|  |
| --- |
| C:\year 5\Git\Report\FYP-Report\gmit-logo-2012rgb.jpg |
| Music Host Interface |
| B.Eng (Hons) in Computer and Electronic Engineering  GMIT |
|  |
| **Thomas Flynn** |
| **May, 2016** |

|  |
| --- |
|  |

# 

# 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

First of all, I would like to thank my classmates. Throughout the past few years they have assisted and encouraged my course work and provided some invaluable feedback on this project. Developing a project with a completely personal scope can result in some disorientation and uncertainty of progress and evaluation. I could not have completed this without their help and I wish them the best for the future.

I would also like to thank the course lecturers, staff and project supervisors for their guidance and assistance during the course of this project and over the past few years. They have always encouraged creativity in the course and I feel this has motivated me to undertake a project at this scope.

A special thanks to my project supervisor Brian O'Shea for providing me with valuable insight and support throughout the development of this project.

And a final thanks to my close friends and supportive family for all their help, both related and unrelated, to the development of this project and over the duration of my college years.

# Summary

The goal of 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

Contents

[Declaration 3](#_Toc449742767)

[Acknowledgements 4](#_Toc449742768)

[Summary 5](#_Toc449742769)

[Contents 6](#_Toc449742770)

[1. Introduction 10](#_Toc449742771)

[1.1 Project goals 10](#_Toc449742772)

[1.2 Project motivation 10](#_Toc449742773)

[1.3 About Music Hosting 10](#_Toc449742774)

[1.4 Report Overview 10](#_Toc449742775)

[2. Project Plan 11](#_Toc449742776)

[2.1 Gantt chart 11](#_Toc449742777)

[2.2 Trello and Time Management 11](#_Toc449742778)

[3 Block Diagram 12](#_Toc449742779)

[4 Flow Charts 13](#_Toc449742780)

[4.1 JavaFX Music Host 13](#_Toc449742781)

[4.2 Android Client Connecting to Music Host 14](#_Toc449742782)

[4.3 Music Host - Android Client Communication Flow Charts 15](#_Toc449742783)

[4.3.1 Song Request Flow Chart 15](#_Toc449742784)

[5 JavaFX Music Host Graphical User Interface 17](#_Toc449742785)

[5.1 Research and Investigation 18](#_Toc449742786)

[5.1.1 Graphical User Interface API 18](#_Toc449742787)

[5.1.2 Cloud Database 19](#_Toc449742788)

[5.1.2 Audio Format 19](#_Toc449742789)

[5.1.3 Bluetooth 19](#_Toc449742790)

[5.2 Requirements 19](#_Toc449742791)

[5.2.1 Graphical User Interface 20](#_Toc449742792)

[5.2.2 Sound Engine 20](#_Toc449742793)

[5.2.3 Database as a service 20](#_Toc449742794)

[5.2.4 Bluetooth Server 20](#_Toc449742795)

[5.2.4 JSON library 21](#_Toc449742796)

[5.3 Tool Choices 22](#_Toc449742797)

[5.3.1 Microsoft Azure SQL Server 22](#_Toc449742798)

[5.3.2 GUI 23](#_Toc449742799)

[JavaFX 2.0 and later is implemented as a native Java library, and applications using JavaFX are written in native Java code. JavaFX 1.1 was based on the concept of a "common profile" that is intended to span across all devices supported by JavaFX. This approach makes it possible for developers to use a common programming model while building an application targeted for both desktop and mobile devices and to share much of the code, graphics assets and content between desktop and mobile versions. 23](#_Toc449742800)

[5.3.3 Bluetooth Server 24](#_Toc449742801)

[5.4 Foundation 25](#_Toc449742802)

[5.5 Realisation 26](#_Toc449742803)

[5.5.1 MusicHostFramework 26](#_Toc449742804)

[5.5.2 ScreensController 27](#_Toc449742805)

[5.5.3 LoginScreenController 30](#_Toc449742806)

[5.5.4 MainSceneController 33](#_Toc449742807)

[5.6.5 Model 46](#_Toc449742808)

[5.6.5.1 Model Properties 46](#_Toc449742809)

[5.6.5.1 Model Methods 46](#_Toc449742810)

[5.6.6.1 Azure SQL Server Database 48](#_Toc449742811)

[5.6.6.1.1 FYP.dbo.UserLogin Table 49](#_Toc449742812)

[5.6.6.1.1.1 FYP.dbo.UserLogin Table Fields 49](#_Toc449742813)

[5.6.6.1.1.2 FYP.dbo.UserLogin Populated Table 49](#_Toc449742814)

[5.6.6.1.2 FYP.dbo.UserSongs Table 49](#_Toc449742815)

[5.6.6.1.2.1 FYP.dbo.UserSongs Table Fields 49](#_Toc449742816)

[5.6.6.1.1.2 FYP.dbo.UserLogin Populated Table 50](#_Toc449742817)

[5.6.6.2 DB 51](#_Toc449742818)

[5.6.6.2.1 DB Constructor 51](#_Toc449742819)

[5.6.6.2.2 DB Fields 52](#_Toc449742820)

[5.6.6.2.3 DB Methods 53](#_Toc449742821)

[5.6.7 HandleFileIO 54](#_Toc449742822)

[5.6.7.1 HandleFileIO Call Fields 54](#_Toc449742823)

[5.6.7.1 HandleFileIO Call method 54](#_Toc449742824)

[5.6.8 ProcessConnectionThread 56](#_Toc449742825)

[5.7 Operation 57](#_Toc449742826)

[5.7.1 Setup Sequence Diagram 57](#_Toc449742827)

[5.7.2 Login Sequence Diagram 59](#_Toc449742828)

[5.7.3 Initialize Button Sequence Diagram 61](#_Toc449742829)

[5.7.4 Add Button Sequence Diagram 62](#_Toc449742830)

[5.7.5 Add Song Animation Ended Event Sequence Diagram 63](#_Toc449742831)

[5.7.6 Song Added Event Sequence Diagram 64](#_Toc449742832)

[5.7.6 Song Removed/Ended Event Sequence Diagram 66](#_Toc449742833)

[5.7.6 Logout Sequence Diagram 68](#_Toc449742834)

[5.7.6 Server Button Sequence Diagram 70](#_Toc449742835)

[5.7.6 ProcessConnectionThread Sequence Diagram 72](#_Toc449742836)

[5.7.6 WhatToDoFunc Options, Song Request, Song Selected Sequence Diagram 74](#_Toc449742837)

[5.7.6 WhatToDoFunc DJ Comment Sequence Diagram 75](#_Toc449742838)

[5.7.6 WhatToDoFunc Skip Song Sequence Diagram 77](#_Toc449742839)

[5.8 Operation 78](#_Toc449742840)

[5.8.1 Login View 78](#_Toc449742841)

[5.8.2 Main View 79](#_Toc449742842)

[5.8.3 Use Case - Setup 80](#_Toc449742843)

[5.8.3 Use Case - Song Added 81](#_Toc449742844)

[5.8.3 Use Case - Song Skipped / Song Ended 82](#_Toc449742845)

[5.8.3 Use Case - Play / Pause 83](#_Toc449742846)

[6 Android Music Host Client 84](#_Toc449742847)

[6.1 Research and Investigation 84](#_Toc449742848)

[6.1.1 Graphical User Interface API 84](#_Toc449742849)

[6.1.2 Cloud Services 84](#_Toc449742850)

[6.1.3 Bluetooth 84](#_Toc449742851)

[6.2 Requirements 84](#_Toc449742852)

[6.2.1 Graphical User Interface 85](#_Toc449742853)

[6.2.2 Sound Engine 85](#_Toc449742854)

[6.2.3 Database as a service 85](#_Toc449742855)

[6.2.4 Bluetooth Server 85](#_Toc449742856)

[6.4 Foundation 85](#_Toc449742857)

[6.4.1 Foundational Architecture 85](#_Toc449742858)

[6.5 Realisation 87](#_Toc449742859)

[6.5.1 Music Host Client Architecture 87](#_Toc449742860)

[6.5.1 Music Host Client Architecture 88](#_Toc449742861)

[6.5.1 Music Host Client Architecture 88](#_Toc449742862)

[6.5.1 Music Host Client Architecture 89](#_Toc449742863)

[6.5.1 Music Host Client Architecture 90](#_Toc449742864)

[6.5.1 Music Host Client Architecture 91](#_Toc449742865)

[6.5.1 Music Host Client Architecture 92](#_Toc449742866)

[6.5.1 Music Host Client Architecture 92](#_Toc449742867)

[6.5.1 Music Host Client Architecture 94](#_Toc449742868)

[6.5.1 Music Host Client Architecture 96](#_Toc449742869)

[6.5.1 Music Host Client Architecture 98](#_Toc449742870)

[6.5.1 Music Host Client Architecture 99](#_Toc449742871)

[6.5.1 Music Host Client Architecture 100](#_Toc449742872)

[6.5.1 Music Host Client Architecture 101](#_Toc449742873)

[6.7.1 Use Case - Open App 102](#_Toc449742874)

[6.7.2 Use Case - Search For Music Host (Bluetooth) 103](#_Toc449742875)

[6.7.3 Use Case - Connected 104](#_Toc449742876)

[6.7.4 Use Case - Song Request 105](#_Toc449742877)

[6.7.5 Use Case - Song Accepted / Not Accepted 106](#_Toc449742878)

[6.7.6 Use Case - DJ Comment 107](#_Toc449742879)

[6.7.7 Use Case - Skip Song 108](#_Toc449742880)

[6.7.4 Use Case - Search For Music Host (GPS) 109](#_Toc449742881)

# 1. Introduction

## 1.1 Project goals

The first goal for this project was to provide the Music Host with an intuitive and responsive Graphical User interface for playing the role of the Music Host. In this role the user should be able to perform all the basic operations that a standard media player Graphical User Interface provides.

The second goal was to integrate a cloud database into the project. This provided an extra level of functionality where by the Music Host was not bound to local song files stored on their computer. This feature allows them to play the role of the Music Host anywhere in the world that implements a Music Host Interface.

The third goal was to develop a simple but effective Android Application for the Music Host Client. After using the application the user should feel the urge to tell other people about the application and it's functionality.

Jukeboxes have faded from popularity over the last decade. They not only provided us with a democratic way of selecting music but also limited the scope of the selection to ....words

Disk Jockeys are more commonplace these days however. They have replaced the Jukebox by allowing their audience to take song requests. This is the baseline for this project The Music Host Interface. This interface provides an application for both the DJ and the audience.

## 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 being 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 able 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. At times this was the case but overcoming these obstacles has not only made a stronger programmer but a stronger person as well.

## 1.3 About Music Hosting

The following is a summary of the various aspects and problems associated with being a Music Host.

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. This can be problematic for a few reasons. The first is that the DJ might not want to play the requested song due to their personal taste in music. The second is that the DJ's performance is impaired when they are being constantly distracted by the audience members song requests.

A DJ is not the only one who plays the role of a Music Host however. A venue that has a selection of music that plays over a duration of time 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 choosing a song to play from this selection. Sadly Jukeboxes are not as popular as they used be.

## 1.4 Report Overview

# 2. Project Plan

I used the following tools to construct a cohesive project plan that I could stick to through the project's lifecycle.

## 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.



## 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.

#### Trello labels



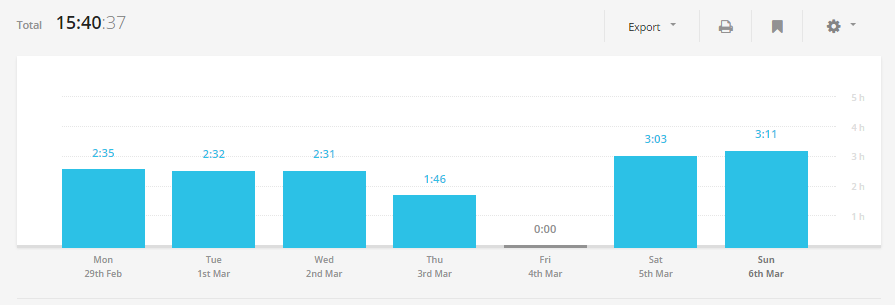
#### Trello Project Management Board



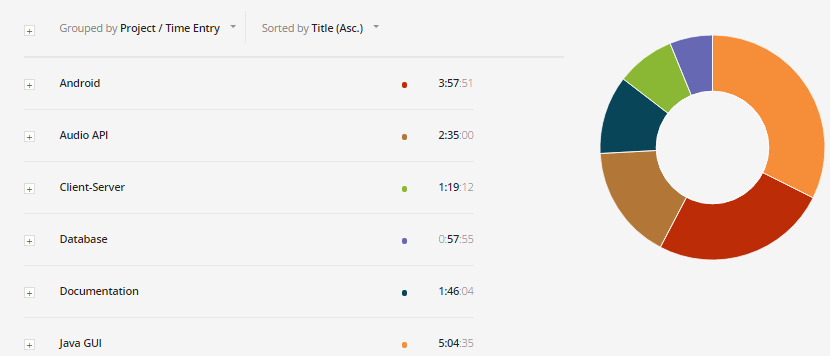
## C:\year 5\Git\Report\FYP-Report\Report\report lib\new\toggl.png2.3 Toggl time management

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.

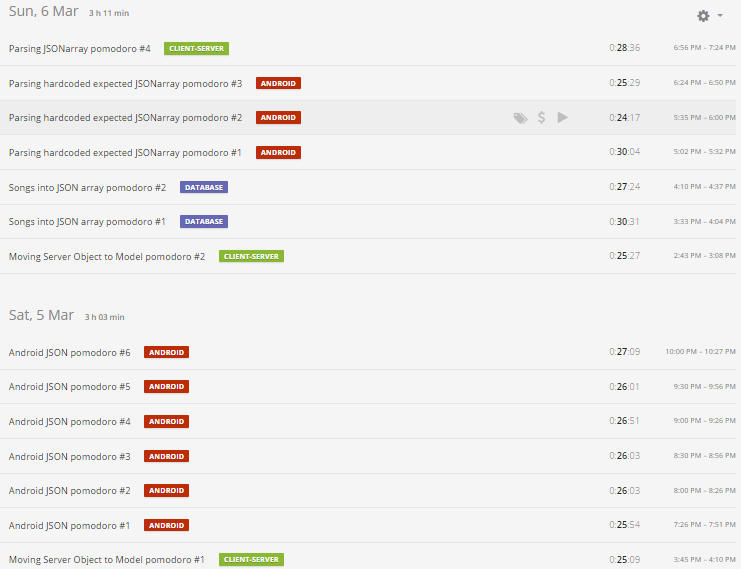
#### Sample Weekly time bar chart



#### Sample weekly time donut chart



#### Sample weekly time log



# 3 Block Diagram



# 4 Flow Charts

The following flow charts will provide you with an overview of the project.

## 4.1 JavaFX Music Host

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

## 4.2 Android Client Connecting to Music Host



## 4.3 Music Host - Android Client Communication Flow Charts

## 4.3.1 Song Request Flow Chart







# 5 JavaFX Music Host Graphical User Interface

The workflow for this project was relatively straightforward. Research and Investigation was handled through prototype development. These prototypes then segmented the project into core functionality features and ultimately decided the most efficient method to implement these features into an application. At this stage, the basic requirements for operation were listed so a rational decision could be made for which tools to use to implement the main features. Once the tools had been selected, features were gradually implemented with core functionality features being a priority. As the project deadline approached, the final feature set was completed so the project could reach a "finished" stage.

My mentality for building this project was to avoid any guidelines or set paths, so I could be free to include any functionality I deemed relative to the project at any stage without diverging from a set plan or timescale. This made the entire process much more adaptive and creative allowing for a much more instinctive workflow.

## 5.1 Research and Investigation

The solution for this part of the project was designed to be a software application with a graphical user interface for the music host. From a basic perspective, this solution requires knowledge of a library for a graphical user interface, cloud services for a database as a service and Bluetooth library for a server and how all these elements can be integrated together to provide a working solution.

### 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).

ref 1- https://en.wikipedia.org/wiki/Graphical\_user\_interface

### 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 models: 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.

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

### 5.1.2 Audio Format

MPEG-1 or MPEG-2 Audio Layer III,[[4]](https://en.wikipedia.org/wiki/MP3#cite_note-rfc5219-4) 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).

The use of lossy compression is designed to greatly reduce the amount of data required to represent the audio recording and still sound like a faithful reproduction of the original uncompressed audio for most listeners. An MP3 file that is created using the setting of 128 [kbit/s](https://en.wikipedia.org/wiki/Kbit/s) will result in a file that is about 1/11 the size of a [CD](https://en.wikipedia.org/wiki/Red_Book_(audio_CD_standard))-quality file.

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

### 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[[4]](https://en.wikipedia.org/wiki/Bluetooth#cite_note-4)) 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,[[5]](https://en.wikipedia.org/wiki/Bluetooth#cite_note-5) 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.

ref - https://en.wikipedia.org/wiki/Bluetooth

## 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

A Graphic User Interface (GUI) should provide the user with feedback regarding their input and the current state of the application. The 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 information and 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

This is where all the sound generation and manipulation should be handled in the application. 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 database will have to be capable of being accessed from anywhere around the world. This will provide Music Host's with the ability play the role of Music Host where ever this software interface is available.

### 5.2.4 Bluetooth Server

The Bluetooth library should provide the necessary API to run a server within the Music Host Application in order to accept connections from Music Host Client's.

### 5.2.4 JSON library

The Music Host's song selection will need to be parsed into a JSON array for sending to the Music Host Client over the Bluetooth network.

## 5.3 Tool Choices

The requirements listed in the previous section are primarily software based problems. The solutions are in the form of software APIs for functionality in the core elements and a thought out solution to link these interfaces together to form the complete application.

### 5.3.1 Microsoft Azure SQL Server

The requirements for the remote database specify that the database be accessible 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

**Other 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.

ref - https://en.wikipedia.org/wiki/SQL\_Azure

### 5.3.2 GUI

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
* Powerful Sound Engine

Figure : JavaFX Logo

**Other 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), as well as [rich internet applications (RIAs)](https://en.wikipedia.org/wiki/Rich_Internet_application) 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).

### JavaFX 2.0 and later is implemented as a native Java library, and applications using JavaFX are written in native Java code. JavaFX 1.1 was based on the concept of a "common profile" that is intended to span across all devices supported by JavaFX. This approach makes it possible for developers to use a common programming model while building an application targeted for both desktop and mobile devices and to share much of the code, graphics assets and content between desktop and mobile versions.

ref - https://en.wikipedia.org/wiki/JavaFX

### 5.3.3 Bluetooth Server

The requirements specified for the Bluetooth Server state that the Bluetooth server is capable of running within the Music Host Application.

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

**Reasons:**

* C:\year 5\Project presentation\pics\bluecove.pngExtensive online documentation
* Open Source library
* Bluecove Stack implements Service Discovery Protocol
* Interfaces with Microsoft Bluetooth stack

Figure : BlueCove Logo

**Other 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.

### 5.4 Foundation

The foundation for the GUI was built on a JFX-MultiScreen example provided by oracle on Github.

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

Taking advantage of the functionality of the MultiScreen example. where the addition and removal Pane Nodes to the scene graph had already acheived. This provided me w

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

## 5.5 Realisation

With all of the tools selected and basic functionality already achieved, the realisation process begun. The application went through many iterations before reaching the final stage. Features were implemented gradually in different versions starting with core functionality such as the playing and queuing of mp3 files in a local directory.

### 5.5.1 MusicHostFramework

*MusicHostFramework* extends Application, allowing it to run as a JavaFX application. *ScreenID* is used to associate the fxml file specified by *ScreenFile* String in a hashmap stored by the *ScreensController.*

#### start

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 within 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 tbf Shows the relationship that was previously described between the classes.



Figure: MusicHostFramework UML

### 5.5.2 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.

#### loadScreen

This is method is called by the MusicHostFramework within it's 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.

#### setScreen

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

Figure tbf below shows the relationship that was previously described between all the classes in the application. tbf



Figure: ScreensController UML

### 5.5.3 LoginScreenController



Figure: LoginSceneController Class

The LoginScreenController shares the screenParent property with the MainSceneController. This provides the necessary functionality for access to shared resources held by the model property.

#### initialize

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 *loginView.fxml* file have been completely loaded. This allows the LoginSceneController to perform any necessary post-processing on the content.

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.

#### loginAction

This method calls the login method which 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

### 5.5.4 MainSceneController

The MainSceneController is where the primary functionality of this application resides. It's role is to act as the business logic for the MainView.fxml file. It consists of a Media Engine, Business logic for SQL queries and a Bluetooth SDP server.

In order to break the fields down and how they relate to the use cases defined below. I will describe the essential fields first and then the methods that those fields relate to.

The controllers main functionality is broken down into the following use cases.

**Use cases:**

1. Init button clicked

2. Add song button clicked

3. Skip button clicked

4. Play/Pause button clicked

5. Logout button clicked

6. Music Host Option buttons Clicked

7. Server button clicked

#### 5.5.4.1 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**

**Button boolSkip**

Use case 6. Music Host Option buttons Clicked.

**@FXML**

**Button serverButton**

7. Server button clicked

**@FXML**

***ListView queueList***

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

**@FXML**

***ListView selectionList***

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

**@FXML**

***ListView dJComments***

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

***FXCollections.observableList SongQueueObservableList***

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

**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 user.

**MediaView mediaView**

This acts a container for the *currentPlayer*. The *changeListener* associated provides the necessary functionality for stopping and removing the *oldPlayer* and then subsequently playing the *newPlayer*.

**AtomicInteger queueSizeAtomic**

This variable is either decremented or incremented by the queueListener.

**ListChangeListener queueListener**

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

**byte[] nextPlayerBytes**

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

**ExecturoService 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.

**Path addSongPath**

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

#### 5.5.4.2 MainSceneController Methods

****

Figure: MainSceneController Methods

#### 5.5.4.2.0 Initialize

**initialize**

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.

**setGUIOptions**

Initializes all the elements of the boolOptionsArray to false.

Sets all the various Nodes to default values. 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.

**setCellFactoryForListViews**

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

**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.

**addVolumeandTimeSlider Listeners**

Binds the timeSlider and the volumeSlider to the mediaView.mediaPlayerProperty for UI control.

**buildSongPlayAnimation**

Creates an ellipsePath for the playPath field.

#### 5.5.4.2.1 Init button

**@FXML**

**public 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 initiSongSelection method.

**public void addFXObservableListeners**

Adds Change listeners for the SongQueueObervableList, ObservableQueueList and the observableDJComments.

These listeners are listening for changes in the model object.

**public 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.

#### 5.5.4.2.2 Add Song

**@FXML**

**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.

**public synchronized boolean addSongTask (int index)**

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

2- Creates a QueueSong object.

3- Sets the progressBall to DEEPSKYBLUE

4- Passes the QueueSong to SongAnimationSetup method.

5- Plays the addSongTransition animation.

**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 finished. The event handler will add the QueueSong to the SongQueueObservableList.

**public synchronized boolean searchQueueForMatch(String)**

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

**public void writeToCurrentPlayer(MediaPlayer)**

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

**private 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.

**private 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.

**private void addAmITheLastSong(MediaPlayer)**

Called under special circumstances that creates an anonymous endOfMedia Listener that will simply remove the MediaPlayer from the song queue because there are no songs to follow.

#### 5.5.4.2.3 Skip Button / Song Ended

**@FXML**

**public void iSkip(ActionEvent event)**

Event fired when the user hits the skip button.

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

**public 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 queueSizeAtomic 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.

#### 5.5.4.2.4 Skip Play/Pause button clicked

**@FXML**

**public 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.

#### 5.5.4.2.5 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.

#### 5.5.4.2.6 Music Host options buttons clicked

**@FXML**

**private void setSkipSongBool**

Toggles an element the boolOptionsArray.

**@FXML**

**private void setSongRequestBool**

Toggles an element the boolOptionsArray.

**@FXML**

**private void setDJCommentsBool()**

Toggles an element the boolOptionsArray.

#### 5.5.4.2.6 Server button clicked

**@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.

**public void startServer**

Creates a newSingleThreadExecutor for executorService1 which will be used for waiting on Android Clients to connect.

**public void stopServer**

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

****

### 5.6.5 Model

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.

### 5.6.5.1 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.

### 5.6.5.1 Model 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.

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

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

**public String songSelectionToJson**

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

**public String songQueueToJson**

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

**public String songQueueToJson**

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

**public void clearValuesBeforeLoggingOut**

Clears all the properties in the model object.

### 5.6.6.1 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.

### 5.6.6.1.1 FYP.dbo.UserLogin Table

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

### 5.6.6.1.1.1 FYP.dbo.UserLogin Table Fields

**userName (VARCHAR 255):**

Stores the name of the Music Host.

**passsword (VARCHAR 255):**

Stores the password of the Music Host.

### 5.6.6.1.1.2 FYP.dbo.UserLogin Populated Table

The following figure shows the values 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

### 5.6.6.1.2 FYP.dbo.UserSongs Table

The UserSongs Table is used to store the songs of each individual Music Host.

### 5.6.6.1.2.1 FYP.dbo.UserSongs 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

### 5.6.6.1.1.2 FYP.dbo.UserLogin 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

### 5.6.6.2 DB

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

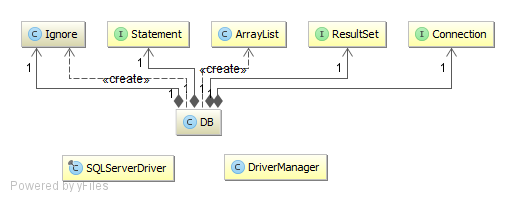


Figure : DB UML

### 5.6.6.2.1 DB Constructor

Creates a connection to the database.

### 5.6.6.2.2 DB 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 with the database. The SQL statements are executed and results are returned within the context of this connection.

### 5.6.6.2.3 DB 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.

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.

### 5.6.7 HandleFileIO

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

Figure : HandleFileIO class

### 5.6.7.1 HandleFileIO Call 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.

### 5.6.7.1 HandleFileIO Call method

**public MediaPlayer call() throws Exception**

Called 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.

### 5.6.8 ProcessConnectionThread

A ProcessConnectionThread that implements runnable is 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

## 5.7 Operation

This section describes how the application works at run time and how the various functional elements interact with each other.

### 5.7.1 Setup Sequence Diagram



Figure : Setup 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

### 5.7.2 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

### 5.7.3 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

### 5.7.4 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

### 5.7.5 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

### 5.7.6 Song Added Event Sequence Diagram



Figure : Song Added Event 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

Alt: [QueueSize == 1]

1.1.4.2.2 Assign the newly created MediaPlayer to the *currentPlayer* field.

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 *nextPlayer* field.

1.1.4.2.6 Add an end of media listener to the *currentPlayer* that will play the *nextPlayer* when it ends*.*

Alt: [default]

1.1.4.2.7 Return

### 5.7.6 Song 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

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

### 5.7.6 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)

### 5.7.6 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

1- Get the local bluetooth device.

2- Create a String url specifying Service Discovery Protocol.

3- Create a StreamConnectionNotifer with this protocol.

4- Start Accepting connections.

5- Enter a while loop

6- Create a newSingleThreadExecutor for the clientExecutor.

7- Pass the socket that the Android Client connected to, to the constructor of the ProcessConnectionThread.

8- clientExecutor is used to execute ProcessConnectionThread's runnable method that will handle the communication protocol for the Android Client.

7- The clientExecutor is tasked to shutdown in 20 seconds. This gives the Android Client only 20 seconds to make a decision.

### 5.7.6 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

### 5.7.6 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)**

### 5.7.6 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**

### 5.7.6 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**

## 

## 5.8 Operation

## 5.8.1 Login View

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

## 5.8.2 Main View

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

## 5.8.3 Use Case - Setup



## 5.8.3 Use Case - Song Added



## 5.8.3 Use Case - Song Skipped / Song Ended

****

## 5.8.3 Use Case - Play / Pause

****

# 6 Android Music Host Client

The workflow for this project was relatively straightforward. Research and Investigation was handled through prototype development. These prototypes then segmented the project into core functionality features and ultimately decided the most efficient metho tbf

## 6.1 Research and Investigation

tbf

### 6.1.1 Graphical User Interface API

tbf

### 6.1.2 Cloud Services

Cloud services are tbf.

AWT and Google Cloud are tbf..

### 6.1.3 Bluetooth

Bluetooth has been around since tbf. It operates using frequency hopping spread dspectrum technique at 2.14 GHZ. tbf

## 6.2 Requirements

From the features defined, a number of requirements and dependencies were raised for the project to meet functionality requirements. These requirements were then segmented into 3 core feature elements. tbf

### 6.2.1 Graphical User Interface

Microsoft JDBC 4.0 is the minimum driver version required in order to make a connection to a Microsoft Azure SQL Server. tbf

### 6.2.2 Sound Engine

The sound engine should be capable of playing mp3 files. tbf

### 6.2.3 Database as a service

The remote database...tbf

### 6.2.4 Bluetooth Server

The Bluetooth library should provide the necessary API to run a server.

### 6.4 Foundation

The foundation for the GUI was built on tbf@github example. Taking advantage of the functionality already implemented in the scene switching feature. The example has the ability to add and remove Pane Nodes to the scene graph.

This provided the necessary functionality for the *LoginView.fxml* and *MainView.fxml* to be able to fade in and out. Business logic for logging in and out was built on this transition feature.

### 6.4.1 Foundational Architecture

Figure 1 shows tbf

Figure : Foundational Architecture Design

## 6.5 Realisation

With all of the tools selected and a basic architectural layout designed, the realisation process begun. The application went through much iteration before reaching the final stage. Features were implemented gradually in different versions[[1]](#endnote-2) starting with core functionality such as motion tracking and audio file playback.

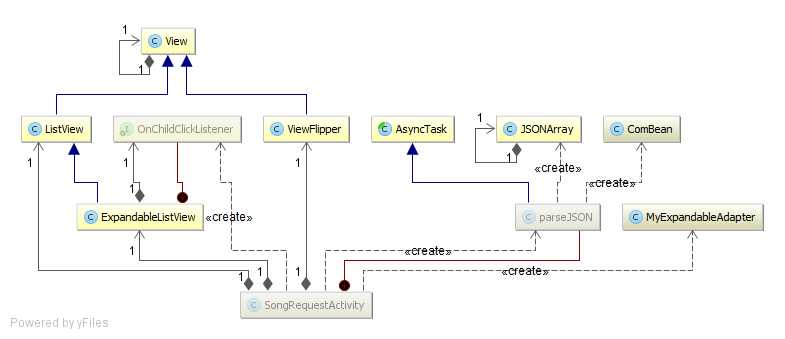
### 6.5.1 Music Host Client Architecture



### 6.5.1 Music Host Client Architecture



### 6.5.1 Music Host Client Architecture



### 6.5.1 Music Host Client Architecture



### 6.5.1 Music Host Client Architecture



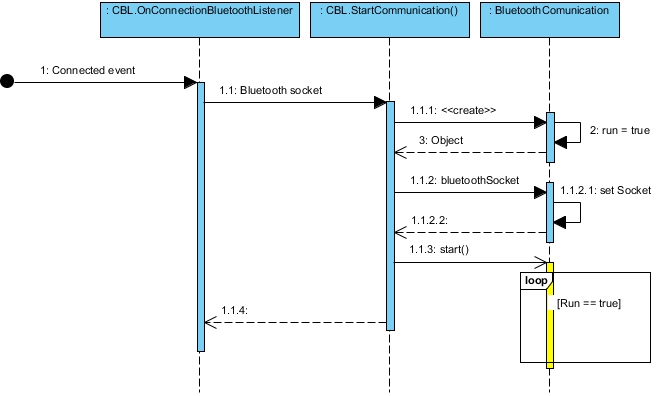
### 6.5.1 Music Host Client Architecture



### 6.5.1 Music Host Client Architecture



### 6.5.1 Music Host Client Architecture



### 6.5.1 Music Host Client Architecture



### 6.5.1 Music Host Client Architecture



### 6.5.1 Music Host Client Architecture



### 6.5.1 Music Host Client Architecture



### 6.5.1 Music Host Client Architecture



### 6.5.1 Music Host Client Architecture



## 6.7.1 Use Case - Open App



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



## 6.7.3 Use Case - Connected



## 6.7.4 Use Case - Song Request



## 6.7.5 Use Case - Song Accepted / Not Accepted



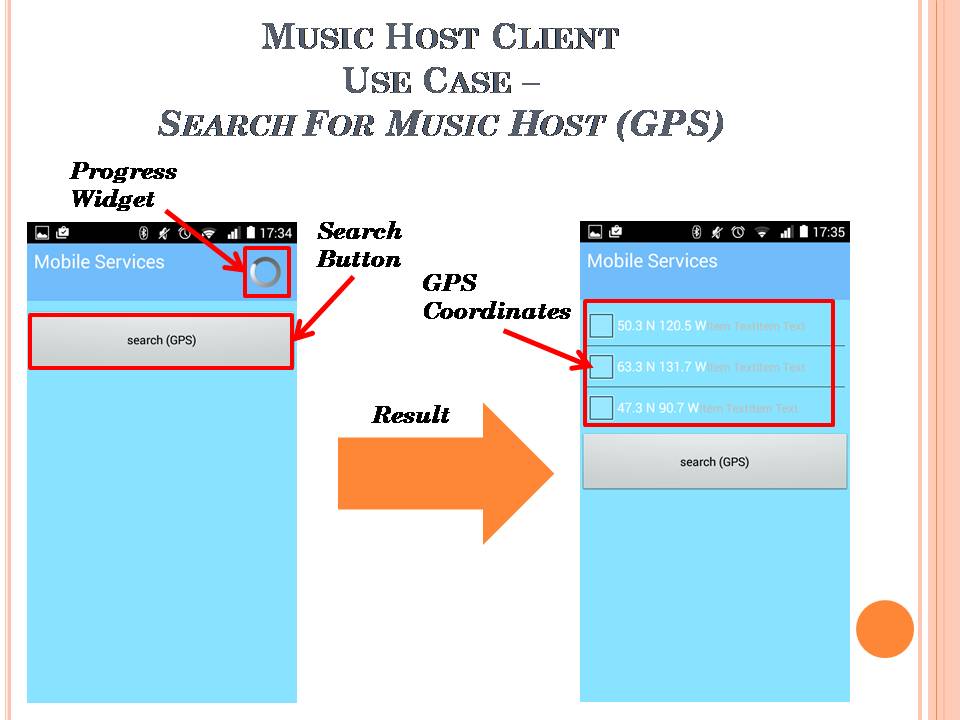
## 6.7.6 Use Case - DJ Comment



## 6.7.7 Use Case - Skip Song



## 6.7.4 Use Case - Search For Music Host (GPS)



# Conclusion

From 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 accurate realisation of these skills. I have managed to incorporate my own personality and interests into the realisation of this project.

From building this project I have also gained experience with invention. The overall design was based on ideas I had into how I felt it should work. 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.

The feedback I have received from both DJs and lovers of music after testing the finished realisation in a real world environment 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 Android application as that's where the market is.

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

The goal of 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.

1. Github Version Commits: github.com/freshfunkee/KinectMusicController/commits/master [↑](#endnote-ref-2)