|  |
| --- |
| C:\year 5\Git\Report\FYP-Report\gmit-logo-2012rgb.jpg |
| **Music Host Interface**  **Thomas Flynn**  Bachelor of Electronic & Computer Engineering  2015/16  Supervisor: Brian O'Shea, Galway-Mayo Institute Technology |
|  |
|  |

# Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering in Computer & Electronic Engineering at Galway Mayo Institute of Technology. This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

# Acknowledgements

I would like to thank my project supervisor Brian O'Shea for providing me with valuable insight and support throughout the development of this project.

# Summary

The goal for this project was to combine the features of a Jukebox and the role that a DJ plays. This combination is what I have titled as the Music Host Interface. It defines two roles, the Music Host which consists of a desktop application and the Music Guest which consists of an Android application.

The scope of this project was primarily based on interfacing these two elements together into one cohesive solution using the following technologies, Bluetooth, Cloud database, and JavaFX.

The approach for this project consisted of dividing up the work to be completed into 9 sprints that each required incremental degrees of functionality to be achieved. The following two paragraphs are a summary of what was achieved from using this approach.

Using the graphical user interface the Music Host has the following abilities. The first is the ability to dictate what the guests can or can't do to suit their own hosting preferences. The second is the ability to play music to the guests. The third is the ability to login and gain access to their own private selection of songs from anywhere around the world that has a Music Host Interface. This is achieved due to the fact that the host's private login credentials and songs are stored on a cloud database.

Using the Android Application the guest has the following abilities. The first is the ability to choose a song to play from the host's selection of songs. The second is the ability to request a song from the host in the form a text message. The third is the ability to veto the current song.

In conclusion the final application successfully provides a solution for both the Music Host and the Music Guest to be able to communicate with one another using this technology.

Contents

[Declaration 2](#_Toc449924256)

[Acknowledgements 3](#_Toc449924257)

[Summary 4](#_Toc449924258)

[1. Introduction 9](#_Toc449924259)

[1.1 Project goals 9](#_Toc449924260)

[1.2 Project motivation 10](#_Toc449924261)

[1.3 Music Host Problems 10](#_Toc449924262)

[1.4 Report Overview 11](#_Toc449924263)

[2. Project Plan 13](#_Toc449924264)

[2.1 Gantt chart 13](#_Toc449924265)

[2.2 Trello Project Management 13](#_Toc449924266)

[2.3 Toggl time management 14](#_Toc449924267)

[3 Block Diagram 15](#_Toc449924268)

[4 Flow Charts 16](#_Toc449924269)

[4.1 JavaFX Music Host 16](#_Toc449924270)

[4.2 Android Client Connecting to Music Host 16](#_Toc449924271)

[4.3 Music Host - Android Client Communication Flow Charts 16](#_Toc449924272)

[4.3.1 Song Request Flow Chart 16](#_Toc449924273)

[5 JavaFX Music Host Graphical User Interface 18](#_Toc449924274)

[5.1 Research and Investigation 18](#_Toc449924275)

[5.1.1 Graphical User Interface API 18](#_Toc449924276)

[5.1.2 Cloud Database 20](#_Toc449924277)

[5.1.2 Audio Format 20](#_Toc449924278)

[5.1.3 Bluetooth 20](#_Toc449924279)

[5.2 Requirements 21](#_Toc449924280)

[5.2.1 Graphical User Interface 21](#_Toc449924281)

[5.2.2 Sound Engine 21](#_Toc449924282)

[5.2.3 Database as a service 21](#_Toc449924283)

[5.2.4 Bluetooth Server 21](#_Toc449924284)

[5.3 Tool Choices 22](#_Toc449924285)

[5.3.1 Microsoft Azure SQL Server 22](#_Toc449924286)

[5.3.2 GUI 23](#_Toc449924287)

[5.3.3 Bluetooth Server 24](#_Toc449924288)

[5.4 JavaFX Overview 25](#_Toc449924289)

[5.4.1 Application class 25](#_Toc449924290)

[5.4.2 Stage class 25](#_Toc449924291)

[5.4.3 Scene class 25](#_Toc449924292)

[5.4.4 Node class 25](#_Toc449924293)

[5.4.5 FXML 26](#_Toc449924294)

[5.4.6 *Model* View Controller 26](#_Toc449924295)

[5.5 Foundation 28](#_Toc449924296)

[5.5.1 MusicHostFramework 28](#_Toc449924297)

[5.5.2 ScreensController 30](#_Toc449924298)

[5.5.3 Setup Sequence Diagram 32](#_Toc449924299)

[5.6 LoginScreenController class 34](#_Toc449924300)

[5.6.1 Login Sequence Diagram 36](#_Toc449924301)

[5.6.2 Login View Operation 37](#_Toc449924302)

[5.7 MediaPlayer Application Realisation 38](#_Toc449924303)

[5.7.3 *Model* class 50](#_Toc449924304)

[5.7.4 Azure SQL Server Database 52](#_Toc449924305)

[5.7.5 DB class 54](#_Toc449924306)

[5.7.5 SelectionSong class 56](#_Toc449924307)

[5.7.6 Queue Song class 57](#_Toc449924308)

[5.7.7 HandleFileIO class 58](#_Toc449924309)

[5.8 Music Host MediaPlayer Sequence Diagrams 59](#_Toc449924310)

[5.8.1 Initialize Button Sequence Diagram 59](#_Toc449924311)

[5.8.2 Add Button Sequence Diagram 60](#_Toc449924312)

[5.8.3 Add Song Animation Ended Event Sequence Diagram 62](#_Toc449924313)

[5.8.4 Song Added Event Sequence Diagram 63](#_Toc449924314)

[8.8.5 Removed/Ended Event Sequence Diagram 65](#_Toc449924315)

[5.8.6 Logout Sequence Diagram 67](#_Toc449924316)

[5.9 Music Host MediaPlayer Operation 69](#_Toc449924317)

[5.9.1 Use Case - Logged In 69](#_Toc449924318)

[5.9.2 Use Case - Setup 70](#_Toc449924319)

[5.9.3 Use Case - Song Added 71](#_Toc449924320)

[5.9.4 Use Case - Song Skipped / Song Ended 72](#_Toc449924321)

[5.9.5 Use Case - Play / Pause 73](#_Toc449924322)

[5.10 Music Host Bluetooth 74](#_Toc449924323)

[5.10.1 Serial Port Profile Connection 74](#_Toc449924324)

[5.10.2 Bluetooth Communication Diagrams 74](#_Toc449924325)

[5.10.3 *MainSceneController* Server related elements 76](#_Toc449924326)

[5.10.4 ProcessConnectionThread 78](#_Toc449924327)

[5.10.5 Server Button Sequence Diagram 83](#_Toc449924328)

[5.10.6 ProcessConnectionThread Sequence Diagram 85](#_Toc449924329)

[5.10.7 WhatToDoFunc Options, Song Request, Song Selected Sequence Diagram 87](#_Toc449924330)

[5.10.8 WhatToDoFunc DJ Comment Sequence Diagram 89](#_Toc449924331)

[5.10.9 WhatToDoFunc Skip Song Sequence Diagram 91](#_Toc449924332)

[6 Music Host Client Android Application 93](#_Toc449924333)

[6.1 Foundation 94](#_Toc449924334)

[6.1.1 ChatBusinessLogic class 94](#_Toc449924335)

[6.2 Realisation 98](#_Toc449924336)

[6.2.1 MainActivity 98](#_Toc449924337)

[6.2.2 SongRequestActivity 102](#_Toc449924338)

[6.2.3 DJActivity 104](#_Toc449924339)

[6.2.4 Bluetooth Communication 105](#_Toc449924340)

[6.3 Sequence Diagrams 107](#_Toc449924341)

[6.3.1 Open Application SD 107](#_Toc449924342)

[6.5.2 Bluetooth Search SD 109](#_Toc449924343)

[6.5.3 Events Bluetooth Receiver SD 110](#_Toc449924344)

[6.5.4 Connecting to Music Host SD 112](#_Toc449924345)

[6.5.5 Connected to Music Host SD 113](#_Toc449924346)

[6.5.6 Bluetooth Communication Thread SD 114](#_Toc449924347)

[6.5.7 Main Activity Handler SD 116](#_Toc449924348)

[6.5.8 onCreate SongRequestAcitivity SD 118](#_Toc449924349)

[6.5.9 Parse JSON Async Task SD 120](#_Toc449924350)

[6.5.10 SongRequestAcitivity song selected SD 122](#_Toc449924351)

[6.5.11 Song Request, DJ comment, Skip button SD 123](#_Toc449924352)

[6.6 Operation 124](#_Toc449924353)

[6.6.1 Use Case - Open App 124](#_Toc449924354)

[6.6.2 Use Case - Search For Music Host (Bluetooth) 125](#_Toc449924355)

[6.6.3 Use Case - Connected 126](#_Toc449924356)

[6.6.4 Use Case - Song Request 127](#_Toc449924357)

[6.7.5 Use Case - Song Accepted / Not Accepted 128](#_Toc449924358)

[6.7.6 Use Case - DJ Comment 129](#_Toc449924359)

[6.7.7 Use Case - Skip Song 130](#_Toc449924360)

[7 System Integration 131](#_Toc449924361)

[8 Project Statistics 132](#_Toc449924362)

[8.1 Github Repositories 132](#_Toc449924363)

[8.1.1 FYP-GUI Repository 132](#_Toc449924364)

[8.1.2 FYP-Android Repository 132](#_Toc449924365)

[8.2 Toggl Time Doucmentation 133](#_Toc449924366)

[8.2.1 September - December 133](#_Toc449924367)

[8.2.2 January - April 134](#_Toc449924368)

[9 Conclusion 135](#_Toc449924369)

# 1. Introduction

## 1.1 Project goals

The scope of this project involves interfacing a JavaFX desktop application that can play music that is stored on a cloud database with an Android application that can connect to the desktop application using Bluetooth.

The first goal for the desktop application was to provide the user with an intuitive and responsive graphical user interface for playing the role of the Music Host. In this role the user can perform all the basic operations that a standard media player provides such as the adding and removing of songs from a queue.

The second goal was to integrate a cloud database for storing music. The reasoning for this was to prevent the user from being limited to the song files stored locally on the computer. The database also serves as a way of privatising an individual's song selection, this is achieved by requiring the user to enter a username and password in order to gain access to their songs on the cloud.

The third goal was to develop an iterative Bluetooth server that runs within the desktop application. The server facilitates requests from the Android client.

The first goal for the Android application was to provide the user with the ability to connect to the desktop application using Bluetooth.

The second goal was allow the user to view the selection of songs that the desktop application has once connected. The user can then choose a song from this selection to be played at a later time by the Music Host.

The third goal was to allow the user to send a text message to the Music Host.

The fourth and last goal was to allow the user to be able to vote to skip the current song that the Music Host is playing.

## 1.2 Project motivation

I had many powerful motivators throughout the duration this project. The first was my own personal interest in playing the role of a Music Host and the problems that are encountered with it. I wanted to use this personal interest to provide a clean, intuitive and practical solution to these problems.

The second was the chance to display my abilities as a software engineer, taking advantage of the various software engineering skills that I learnt throughout my years in the course. Being given the chance to combine these skills and apply them to something that I am very passionate about was a most welcome opportunity afforded to me by this course.

The last and most powerful motivator for me was the scope of the project. At first I did not think I was capable of such a feat, that I would be overwhelmed by the workload or lose discipline and at times this was the case, but overcoming these obstacles has not only made a stronger programmer but also a stronger person as well.

## 1.3 Music Host Problems

Music hosting is the role someone takes on when they are in the position of dictating what music others around them hear. This role is most commonly referred to as the DJ.

The DJ in this role very commonly takes song requests from his or her audience through vocal communication. This can be problematic for two reasons. The first is that the DJ's performance is impaired when being constantly distracted by the audience members song requests. The second is that due to the loudness of the music, it can be difficult for the audience member to communicate what song he or she wants the DJ to play.

A DJ is not the only one who plays the role of a Music Host however. A venue in the public domain that has a selection of music that plays over a duration of time on a sound system is playing this role. In this case however there is no straight forward way for a guest to request a song from this selection.

Jukeboxes were more commonplace in the past. They provided people with an interface for viewing a selection of songs and the ability to choose a song to play from this selection. This project is an attempt to bring back the Jukebox interface into the public domain.

## 1.4 Report Overview

This report first covers my project plan followed by the body of the report. The body is broken up into two main sections numbered 5 and 6, the desktop application and the Android application respectively.

Each subsection of the desktop application discusses the following.

5.1 The research carried out.

5.2 The requirements for the application.

5.3 The tools chosen and the reasons why.

5.4 An overview of the API used to build the desktop application.

5.5 The code that the application was built on.

5.6 The login screen aspect of the application.

5.7 This sub section is quite dense. It discusses the media player aspect of the application, the individual classes used within it and the database that stores the media.

5.8 Consists of sequence diagrams relating to the operation of the media player aspect of the application.

5.9 Consists of screenshots of the media player in operation and an explanation of these features at a high level of abstraction.

5.10 Covers the Bluetooth aspects of the desktop application and how it interfaces with the Android application. It has high level abstraction communication diagrams and low level abstraction sequence diagrams.

Section 6 covers the foundation of the Bluetooth Chat Application

# 2. Project Plan

## 2.1 Gantt chart

I divided up my allotted time for the project into 9 sprints. This allowed me to gauge my progress throughout the project development.



Figure : Gantt chart

## C:\year 5\Git\Report\FYP-Report\Report\report lib\new\trello.png2.2 Trello Project Management

Trello is a web based project mangement tool. It proved to be a vital part of my project plan as it allowed me to quickly and easily add and remove tasks within a sprint as the priority of those tasks changed.

Figure : Trello logo

#### Trello Project Board labels



Figure : Trello project labels

#### Trello Project Management Board



Figure : Trello project management board

## C:\year 5\Git\Report\FYP-Report\Report\report lib\new\toggl.png2.3 Toggl time management

Toggl provided me with a way of keeping track of my time. It also proved to be an excellent source of documentation for my project logs.

Figure : Toggl logo

# 3 Block Diagram



Figure : Block diagram

# 4 Flow Charts

## 4.1 Music Host desktop application

C:\year 5\Git\Report\FYP-Report\Report\report lib\new\Flowchart1.png

Figure : Song Request flowchart

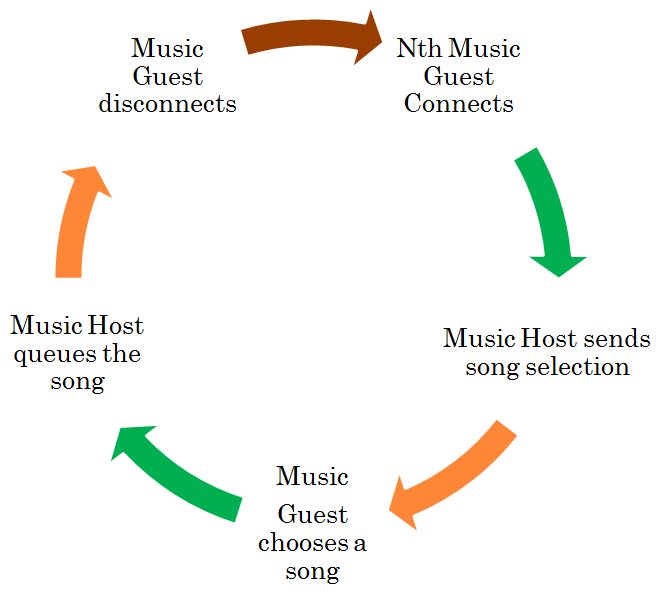
4.2 Song Request Flowchart

Figure : Song Request flowchart

# 5 Music Host desktop application

Research and investigation for the desktop application was carried out in the first sprint of the project life cycle. At this stage, the basic requirements for operation were listed so an informed decision could be made on which tools to implement in order to build the application.

With the tools selected I carried out a second round of research and investigation in sprint 2, this involved researching existing sample code for each of the tools chosen.

In Sprint 3, separate prototype development was carried out for the Bluetooth server, database and media player features on the following GitHub repositories respectively, FYP-Server **[x]** , FYP-Database **[x]** and FYP-GUI. **[x]**

By sprint 4 I had successfully managed to integrate the basic functionality of all 3 prototypes into the desktop application. The subsequent sprints involved building on this within the FYP-GUI repository.

**https://github.com/g00291875/FYP-Database**

**https://github.com/g00291875/FYP-Server**

**https://github.com/g00291875/FYP-GUI**

## 5.1 Research and Investigation

The solution for the desktop application was based on designing a graphical user interface for the user to be able to control the playing of mp3 files that are stored on a remote database and to be able to dictate the communication interaction of the connecting clients to the Bluetooth server. From a basic perspective, this solution requires knowledge of a library for a graphical user interface, a media player API, a cloud database as a service and a library for a Bluetooth server.

### 5.1.1 Graphical User Interface API

A graphical user interface or GUI, is a type of [interface](https://en.wikipedia.org/wiki/User_interface) that allows [users](https://en.wikipedia.org/wiki/User_(computing)) to [interact with electronic devices](https://en.wikipedia.org/wiki/Human%E2%80%93computer_interaction) through graphical [icons](https://en.wikipedia.org/wiki/Computer_icon) and visual indicators such as [secondary notation](https://en.wikipedia.org/wiki/Secondary_notation), as opposed to [text-based interfaces](https://en.wikipedia.org/wiki/Text-based_user_interface), typed command labels or text navigation. GUIs were introduced in reaction to the perceived steep [learning curve](https://en.wikipedia.org/wiki/Learning_curve) of [command-line interfaces](https://en.wikipedia.org/wiki/Command-line_interface), which require commands to be typed on the [keyboard](https://en.wikipedia.org/wiki/Computer_keyboard).

[x] - **http://www.computerhope.com/issues/ch000619.htm**

### 5.1.2 Cloud Database

A cloud database is a [database](https://en.wikipedia.org/wiki/Database) that typically runs on a [cloud computing](https://en.wikipedia.org/wiki/Cloud_computing) platform. There are two common deployment *model*s: users can run databases on the cloud independently, using a [virtual machine](https://en.wikipedia.org/wiki/Virtual_machine) image, or they can purchase access to a database service, maintained by a cloud database provider. Of the databases available on the cloud, some are [SQL](https://en.wikipedia.org/wiki/SQL)-based and some use a [NoSQL](https://en.wikipedia.org/wiki/NoSQL) data *model*. **[x]**

ref 2 - **https://en.wikipedia.org/wiki/Cloud\_database**

### 5.1.2 Audio Format

MPEG-1 or MPEG-2 Audio Layer III, **[x]**  more commonly referred to as MP3, is an [audio coding format](https://en.wikipedia.org/wiki/Audio_coding_format) for [digital audio](https://en.wikipedia.org/wiki/Digital_audio) which uses a form of [lossy data compression](https://en.wikipedia.org/wiki/Lossy_data_compression). It is a common audio format for consumer audio [streaming](https://en.wikipedia.org/wiki/Streaming_media) or storage, as well as a [de facto standard](https://en.wikipedia.org/wiki/De_facto_standard) of digital audio compression for the transfer and playback of music on most [digital audio players](https://en.wikipedia.org/wiki/Digital_audio_player).

**ref 3- https://en.wikipedia.org/wiki/MP3**

**https://tools.ietf.org/html/rfc5219**

### 5.1.3 Bluetooth

Bluetooth is a [wireless](https://en.wikipedia.org/wiki/Wireless) technology standard for exchanging data over short distances (using short-wavelength [UHF](https://en.wikipedia.org/wiki/UHF) [radio waves](https://en.wikipedia.org/wiki/Radio_waves) in the [ISM band](https://en.wikipedia.org/wiki/ISM_band) from 2.4 to 2.485 GHz) from fixed and mobile devices, and building [personal area networks](https://en.wikipedia.org/wiki/Personal_area_network) (PANs). Invented by telecom vendor [Ericsson](https://en.wikipedia.org/wiki/Ericsson) in 1994, it was originally conceived as a wireless alternative to [RS-232](https://en.wikipedia.org/wiki/RS-232) data cables. It can connect several devices, overcoming problems of synchronization. **[x]**

**https://www.bluetooth.org**

## 5.2 Requirements

From the research carried out, a number of requirements and dependencies were raised for the project to meet functionality requirements. These requirements were then segmented into 4 core elements.

### 5.2.1 Graphical User Interface

The Graphic User Interface (GUI) will be used to provide the user with a source of input for playing the role of the music host while acting as a translator between the inputs received and sound engine functionality. The GUI should also provide enough feedback so the user is aware of what actions they can perform and how to play the role of the Music Host.

### 5.2.2 Sound Engine

The sound engine should be capable of playing mp3 files due to the overwhelming popularity of mp3 files as well as meet all the minimum requirements for playback control.

### 5.2.3 Database as a service

The solution will use a specific form of cloud service known as Database as a service (DBaas)**.** The database will store the collection of Music Host's login credentials as well as their associated song selection. This will provide the user with the ability to access their private music selection from anywhere that has the desktop application and an internet connection. This will require the database to be SQL based with sufficient data throughput to be able to access and download mp3 files in real time.

### 5.2.4 Bluetooth Library

The Bluetooth library should provide the necessary functionality to run a server within the desktop application in order to accept connections from users of the Android application.

## 5.3 Tool Choices

The requirements listed in the previous section have led me to the selection of the following tools.

### 5.3.1 Graphical user interface and sound engine

The requirements for the graphical user interface specify that the API is capable of providing a responsive and intuitive interface for the Music Host.

**Choice:** *JavaFX API*

**Reasons:**

* API is built into the JDK
* Access to primitives
* Extensive Animation API
* Sound engine can play mp3 files

Figure : JavaFX Logo

**Notes:**

JavaFX is a [software platform](https://en.wikipedia.org/wiki/Computing_platform) for creating and delivering [desktop applications](https://en.wikipedia.org/wiki/Application_software) that can run across a wide variety of devices. JavaFX is intended to replace [Swing](https://en.wikipedia.org/wiki/Swing_(Java)) as the standard [GUI](https://en.wikipedia.org/wiki/Graphical_User_Interface) library for [Java SE](https://en.wikipedia.org/wiki/Java_Platform,_Standard_Edition).

The API is implemented as a native Java library, and applications using JavaFX are written in native Java code.**[x]**

The JavaFX media concept is based on the following entities.

Media – A media resource, containing information about the media, such as its source, resolution, and metadata.

MediaPlayer – The key component providing the controls for playing media.

MediaView – A Node object to support animation, translucency, and effects.

Each element of the media functionality is available through the javafx.scene.media package. **[x]**

Appendix 1 **[x]** provides an overview of JavaFX for readers who are not familiar with the API already.

**http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm**

**https://docs.oracle.com/javafx/2/media/overview.htm**

## 5.3.2 Cloud database

The requirements for the remote database specify that the database can be accessed globally and that it is SQL based with sufficient Data Throughput for the accessing and downloading of mp3 files.

**Choice:** *Microsoft SQL Azure*

**Reasons:**

* Extensive online documentation
* Powerful JDBC Driver
* Affordable
* Account Subscription is compatible with student email
* Competitive Data Throughput

Figure : Microsoft SQL Azure Logo

**Notes:**

Microsoft Azure SQL Database is a [cloud based](https://en.wikipedia.org/wiki/Cloud_computing) [service](https://en.wikipedia.org/wiki/Software_as_a_service) from [Microsoft](https://en.wikipedia.org/wiki/Microsoft) offering data-storage capabilities as part of the [Azure Services Platform](https://en.wikipedia.org/wiki/Azure_Services_Platform).

SQL Database uses a special version of [Microsoft SQL Server](https://en.wikipedia.org/wiki/Microsoft_SQL_Server) as its backend. It provides [high availability](https://en.wikipedia.org/wiki/High_availability) by storing multiple copies of databases, [business continuity](https://en.wikipedia.org/wiki/Business_continuity) and [disaster recovery](https://en.wikipedia.org/wiki/Disaster_recovery) with backups and geo-replication, elastic scale and rapid provisioning.

Microsoft Azure SQL Database uses an [XML](https://en.wikipedia.org/wiki/XML)-based format for data transfer. Like [Microsoft SQL Server](https://en.wikipedia.org/wiki/Microsoft_SQL_Server), SQL Database uses [T-SQL](https://en.wikipedia.org/wiki/T-SQL) as the query language and [Tabular Data Stream](https://en.wikipedia.org/wiki/Tabular_Data_Stream) (TDS) as the protocol to access the service over the Internet. **[x]**

**http://www.amazon.com/Microsoft-Azure-Enterprise-Application-Development/dp/1849680809/ref=ntt\_at\_ep\_dpt\_2**

### 5.3.3 Bluetooth Server

The requirements specified for the Bluetooth Server state that the Bluetooth server is capable of running concurrently within the desktop application.

**Choice:** *bluecove-2.1.1-SNAPSHOT*

**Reasons:**C:\year 5\Project presentation\pics\bluecove.png

* Open Source library
* Bluecove Stack implements Service Discovery Protocol
* Interfaces with the Microsoft Bluetooth stack

Figure : BlueCove Logo

**Notes:**

BlueCove is a JSR-82 implementation on Java Standard Edition (J2SE) on BlueZ Linux, Mac OS X, WIDCOMM, BlueSoleil and Microsoft Bluetooth stack on WinXPsp2 and newer. Originally developed by Intel Research and currently maintained by volunteers. **[x]**

**http://bluecove.org/**

## 5.4 Foundation

The foundation for the GUI was built on a JavaFX multi-screen example provided by Oracle on Github. [x]

**ref - https://github.com/acaicedo/JFX-MultiScreen**

This example provided me with the necessary functionality for the *LoginView.fxml* and *MainView.fxml* to be swapped in and out of the scene graph. Business logic for logging in and out was built on top of this feature.

### 5.4.1 MusicHostFramework

**

*MusicHostFramework* extends Application, allowing it to run as a JavaFX application. The Music Host Application is launched by this class.

#### Fields

**String loginScreenID**

*A*ssociates the fxml file specified by login*ScreenFile* in a hashmap stored by the *ScreensController.*

**String loginScreenFile**

Value = *LoginView*.fxml

**String mainScreenID**

*A*ssociates the fxml file specified by login*ScreenFile* in a hashmap stored by the *ScreensController.*

**String mainScreenFile**

Value = *MainView*.fxml

#### Methods

**@Overide**

**public void start(Stage)**

The main entry point for all JavaFX applications. This method is called after the init method has returned, and after the system is ready for the application to begin running.

Creates a ScreensController object that will store the *LoginView.fxml* and the *MainView*.fxml nodes. Each of these nodes have a controller associated specified by their respective *.fxml*.

Once the fxml files have been loaded by the *ScreensController* a Group object is created and it's child Node is defined as the *ScreensController StackPane Node*.

A *Scene* object is created to be used on the *Stage*. The *Group Node* which now contains a *StackPane* consisting of the *LoginView* and *MainView Pane Nodes* is added to the Scene graph.



Figure: MusicHostFramework UML

### 5.4.2 ScreensController



Figure: ScreenController UML

The ScreensController extends StackPane allowing it to act as a container for Pane Nodes. These Pane Nodes are the *LoginView.fxml* and *MainView.fxml*.

The hashmap *screens* provides the necessary functionality for swapping the nodes by storing a screenID String along with the associated Node.

It also holds the *model* object which is accessed from both the LoginSceneController and the *MainSceneController*.

#### Methods

**public boolean loadScreen(String,String)**

This method is called by the MusicHostFramework within its start method. It loads the fxml file specified by the second parameter. Then class casts the controller associated with the fxml to a ControlledScreen interface object. This allows the injection of the *ControlledScreen* to both the *LoginSceneController* and the *MainSceneController*.

Because both Controllers implement the *ControlledScreen* interface they can gain access to common methods and resources such as the *model* object.

**public boolean *setScreen*(String)**

This method tries to display the screen with a predefined name. First it makes sure the screen specified has already been loaded. Then if there is more than one screen, the new screen is added second and the current screen is removed. If there isn't any screen being displayed. The new screen is simply added to the root.



Figure: ScreensController UML

### 5.5.3 Setup Sequence Diagram



Figure : Setup Sequence Diagram

## 5.6 LoginScreenController class



Figure: LoginSceneController Class

The *LoginScreenController* acts as the controller for the *LoginView.fxml* file. It provides the necessary business logic for allowing the user to login to the remote database in order to be able to start hosting music on the *MainView.fxml*.

#### Fields

**@FXML  
private TextField *userLogin*;**

The user enters their user name here. **@FXML  
private PasswordField passwordField;**

The user enters their username here. The Password field provides the necessary functionality for hiding the user's password from the view. **@FXML  
private Rectangle loginRect;**

Used in an ellipse animation. It informs the user if their login attempt was correct by changing green or red if not. **private PathTransition pathTransitionCircle;**

Plays the animation for the *loginRect*.

**ScreensController myController;**

Contains the reference to the *ScreensController* object. Provides the necessary functionality for accessing methods of *ScreensController* which holds the *model* object.

#### Methods

**@Override  
public void initialize(URL url, ResourceBundle rb)**

An animation is built for the *loginRect* primitive using the *PathTransition* class. This transition is specified to perform an ellipse path that will run indefinitely until told to stop.

**@FXML  
private void loginAction(ActionEvent event)**

Calls the login functions whenever the user hits the login button.

**private void Login()**

Starts an asynchronous task that performs the necessary business logic for confirming the credentials that were entered in the *userLogin* and *password* fields.

If the user entered an incorrect password, the *loginRect* will display red for 2.5 seconds before turning back to blue.

However if the user enters correct credentials then the *loginRect* will display green for 2.5 seconds before stopping the *transition* animation and then calling the *setScreen* method implemented by the *ScreensController*.



Figure: LoginSceneController UML

### 5.6.1 Login Sequence Diagram



Figure : Login Sequence Diagram

**1. Handle Event**

**1.1 Start asynchronous SongFileIOFunc task**

**Alt: [QueueSize <= 2]**

**1.1.1 Acquire the foreign key of the added song.**

**1.1.2 Call download function**

**1.1.2.1 Call download function**

**1.1.2.1.1 Create DB object**

**1.1.2.1.1. Download the bytes of the added song.**

**1.1.3 Bytes returned after downloading.**

**1.1.4 Start Async Future HandleFileIO**

**1.1.4.1 Create an mp3 file from the downloaded bytes and then construct a MediaPlayer.**

**1.1.4.2.1. Get Future MediaPlayer**

### 5.6.2 Login View Operation

# C:\year 5\Git\Report\FYP-Report\Report\report lib\slides\Thomas_Flynn_FYP_Presentation\Slide13.PNG

Figure loginView

**Notes:**

When the user of the Music Host Desktop Application runs the application they will be presented with the *LoginView.fxml node* that can be seen in the above figure. The user enters their username and password and then clicks the login button in order to gain access to the *MainView.fxml* node.

## 5.7 MediaPlayer Application Realisation

This section will explain the operation of the *MainSceneController* class and the classes that it uses in relation to its role as a MediaPlayer Application.

The *MainSceneController* class is where the primary functionality of this application resides, operating as the controller for the *MainView.fxml* file. It consists of a Media Engine for playing mp3 files, Business logic for communicating with the remote database and a Bluetooth SDP server for accepting Android Client connections.

In order to help coherently explain the context of the member fields and member functions. I have defined a set of use cases for this class. The explanation of the fields and functions will be done in relation to these use cases.

For the purpose of coherence, the details of the Bluetooth aspect of this class have been omitted and will be discussed in full in the next section of the report.

**Use cases:**

1. Init button clicked

2. Add song button clicked

3. Skip button clicked

4. Play/Pause button clicked

5. Logout button clicked



Figure: MainSceneController UML

### 5.7.1 MainSceneController Fields



Figure: MainSceneController Fields

**@FXML**

**Button initbtn**

Use case 1. Init button clicked.

**@FXML**

**Button addbtn**

Use case 2. Add song button clicked.

**@FXML**

**Circle progressBall**

Use case 2. Add song button clicked.

This Circle is used in the add song animation.

**@FXML**

**Button skipButton**

Use case 3. Skip button clicked.

**@FXML**

**Button playButton**

Use case 4. Play/Pause button clicked.

**@FXML**

**Button logOutButton**

Use case 5. Logout button clicked.

**@FXML**

**Button boolRequest**

Use case 6. Music Host Option buttons Clicked.

**@FXML**

**Button boolDJComment**

Use case 6. Music Host Option buttons Clicked.

**@FXML**

**Button boolSkip**

Use case 6. Music Host Option buttons Clicked.

**@FXML**

**ListView queueList**

This graphical Node displays the list of songs that are currently in the queue.

**FXCollections.observableList SongQueueObservableList**

This watches the list of *QueueSong* objects in the *model.queueList*. It is then bound to the *queueList*. Any changes in the *model.queueList* will result in a change in the *queueList* graphical Node.

**ListChangeListener queueListener**

This listens for a *QueueSong* to be added or removed from the *SongQueueObservableList* and then calls either *songAddedFileIOFunc* or *songRemovedFileIOFunc* respectively.

**AtomicInteger queueSizeAtomic**

This variable is either decremented or incremented by the *queueListener* depending on if a song was added or removed from the *SongQueueObservableList*.

**@FXML**

**ListView selectionList**

This graphical Node displays the list of songs that the currently logged in user has available in their selection.

**@FXML**

**ListView dJComments**

This graphical Node displays a list of comments that the Music Host has been receiving from the Android Clients.

**@FXML**

**ProgressBar songProgBar**

Used to inform the user of the progress of the downloading song from the remote database.

**@FXML  
Slider timeSlider**

Allows the user to seek through the currently playing song.

**@FXML  
Label timeLabel**

Informs the user of the progress of the song playing in the format hours : minutes : seconds.

**MediaPlayer currentPlayer**

Used as the primary reference to the currently playing song on the application. The MediaPlayer API has all the necessary functionality for playing and pausing a song.

**MediaPlayer nextPlayer**

This *MediaPlayer* is used to load the next *MediaPlayer* that will be played when the *currentPlayer* has ended or has been skipped by the FX user or Android Client.

**byte[] nextPlayerBytes**

Stores the bytes necessary for creating an mp3 file for the *nextPlayer.*

**MediaView mediaView**

This acts a container for the *currentPlayer*. The anonymous *ChangeListener* associated with its *mediaView.mediaProperty* provides the necessary functionality for stopping and removing the *oldPlayer* and then subsequently playing the *newPlayer*.

**Path addSongPath**

This represents a simple shape and provides facilities required for basic construction and management of a geometric path. In our case the path is given the element *CubicCurbeTo* which has the specifics of the animation path already defined. The Path is then added to *addSongTransitionPath*.

**PathTransition addSongPathTransition**

This Transition creates a path animation that spans its duration. The translation along the path is done by updating the *translateX* and *translateY* variables of the *progressBall* Node.

**ScreensController myController**

Contains the reference to the *ScreensController* object. Provides the necessary functionality for accessing methods of *ScreensController* which holds the *model*.

### 5.7.2 **MainSceneController** Methods

****

Figure: MainSceneController Methods

#### 5.7.2.0 Initialize

**@Overide**

**void initialize(URL,ResourceBundle)**

This method is called after all @FXML annotated members have been injected.

It is only called once on the implementing controller when the contents of the *MainView.fxml* file have been completely loaded. This allows the *MainSceneController* to perform any necessary post-processing on the content. The methods listed below are called in order within initialize.

**void setGUIOptions**

Sets all the various Nodes to default values and sets tooltips for all the buttons which provide additional information to the user as to what the buttons do when the user hovers over the respective button. This method is also called when the user logs out.

**void setCellFactoryForListViews**

The *selectionView* and *queueListView* have their respective cells configured for the *SelectionSong* and *QueueSong* objects.

**void addMediaViewPropertyListener**

Adds a *ChangeListener* to the *mediaView.MediaPlayerProperty*. This Listener is instructed to stop, dispose and unbind an *oldPlayer*.

The *newPlayer* is then set to be the *currentPlayer* followed by the *setCurrentlyPlaying* method being called. The *currentPlayer* starts playing along with the *pathTransitionCircle* animation.

**void addVolumeandTimeSlider Listeners**

Binds the *timeSlder* and the *volumeSlder* fields to the *mediaView.MediaPlayerProperty* for UI control.

**void buildSongPlayAnimation**

Creates an *ellipsePath* for the *playPath* field.

#### 5.7.2.1 Use Case - Init button

**@FXML**

**void init(Action Event)**

Event fired when the user hits the *init* button. It sets the button from green to red and then calls *addFXObservableListeners* followed by calling the *initSongSelection* method.

**void addFXObservableListeners**

Adds *ChangeListeners* to the *SongQueueObservableList*, *ObservableQueueList* and the *observableDJComments* fields.

These listeners are listening for changes in the associated properties of the *model* object which is held in *myController*.

**void initSongSelection**

Starts an asynchronous task that uses *myController* to call *model.initsongs* method. Once this method returns the *selectionView* is updated with a list of songs that the currently logged in user has.

#### 5.7.2.2 Use Case - Add Song

**@FXML**

**void addSongButtonFunc(ActionEvent event)**

Event fired when the user hits the add button. It gets the index of the song highlighted in the *selectionView* and passes it to the *addSongTask* method.

**synchronized boolean addSongTask (int index)**

This method is executed on the JavaFX Application Thread, despite the misleading title of the method.

**1-** Checks if the selected song is already in the queue by calling *searchQueueForMatch*.

**2-** Creates a *QueueSong* object from the selected *SelectionSong* object.

**3-** Sets the *progressBall* to *DEEPSKYBLUE.*

**4-** Passes the *QueueSong* to the *SongAnimationSetup* method.

**5-** Plays the *addSongTransition* animation.

**void songAnimationSetup(QueueSong)**

**1-** Creates a new path for the *addSongPath* that will have its destination adjusted by the size of the queue.

**2-** Builds a new *PathTransition* for *addSongPathTransition* by setting the animation path to *addSongPath* and the Node for the animation to *progressBall*.

**3-** Finally it defines an event handler for when the animation has finished. The event handler will add the *QueueSong* to the *SongQueueObservableList*.

**synchronized void songAddedfileIOFunc**

Called by *queueListener* when a song has been added to the *SongQueueObservableList*. The method starts an asynchronous task that binds itself to the *songProgBar* field. If the queue size is less than three, it will perform a download operation by getting the *index* value that the added *QueueSong* contains and then passes that *index* to *DB.downloadSongBytes* method which returns the added song's mp3 file from the remote database in the form of bytes. The bytes are then converted into an mp3 file in the project destination folder. This file will then be used to make a *Media* object for the *currentPlayer* or the *nextPlayer* depending on the current size of the queue. After each of these key steps, the *songProgBar* field is updated for the user to see the progress of the download operation.

**synchronized boolean searchQueueForMatch(String)**

Search the song queue for a match using the java 8 stream API. Returns true if it finds a match.

**void writeToCurrentPlayer(MediaPlayer)**

Obtains a *writeLock* for assigning a new *MediaPlayer* to the *currentPlayer*.

**void addEndofMediaListener(MediaPlayer, MediaPlayer)**

Adds an end of media listener to the first argument that instructs it to set the *mediaView.MediaPlayerProperty* to the second argument.

**void setCurrentlyPlaying(MediaPlayer)**

Called when the *mediaView.MediaPlayerProperty* changes. It binds the *progressChangeListener* to the passed argument. The *progressChangeListener* updates the *songProgressBar* and the *timeLabel*.

**void *addAmITheLastSong(MediaPlayer)***

Called when the queue size is equal to one. It creates an anonymous end of media listener that will simply remove the *MediaPlayer* from the song queue because there are no songs to follow. This listener is disregarded when a new song is added to the queue.

### 5.7.2.3 Use Case - Skip Button / Song Ended

**@FXML**

**void iSkip(ActionEvent event)**

Event fired when the user hits the skip button.

Provided the song queue is greater than or equal to two, *mediaView* gets assigned *nextPlayer*. The change listener assigned to the *mediaView* will stop and remove it's *oldPlayer* and start playing the *newPlayer*.

**synchronized void songRemovedFileIOFunc(QueueSong)**

Called by *queueListener* whenever a song has been removed from the song queue. It starts an asynchronous task that binds itself to the *songProgBar*. It's operation is determined by the size of *queueSizeAtomic*. If the size is greater than 1 then it will

**1-** Download the bytes from the db0.UserSongs table.

**2-** Start a Future task that will create an mp3 file from the downloaded bytes and then return a *MediaPlayer* object once finished.

**3-** The returned *MediaPlayer* is assigned to the *nextPlayer* field.

**4-** cleanUpUnusedFiles method is called.

When each of these steps complete, the *songProgBar* is updated for the user to see in the view.

**cleanUpUnusedFiles**

Called whenever a song has been removed from the song queue. It attempts to delete all files in the song file directory. The files still held by the JVM are ignored.

**writeToNextPlayer**

Obtains a *writeLock* for assigning a new *MediaPlayer* to the *nextPlayer*.

#### 5.7.2.4 Use Case - Skip Play/Pause button clicked

**@FXML**

**void iPlay(ActionEvent event)**

Event fired when the user hits the play button. This method toggles the text assigned to the button between "Pause" and "Play". Depending on the previous text assigned to the button, the *currentPlayer* will either be paused or resumed. The *pathTransitionCircle* animation works in sync with this operation.

#### 5.7.2.5 Logout button clicked

**@FXML**

**private void logout(ActionEvent event)**

Event fired when the user hits the logout button. The flow of operation is as follows.

**1-** Stop the current song.

**2-** Call *clearValuesBeforeLoggingOut*.

**3-** Stop pathTransitionCircle animation

**4-** Restart *LoginView*'s animation rectangle animation.

**5-** Set *myController* to the *LoginView* Pane Node.

**public void clearValuesBeforeLoggingOut**

Provides the necessary cleanup for a new user to login.

Order of operation is as follows.

**1-** Remove the *queueListener* from the *SongQueueObservableList* in order to prevent *SongRemovedFileIO* method being called multiple times.

**2-** Set *SongSelectionObservableList*, *observableDJComments* and *SongQueueObservablist* all to null.

**3-** Clear all the values in the *model* object.

**4-** Call *setGUIOptions* method.

**5-** Set *queueSizeAtomic* to 0.

### 5.7.3 Model class

The *Model* class holds the key properties for the *LoginController* and the *MainSceneController*. It performs 2 roles.

Firstly it uses the *DB* object to gain access to the SQL database for performing the necessary SQL queries for the *LoginSceneController* and the *MainSceneController*.

Secondly it returns JSON arrays of *ComBean* objects from the *SongQueue*, *selection* and *DJCommentsData* fields.

#### Model Properties

**Int userID**

Stores the primary key of the logged in user from the *dbo.UserLogin* table.

**List <String> DJCommentsData**

Stores the DJ Comment messages received from the Android Client.

**List <SelectionSong> selection**

Stores the list of *SelectionSongs* for the logged in user.

**List <QueueSong> songQueue**

Stores the list of *QueueSongs* for the logged in user.

#### Methods

**public boolean confirmLogin (userName,password)**

Creates a new DB object that checks if *userName* and *password* arguments match any row in the dbo.*UserLogin* table.

Returns true or false based on the success or failure of this operation.

**public void initSongs**

Creates a new *DB* object that performs an operation that returns a list of *SelectionSong* objects for every song that the logged in user has in the database. The list returned is added to the *selection* property.

**public byte [] downloadSongBytes(index)**

Called from the *MainSceneController* when a song has been added or removed from the queue.

Creates a new *DB* object that calls its own *downloadSongBytes* method.

Once finished, this method returns the bytes to the caller.

**public String songSelectionToJson**

Called when an Android Client connects and wants to request a song.

Returns a JSON array of *ComBean* objects from the list of *SelectionSong* objects.

**public String songQueueToJson**

Called when an Android Client connects and wants to request a song.

Returns a JSON array of *ComBean* objects from the list of *QueueSong* objects.

**public String dJcommentsToJson**

Called when an Android Client connects and sends a DJ Comment message.

Returns a JSON array of *ComBean* objects from the list of *DJCommentsData*.

**public void clearValuesBeforeLoggingOut**

Clears all the properties in the *model* object.

### 5.7.4 Azure SQL Server Database

Using my student email address I was able to setup a Microsoft Azure SQL Database through the Azure portal that can be seen below.

I signed up for a pay as you go subscription for the muzikhostserver.

**Server:** muzikhostserver.database.windows.net

**Database:** FYP

The figure below shows the relationship between the Azure portal and the hierarchy of the SQL database.

Figure : Azure portal

#### 5.7.4.1 FYP.dbo.UserLogin Table

The *UserLogin* Table is used to store the Music Host's login credentials.

#### Table Fields

**userName (VARCHAR 255):**

Stores the name of the Music Host.

**passsword (VARCHAR 255):**

Stores the password of the Music Host.

#### Populated Table

The following figure shows the values that the *UserLogin* Table was populated with in order to carry out testing for building the application.

### C:\year 5\Git\Report\FYP-Report\Report\report lib\report pics\Azure\loginTable.PNG

Figure : UserLogin Table

#### 5.7.4.2 FYP.dbo.UserSongs Table

The *UserSongs* Table is used to store the songs of each individual Music Host. It does this associating each song with the appropriate Id of the user in the *UserLogin* table.

#### Table Fields

**S\_Id (INT):**

Primary of the song.

**songName (VARCHAR):**

Stores the title of the song.

**artistName (VARCHAR):**

Stores the artist of the song.

**dataSize (INT):**

Stores the size of the mp3 file in bytes.

**data(VARBINARY):**

Stores the bytes of the mp3 file.

**Id(INT):**

Foreign key that points to the primary key *ID* in the *UserLogin* table

#### Populated Table

The following figure shows the values the *UserSongs* Table was populated with in order to carry out testing for building the application.

### C:\year 5\Git\Report\FYP-Report\Report\report lib\report pics\Azure\songTable.PNG

### 5.7.5 DB class



The *DB* class holds the connection to the remote database. It obtains it's connection string from the *Ignore* class *getCon()* method.

Figure : Model Class

#### Constructor

Creates a connection to the remote database using the *com.microsoft.sqlserver.jdbc.SQLServerDriver* driver class.

#### Fields

**Ignore ignore**

Holds the following connection String for making a successful connection to the database.

*"jdbc:sqlserver://muzikhostserver.database.windows.net:1433;" +  
 "database=FYP;" +  
 "user=thomas11811@muzikhostserver;" +  
 "password=Zqlllx$8;*

**Resultset rs**

A table of data representing the database result set.

**Statement state**

Used for executing the static SQL statements and returning the results it produces.

**Connection connection**

Provides the connection to the database. The SQL statements are executed and results are returned within the context of this connection.

#### Methods

All of the following methods close the *rs*, *state*, and *connection* fields after they finish executing.

**public int confirmLogin (userName,password)**

Called when the user hits the login button by the *model*.

Checks if *userName* and *password* arguments match any row in the *dbo.UserLogin* table by performing the following query.

*"select id from UserLogin where userName = userName and password = password"*

The result set stores the id of the user if there is a match and then this ID is returned to the caller.

If there is no result set then return a -1 to the caller.

**public List<SelectionSong> initSongs(userID)**

Performs the following SQL query on the *dbo.UserSongs* table.

*"select S\_Id , songname, artistname from UserSongs where Id = userID"*

From this query A *SelectionSong* object is constructed for every result set returned and is then added to a list of *SelectionSong* objects. This list is returned to the caller.

**public byte [] downloadSongBytes(index)**

Called from the *MainSceneController* when a *MediaPlayer* is needed to be constructed.

The argument *index* is the foreign key from the *dbo.UserSongs* table that is stored by the individual *QueueSong* object. With this *index* the following SQL query is performed.

*"select data from UserSongs where S\_Id = index"*

The bytes from this query are returned to the caller.

### C:\year 5\Git\Report\FYP-Report\Report\report lib\new\selectionsong.PNG5.7.5 SelectionSong class

*SelectionSong* operates as a bean for communicating between the remote database and the FX Application. It populates the *selectionView* Node on the GUI. This object gets passed into the constructor of a *QueueSong* when it has been selected to be added to the queue.

#### Constructor

Called by *DB* when it has found a song in the *dbo.UserSongs* table that belongs to the logged in user. It gets passed the 3 parameters.

**1-** The name of the song in the row.

**2-** The name of the artist in the row.

**3-** The primary key of the row.

#### Fields

**String artist**

Stores artist who wrote the song.

Parsed into a *ComBean* JSON object when communicating with the Android Client.

**String song**

Stores the name of the song.

Parsed into a *ComBean* JSON object when communicating with the Android Client.

**int id**

Stores the primary key associated with the song.

#### Methods

All methods are getters for the fields of this class.

### 5.7.6 Queue Song class



The *QueueSong* object performs the role of informing the user and the Android Client what songs are currently in the queue. What separates the *QueueSong* from the *SelectionSong* class is the fact that it has a *votes* field.

#### Constructor

Called by the *MainSceneController* when a song from the selection has chosen to be added to the queue. The parameter passed is a *SelectionSong* object, all if it's fields are copied. The *votes* field is initialized to 2 here.

#### Fields

**String artist**

Stores the artist who wrote the song.

**String song**

Stores the name of the song.

**int id**

Stores the primary key associated with the song.

**AtomicInteger votes**

This variable triggers the current song in the queue to be skipped when it reaches 0.

#### Methods

**public int decrementAndGetVotes**

Called when the Android Client has voted to skip the current song in the queue. It decrements the *votes* field by one and then returns the updated value.

**public void incrementAntiSkipVote**.

Called when the Android Client has chosen a song that is already in the queue. It increments the *votes* field by 1.

### 5.7.7 HandleFileIO class

*HandleFileIO* implements the callable interface allowing it be called as a Future Executor Thread. The class handles the necessary File IO in order to convert bytes into a *MediaPlayer* object.

Figure : HandleFileIO class

#### Fields

**byte[] songData**

Stores the bytes queried from the data field in the *UserSongs* table.

**String songName**

Stores the name of the file queried from the *songName* field in the *UserSongs* table.

#### Methods

**public MediaPlayer call() throws Exception**

Called from the *MainSceneController* when a new *MediaPlayer* needs to be created. It creates an mp3 file using the *songData* and *songName* fields that were assigned when the object was constructed. The URI of the mp3 file is used to create a new *Media* object that will be used to create a new *MediaPlayer* object

## 5.8 MediaPlayer Sequence Diagrams

### 5.8.1 Initialize Button Sequence Diagram



Figure : Initialize Button Sequence Diagram

**1. User clicks the init button.**

**1.1 addFXObservableListeners()**

**1.2 InitSongSelection().**

**1.2.1 initSongs()**

**1.2.1 initSongs()**

**1.2.1.1 initSongs(userID)**

**1.2.1.1.1 initSongs(userID)**

**1.2.1.1.1.2 list of songs**

**1.2.1.1.2 add to selection**

**1.2.3 observable list updates GUI**

### 5.8.2 Add Button Sequence Diagram



Figure : Initialize Button Sequence Diagram

**1. Click**

**1.1 Selection listView index (int)**

**1.1.1 SongSelection.get(index).getSong()**

**1.1.2 SongName(String)**

**1.1.3 song in queue (Boolean)**

**1.1.4 <<Create>>**

**2 Constructor**

**3 QueueSong object**

**1.1.5 SetUpAnimation(QueueSong)**

**1.1.6 Play Animation**

**1.1.7 Animation finished**

### 5.8.3 Add Song Animation Ended Event Sequence Diagram



Figure : Initialize Button Sequence Diagram

**1. Handle Event**

**1.1 Start asynchronous SongFileIOFunc task**

**Alt: [QueueSize <= 2]**

**1.1.1 Acquire the foreign key of the added song.**

**1.1.2 Call download function**

**1.1.2.1 Call download function**

### 5.8.4 Song Added Event Sequence Diagram



Figure : Song Added Event Sequence Diagram

**1.1.1 Acquire the foreign key of the added song.**

**1.1.2 Call download function**

**1.1.2.1 Call download function**

**1.1.2.1.1 Create DB object**

**1.1.2.1.1. Download the bytes of the added song.**

**1.1.3 Bytes returned after downloading.**

**1.1.4 Start Async Future HandleFileIO**

**1.1.4.1 Create an mp3 file from the downloaded bytes and then construct a MediaPlayer.**

**1.1.4.2.1. Get Future MediaPlayer**

**Alt: [QueueSize == 1]**

**1.1.4.2.2 Assign the newly created MediaPlayer to the MediaPlayerfield.**

**1.1.4.2.3 Add an end of media listener to the MediaPlayer that just removes itself from the queue once it reaches the end of media.**

**1.1.4.2.4 Assign the new MediaPlayer to the MediaView triggering an event within the MediaView.**

**Alt: [QueueSize == 2]**

**1.1.4.2.5 Assign the newly created MediaPlayer to the MediaPlayerfield.**

**1.1.4.2.6 Add an end of media listener to the MediaPlayerthat will play theMediaPlayerwhen it ends*.***

**Alt: [default]**

### 8.8.5 Removed/Ended Event Sequence Diagram



Figure : Song Removed/Ended Event Sequence Diagram

**1. Handle Event**

**1.1.1 SkipOK False**

**1.1.2 deleteFailedDeletions**

**1.1.3 deleteRemovedSongFile**

**1.1.4 ball visible false**

**1.1.6 writeToCurrentPlayer(nextPlayer)**

**1.1.7 addAmITheLastSong(currentPlayer)**

**1.1.8 deleteFailedDeletions**

**1.1.9 deleteRemovedSongFile**

**1.1.10 move ball up 1 slot**

**ALT [ queuesize >= 2]**

**1.1.12 get next song Azure foreign Key**

**1.1.13 download(foreignKey)**

**1.1.13.1 download(foreignKey)**

**1.1.13.1.1 <<create>>**

**1.1.13.1.1 download**

**1.1.15 AsyncFuture(bytes, song)**

**1.1.15.1 <<create>>**

**1.1.15.1.1 create MediaPlayer**

**1.1.15.2 return Future**

**1.1.15.2.1 deleteFailedDeletions()**

**1.1.15.2.1 deleteRemovedSongFile()**

**1.1.15.2.3 Get Future Media Player**

**1.1.15.2.4 WriteToNextPlayer(futurePlayer)**

**1.1.15.2.5 addEndOfMediaListeners(CP,NP)**

**1.1.15.2.6 SkipOK True**

**1.1.15.2.6 move ball up 1 slot**

### 5.8.6 Logout Sequence Diagram



Figure Logout Sequence Diagram

**1. clicks**

**ALT: init button hasn't been pressed**

**1.1 stop current song**

**1.2 remove IO stream**

**1.3 clear values**

**2 stop server**

**3 clear selection**

**4 clear queue**

**5 reset GUI options and label**

**1.4 restartAnimationUponLogout**

**1.4.1 restart login animation**

**1.5 logout**

**ALT: init button hasn't been pressed**

**1.6 clear values**

**1.6.1 stopServer**

**1.6.2 reset GUI options and labels**

**1.7 restartLoginAnimation**

**1.8 logout**

**1.8.1 SetScreen(LoginView)**

## 5.9 Music Host Media Player Operation

### 5.9.1 Use Case - Logged In

### C:\year 5\Git\Report\FYP-Report\Report\report lib\slides\Thomas_Flynn_FYP_Presentation\Slide14.PNG

Figure Main View

View

**Notes:**

After the user successfully logs in. They are presented with this screen.

The yellow, blue and red columns represent the song queue, song selection and DJ comments history respectively.

### 5.9.2 Use Case - Setup



Figure Use Case - Setup

**Notes:**

In the example shown in the above figure you can see that the initialize button in the bottom right hand corner has been pressed because it has turned from green to red.

The blue column in the centre that represents the song selection has now been populated with songs that the logged in user has on the cloud database as a result of pressing the initialize button. This button cannot be pressed again until the user logs out and logs back in again.

In the top left hand side of the above figure you can see that the guest options have been specified and in the bottom left hand side of the above figure you can see that the server has been enabled.

### 5.9.3 Use Case - Song Added



Figure Use Case - Song Addded

**Notes:**

In the above figure you can see that the add button has been pressed. This starts an animation with a blue circle that begins at the location of the add button and finished at the location of the last song added in the queue. The song highlighted in the song selection column is then added to the song queue column and the user is notified that the song has started downloading.

### 5.9.4 Use Case - Song Skipped / Song Ended

****

Figure Use Case - Song Skipped / Song Ended

**Notes:**

When a song has ended or has been skipped. The next song in the queue starts playing. The add song animation circle moves up during this operation to keep sync with the last added song in the queue. After this, the next song in the queue starts downloading, provided there is one available.

### 5.9.5 Use Case - Play / Pause

****

Figure Use Case - Play / Pause

**Notes:**

The play animation stops and turns circle red if there is no music playing otherwise animation plays and turn the circle green.

## 5.10 Music Host Bluetooth

This section explains how the desktop application interfaces with the Android application.

### 5.10.1 Serial Port Profile Connection

In the Bluetooth Stack the Service Discover Protocol is bound to the L2CAP. SDP is used to allow devices to discover what services are supported by each other, and what parameters to use to connect to them. **[x]** This Server uses Serial Port Profile which means it's an iterative server because SDP only allows for one connection at a time. SDP defines two roles an initiator and an acceptor which can be seen in the figure below.

**https://developer.bluetooth.org/TechnologyOverview/Pages/SPP.aspx**

### 5.10.2 Bluetooth Communication Diagrams



Figure : Serial Port Profile Connection Diagram

### C:\year 5\Git\Report\FYP-Report\Report\report lib\slides\1\Slide2.PNG

Figure : Song Request Communication Diagram



Figure : DJ Comment Communication Diagram



Figure : Skip Song Communication Diagram

## 5.10.3 MainSceneController Server related elements

#### MainSceneController Fields

**ExectutorService exectorService1**

Runs the wait for connection thread for the Bluetooth SDP server.

**ExecutorService clientExecutor**

Runs the ProcessConnectionThread when an Android Client connects.

**ProcessConnectionThread implements Runnable**

Handles the communication protocol between the Android client and the Music Host.

**Boolean[] boolOptionsArray**

Used to store the current state of the Music Host options. The array is transmitted to the Android Client upon connecting to the Music Host in order to inform the Android Client what options are available to them.

#### MainSceneController Methods

**@FXML**

**startServer(ActionEvent)**

Event fired when the user hits the server button. Depending on the previous state of the text assigned to the server button, call either the *stopServer* method or *startServer* method.

**void startServer**

Creates a *newSingleThreadExecutor* for *executorService1* which will be used for waiting on Android Clients to connect. Once executed the *executorService1* thread gets the local Bluetooth device and sets it to be discoverable. Then it creates a UUID that defines the Bluetooth profile to be SPP. After this it opens a socket and waits for connections.

**void stopServer**

Shuts down the *executorService1* thread that waits on Android Clients to connect.

**@FXML**

**void setSkipSongBool**

Toggles an element in *boolOptionsArray*.

**@FXML**

**void setSongRequestBool**

Toggles an element in *boolOptionsArray*.

**@FXML**

**void setDJCommentsBool()**

Toggles an element in *boolOptionsArray*.

### 5.10.4 ProcessConnectionThread

The *ProcessConnectionThread* is an inner class of the *MainSceneController*. It has the advantage of being able to access all of the media player functionality of the *MainSceneController*. It's created whenever an Android Client makes a successful connection to the SDP server. It handles the communication protocol for the duration of the connection.



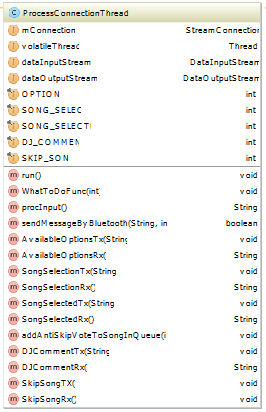


Figure : ProcessConnectionThread UML

Figure : ProcessConnectionThread Fields and Methods

#### Constructor

Takes the socket that the Android client connected to.

#### Fields

**StreamConnector mConnection**

The socket that the communication will take place on.

**Thread volatile Thread**

Used as a reference to the current thread for comparing to ensure an element of thread safety.

**DataInputStream dataInputStream**

Data from the Android client is read from this Stream.

**DataOutputStream dataOutputStream**

Data is sent to the Android client from this Stream.

#### Binary communication protocol constants

**int OPTIONS**

Value = 0

The Android Client sends this value to the server upon connecting to find out what options it has available.

**int SONG\_SELECT**

Value = 1

The Android Client sends this value to the server when requesting to view the Music Host's song selection.

**int SONG\_SELECTED**

Value = 2

The Android Client sends this to the server after making a selection from the Music Host's selection.

**int DJ\_COMMENT**

Value = 3

The Android Client sends this value to the server when making a comment to the Music Host.

**int SKIP\_SONG**

Value = 4

The Android Client sends this value to the server when voting to skip the current song.

#### Methods

**public void run()**

Runs in a loop reading the *dataInputStream* until communication has been terminated.

**public synchronized String procInput()**

Returns a string representation of the bytes read from the dataInputStream.

**public synchronized void sendMessageByBluetooth(String,int)**

Sends an int followed by a String to the Android Client over the dataOutputStream. The Android Client also uses this exact same method.

**public synchronized void WhatToDoFunc(int)**

Called when the Android client sends a value from the binary communication protocol.

Handles the business logic of the binary communication protocol by calling the appropriate implemented *MusicHostCommunication* methods. It does this by switching on the parameter passed.

**public void addAntiSkipVoteToSongInQueue(int)**

Called when the song selected by the Android client is already in the queue. It increments the skipVote member field of the currently playing *QueueSong* object.

#### Implemented MusicHostCommunication Methods

**public String AvailableOptionsRx()**

Called when the Android client sends an OPTIONS value. This happens when the upon connection.

Returns *Arrays.toString(boolOptionsArray)*

*boolOptionsArray* is set by the Music Host toggle option buttons.

**public void AvailableOptionsTx(String)**

The parameter passed is the value returned from *AvailableOptionsRx*.

This parameter is then sent to the Android client who in turn figures out what options they have available to them.

**public String SongSelectionRx()**

Called when the Android client sends a *SONG\_SELECT* value.

It returns the String of the input read by *procInput*.

This String is never used

**public void SongSelectionTx(String)**

Calls *model.songSelectionToJson* and *model.songQueueToJson*. These two JSON Strings are concatenated together with an '&' character. This concatenated String is then sent to the Android Client.

**public String SongSelectedRx()**

Called when the Android client sends a *SONG\_SELECTED* value. It reads the value returned from *procInput*. This value is the song chosen by the Android Client. The chosen song is then checked to see if it's already in the queue.

If it is, it finds the index of the song in the queue by calling the *searchQueueForIndex* method. It then passes this index to the *addAntiSkipVoteToSongInQueue* method which in turn increments the amount of votes needed to skip that song by 1. After this it returns the String

**"The song "** + song + **" is already in the queue, \nRemember to swipe right to check the queue before making a selection"**

Otherwise if the selected song is not in the queue already it calls the *addSongTask* method and then returns the String

**"The song "** + song + **" has been added to the queue"**

**public void SongSelectedTx(String)**

The parameter passed is the value returned from *SongSelectedRx.* This value is sent to the Android Client.

**public String DJCommentRx()**

Called when the Android Client sends a *DJ\_COMMENT* value.

Returns the String of the input read by *procInput*.

This String is the DJ Comment that was sent from the Android Client.

**public void DJCommentTx(String)**

The parameter passed is the value returned from *DJCommentRx.*

Adds the parameter passed to *observableDJComments*, which in turn updates the *dJComments ListView* node. This operation is performed on the JavaFX Application thread.

Then it calls *model.DJCommentsToJson* and *model.songQueueToJson*. These two JSON Strings are concatenated together with an '&' character. This concatenated String is then sent to the Android Client.

**public void SkipSongRx()**

Called when the Android Client sends a *SKIP\_SONG* value.

It decrements the AtomicInteger *skipVote* field of the currently playing *QueueSong* object.

If this field is less than or equal to zero then it starts an asynchronous task to perform the *iSkip* method.

**public void SkipSongTx()**

Calls *model.DJCommentsToJson* and *model.songQueueToJson*. These two JSON Strings are concatenated together with an '&' character. This concatenated String is then sent to the Android Client.

## 5.10.5 Server Button Sequence Diagram



Figure : Logout Sequence Diagram

**1 click**

**1.1 set button on**

**1.2 startServer()**

**1.2.1 getLocalBTdevice**

**1.2.2 setDiscoverable(TRUE)**

**1.2.3 create UUID**

**1.2.4 create BT URL**

**1.2.5 Connector.open(url)**

**1.2.6 notifer.accetAndOpen()**

**1.2.7 Executors.newSingleThreadExecutor()**

**1.2.8 <create>>ProcessConnectionThread**

**1.2.9 Shutdown Thread in 40 seconds**

**1.4 Set Button ON**

**1.5 Set Discoverable True**

**1.7 Set Button OFF**

**1.8 Set Discoverable False**

### 5.10.6 ProcessConnectionThread Sequence Diagram



Figure : Logout Sequence Diagram

**1. <<create>>**

**1.1 Constructor(connection)**

**2 Executorservice.submit()**

**2.1 new OutputStream**

**2.2 new InputStream**

**2.3 ID.readInt()**

**2.4 WhatToDoFunc(int)**

**2.5 IS.readInt()**

**2.6 WhatToDo(int)**

**2.7 End Communication**

**2.8 WhatToDo(Int)**

**2.9 Return**

### 5.10.7 WhatToDoFunc Options, Song Request, Song Selected Sequence Diagram



Figure : WhatToDoFunc, Options, Song Request, Song Selected Sequence Diagram

**1. whatToDoInt**

**ALT: OPTIONS**

**1.1 GET Options**

**1.1.1 calc options**

**1.1.2 Options (Boolean array)**

**1.2 SEND(OPTIONS)**

**1.2.1 To Client**

**ALT: SONG\_REQUEST**

**1.3 SongSelectionRX()**

**1.4 SongSelectionTX()**

**2 selection.toJSON**

**1.4.1 Get Selection**

**1.4.2 JSON\_ARRAY**

**1.4.3 JSON to Client**

**ALT: SONG\_SELECTED**

**3 procInput()**

**4 Search Queue(song)**

**ALT: Song NOT in Queue**

**5 AddSongAsync(song)**

**6 SONG OK True**

**ALT: Song NOT in Queue**

**7 ANTISkipVotes**

**8 SONG OK false**

**1.6 Send SONG OK (Boolean)**

### 5.10.8 WhatToDoFunc DJ Comment Sequence Diagram



Figure : WhatToDoFunc DJ Comment Sequence Diagram

**1. whatToDoInt**

**ALT: DJ\_COMMENT**

**1.1 Get DJ Comment**

**1.1.1 procInput()**

**1.1.2 DJ\_Comment**

**1.2 DJCommentTx(String)**

**1.2.1 addToDJComments**

**1.2.2 Get DJ Comments**

**1.2.2.1 DJ to JSON**

**1.2.2.1 Comments as JSON**

**1.2.3 Get Queue**

**1.2.3.1 Queue to JSON**

**1.2.3.1 Queue as JSON**

**1.2.4 Concat JSON**

**1.2.5 Send to Client**

### 5.10.9 WhatToDoFunc Skip Song Sequence Diagram



Figure : WhatToDoFunc Skip Song Sequence Diagram

**1. whatToDoInt**

**ALT: SKIP\_SONG**

**1.1 SkipSongRx()**

**2 currentSong.decrementVotes**

**3 Skip Current Song**

**1.2.1 Get DJ Comments**

**1.2.1.1 DJ to JSON**

**1.2.1.2 Comments as JSON**

**1.2.2 Get Queue**

**1.2.2.1 Queue to JSON**

**1.2.2.2 Queue as JSON**

**1.2.3 Concat JSON**

**1.2.4 Send to Client**

# 6 Music Host Client Android Application

The user of the Android Application connects to a Music Host through Bluetooth. Once connected they will have the following options, provided the Music Host has enabled them.

The first is the ability to choose a song from the Music Host's song selection, the user will be prompted with the Music Host's song selection and the song Queue. From this information the user can make an informed decision as to what song from the selection they should choose. Once the user has selected a song the choice is sent to the Music Host who will in turn add it to the song queue provided it is not there already. The user will be informed by a toast message if the operation was successful or not and then the connection is terminated to allow other users of the Application to connect to the Music Host.

The second is the ability to send a text message to the Music Host. The context of the text message is generally in the form of a song request. The user will be presented with a history of text messages that the Music Host has received along with the current song queue. The connection is then terminated.

The third is the ability to vote to skip the current song that is playing. This intention is sent to the Music Host which then in turn tallies the amount of skip votes the current song has. Provided the current song has received the necessary amount of skip votes. The user will hear on the local sound system that the current song has been skipped. This feature also has the same response operation as the second ability by where the user is presented with the history of text messages and the list of the current songs in the queue after performing the described action. The user will be able to see the current tally of skip votes that each song in the queue has.

## 6.1 Foundation

The Music Host Client Android application was built on the Bluetooth Chat Android application developed by the Github user marcuspimenta [x]. It provided me with all the necessary functionality needed in order to give me a head start on the project.

**https://github.com/marcuspimenta/Chat-Bluetooth-Android**

#### About the Bluetooth Chat Application

The Bluetooth Chat Application was developed to connect to other users with the same application. Once connected communication began. It's primary functionality resides within the *ChatBusinessLogic* class which performs the necessary business logic for connecting and communicating to other Bluetooth devices

### 6.1.1 ChatBusinessLogic class

The *ChatBusinessLogic* class implements *OnConnectionBluetoothListener*, *OnSearchBluetoothListener* and OnbluetoothDeviceSelectedListener. All communication business logic takes place within this class.



Figure ChatBusinessLogic UML

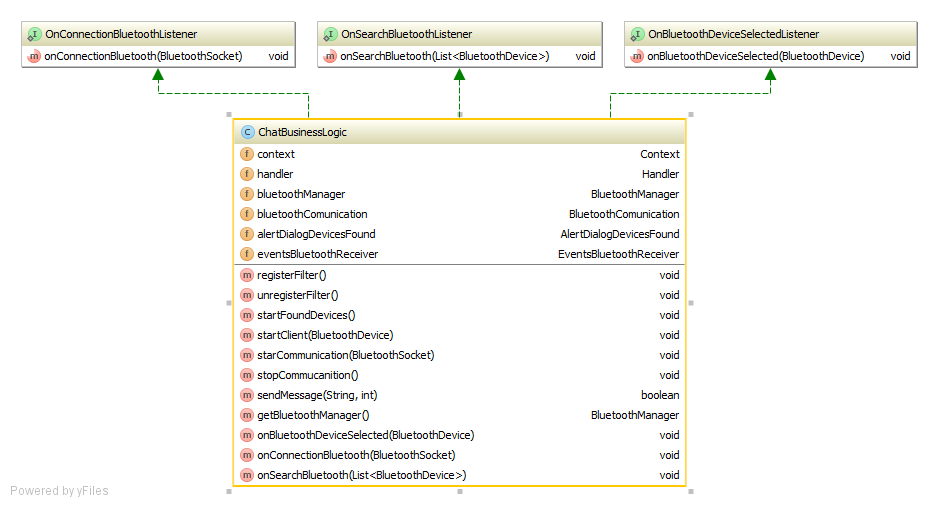


Figure ChatBusinessLogic Fields and Methods

#### Fields

**Context context**

Used as a reference to the MainActivity for passing to the BluetoothCommunication constructor.

**Handler handler**

Used as a reference to the MainActivity's embedded handler for passing to the BluetoothCommunication constructor. It receives an int and a String from the BluetoothCommunication thread whenever the Music Host responds to the Android Client. The int is used to decide what action the MainActivity should perform next.

**BluetoothManager bluetoothManager**

Gets the default BluetoothAdaptor for the device.

**BluetoothCommunication bluetoothCommunication**

Extends thread. Handles the binary communication protocol for communicating with the Music Host. When it receives a value from this protocol it sends it to the embedded handler in the *MainActivity*.

**AlertDialogue alertdialogue**

Implements *onClickListene*r. It is used to inform the Android Client if a Music Host has been found or not.

**EventsBluetoothReceiver eventsBluetoothReceiver**

Extends Broadcast receiver. It holds a list of Bluetooth devices and also registers two intents to listen for.

1- Bluetooth device found.

Whenever a Bluetooth device is found, it gets added to the list of Bluetooth devices.

2- The search for Bluetooth devices has ended.

Once the search for a Music Host has ended, it cycles through all the names of the Bluetooth devices found. If it finds the one it's looking for it displays it on the AlertDialogue for the user to see.

#### Methods

**public void startFoundDevices()**

Starts an implicit intent on the default Bluetooth adaptor that starts searching for nearby Bluetooth devices.

**public void startClient(BluetoothDevice)**

Starts an asynchronous task that establishes a secure Bluetooth socket connection to the Music Host once finished.

**public void starCommunication(BluetoothSocket)**

Creates and starts a BluetoothCommunication thread and passes it the BluetoothSocket.

**public void stopCommucanition()**

Calls BluetoothCommunication.stopCommunication() which terminates the connection to the Music Host.

**public boolean sendMessage(String,int)**

Calls BluetoothCommunication.sendMessageByBluetooth and passes it the String and int parameters. The int is used in the binary communication protocol for communicating with the Music Host.

**@Override  
public void onBluetoothDeviceSelected(BluetoothDevice bluetoothDevice)**

Calls the startClient method when the user attempts to connect to the Music Host.

**@Override  
public void onConnectionBluetooth(BluetoothSocket bluetoothSocket)**

Once a BluetoothSocket connection has been esablished to the Music Host. This implemented listener calls the startCommunication method.

**@Override  
public void onSearchBluetooth(List<BluetoothDevice> devicesFound)**

Updates the AlertDialogue to inform the user if a Music Host was found or not.

## 6.2 Realisation

With all the basic functionality for the Android Client Application in place. The realisation process began. I first started by building on the Main Activity of the Bluetooth Chat application by laying out all the integral GUI widgets needed for the Android Client Application. I then began development of the SongRequestActivity and the DJActivity.

### 6.2.1 MainActivity

The MainActivity is created when the user opens the application. It performs the primary business logic operations for the application. The figure below describes how the MainActivity and its components relate to the ChatBusinessLogic object.



Figure MainActivity UML

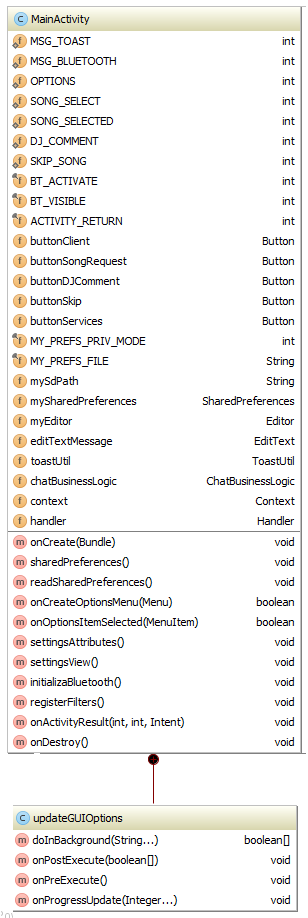
Fields **final int ACTIVITY\_RETURN**

Figure MainActivity Fields and Methods

Used in onActivity result to check if the user pressed the return button to exit the Activity. **Button buttonClient**

Starts a search for Music Hosts. **Button buttonSongRequest**

Set to invisible by default. It sends a SONG\_SELECT constant to the Music Host. **Button buttonDJComment**

Set to invisible by default. It sends a DJ\_COMMENT constant to the Music Host followed by a String.

**EditText editTextMessage**

The user enters the DJ Comment they would like to make here. This field is read when the buttonDJComment is pressed. **final String MY\_PREFS\_FILE**

Stores the users name. This value is concatenated to the DJ Comment text message. **Button buttonSkip**

Set to invisible by default. It sends a SKIP\_SONG constant to the Music Host.

**ToastUtil toastUtil**

The reference to this object is passed to other classes that do not have the ability do a Toast. **ChatBusinessLogic chatBusinessLogic**

Handles the business logic for connecting and communicating to the music host. **Context context**

Contains the context of the MainActivity. The context is passed to other classes.

**Handler handler**

Performs the role of deciding what operation to initiate after a response from the Music Host has been received. It handles messages sent from the BluetoothCommunication thread. These messages come in the form of an int followed by a String. The handler performs the appropriate action based on the int it receives.

#### Binary communication static constants

These constants are used for the binary communication protocol by both MainActivty's embedded Handler and the BluetoothCommunication thread. The context of their use will be explained later.

**public static int *OPTIONS* = 0  
public static int *SONG\_SELECT* = 1  
public static int *SONG\_SELECTED* = 2  
public static int *DJ\_COMMENT* = 3  
public static int *SKIP\_SONG* = 4**

#### Methods

**@Override  
public void onCreate(Bundle)**Called when the user opens the application. It sets the MainActivity View to *chat\_activity.*xml after this it calls the following methods in turn.

**public void settingsAttributes**

Creates new ToastUtil and ChatBusinessLogic objects. **public void settingsView()**

Assigns anonymous OnClickListeners for the GUI widgets.

**public void initializeBluetooth**

Ensures that the user has enabled Bluetooth on their device.

**public void registerFilters**

Registers the intents for the EventsBluetoothReceiver to listen for.

**public void sharedPreferences()**

Writes the value "Tom" to the users MY\_PREF\_FILE. This value will be used when sending a text message to the Music Host.

#### updateGUIOptions

Extends AsyncTask. This is an inner class of the MainActivity. It gets Executed when the MainActivity's embedded handler receives an *OPTIONS* int from the BluetoothCommunication Thread. It's role is to parse the array of booleans that the Music Host sends to the Android Client. This array represents the options that the Music Host has enabled for its Android Clients. Once the task has finished parsing the array, it updates the *buttonSongRequest, buttonDJComment* and *buttonskip* fields to visible depending on the values inside the array.

### 6.2.2 SongRequestActivity

The SongRequestActivity gets created when the MainActivity's embedded handler receives an SONG\_SELECT int from the BluetoothCommunication Thread.

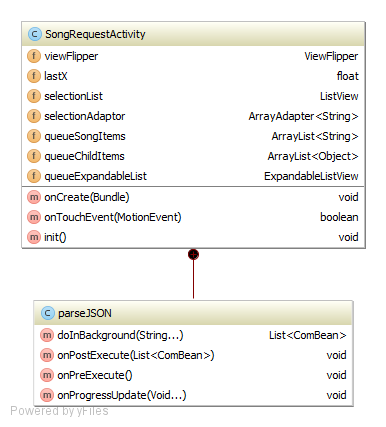


Figure SongRequestActivity UML

#### Fields

**ViewFlipper viewFlipper**

[ViewAnimator](http://developer.android.com/reference/android/widget/ViewAnimator.html) that will animate between the *selectionList* and queueExpandableList views that have been added to it. Only one child is shown at a time.  **ListView selectionList**

Displays the list of songs that the Music Host has in their selection. **ArrayAdapter<String> selectionAdaptor**

A BaseAdapter that is backed by an array of Strings for each song in *selectionList*. **ArrayList<String> QueueSongItems**

Contains the song name in the parent of the queueExpandableList node. **ArrayList<Object> queueChildItems**

Contains the artist of the song and the tally of skip votes it has in the child of the queueExpandableList node.

**ExpandableListView queueExpandableList**

Displays the list of songs in the queue of the Music Host.

#### Methods

**@Override  
protected void onCreate(Bundle)**

When called the View is set to the layout of view\_flipper\_main.xml.

The fields *selectionList* and queueExpandableList are set to the appropriate widgets defined in the view\_flipper\_main.xml.

Anonymous OnClickListeners are then assigned to both these fields.

The Bundle parameter passed has a JSON array which contains the song selection and song queue of the Music Host. This JSON array is put into the constructor of the AsyncTask ParseJSON and then executed.

**boolean onTouchEvent(MotionEvent)**

Method to handle touch event like left to right swap and right to left swap for switching between the viewFlipper's children. This allows the user to view both the song selection and the song queue.

#### ParseJSON

Extends AsyncTask. It's constructor receives a JSON array which contains the song selection and the song queue that the Music Host has. When this task is executed it's job is to first separate the JSON array into two separate arrays.

It does this by searching for the '&' character. Then it processes the two JSON arrays separately. Just the song name is extracted from each object in the first JSON array. Then the song name, song artist and song votes are extracted from the second JSON array. Once finished processing both arrays it updates the *selectionList* and the queueExpandableList for the user to see.

### 6.2.3 DJActivity

This Activity is created when the Music Host responds to either a DJ\_COMMENT or SKIP\_SONG intention. The DJActivity is virtually the same as the SongRequestActivity. The main difference is that the *selectionList* field displays the list of DJ comments that the Music Host has been receiving instead of the selection of songs.

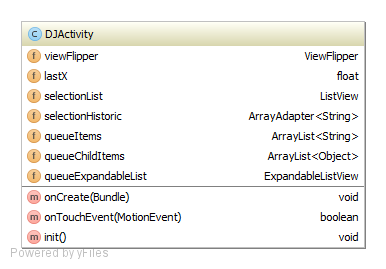


Figure DJActivity Fields and methods

### 6.2.4 Bluetooth Communication

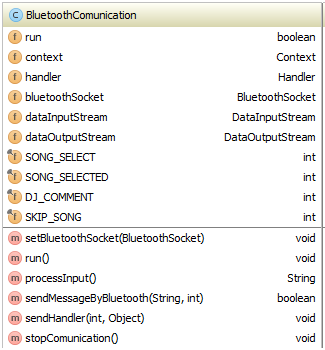


Figure BluetoothCommunication Fields and methods

#### Fields

**boolean run**

Used to exit the loop that executes in the run method. **Context context**

Contains the reference to the MainActivity. **Handler handler**

Contains the reference to the MainActiity's Handler. **BluetoothSocket bluetoothSocket**

Contains the reference to the connected BluetoothSocket. **DataInputStream dataInputStream**

Data sent from the Music Host is read from this stream. **DataOutputStream dataOutputStream**Data sent to the Music Host is written to this stream.

#### Methods

**void setBluetoothSocket(BluetoothSocket)**

The parameter passed is the reference to the connected BluetoothSocket. This is set to the BluetoothCommunication's bluetoothSocket field.

**void run()**

Runs a loop that handles communication with the Music Host until the boolean flag *run* is set to false.

**String processInput()**

Reads the bytes from the dataInputStream and returns a String.

**void SendMessageByBluetooth(String,int)**

Called when sending an intention to the Music Host.

Sends an int followed by writing a String to the dataOutputStream.

**void sendHandler(int,Object)**

Called when the BluetoothCommunication thread has finished reading an int from the dataInputStream. The int parameter passed is the value received from the Music Host. The Object parameter is a String received from the Music Host. These parameters are then sent to the MainActivity's embedded Handler.

**void stopCommunication**

Sets the boolean flag *run* to false. Then closes the bluetoothSocket followed by closing the dataInputStream and dataOutputStream.

## 6.3 Sequence Diagrams

### 6.3.1 Open Application SD



Figure Open Application SD

**1 openApp()**

**1.1 setContentView**

**1.2 <<create>>**

**1.2.1 <<create>>**

**1.2.1..1 get default BTAdatpor**

**1.2.1.2 BTM object**

**1.2.2 new AlertDialogDevicesFound**

**1.2.3 onSearchBluetoothListener**

**3 Set onReceive**

**4 EBTR object**

**1.3 CBL object**

**1.4 add button listeners**

**1.5 initializeBluetooth**

**1.6 registerFilter**

**1.6.1 registerFilter**

**1.6.1.1 register receivers**

**1.8 sharedPreferences**

**1.9 OnStart()**

### 6.3.2 Bluetooth Search SD



Figure Bluetooth Search SD

**1 clicks**

**1.2 startFoundDevices**

**1.1.1 stopCommunication**

**1.1.2 showProgress()**

**1.1.2.1 clear BT devices**

**1.1.2.1 clear BT devices**

**1.1.2.2 show progress dialog**

**1.1.3 startDiscovery**

**1.1.3.1 adaptor.startDiscovery**

### 6.3.3 Events Bluetooth Receiver SD



Figure Events Bluetooth Receiver SD

**1 intent**

**ALT: BluetoothDevice.ACTION\_FOUND == intent**

**1.1 intent.getParceableExtra**

**1.2 addbluetoothdevice**

**ALT: BluetoothDevice.ACTION\_DISCOVERY\_FINISHED == intent**

**1.3 closeProgress()**

**ALT: devices found > 0**

**1.4 devicesfound**

**1.4.1 onSearchBluetoothListener**

**1.4.1.1 devicesFound**

**1.4.1.1.1 add device to alert dialog**

**1.4.1.1.2 show alert dialog**

**ALT: no devices found**

**1.6 Toast "not found"**

### 6.3.4 Connecting to Music Host SD



Figure Connecting to Music Host SD

**1 clicks**

**1.1 <<create>>**

**1.2 BCT object**

**1.3 execute(bluetoothDevice)**

**1.3.1 onPostxecute(connected)**

### 6.3.5 Connected to Music Host SD

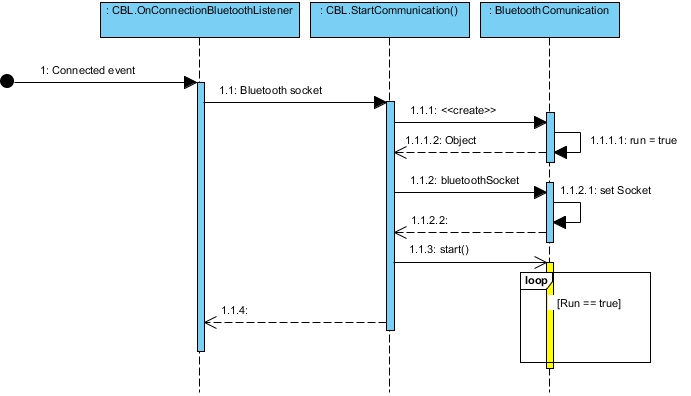


Figure Connected to Music Host SD

**1 Connected event**

**1.1 Bluetooth socket**

**1.1.1 <<create>>**

**1.1.1.1 run = true**

**1.1.1.2 object**

**1.1.2 bluetoothSocket**

**1.1.2.1 setSocket**

**1.1.2 start()**

### 6.3.6 Bluetooth Communication Thread SD



Figure Bluetooth Communication Thread SD

**1 (Context, Handler)**

**1.1 set context**

**1.2 set handler**

**1.3 run = true**

**2 bluetoothsocket**

**2.1 setBluetoothSocket**

**3 Run()**

**3.1 new dataInputStream**

**3.2 new dataOutputStream**

**3.3 readInt()**

**3.5 sleep (1sec)**

**ALT: OPTIONS RX**

**3.6 processInput()**

**3.7 sendHandler**

**ALT: SONG\_SELECT RX**

**3.8 processInput()**

**3.9 sendHandler**

**ALT: SONG\_SELECTED RX**

**3.10 processInput()**

**3.11 sendHandler**

**ALT: DJ\_COMMENT RX**

**3.12 processInput()**

**3.13 sendHandler**

**ALT: SKIP RX**

**3.12 processInput()**

**3.13 sendHandler**

### 6.3.7 Main Activity Handler SD



Figure Main Activity Handler SD

**1 handleMessage**

**1.1 updateGUIOptions()**

**1.2 new Intent()**

**1.3 Bundle.add(message)**

**1.4 add bundle to intent**

**1.5 startActivityForResult**

**1.6 toast message**

**1.7 stopCommunication**

**1.8 new Intent()**

**1.9 Bundle.add(message)**

**1.10 add bundle to intent**

**1.11 startActivitity for result**

**1.12 new Intent()**

**1.13 Bundle.add(message)**

**1.14 add bundle to intent**

**1.15 startActivityForResult**

### 6.3.8 onCreate SongRequestAcitivity SD



Figure onCreate SongRequestAcitivity SD

**1 <<create>>**

**1.1 setContentView**

**1.2 set selection listview**

**1.3 new adaptor**

**1.4 add adaptor to selection listView**

**1.5 selectionList.setOnItemClickListener**

**1.6 set queue listView**

**1.7 new adaptor**

**1.8 add adaptor to queue listView**

**1.9 get Intent**

**1.10 Bundle = Intent.getExtras**

**1.11 Bundle.get(JSON)**

**1.12 <<create>>**

**2 constructor**

**3 object**

**1.13 execute**

**1.13.1 parse JSON**

**1.13.2 update GUI**

### 6.3.9 Parse JSON Async Task SD



Figure Parse JSON Async Task SD

**1 execute**

**1.1 String**

**1.1.1 split String**

**1.1.2 new JSONArray**

**1.1.3 new ComBean**

**1.1.4 comBean = parseJSON**

**1.1.5 returnList.add(comBean)**

**1.1.6 newJSONArray**

**1.1.7 new ComBean**

**1.1.8 comBean = parseJSON**

**1.1.9 RETURNlIST.ADD(ComBean)**

**1.1.10 returnList**

**1.1.10.2 add to queue**

**1.1.10.3 update GUI**

### 6.3.10 SongRequestAcitivity song selected SD



Figure SongRequestAcitivity song selected SD

**1 clicks**

**1.1 getIntent()**

**1.2 new Bundle**

**1.3 selection.getItem(position)**

**1.4 put song in bundle**

**1.5 put bundle in intent**

**1.6 setResult(RESULT\_OK)**

**1.7 finish()**

**1.7.1 Bundle.get(song)**

**1.7.2 toast "song has been added to the queue"**

**1.7.3 SONG\_SELECTED**

**1.7.3.1 send message**

**1.7.3.1.1 sendMessageByBluetooth**

**1.7.3.1.2 success(Boolean)**

**1.7.4 succes (boolean)**

### 

### 6.3.11 Song Request, DJ comment, Skip button SD



Figure Song Request, DJ comment, Skip button SD

**1 clicks**

**1.1 intention**

**1.1.1 sendMsg**

**1.1.1.1 sendMsgByBluetooth**

**1.1.1.2 Success (Boolean)**

**1.2 Success (Boolean)**

**2 update GUI**

## 6.4 Operation

This section explains in terms of use cases how the application operates with aid of screenshots from the application in operation.

### 6.4.1 Use Case - Open App



Figure Use Case - Open App

**Notes:**

When the Music Host Client opens up the application, this is what they see. Please take note of the invisible buttons highlighted in the above figure. This aspect will be explained further in the "Connected" use case. The following use case will demonstrate clicking the "Search for Music Host (Bluetooth)" button that can be seen in the figure above.

### 6.4.2 Use Case - Search For Music Host (Bluetooth)



Figure Use Case - Search for Music Host (Bluetooth)

**Notes:**

After clicking the "Search for Music Host (Bluetooth)" button. The user now has to wait to connect to the Music Host. Once prompted, the user clicks on the Music Host in the Alert Dialogue box. This will attempt to connect the user the selected Music Host.

### 6.4.3 Use Case - Connected



Figure Use Case - Connected

**Notes:**

Now that the user has successfully connected to the Music Host, the 3 previously invisible buttons will now become visible depending on the options the Music Host has delegated. The following use will discuss the clicking of the "Request a song" button.

### 6.4.4 Use Case - Song Request



Figure Use Case - Song Request

**Notes:**

After the user clicks the "Request a song" button they will be prompted with the "Song Selection" screen that can been seen on the left hand side in the above figure. The user can swipe to the right to view the current song Queue in order to make an informed decision about what song they should pick to add to the song queue.

### 6.4.5 Use Case - Song Accepted / Not Accepted



Figure Use Case - Song Accepted / Not Accepted

**Notes:**

After the user has selected a song they will be returned to the main activity. A toast from the Music Host will be displayed shortly after to inform the user if their song selection was successful or not.

On the left hand side of the above figure you can see the toast *"The song La Partida has been added to the queue"*.

However the right hand side of the above figure displays a toast informing the user that "*The song La Partida is already in the queue, Remember to swip right to check the queue before making a selection"*.

### 6.4.6 Use Case - DJ Comment



Figure Use Case - DJ Comment

**Notes:**

This use case discusses the DJ comment feature. On the MainActivity the user enters a message in the EditTextField that can be seen on the left hand side in the above figure. Then the user hits the "Shout out to DJ" button to send the message. The user is then prompted with the entire history DJ Comments chat session. From here the user also has the option to swipe right in order to view the current song queue, similar to the Song Request use case.

### 6.4.7 Use Case - Skip Song



Figure Use Case - Skip Song

**Notes:**

This use case discusses the skip song feature. After user has click the "skip" button. They will be prompted with the Song Queue. The user can expand each song and view the artist and the "skip votes required" field. If you take a close look at the right hand side of the above figure, you will see that the current song has only 1 more vote needed in order to be skipped while the others have 2. If another user of the application hit the skip button the current song would be skipped and the next song in the queue would play.

# 7 System Integration

Integrating the JavaFX application with the remote database was not particularly difficult to obtain basic functionality. What proved to be the challenge with this aspect of the project was limiting the time spent connected to the database. To achieve this I designed a Song class that stored locally all the vital information in a given row from the UserSongs table. This class acted as a bean for representing songs in the database while being disconnected from it.

To further limit time connected to the database I devised my own sequence of operation for the song queue, whereby only two mp3 files from the remote database needed to be ready locally on the machine at any given time. This allowed the user of the desktop application to add multiple songs to the queue without having to wait for each one to finish downloading. This was my proudest achievement.

Integrating the JavaFX application with the Android application proved to be quite a difficult task. Luckily I was able to make a major breakthrough early on in the project development lifecycle. This breakthrough came towards the end of sprint 3 in December 2015. It involved successfully connecting to the Music Host with the Android application and sending a message. This was the result of using the same communication methods for both entities.

Once I had established a foothold on integrating the two systems I continued to build on this functionality. I repeatedly ended up breaking the communication functionality during the project's lifecycle. To solve this I used Git version control to return to a previous version of successful communication. Git proved to be the single most important tool in the integration of these two systems. It not only provided me with a way of undoing a mistake but it also allowed me to closely examine how the mistake broke the system by comparing the two different versions in question.

In order to get the desired functionality in the time required I developed my own binary protocol of writing and reading integers. This basic communication protocol although limited, allowed me to build the integral features quickly and efficiently.

# 8 Project Statistics

I managed to keep up a consistent level of documentation throughout the entire development of this project. This is evident in the project logbook I used which detailed tasks completed and time spent working on the project.

## 8.1 Github Repositories

Github repositories were used for version control in building of this project. I pushed local commits for the Music Host desktop application onto the FYP-GUI repository and the Music Host Client application local commits onto the FYP-Android repository.

### 8.1.1 FYP-GUI Repository

**Commit graph:**

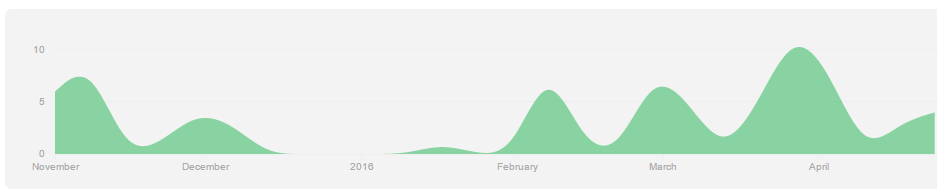


Figure FYP-GUI Commit Graph

**Total commits to master branch:** 79

### 8.1.2 FYP-Android Repository

**Commit graph:**

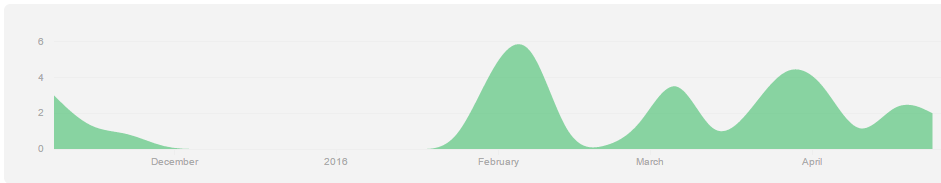


Figure FYP-Android Commit Graph

**Total commits to master branch:** 39

## 8.2 Toggl Time Documentation

I have seperated the total time spent working on the project that was documented by Toggl into the following college semesters.

#### 8.2.1 September - December

**Time bar chart**

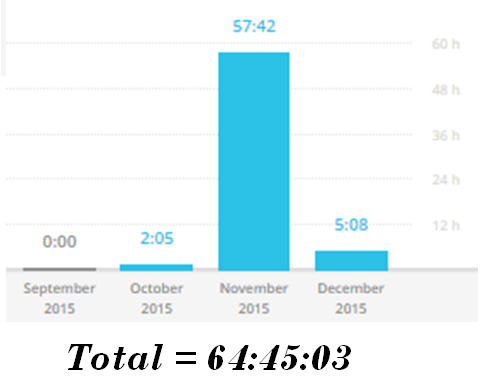
****

Figure September - December time bar chart

**Time Donut Chart**

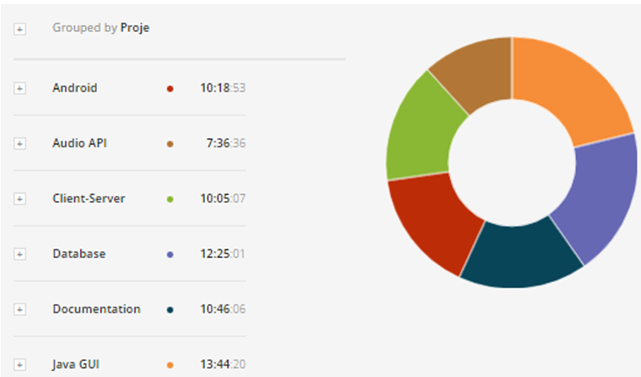
****

Figure September - December donut time chart

#### 8.2.2 January - April

**Time bar chart**

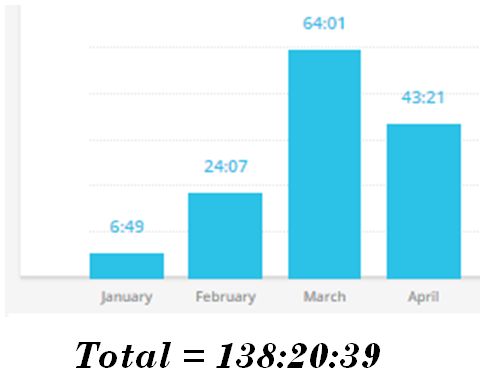


Figure January - April time bar chart

**Time Donut Chart**



Figure January - April time donut chart

# 9 Conclusion

Since the very beginning, this project set out to be a demonstration of my abilities and what I have learned from this course. It is my opinion that the finished product provides an honest realisation of these skills.

From building this project I have gained experience with invention. The overall design was based on ideas of how I felt the two systems should operate. The GUI design for both the desktop application and the Android application are a direct realisation of what I wanted the eventual product to look like.

After testing the finished realisation in a real world environment the feedback I received from both people in the public domain and DJs is that the application is pragmatic and essential. DJs also stated however that they would find my desktop application too limiting for playing music compared to others that are freely available.

On reflection I realised that more time should have been spent on the development of the Android application as that's where the market is. If I were to start this project again I would develop the Music Host Application as a plug-in for a media player that already had the essential functionality of playing music established. This would be far more practical and marketable than my custom made media player application that is dependent on a remote database for playing music.

In conclusion I would consider the project to be a success. I am proud to present this project as my own creation. I have learned a great deal about the creation and realisation of ideas into the real world environment.

# Bibliography

graphs and commits android

https://github.com/g00291875/FYP-Android/commits/master

https://github.com/g00291875/FYP-Android/graphs/contributors

**activites**

https://github.com/g00291875/FYP-Android/blob/master/Client/src/com/chat/bluetooth/activity/AzureMobileServicesActivity.java

https://github.com/g00291875/FYP-Android/blob/master/Client/src/com/chat/bluetooth/activity/DJActivity.java

https://github.com/g00291875/FYP-Android/blob/master/Client/src/com/chat/bluetooth/activity/GenericActivity.java

https://github.com/g00291875/FYP-Android/blob/master/Client/src/com/chat/bluetooth/activity/MainActivity.java

https://github.com/g00291875/FYP-Android/blob/master/Client/src/com/chat/bluetooth/activity/SongRequestActivity.java

**alertdialog**

https://github.com/g00291875/FYP-Android/blob/master/Client/src/com/chat/bluetooth/alertdialog/AlertDialogDevicesFound.java

**broadcast receiver**

https://github.com/g00291875/FYP-Android/blob/master/Client/src/com/chat/bluetooth/broadcast/EventsBluetoothReceiver.java

**business**

https://github.com/g00291875/FYP-Android/tree/master/Client/src/com/chat/bluetooth/business

**communication**

https://github.com/g00291875/FYP-Android/blob/master/Client/src/com/chat/bluetooth/communication/BluetoothClient.java

https://github.com/g00291875/FYP-Android/blob/master/Client/src/com/chat/bluetooth/communication/BluetoothComunication.java

https://github.com/g00291875/FYP-Android/blob/master/Client/src/com/chat/bluetooth/communication/MusicHostCommunication.java

**manager**

https://github.com/g00291875/FYP-Android/tree/master/Client/src/com/chat/bluetooth/manager

**bluetoothclient task**

https://github.com/g00291875/FYP-Android/blob/master/Client/src/com/chat/bluetooth/task/BluetoothClientTask.java

**fyp gui**

**framework**

https://github.com/g00291875/FYP-GUI/blob/master/FXGUI/src/com/framework/LoginSceneController.java

https://github.com/g00291875/FYP-GUI/blob/master/FXGUI/src/com/framework/*MainSceneController*.java

https://github.com/g00291875/FYP-GUI/blob/master/FXGUI/src/com/framework/MusicHostFramework.java

https://github.com/g00291875/FYP-GUI/blob/master/FXGUI/src/com/framework/*ScreensController*.java

**view**

https://github.com/g00291875/FYP-GUI/blob/master/FXGUI/src/com/View/*LoginView*.fxml

https://github.com/g00291875/FYP-GUI/blob/master/FXGUI/src/com/View/*MainView.fxml*

***model***

https://github.com/g00291875/FYP-GUI/blob/master/FXGUI/src/com/*model*/DB.java

https://github.com/g00291875/FYP-GUI/blob/master/FXGUI/src/com/*model*/*Model*.java

**util**

https://github.com/g00291875/FYP-GUI/blob/master/FXGUI/src/com/util/ComBean.java

https://github.com/g00291875/FYP-GUI/blob/master/FXGUI/src/com/util/HandleFileIO.java

https://github.com/g00291875/FYP-GUI/blob/master/FXGUI/src/com/util/*QueueSong*.java

https://github.com/g00291875/FYP-GUI/blob/master/FXGUI/src/com/util/*SelectionSong*.java

**fyp gui commits and graphs**

https://github.com/g00291875/FYP-GUI/commits/master

https://github.com/g00291875/FYP-GUI/graphs/contributors

**bluetooth chat app**

https://github.com/marcuspimenta/Chat-Bluetooth-Android

**jfx scene switching**

https://github.com/acaicedo/JFX-MultiScreen

**Toggl**

https://www.toggl.com/app/reports/summary/858727

**Bluetooth**

https://developer.bluetooth.org/TechnologyOverview/Pages/SPP.aspx

# 11 Appendices

## 11.1 Appendix A: JavaFX Overview

The information in this appendix is referenced from **https://docs.oracle.com/javase/8/javafx/api/javafx/**

#### Application class

The entry point for JavaFX applications is the Application class. The JavaFX runtime does the following, in order, whenever an application is launched:

Constructs an instance of the specified Application class

Calls the [init()](https://docs.oracle.com/javase/8/javafx/api/javafx/application/Application.html#init--) method

Calls the [start(javafx.stage.Stage)](https://docs.oracle.com/javase/8/javafx/api/javafx/application/Application.html#start-javafx.stage.Stage-) method

**https://docs.oracle.com/javase/8/javafx/api/javafx/application/Application.html**

#### Stage class

The JavaFX Stage class is the top level JavaFX container. The primary Stage is constructed by the platform. Additional Stage objects may be constructed by the application.

Stage objects must be constructed and modified on the JavaFX Application Thread.

**https://docs.oracle.com/javase/8/javafx/api/javafx/stage/Stage.html**

#### Scene class

The JavaFX Scene class is the container for all content in a scene graph. The background of the scene is filled as specified by the fill property.

The application must specify the root Node for the scene graph by setting the root property.

**https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Scene.html**

#### Node class

Base class for scene graph nodes. A scene graph is a set of tree data structures where every item has zero or one parent, and each item is either a "leaf" with zero sub-items or a "branch" with zero or more sub-items.

Each item in the scene graph is called a Node. Branch nodes are of type [Parent](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Parent.html), whose concrete subclasses are [Group](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Group.html), [Region](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/Region.html), and [Control](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/Control.html), or subclasses thereof.

Leaf nodes are classes such as [Rectangle](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/shape/Rectangle.html), [Text](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/text/Text.html), [ImageView](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/image/ImageView.html), [MediaView](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/media/MediaView.html), or other such leaf classes which cannot have children. Only a single node within each scene graph tree will have no parent, which is referred to as the "root" node.

**https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Node.html**

#### FXML

The *fx:controller* attribute allows a caller to associate a "controller" class with an FXML document. A controller is a compiled class that implements the "code behind" the object hierarchy defined by the document.

Controllers are used to implement event handlers for user interface elements defined in markup:

<VBox fx:controller="com.foo.MyController"

xmlns:fx="http://javafx.com/fxml">

<children>

<Button text="Click Me!" onAction="#handleButtonAction"/>

</children>

</VBox>

package com.foo;

public class MyController {

public void handleButtonAction(ActionEvent event) {

System.out.println("You clicked me!");

}

}

The controller can define an initialize() method, which will be called once on an implementing controller when the contents of its associated document have been completely loaded:

public void initialize();

This allows the implementing class to perform any necessary post-processing on the content. It also provides the controller with access to the resources that were used to load the document and the location that was used to resolve relative paths within the document

**http://docs.oracle.com/javase/8/javafx/api/javafx/fxml/doc-files/introduction\_to\_fxml.html**

#### *Model* View Controller

JavaFX incorporates the MVC design pattern. *Model*–view–controller (MVC) is a software [architectural pattern](https://en.wikipedia.org/wiki/Architectural_pattern) for implementing [user interfaces](https://en.wikipedia.org/wiki/User_interface) on computers. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.

**http://www.codeproject.com/Articles/25057/Simple-Example-of-MVC-*Model*-View-Controller-Design**