6d8fa84507 was not completed successfuly (status=ERROR, message=java.lang.NumberFormatException: For input string: "brand leading-edge communities")`

Done
Children: 1 / 8 done
Run time: 00:13:42.865
Complete: about 8 hours ago

MapReduceJob$MapJob
Canceled

ShardedJob
Done
Run time: 16.025 seconds
Complete: about 8 hours ago
Console

ExamineStatusAndReturnResult
Aborted
Abort Message: `Stage map-370a1de2-dfe0-4ac9-a1f0-1a6d8fa84507 was not completed successfuly (status=ERROR, message=java.lang.NumberFormatException: For input string: "brand leading-edge communities")`
Attempt: 3 / 3
Console

MapReduceJob$SortJob
Canceled

MapReduceJob$MergeJob
Canceled

MapReduceJob$ReduceJob
Canceled

MapReduceJob$Cleanup
Canceled

MapReduceJob$Cleanup
Canceled

Waiting

Delete Refresh Auto-pilot

**com.
google.appengine.tools.mapreduce.impl.pipeline.ExamineStatusAndReturnResult**
ID #5abc6901-61e3-4e7e-8984-960a078e5fb5
Status: Aborted
Abort Message: `Stage map-370a1de2-dfe0-4ac9-a1f0-1a6d8fa84507 was not completed successfuly (status=ERROR, message=java.lang.NumberFormatException: For input string: "brand leading-edge communities")`
Attempt: 3 / 3
Links: Console

Retry parameters
Backoff seconds: 15
Backoff factor: 2

map-370a1de2-dfe0-4ac9-a1f0-1a6d8fa84507 Cleanup Job

Job #map-370a1de2-dfe0-4ac9-a1f0-1a6d8fa84507

Overview
- ERROR
- Elapsed time: 00:11:48
- Start time: 2016-10-24T10:29:47Z
- Shards active: 0
- Shards completed: 20
- Shards total: 20

Counters
- mapper-calls: 16204456 (22881.19/sec avg.)
- mapper-walltime-msec: 1135448 (1603.29/sec avg.)

Figure 29 A failing MapReduce job

From the two tables below, we can see that bigger size input files take more time than that of smaller files given the same number of instances running on App Engine and same number of shards. For example, when sharding number is 20, and number of instances is 25, the running time for 1.3GB input file is 5 minutes 49 seconds, comparing to 16 minutes 4 seconds for 5.19GB input file. Whilst for the same size of input files, more instances mean less running time. As it shown in Table 2, it only takes 5 minutes 8 seconds to completes the job for 25 instances, comparing to 10 minutes 18 seconds for 10 instances.

Another interesting result drawn from the two tables is that the running time for a MapReduce job tends to get its minimum when the number of shards is close to the number of instances. For example, for 1.3GB input files and 25 instances, the minimum running time is 5:08 at which the number of shards is 20, while for 10 instances case, the minimum is 7:28 at which the number of shards is 10.

From the data shown here, it is not a good idea to set the number of shards too bigger than the number of instances. MapReduce job will not complete in this case. As it is shown in Table 1 when number of shards is 60 for 25 instance and when number of shards is 25, 50, and 60 for 10 instances. Large number of shards means dividing input files to a lot of pieces, but if the instances that are going to take over the jobs are limited, then waiting time for some of the pieces will be longer. If the waiting time is longer than a maximum number set in the MapReduce API, the thread will be closed which leads to accessing closed thread error.

Number of Shards	Number of Instances	Time (minute:second)	Time (minute:second)	Number of Instances
60	25 instances	Error	Error	10 instances
50		9:49	Error	
25		5:49	Error	
20		5:08	10:18	
10		7:58	7:28	
3		13:43	12:16	

Table 2 Performance for input data of 1.3GB

Number of Shards	Number of Instances	Time (minute:second)	Time (minute:second)	Number of Instances
60	25 instances	Error	Error	10 instances
50		20:05	Error	
25		14:46	Error	
20		16:04	Error	
10		26:34	27:59	
3		47:35	49:32	

Table 3 Performance for input data of 5.19GB

Finally, there are some limitations based on the current set up of the tests above. First, the given limited scale of the input file size, no further conclusion could be made about how the running time changes with input files size. Second, more data needed to figure out what is the optimized configuration to get the minimum running time.

Chapter 4 - Reflections and Future Work

To sum up, the current features cover most of the basic and most frequently used functions that a user can look for from a music player. Also, most of the current features and functionality are well tested, which makes it a reliable and delightful music player to play with. However, it needs more improvements if the goal is to release it to the final users. The following sections give some thoughts about how to improve it based on current set up. In the end, reflections on the whole developing process will be helpful for building similar mobile applications, or small software projects in general.

4.1 Future Work

Based on the current set up, one extension about the back end is to enable its ability to save users' data and do deeper analysis. For example, collect data about the songs on users' devices, trace how often they play a particular song. Analyzing these kinds of data will make it possible to group users based on their preferences, recommend new products to a particular user, recommend new products to groups of users based on their preferences, etc.

In addition, as a possible extension to the Comment feature, server can send emails to the user after they submit any comments. For example, send user a thank you email after they submit comments, or send emails to introduce some of the new features that they did not notice before, or send regular email to all users when there are new features added to the app. By adding the feature of sending emails to user, it is likely to increase user participation of giving more comments or suggestions.

Some improvements about the front end include, allowing user create their own playlists, providing different skins for the music player, allowing user download files from the internet, etc.

These are common features for a music player application. Providing these features will increase users' satisfaction about this app, and keep using it.

4.2 Reflections

Design and overall management of the whole project is very important when a project becomes larger and larger. A good design allows more flexibility for the whole developing process. For example, design that is structured to accommodate change will make coding easier when adding new modules. Also, a good design take test into account in the first place which leads to better maintainability.

Since Android APIs are developing very fast, long lasting projects such as this project, are likely to encounter the problem of using deprecated APIs, a need to update some methods with new APIs, etc. The key point to these problems is always searching for suggestion from the newest documentation, blogs, online communities, etc. Solutions to previous problems may solve current issue but it also comes with a potential problem sometimes, especially when using tools in beta. So using the newest APIs and approaches to solve problems is always the better choice.

References

[1] Material Design for Android. (2015, August).

Retrieved from https://developer.android.com/design/material/index.html

[2] Get the best music experience on your Android phone (2016, April).

Retrieved from http://www.androidcentral.com/best-music-players-android

[3] Supported media formats. (2015, August).

Retrieved from http://developer.android.com/guide/appendix/media-formats.html

[4] Parcelable VS. Java Serialization in Android App development. (2015, August).

Retrieved from http://www.3pillarglobal.com/insights/parcelable-vs-java-serialization-in-android-app-development

[5] The MIT License. (2016, November).

Retrieved from http://opensource.org/licenses/mit-license.php

[6] Performance tips. (2016, November).

Retrieved from http://developer.android.com/training/articles/perf-tips.html

[7] SQL as understood by SQLite. (2015, November).

Retrieved from http://www.sqlite.org/lang_createindex.html

[8] Jobs and stages. (2016, November).

Retrieved from https://github.com/GoogleCloudPlatform/appengine-mapreduce/wiki/1.2-Jobs-and-Stages

[9] App Engine. (2016, November).

Retrieved from https://cloud.google.com/appengine/

[10] App Engine MapReduce wiki (2014, November).

Retrieved from https://github.com/GoogleCloudPlatform/appengine-mapreduce/wiki

[11] Getting started with testing. (2016, November).

Retrieved from https://developer.android.com/training/testing/start/index.html

[12] Tasty mocking framework for unit tests in Java. (2016, February).

Retrieved from http://site.mockito.org/

[13] Robolectric test-drive your Android code. (2016, November).

Retrieved from http://robolectric.org/

[14] Espresso. (2016, November).

Retrieved from https://google.github.io/android-testing-support-library/docs/espresso/

[15] UI Automator. (2016, November).

Retrieved from https://google.github.io/android-testing-support-library/docs/uiautomator/