COMP390 2023/24

RetroSonic Music Player

Student Name: Kevin Rahimi

Student ID:201563237

Supervisor Name: Dr Blaine Keetch

DEPARTMENT OF COMPUTER SCIENCE

University of Liverpool Liverpool L69 3BX

Acknowledgements

I would like to express my gratitude to my parents for their unwavering support throughout my academic journey. Their encouragement has been a cornerstone of my perseverance and success.

Thanks to my supervisor for this project, Blaine who was a huge help in guiding me in how to use my time effectively by setting me checkpoints & helping me become better at writing.

I am grateful to my coaches and friends at Gym21 for providing me with methods to let off steam when I have needed.

Big thanks to my roommates Hugo and Luke, who came Liverpool with me from the same town in Manchester.

Thank you all for being part of this journey & supporting me to the end.

COMP390 2023/24

UNIVERSITY OF LIVERPOOL

RetroSonic Music Player

DEPARTMENT OF COMPUTER SCIENCE

University of Liverpool Liverpool L69 3BX

Abstract

The RetroSonic Music Player project aims to develop a standalone JavaFX application that combines music playback with direct metadata editing capabilities. This application addresses a gap in existing media players by integrating detailed management of MP3 metadata with standard music player functionalities, thereby catering to users who seek a more involved interaction with their digital music libraries. The application supports basic playback controls, playlist management, and metadata editing, for fields such as title, artist, album and year.

Unlike conventional music players that separate media consumption from media management, RetroSonic integrates these aspects into a single, intuitive interface, allowing users to not only upload & play music but also modify song metadata permanently without the need for external software. The project leverages JavaFX for its graphical user interface and the MP3agic library to handle metadata manipulation, providing a robust platform for both novice and experienced users.

The design of RetroSonic emphasises simplicity and user-friendliness, featuring a Simplified User Interface (SUI) that makes navigation and operation accessible to all users, regardless of their technical proficiency.

Throughout its development, the project underwent testing to ensure functionality and usability. The final product is a versatile tool that enhances users' interaction with their music collections, offering a personalised listening experience. The RetroSonic Music Player project sets the foundation for a new type of music player that empowers users to manage their music libraries in a more detailed and integrated manner.

Statement of Ethical Compliance

Data Categories: A

Participant Category: 3

I have complied with the ethical guidance standards of this module.

Table of Contents

Table of Figures (optional)

Introduction & Background-explain what vbox and hbox are

MP3 files, due to their compact size and decent quality, have become one of the most popular audio formats. However, efficiently managing and editing the metadata of these files can be challenging, especially for users looking to personalise their music libraries.

This project focuses on developing a JavaFX application designed to use, edit and manage the metadata of MP3 files all within one environment where songs can be played or added to playlists like Spotify. The difference between RetroSonic and Spotify is not only the metadata reading and writing functionality but also the Simplified User Interface (SUI) which is implemented to allow for easy navigation

within the application even for users who are less technologically adept. RetroSonic focuses on providing a streamlined, focused experience specifically for just playing music and managing and editing MP3 metadata so the song formats properly in your music library.

The aims of this music player were:

1. To be Cross Platform;
2. Implement a Simplified User Interface
3. An app where you upload music;
4. All music is displayed in the library;
5. Music with no tags provided default values;
6. Tags can be edited through the application;
7. Songs can be played, next & previous;
8. Most frequently played songs in the
9. The Music player can be used across every scene with only one instance being created;
10. Songs can be added and removed from playlists;

JavaFX is a software platform for creating and delivering desktop applications. It is intended to replace Swing as the standard GUI library for Java. JavaFX provides a powerful Java-based UI platform which allows for applications to be built for Cross-Platform uses across Linux, Windows and MacOs; consequently allowing RetroSonic to achieve the first aim. Another powerful component used in this project is the JavaFX MediaPlayer, which allows for the handling of audio files and playback controls such as play/pause, next/previous buttons. This helps us achieve aim 6. The extensive api allows for a lot of future extension with features like an equaliser for frequency adjustment.

FXML (FXML Graphical User Interface Markup Language) is an XML-based language created by Oracle for defining the user interface (UI) of a JavaFX application.Using FXML to design the UI of JavaFX applications allows developers to separate the presentation layer from the logic layer, improving code readability and maintainability. This separation follows the Model-View-Controller (MVC)

design pattern which was taught in COMP319-Software Engineering 2. These FXML files can be edited within Scenebuilder.

Scenebuilder is a powerful tool that greatly simplifies the development of JavaFX applications, particularly when aiming to create a Simplified User Interface (SUI). The design of an SUI focuses on ease of use, minimalism, and efficiency, ensuring that users can navigate and utilise the application without unnecessary complexity. Scene Builder provides a WYSIWYG (What You See Is What You Get) environment, enabling developers to design the UI visually. Scene Builder supports the application of CSS styles to JavaFX components which is useful for parts which are hard to design visually. With Scene Builder, prototypes of the user interface can be quickly developed, which was the case with RetroSonic, where the prototype of the project was designed for the proposal, then ended up being used as the base to start developing the front end.

MP3agic is a powerful, open-source Java library used to read and manipulate the metadata and MP3 file information. MP3agic allows the application to read various metadata from MP3 files, such as title, artist, album and year. This capability is essential for organising and displaying songs in your music player in a user-friendly manner. A key feature of RetroSonic is allowing users to edit the metadata of their MP3 files. MP3agic provides the functionality to modify tags, enabling your application to update information like song title, artist, and album directly within the user interface. Metadata extracted using MP3agic can be bound to JavaFX UI components, ensuring that any changes in the metadata are immediately reflected in the user interface

Requirements Analysis (optional)

A survey was taken to find what people require within a music player before the initial proposal of the project.

The people who took the survey were informed and agreed for their results to be used in this dissertation. Their answers from the survey were anonymous.

These are a few of the responses to the questions asked: **Q1; What are some features you would expect in a music player?** & **Q2; What are some extra features you would love to see in a music player app?**

Q1: What are some features you would exp...	Q2: What are some extra features you wou...
Playlists, Shuffling, Queue, Search, Liked songs	Ratings, Suggested songs/albums/artists, Genr
Making playlists, downloading playlists, being a	different types of shuffle in a playlist, and algo
The ability to make multiple playlists	Recommend songs I would possibly like based
Custom playlists, song info like artists genre ye	Can't think of anything Spotify doesn't already
Pause/ play button, skip / rewind button, creati	Be able to download songs as mp3 files to PC,
Ability to create playlists	Share music with other “friends” or other app u
Easy to navigate home page	Extra social interaction, able to share songs ea
Features that allow me to download music so th	Features that make it easy for me to find new s
Background play whilst using other apps	
Music to be ordered appropriately, layout make	Suggestions of music I might like based of the r
Playlists, Albums, Charts	Customisation

Some common requirements were:

- Play/Pause, Rewind, Shuffle and Loop buttons
- a function to upload music to a library
- an ability to edit attributes of songs(photo, name, artist or year)
- a function to create playlists and manipulate them (change order, add/remove songs)
- the ability to function offline
- the ability to function on Windows, MacOs and Linux
- Playlists show length of playlist and how many songs are within playlist.
- Recommend songs to users (offline music player - so cannot recommend songs that haven’t been uploaded however frequently played songs is

recommended to users.)

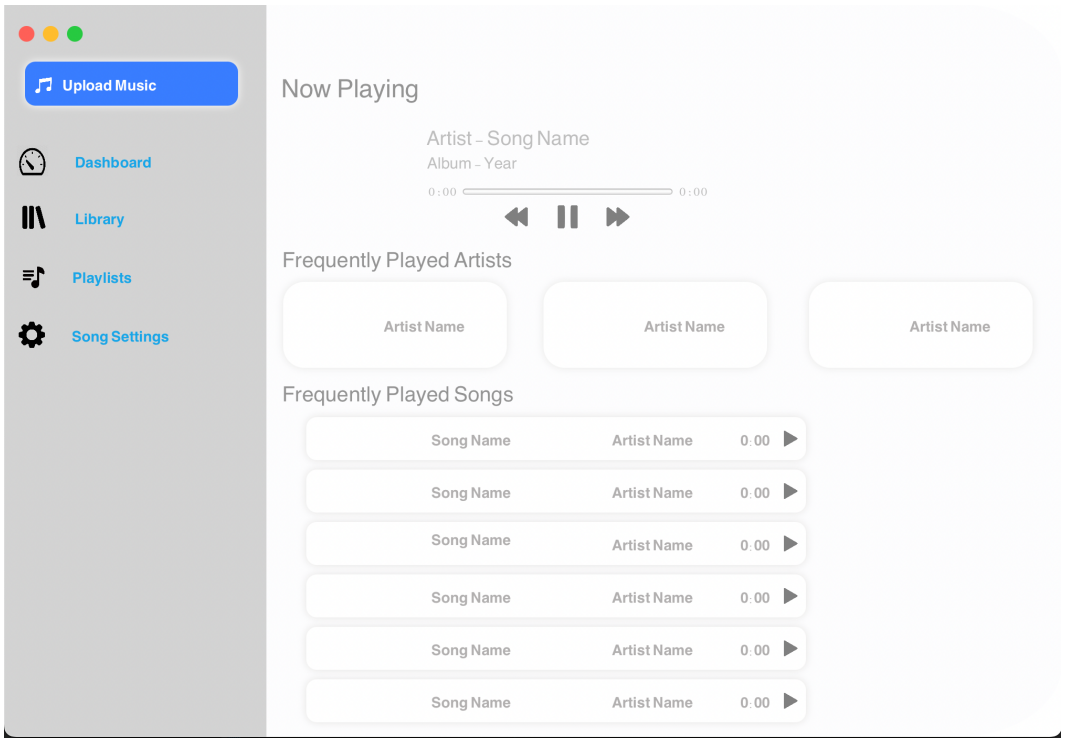
Design & Implementation ——Tell story

Front - End

The front end was the first thing I developed ...

Main.FXML

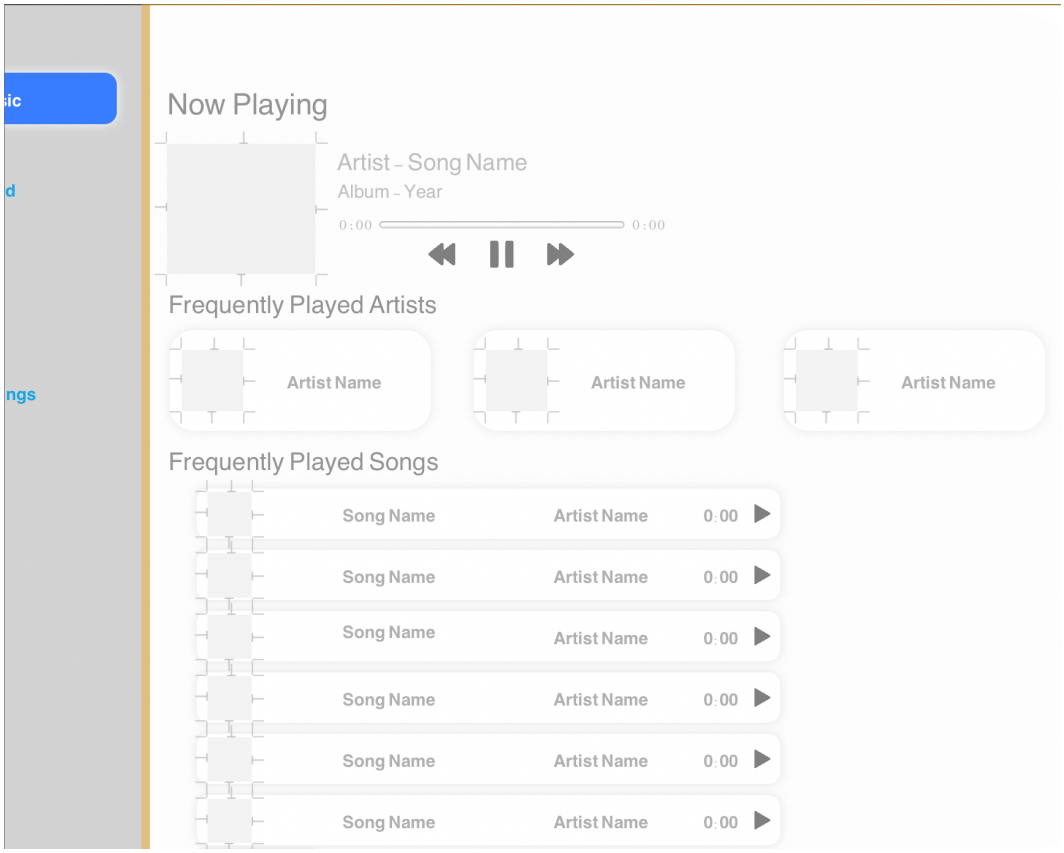
The initial prototype design for the music player dashboard built on SceneBuilder



The scene is built with an initial BorderPane, this splits the UI into five parts (Top,Bottom,Left,Right & Middle) however in this project it only use two (Left & Middle) in these two parts a pane is inserted so that components can be placed on top. The Left Pane contains our Sidebar. This is where we implement our buttons (Upload, Dashboard, Library, Playlists, Settings), in the figure, these are only labels. In the Middle Pane, for our dashboard design, we have implemented the a

few labels representing the song, play, next, previous images for media playback and HBoxs confined within a VBox to represent Frequently Played Songs/Artists (Artists feature was removed by the end of the process.) This VBox containing eight HBoxes can be populated with song values which will be shown later in this report.

There are blank spaces next to the songs and artist name, in actuality these are not blank and contain an “ImageView” container.



In the design stage of this prototype dashboard, more of the features became apparent and clearer:

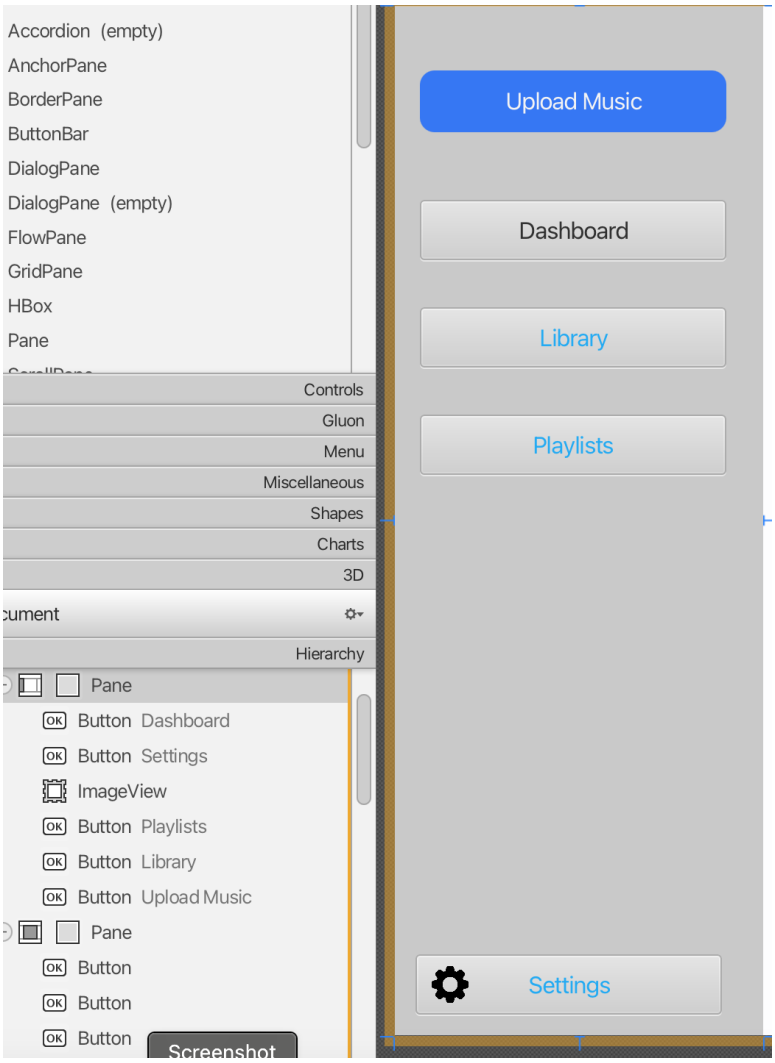
- Upload music button
- Default values for the Music (Image, Title, Artist, Album , Year)
- Music playback (Next/Previous, Pause/Play)
- Frequently Played Songs/Artists
- Buttons to switch between dashboard, library & playlists (Scenes)

- A means for the music player to keep a song instance while switching between scenes

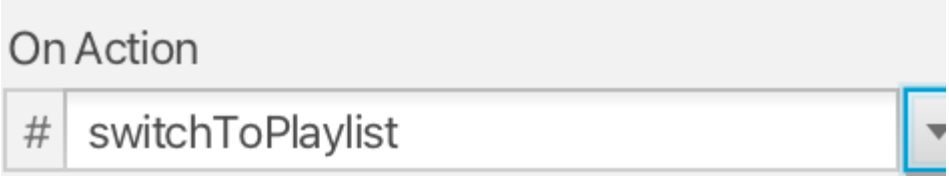
The main premise was to keep things simple to abide by a Simplified User Interface, while using a bright blue colour to make certain components stand out such as the upload music button. The aim was for the user to be able to navigate the app by following the blue subconsciously, as it guides the user through the app.

To get from this prototype to the finished front-end, there were a couple iterative changes that were made.

First of all, all the labels in the sidebar were turned into buttons:



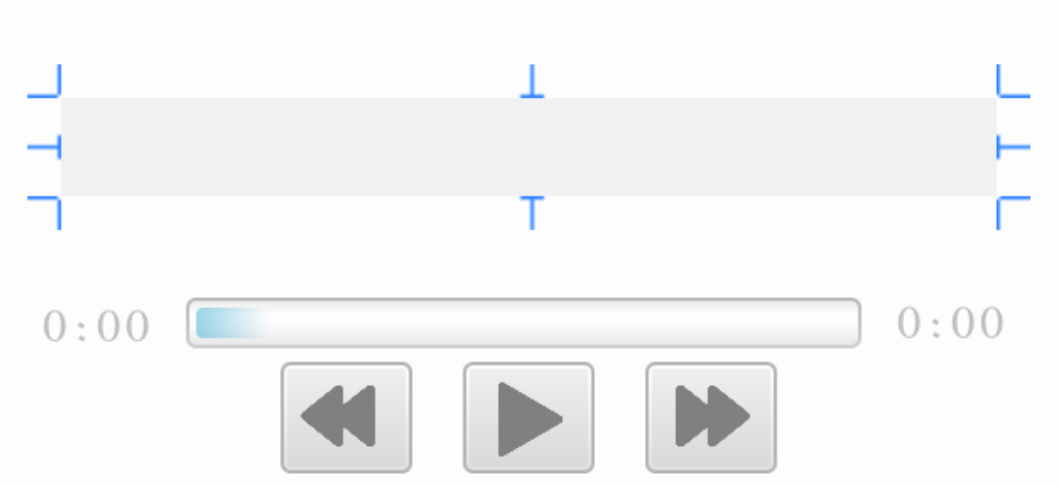
These buttons were are all linked to methods in the back end “On Action”.



Upload Music & Settings contain methods used for specific features like uploading MP3s or editing metadata.

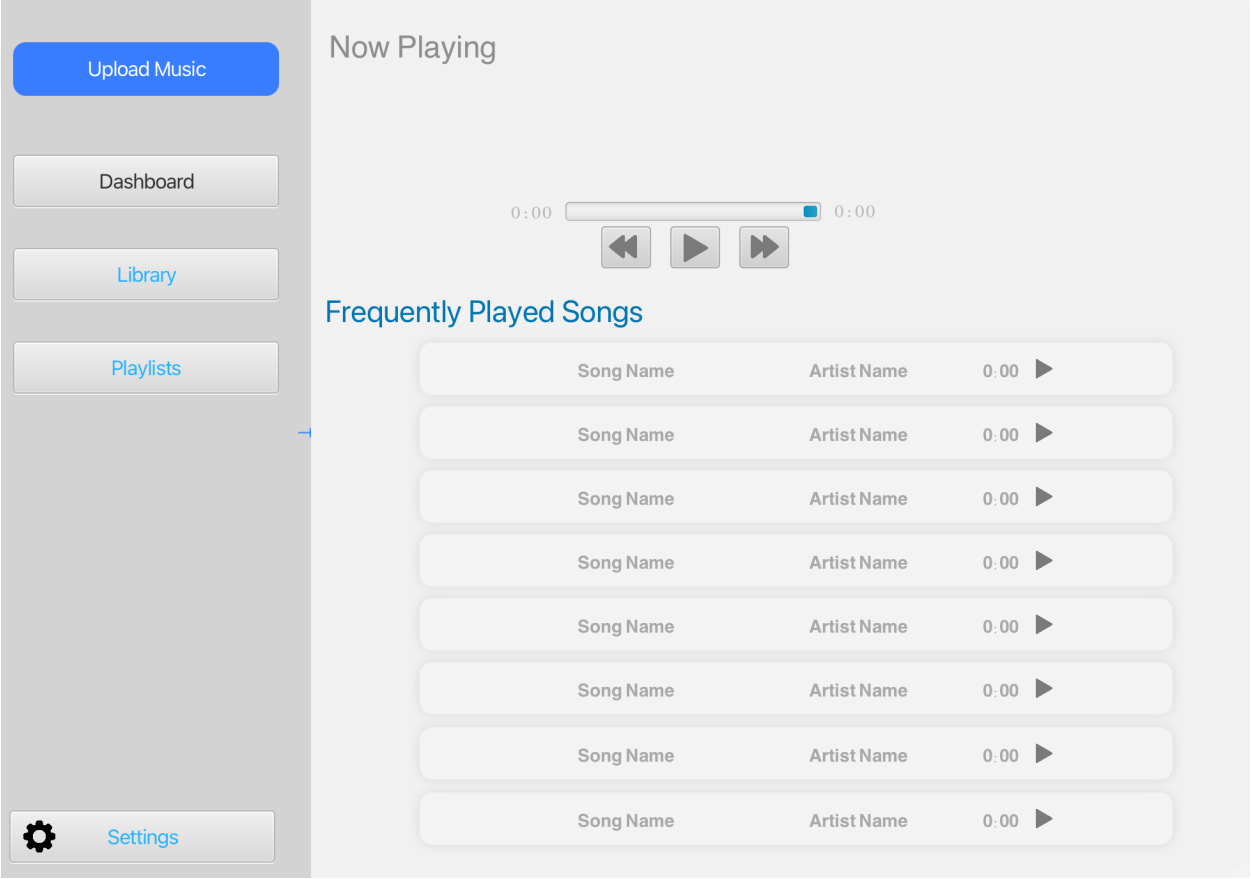
The Dashboard, Library, Playlists are important especially as they contain methods that will allow them to switch between each scene like in the figure above. The label in the button is turns black to show that scene is selected.

The Music Player also had buttons added over the images:

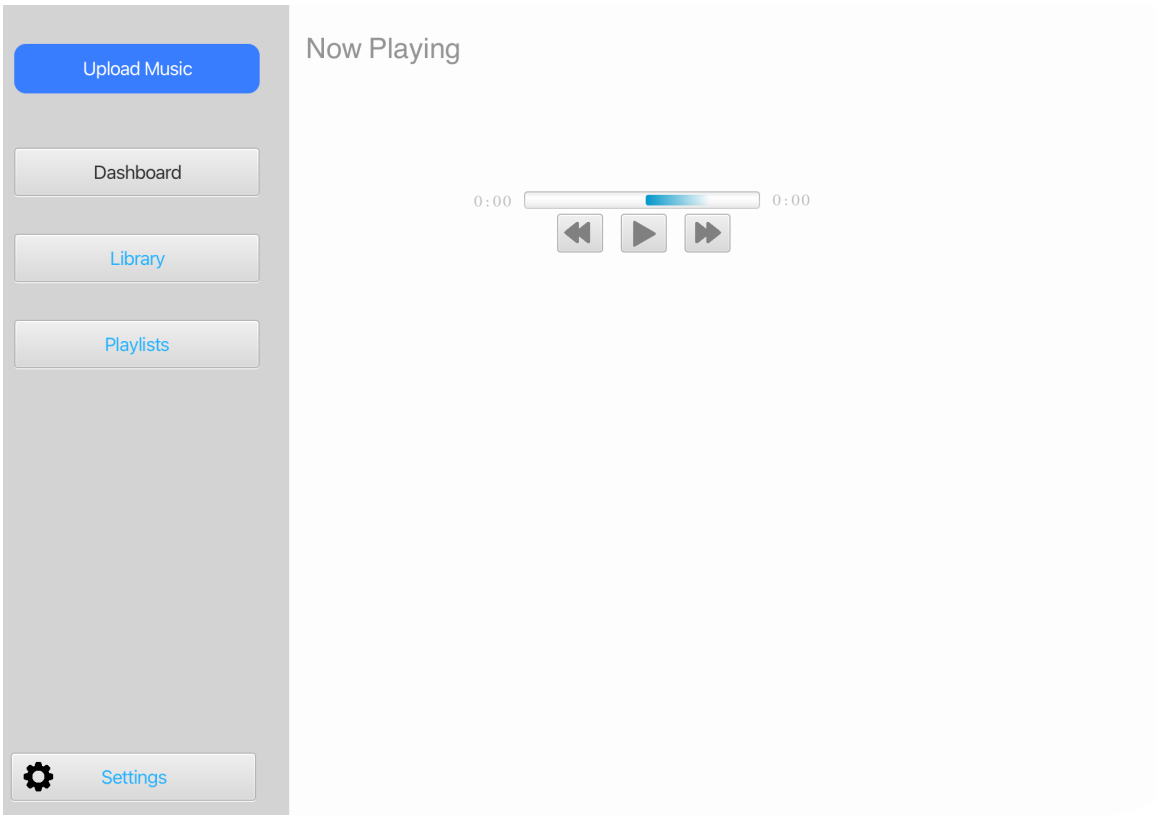


The text in song title, artist, album label was also removed to make the music player a look bit cleaner when no songs are in the app. FXIDs of all these labels have been appended to later be referenced in the back-end.

In the final design for the dashboard, the frequently played artists was removed and replaced by more HBoxes for frequently played songs.



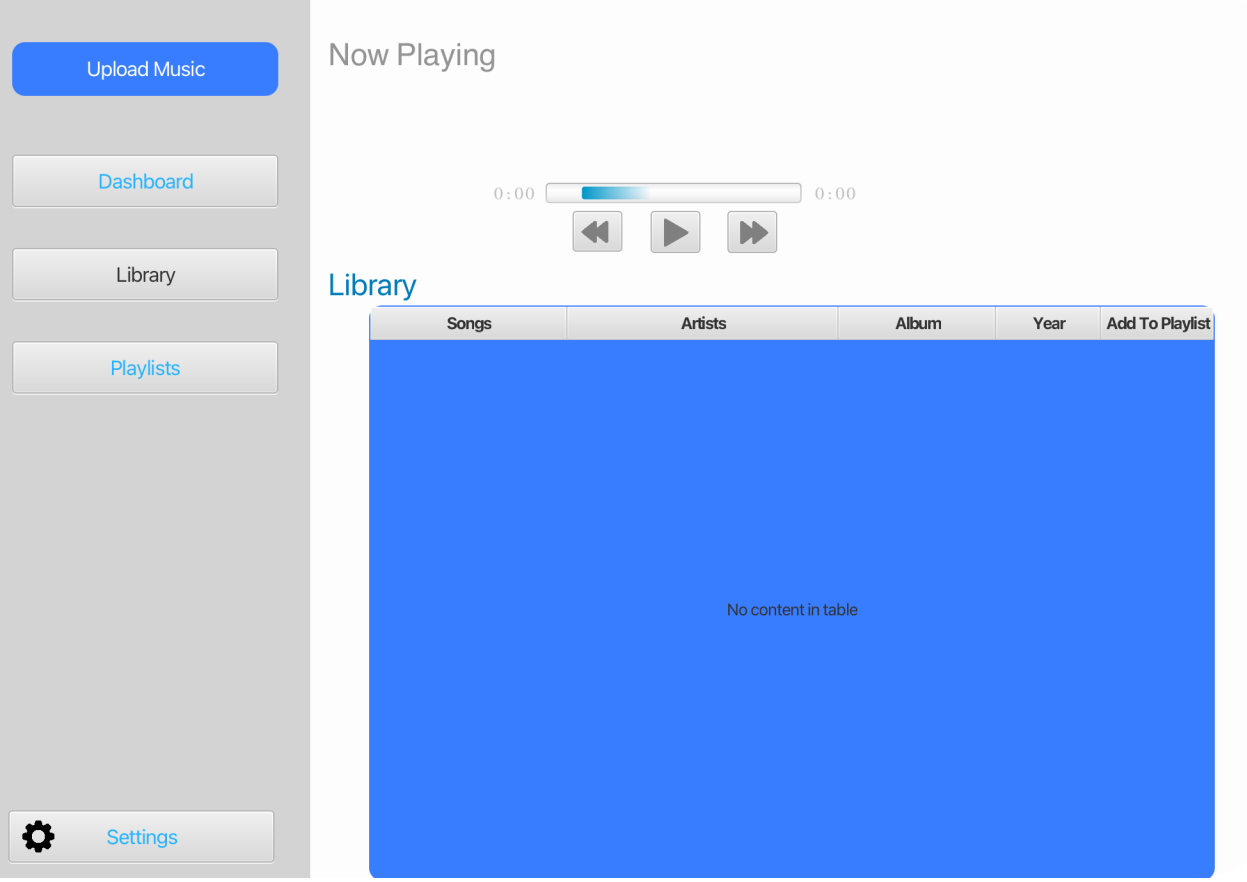
To create the other scenes: Library.FXML and Playlist.FXML, from the main.FXML file, the frequently played section was removed giving a template for the scenes to be created.



Library.FXML

Using the template form the figure above saves a lot of time, all that needs to be implemented is the library as all FX:IDs and on action methods are copied over from the previous scene.

To implement the library, we use a TableView with five cells (Songs,Artists,Album,Year,Add To Playlist) all with their respective FX:IDs to be referred to later in the back-end when we want to populate this table.

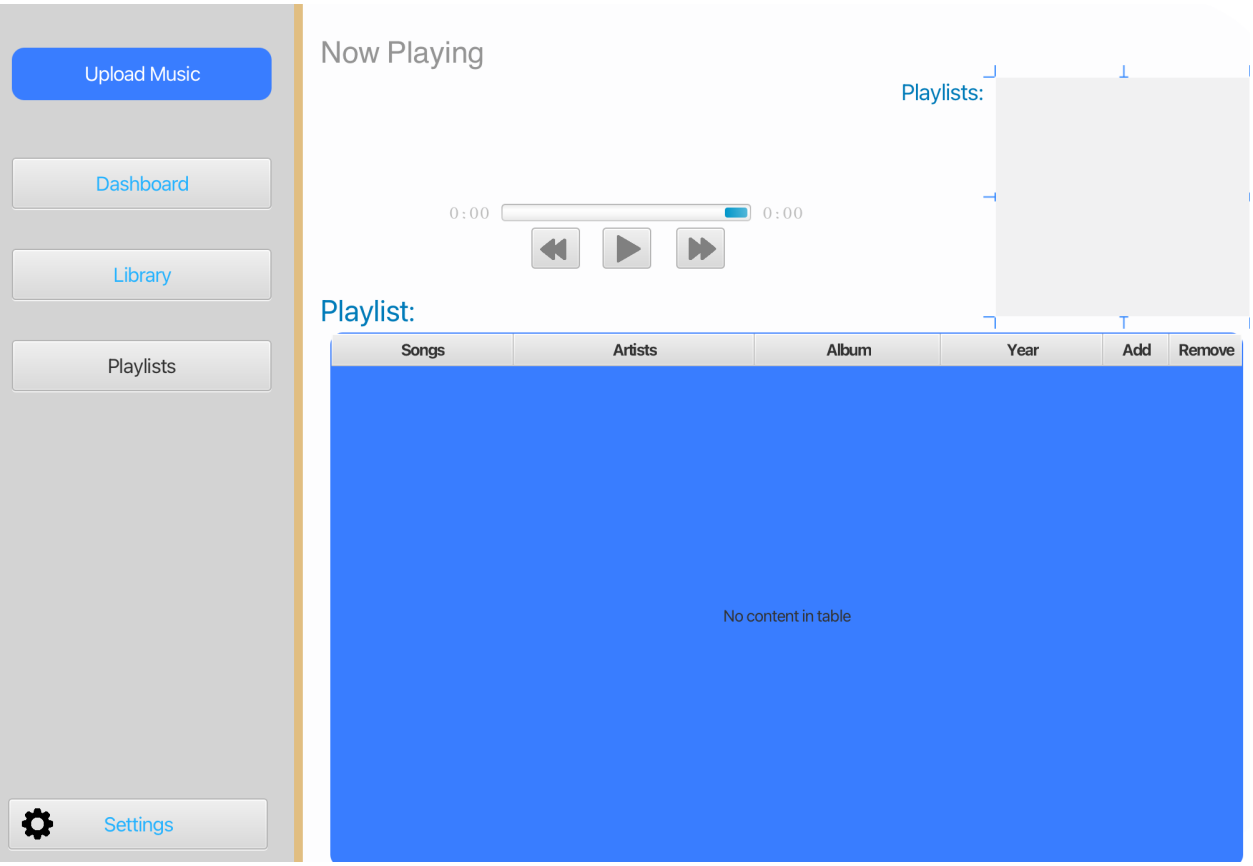


We also make the Library label black to show it is the selected scene.

Playlists.FXML

To implement Playlist.FXML, the template made by Main.FXML could have been used. However, a lot of time can be saved using the new Library.FXML file as a template for Playlist.FXML. This is because they both use a tableview to display songs.

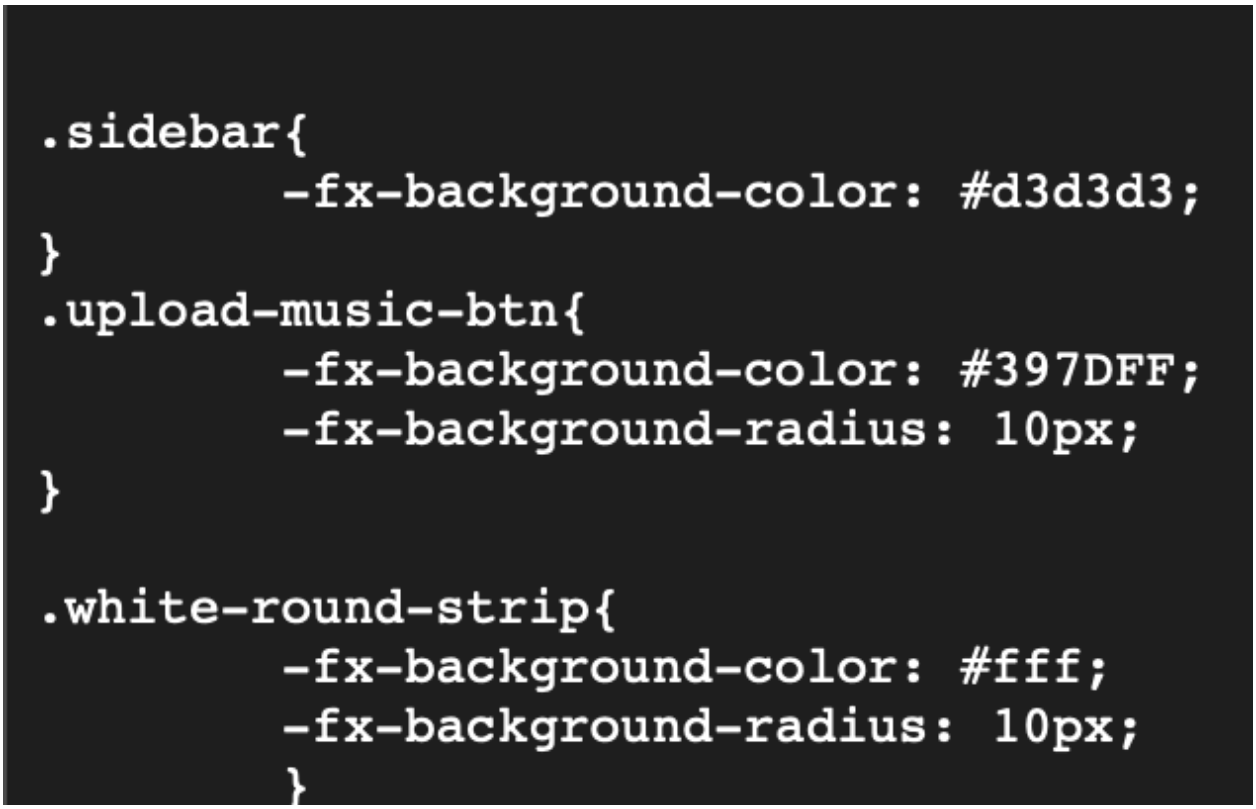
So, with this new template, a new cell is added “Remove” which is linked to new FX:ID. This cell is used to remove songs from the playlist. It will be explained later in the back end.



As can be seen in the top right of this figure, there is something that can be mistaken with an ImageView. This is a FlowPane. It is used as a container to contain the playlist buttons when a new playlist is created. Users will be able to click the buttons to load each playlist.

Application.CSS

This CSS file is used for styling of certain components of the application:



```
.sidebar{
        -fx-background-color: #d3d3d3;
}
.upload-music-btn{
        -fx-background-color: #397DFF;
        -fx-background-radius: 10px;
}

.white-round-strip{
        -fx-background-color: #fff;
        -fx-background-radius: 10px;
        }
```

The components the file edits are:

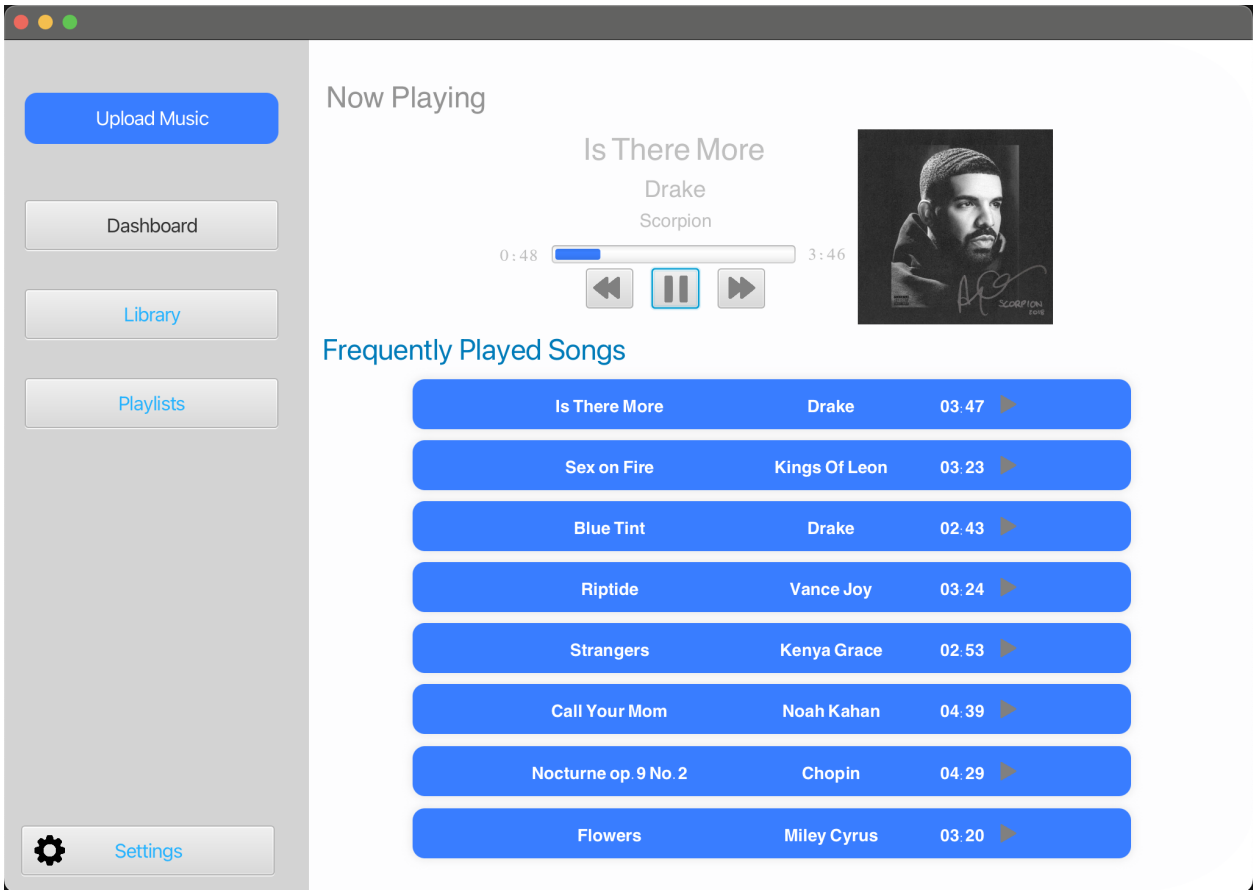
- .sidebar - Makes the sidebar a light grey colour
- .upload-music-btn - Makes the upload music button blue and rounded on the corners.
- .white-round-strip - Used for the HBoxs in Main.FXML, rounding off on the corners.

I
mplemented front-end dashboard with minimalistic design while using bold colours on certain objects to make them stand out such as the upload music button and song pictures look for a study suggesting light colours make pictures stand out more

Final Product of Front-End

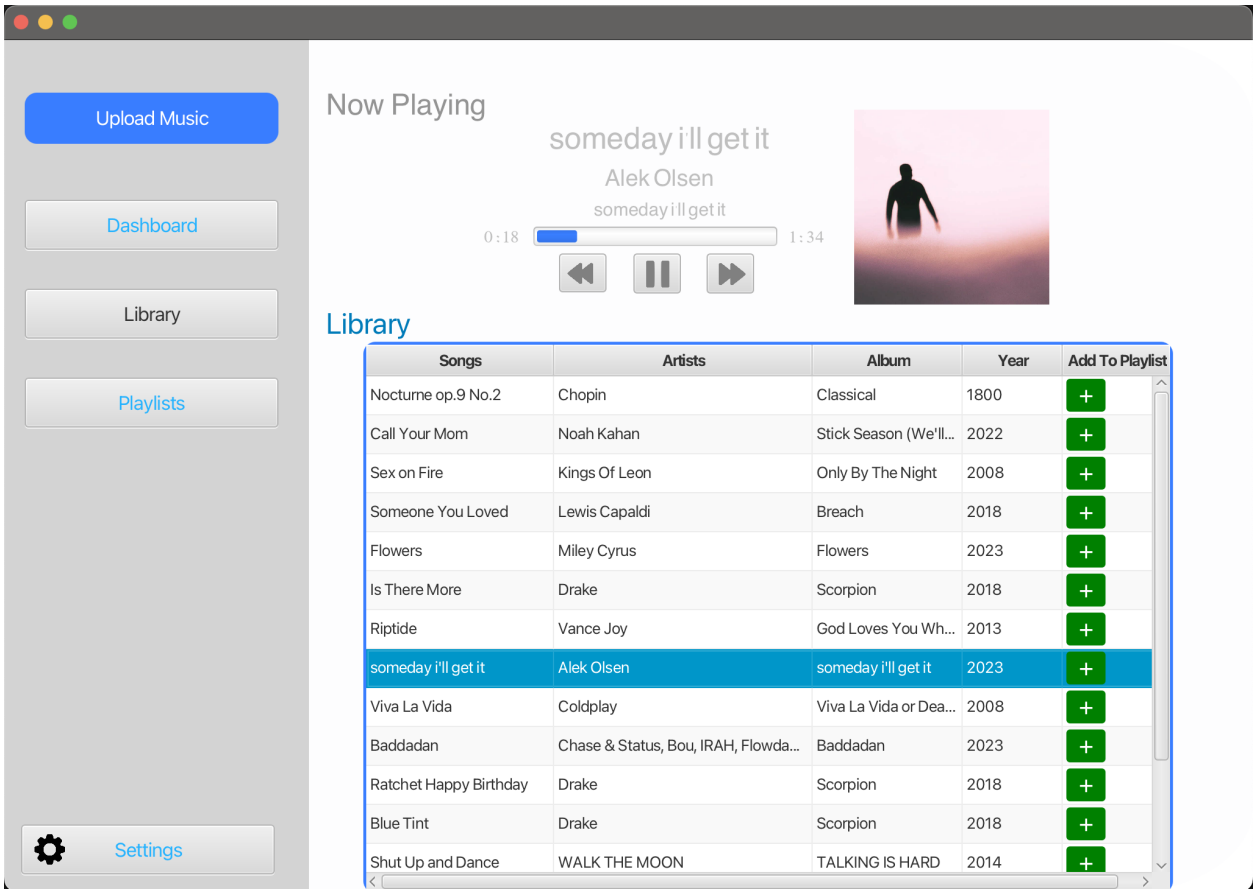
Main.FXML

This is the main dashboard for the RetroSonic application, it is loaded up first by Main.java by default, then it can be switched to by the switchToMain() method in SceneController.Java activated but the Dashboard button:



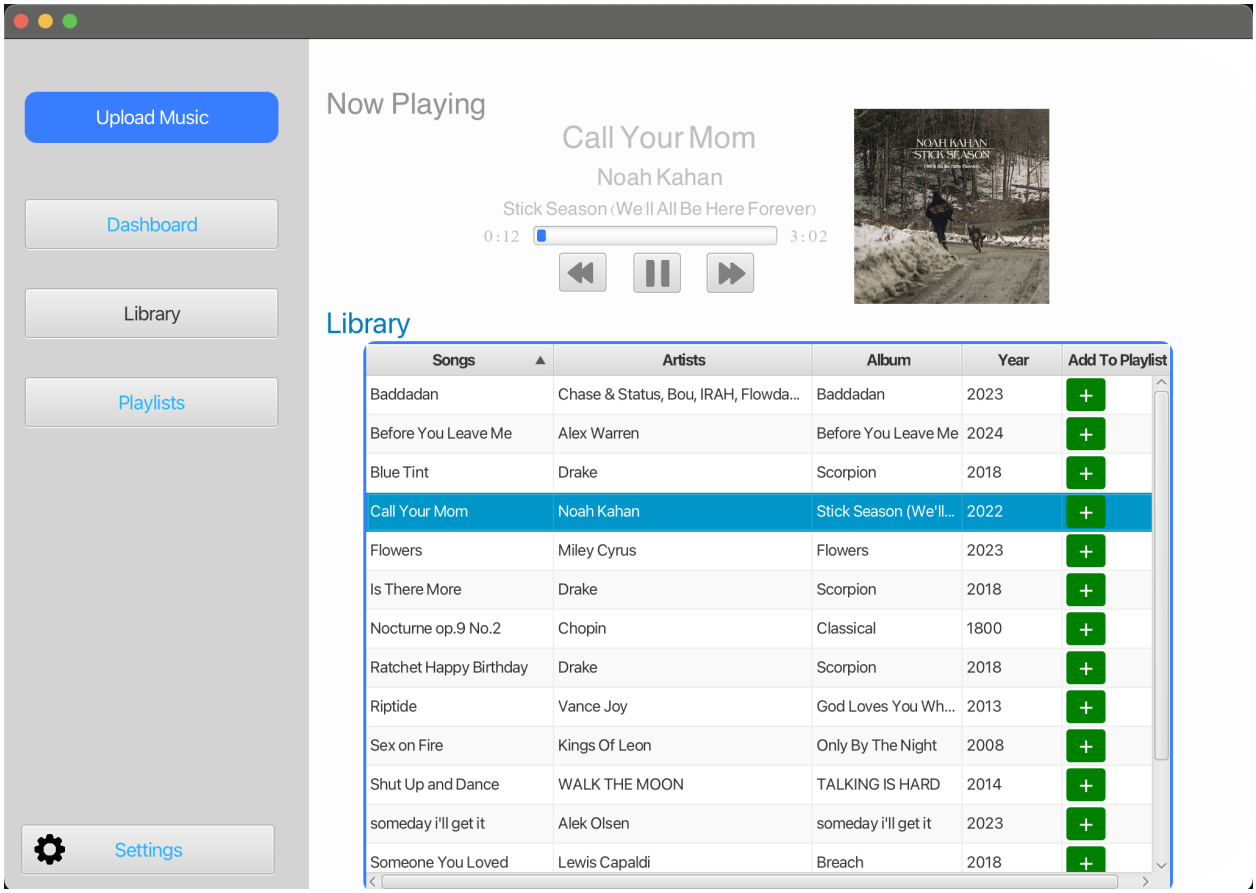
Library.FXML

This is the library scene of the application, it is switched to by the switchToLibrary() method in SceneController.Java activated by the Library button:

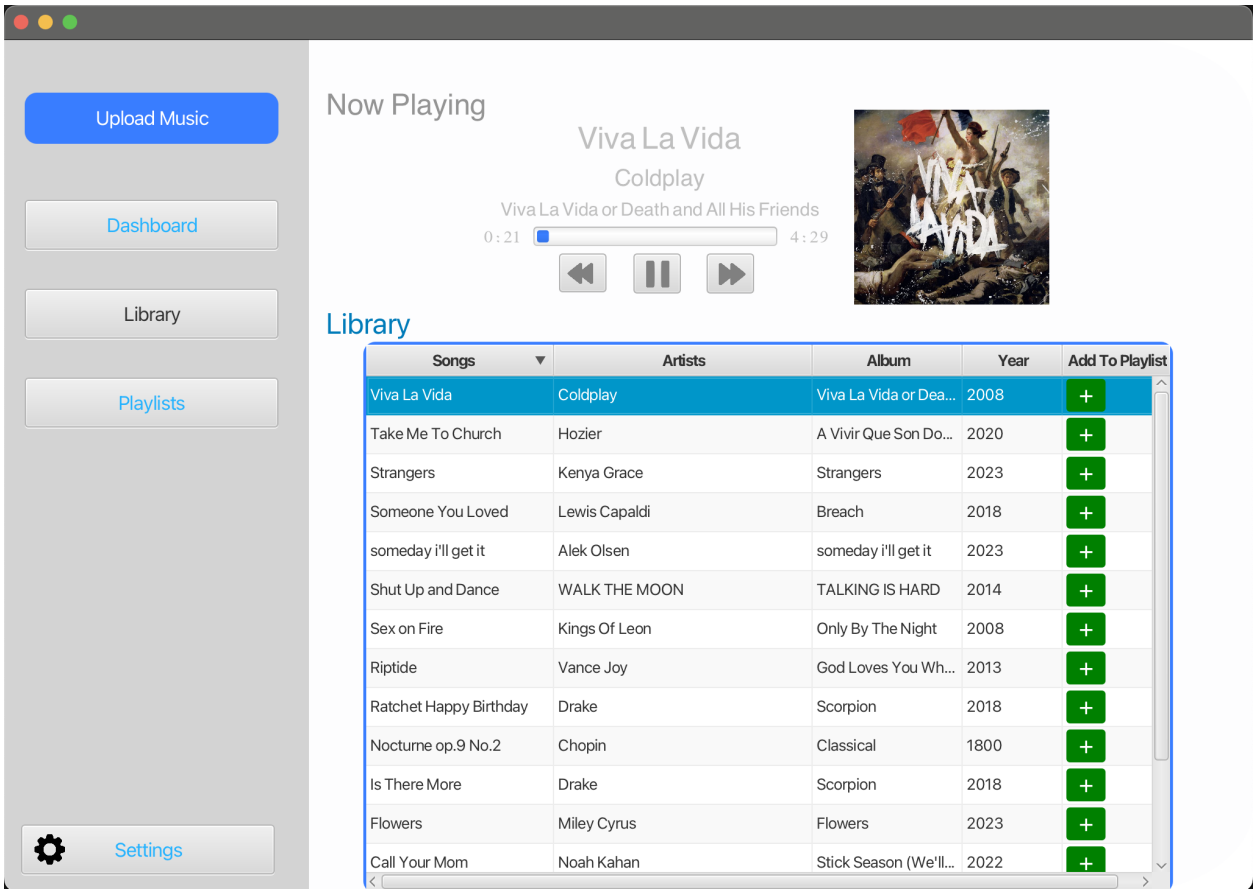


Songs in the tableview can be double clicked to be played due to the method in

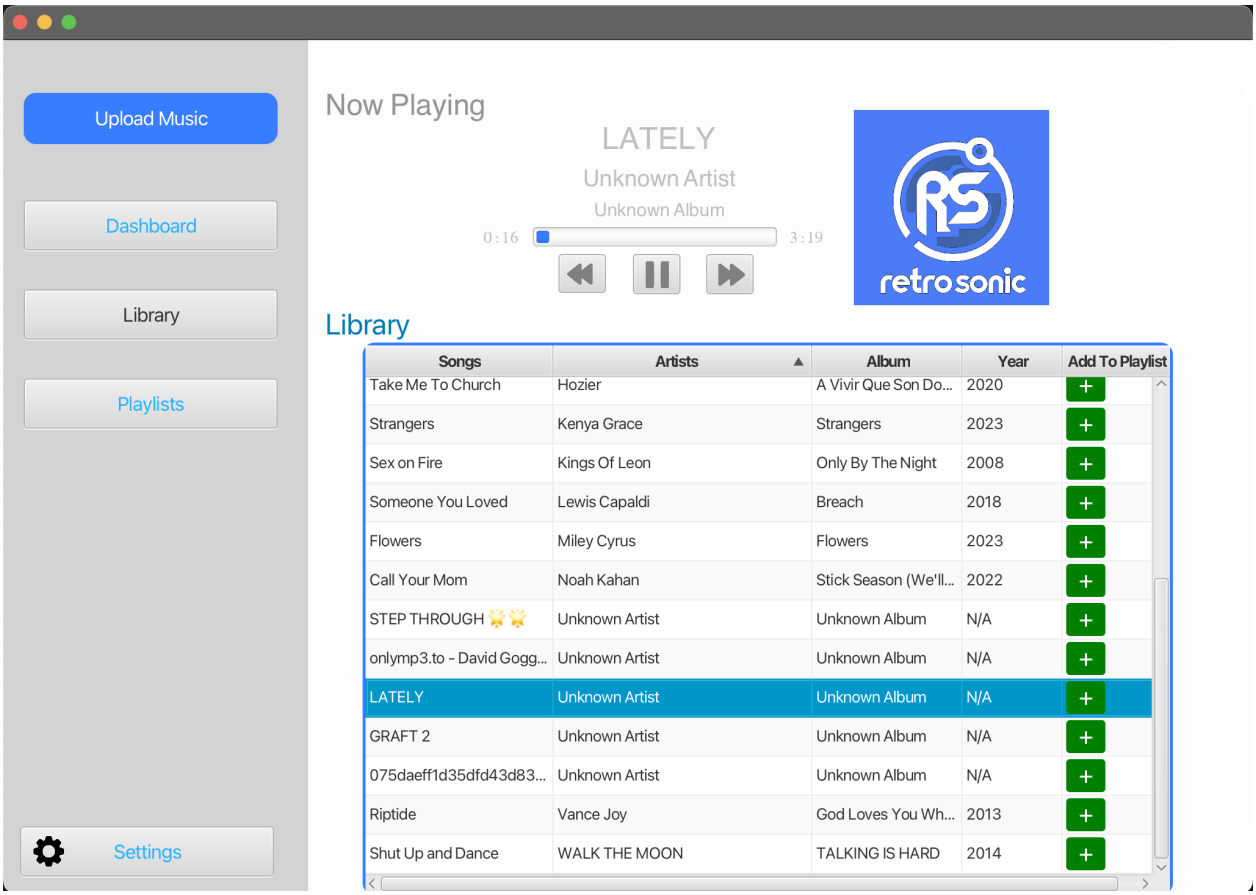
Each cell of the tableview can be adjusted to be viewed in alphabetical order or chronologically in the case of year:



It can also represent songs, artists, albums in reverse alphabetical order (Z-A) and years in reverse chronological order:



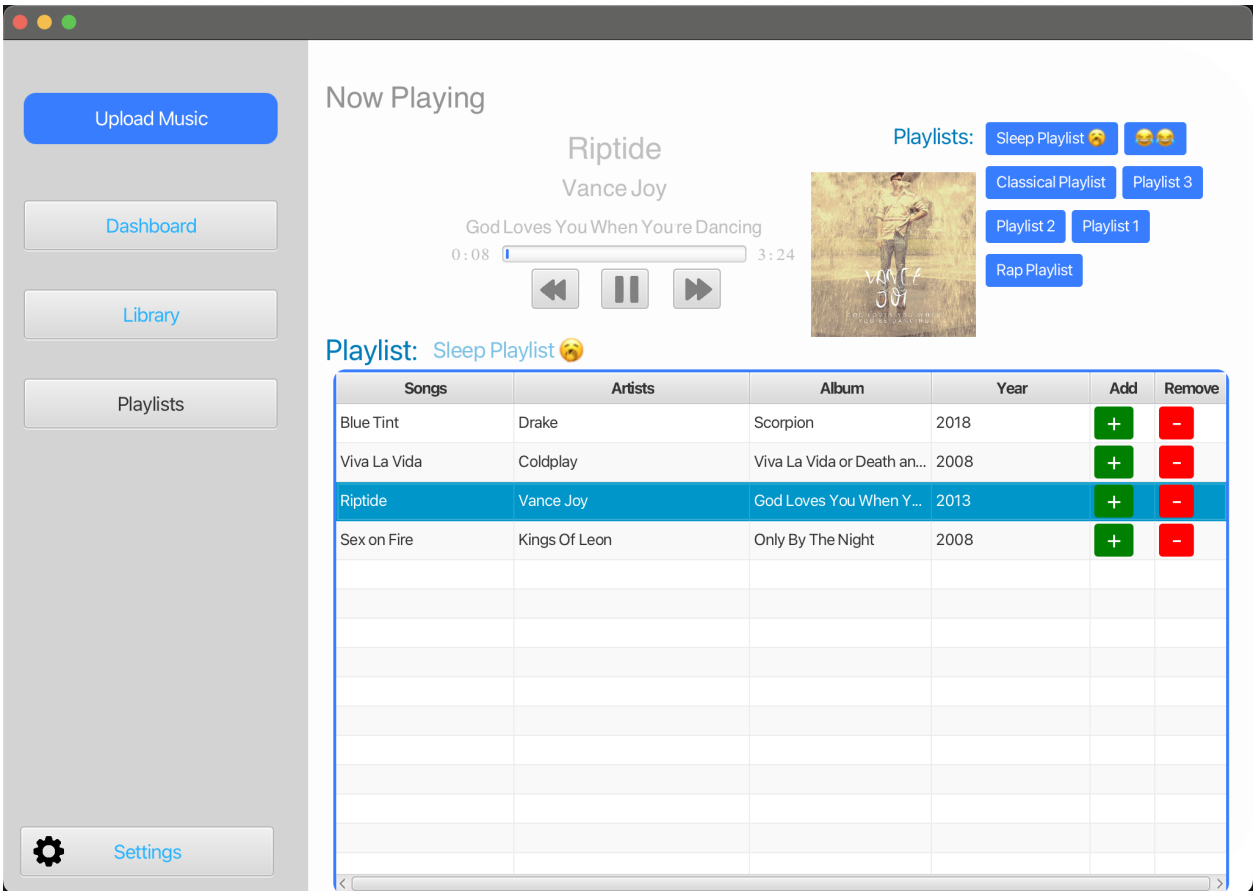
The app also gives default values for songs which don't have metadata tags, this is implemented in Song.java. This allows for songs to be represented like this:



As can be seen, when there are no metadata tags a default image is placed and default values are assigned to fill in the table view cells and the media player labels.

Playlist.FXML

This is the playlist scene of the application, it is switched to using the switchToPlaylist() method in SceneController.Java activated by the Playlists button:

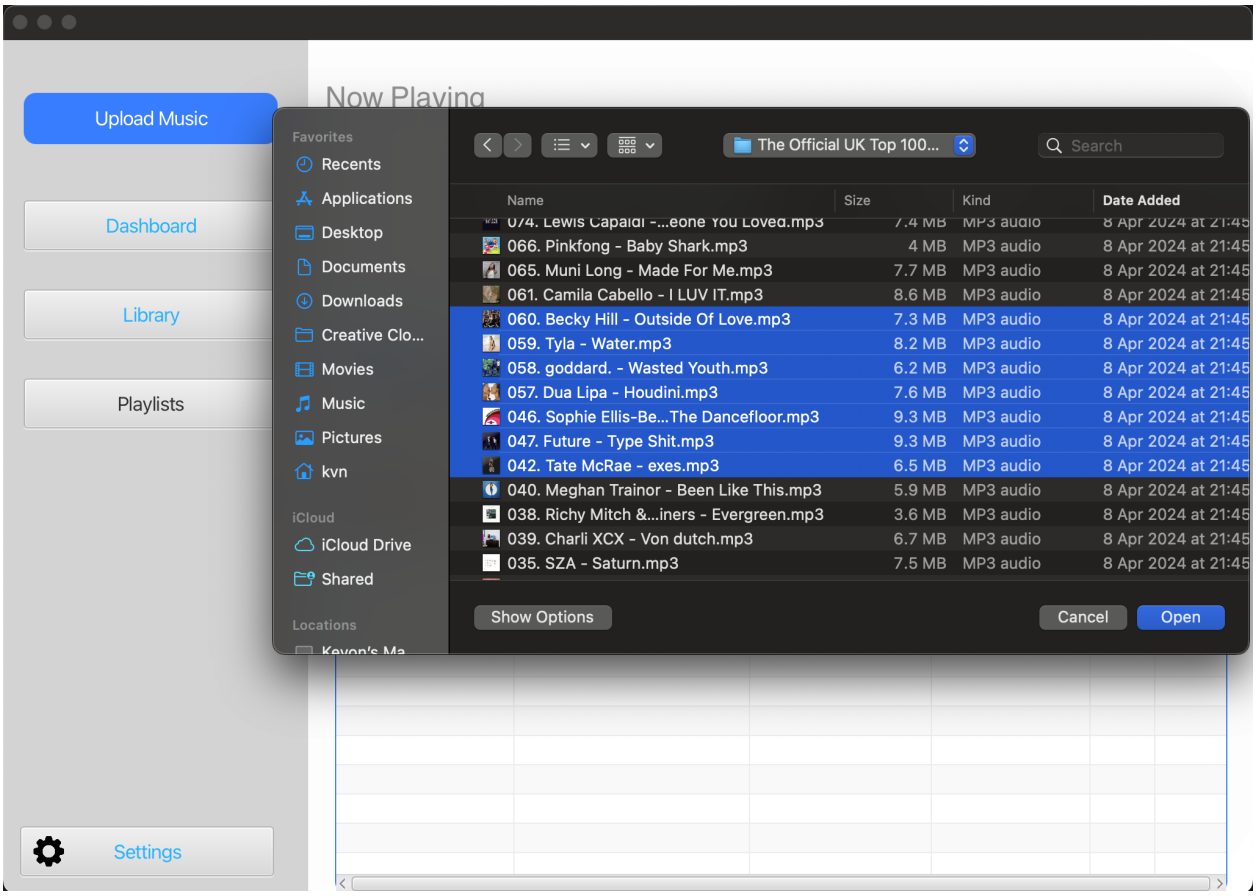


The playlists are displayed in the top right. Press to select one and it will populate one. Functionality to add songs to playlists still exists, songs can be removed from playlists also using the “Remove” cell. Songs can still be double clicked in the table to be selected.

Dialogue Screens

Upload Music Button:

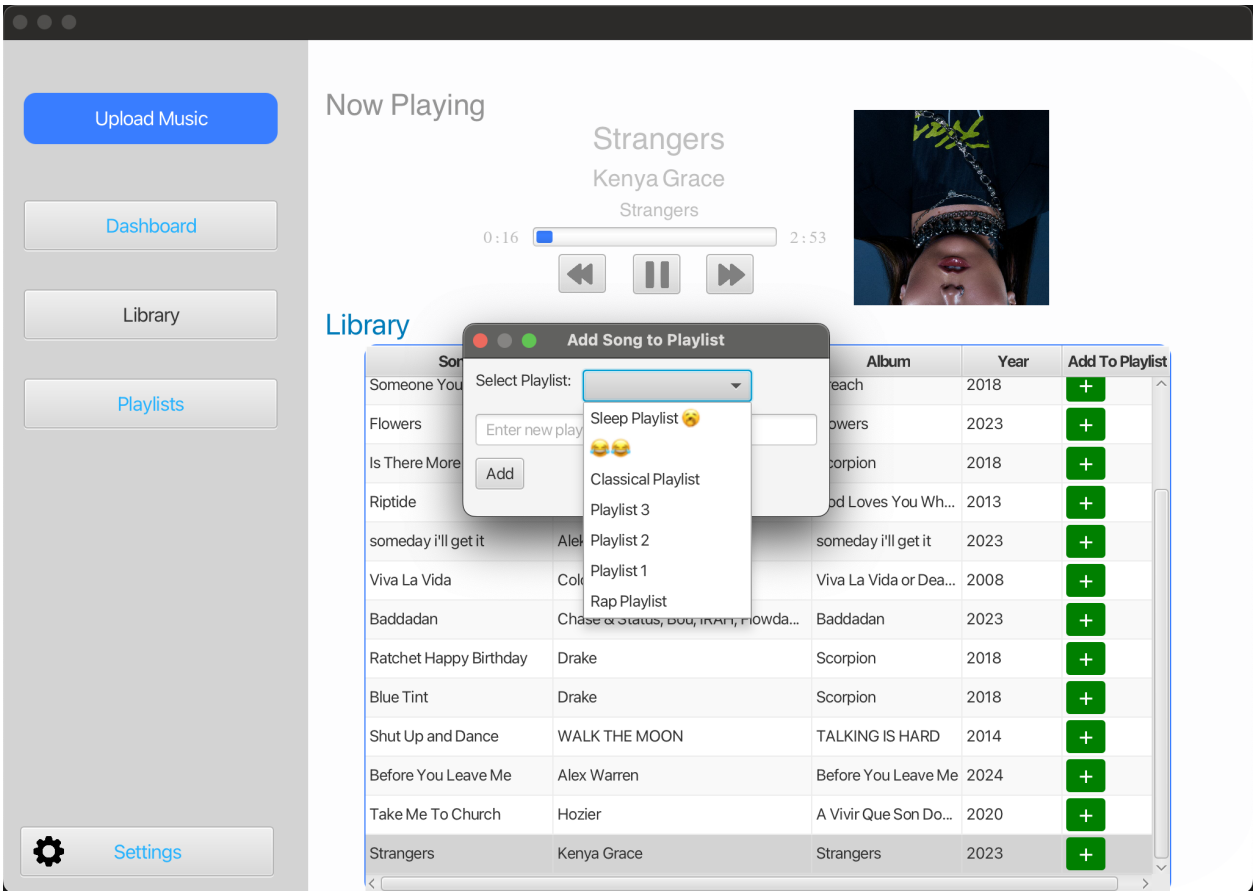
When the “Upload Music” Button is clicked, it’s embedded method is the uploadMusicAction() within SceneController.Java, it opens up the directory for the user to select songs:



This allows users to upload an MP3 file or a directory of MP3 files so that the user doesn't have to upload songs one by one.

***Playlists*:**

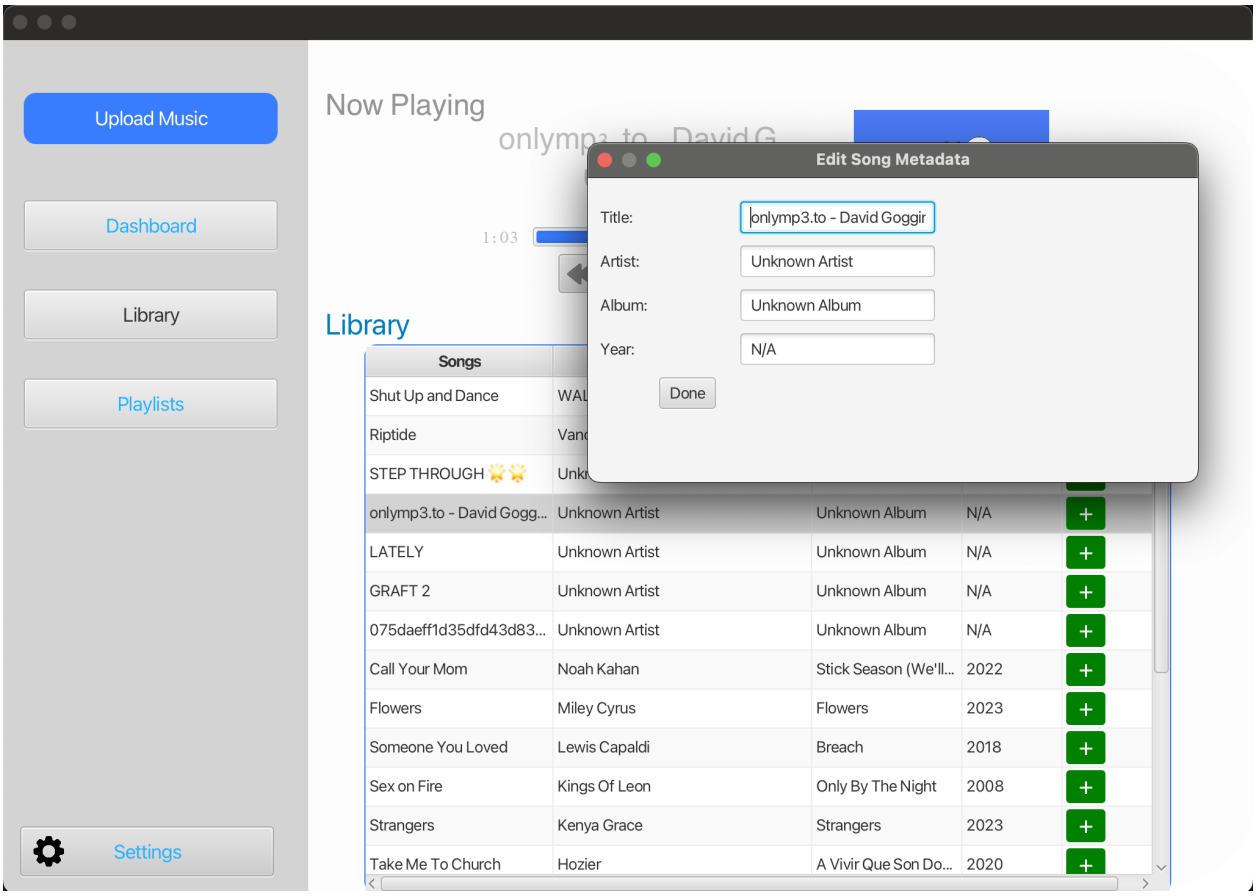
When the "Add to Playlist" Button is clicked on Library.FXML or the "Add" Button is clicked on Playlists.FXML, it uses the showAddToPlaylistDialog() method in the SceneController.Java, causing this dialogue box to pop up:



The user has the ability to type in the text field a new playlist name or they can select a playlist that has already been created to add the selected song.

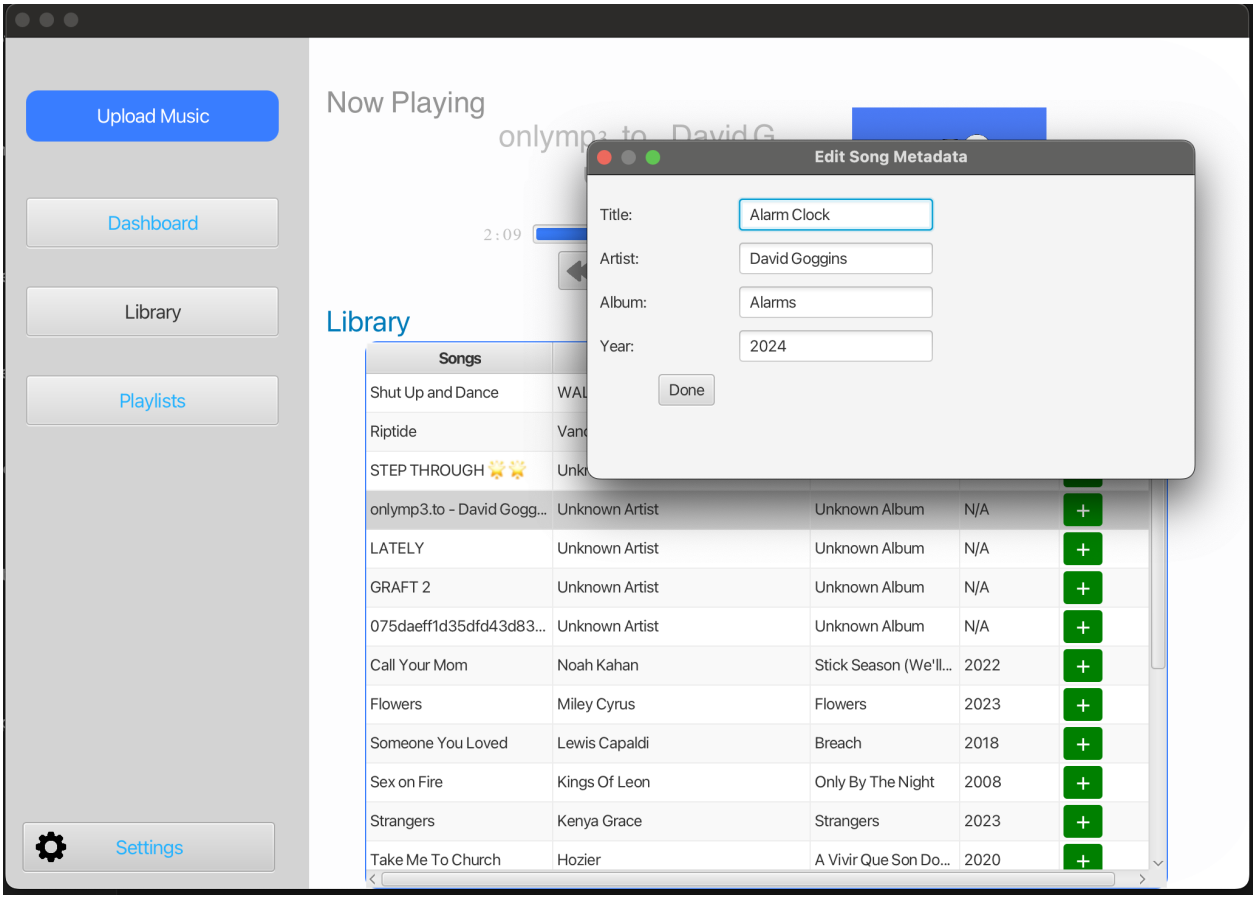
***Settings*:**

When the Settings Button on the sidebar is pressed, the showEditMetadataDialogue() method activates, causing this dialogue box pops up with the ability to edit the currently selected song:



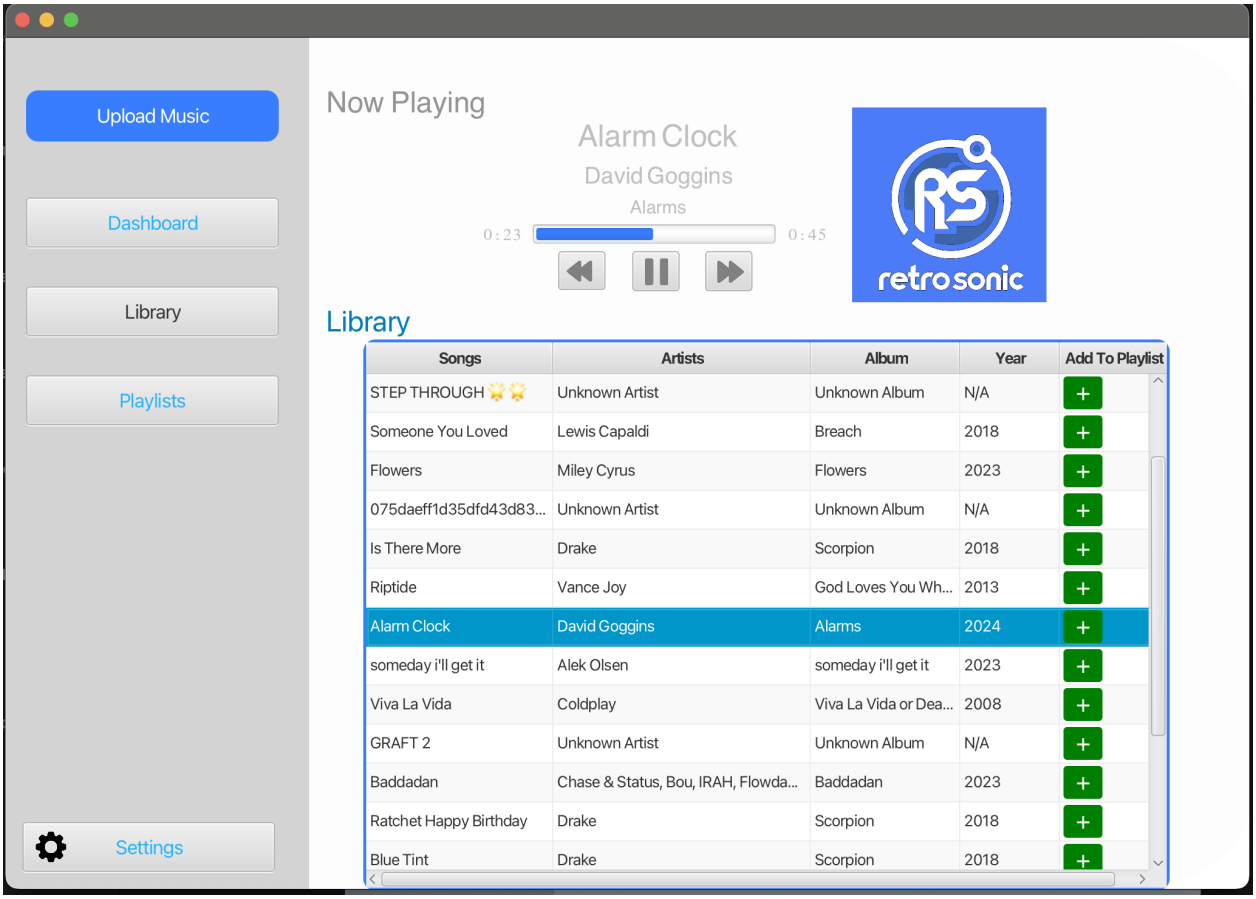
As can be seen in the figure above the song has not been embedded with tags and has been given default values shown in the text fields.

These text fields can be edited as such to fill in tags for songs which are tag-less.



Here the metadata in the fields has been edited to show something new.

When the Done button is selected the changes are made permanent, due to the MP3TagWriter.java class, which takes advantage of the external MP3agic library's extensive getter/setter methods:



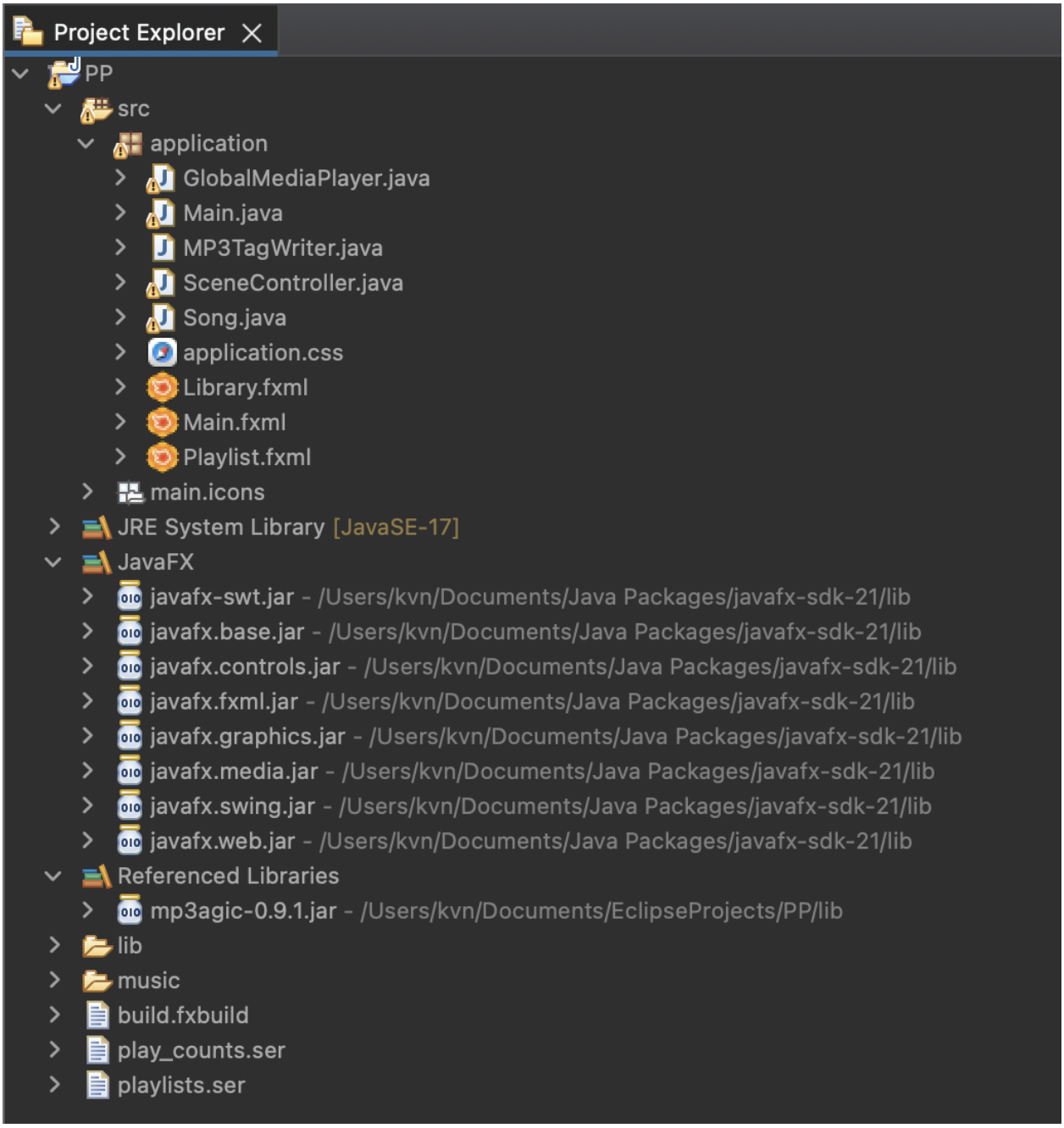
As witnessed from the figure above, the changed tags are made apparent once the scene is refreshed.

Back-End

The back end was developed after the front end FXIDs get linked

Environment

This is the Directory of RetroSonic within Eclipse. Eclipse was my chosen environment for this project due to my experience using it from COMP202-Software Engineering 1. I am using Eclipse with the e(fx)clipse 3.8.0 plugin set which is used to support the experience of JavaFX on Eclipse:



The file is directory is named PP, originally after the previous name given to the application (PurePlay), before deciding the name RetroSonic was more appropriate.

Within this folder contained are:

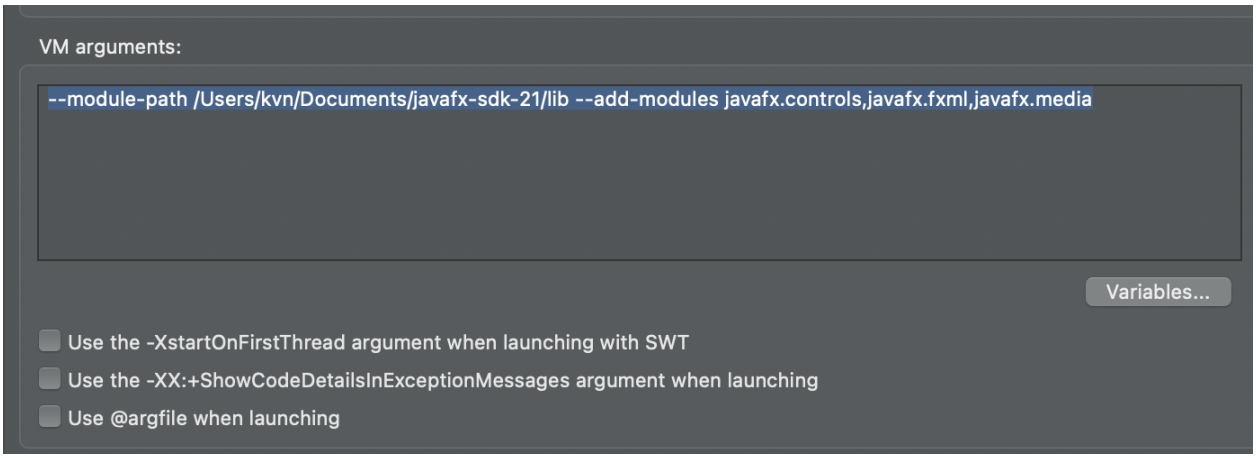
- Libraries

    - JRE - Used for running Java applications by providing Java Runtime Environment
    - JavaFX - Used for creating UI and implementing the MediaPlayer api
    - Mp3Agic- Supports handling and manipulation of MP3 metadata directly. Used within Song.java to extract metadata and within MP3tagwriter.java to write tags effectively.
- Serialisable Files; Data is serialised to the file when session is complete, then deserialised to be used when app is ran again. Contains Persistent Data.
    - play_counts.ser - used for saving data of how many times a song is played, persistently, to display frequently played songs in Main.FXML. This file is made in the saveFrequentlyPlayedList() method in GlobalMediaPlayer.java.
    - playlists.ser - used for saving playlists persistently to display them in Playlist.FXML. This file is made by the savePlaylists() method in SceneController.java.
- build.fxbuild - automatically created file created by JavaFX environment within Eclipse.
- Application Files
    - Front-End
        - Main.FXML - Dashboard
        - Library.FXML - Library
        - Playlist.FXML - Playlists
        - Application.CSS - Styling for Front-End of Application
    - Back-End
        - Main.Java - Starts application by loading Main.FXML
        - GlobalMediaPlayer.Java - Controls all music playback and frequently played songs
        - SceneController.java - Controls all interaction with front-end by giving buttons their respective methods on the back-end.

- Song.Java - Extracts metadata of songs using MP3agic and gives default values for songs with no tags.
- MP3TagWriter.Java - Used to set metadata tags of songs using MP3agic library.

Configurations

These are the Virtual Machine arguments used for configuring the JavaFX runtime when launching the application:



The first part "—module-path" sets the module path for the application. The module path is where the Java Runtime looks for modules to load, it currently points to the javafx-sdk library folder containing the JavaFX modules for this application.

The second part "—add-modules" specifies which modules to run at runtime, currently this is:

- .controls - supports user interface controls such as buttons & tables
- .fxml - supports loading user interface defined in fxml files.
- .media - supports the multimedia components: the media player & playback of audio.

Main.Java

This is the extent of Main.Java:

```java
package application;

import javafx.application.Application;


public class Main extends Application {
    @Override
    public void start(Stage primaryStage) {
        try {
            Parent root = FXMLLoader.load(getClass().getResource("Main.fxml"));
            Scene scene = new Scene(root);
            scene.getStylesheets().add(getClass().getResource("application.css").toExternalForm());
            primaryStage.setScene(scene);
            primaryStage.show();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

The JavaFX application's main entry point is the start() method:

1. It implements FXMLLoader to load in Main.FXML file, containing the User Interface & initialises it into a Parent object. This is the root of the scene graph for the window.
2. A new scene is then created from the Parent root which is then a container for all content in the stage.
3. This scene then adds the stylesheets from Application.css for styling the front-end.
4. This new scene is then set onto the primary stage
5. Then the primary stage is displayed for the user.

This setup clearly separates the application's configuration and startup sequence into well-organised blocks, making use of JavaFX's capabilities to load and style the user interface declaratively using FXML and CSS.

MP3TagWriter.Java -explain beter way of doing it

The MP3TagWriter class implements the MP3agic library to take advantage of the setter methods (Title, Artist, Album, Year)

```java
public class MP3TagWriter
{
    public static void setTitle(Song song, String newTitle) throws NotSupportedException, IOException {
        try
        {
            Mp3File mp3file = new Mp3File(song.getFilePath());
            ID3v2 tag = getOrCreateId3v2Tag(mp3file);
            tag.setTitle(newTitle);
            saveMp3File(mp3file, song.getFilePath(), "temp_title.mp3");  // Using a temporary file for saving
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void setArtist(Song song, String newArtist) throws NotSupportedException, IOException {
        try
        {
            Mp3File mp3file = new Mp3File(song.getFilePath());
            ID3v2 tag = getOrCreateId3v2Tag(mp3file);
            tag.setArtist(newArtist);
            saveMp3File(mp3file, song.getFilePath(), "temp_artist.mp3");  // Using a temporary file for saving
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void setAlbum(Song song, String newAlbum) throws NotSupportedException, IOException {
        try
        {
            Mp3File mp3file = new Mp3File(song.getFilePath());
            ID3v2 tag = getOrCreateId3v2Tag(mp3file);
            tag.setAlbum(newAlbum);
            saveMp3File(mp3file, song.getFilePath(), "temp_album.mp3");  // Using a temporary file for saving
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void setYear(Song song, String newYear) throws NotSupportedException, IOException {
        try
        {
            Mp3File mp3file = new Mp3File(song.getFilePath());
            ID3v2 tag = getOrCreateId3v2Tag(mp3file);
            tag.setYear(newYear);
            saveMp3File(mp3file, song.getFilePath(), "temp_year.mp3");  // Using a temporary file for saving
        }
        catch (Exception e) {
```

The currently selected song's 4 edited textfields from the Settings Dialogue Screen ***[Front-End⇒Dialogue Screens⇒Settings]*** are taken as parameters for setTitle, setArtist, setAlbum & setYear respectively.

This is where the songs file path is used to initialise an Mp3File object - this is object is used for modification.

The getOrCreateId3v2Tag() method reads the current Id3v2Tag of the song or creates one if it doesn't yet exist to the tag variable:

```java
    private static ID3v2 getOrCreateId3v2Tag(Mp3File mp3file) {
        ID3v2 tag = mp3file.getId3v2Tag();
        if (tag == null) {
            tag = new ID3v24Tag();
            mp3file.setId3v2Tag(tag);
        }
        return tag;
    }
```

The input from the textfield is then set onto the tag. The modified MP3 file is then saved to a temporary file. This temporary file then replaces the original source file:

```java
    private static void saveMp3File(Mp3File mp3file, String originalPath, String tempPath) throws NotSupportedException,
        try {
            mp3file.save(tempPath);
            replaceOriginalFile(originalPath, tempPath);
        } catch (Exception e) {
            e.printStackTrace();
            throw new NotSupportedException("Could not save the MP3 file with updated tags: " + e.getMessage());
        }
    }

    private static void replaceOriginalFile(String originalPath, String tempPath) throws IOException {
        File originalFile = new File(originalPath);
        File tempFile = new File(tempPath);

        if (originalFile.delete()) {
            if (!tempFile.renameTo(originalFile)) {
                throw new IOException("Failed to rename " + tempPath + " to " + originalPath);
            }
        } else {
            throw new IOException("Failed to delete the original file: " + originalPath);
        }
    }
}
```

The result is the original file with permanent changes in its metadata tags. This replacement process is done to avoid corruption of the original file during the save operation. Without this method, this error was consistent when trying to change metadata:

```
java.lang.IllegalArgumentException: Save filename same as source filename
        at com.mpatric.mp3agic.Mp3File.save(Mp3File.java:446)
        at application.MP3TagWriter.saveMp3File(MP3TagWriter.java:77)
        at application.MP3TagWriter.setTitle(MP3TagWriter.java:20)
        at application.SceneController.lambda$3(SceneController.java:1074)
        at javafx.base@21/com.sun.javafx.event.CompositeEventHandler.dispatchBubblingEvent(CompositeEventHandler.java:86)
        at javafx.base@21/com.sun.javafx.event.EventHandlerManager.dispatchBubblingEvent(EventHandlerManager.java:232)
        at javafx.base@21/com.sun.javafx.event.EventHandlerManager.dispatchBubblingEvent(EventHandlerManager.java:189)
        at javafx.base@21/com.sun.javafx.event.CompositeEventDispatcher.dispatchBubblingEvent(CompositeEventDispatcher.java:59
        at javafx.base@21/com.sun.javafx.event.BasicEventDispatcher.dispatchEvent(BasicEventDispatcher.java:58)
        at javafx.base@21/com.sun.javafx.event.EventDispatchChainImpl.dispatchEvent(EventDispatchChainImpl.java:114)
        at javafx.base@21/com.sun.javafx.event.BasicEventDispatcher.dispatchEvent(BasicEventDispatcher.java:56)
        at javafx.base@21/com.sun.javafx.event.EventDispatchChainImpl.dispatchEvent(EventDispatchChainImpl.java:114)
        at javafx.base@21/com.sun.javafx.event.BasicEventDispatcher.dispatchEvent(BasicEventDispatcher.java:56)
        at javafx.base@21/com.sun.javafx.event.EventDispatchChainImpl.dispatchEvent(EventDispatchChainImpl.java:114)
        at javafx.base@21/com.sun.javafx.event.EventUtil.fireEventImpl(EventUtil.java:74)
        at javafx.base@21/com.sun.javafx.event.EventUtil.fireEvent(EventUtil.java:49)
        at javafx.base@21/javafx.event.Event.fireEvent(Event.java:198)
        at javafx.graphics@21/javafx.scene.Node.fireEvent(Node.java:8875)
        at javafx.controls@21/javafx.scene.control.Button.fire(Button.java:203)
```

This is also a downside of MP3agic which cannot write changes to an MP3 file directly (in-place modification). I was consider using a different library such as Jaudiotagger which supports in-place modification. This would lead to less lines of code, however, research into in-place editing reveals that it is risky and not generally recommended because of the potential for data loss if an error occurs during the write process. It is safer to write to a temporary file and then replace the original file after the write is successful, like what is done in the saveMP3File() method.

Song.Java

The Song Class implements Serializable interface to allow the object to get serialised for the persistent capabilities of the application:

```
public class Song implements Serializable
{
    private String title;
    private String artist;
    private String album;
    private String year;
    private String duration;
    private File file;
    private transient Image albumArt;//stop it getting serialised into song data

    // Constructor
    public Song(File file)
    {
        // Initialize with default values
        this.title = "";
        this.artist = "";
        this.album = "";
        this.year = "";
        this.file = file;

        extractMetadata();
        extractAlbumArt();
    }
```

The albumArt image is made transient to avoid serialisation of the image. This to avoid a notSerializableException error cause by trying to serialise the image:

This is because images are not serialisable in java by default due to images being large and complex. This deficiency does not command any shortcomings, however, if necessary there exists methods to bypass this like using a BufferedImage to handle the pixel data into a format that can be easily serialised:

```
public byte[] serializeImage(BufferedImage img) throws IOExcepti
    ByteArrayOutputStream stream = new ByteArrayOutputStream();
    ImageIO.write(img, "png", stream);
    return stream.toByteArray();
}
```

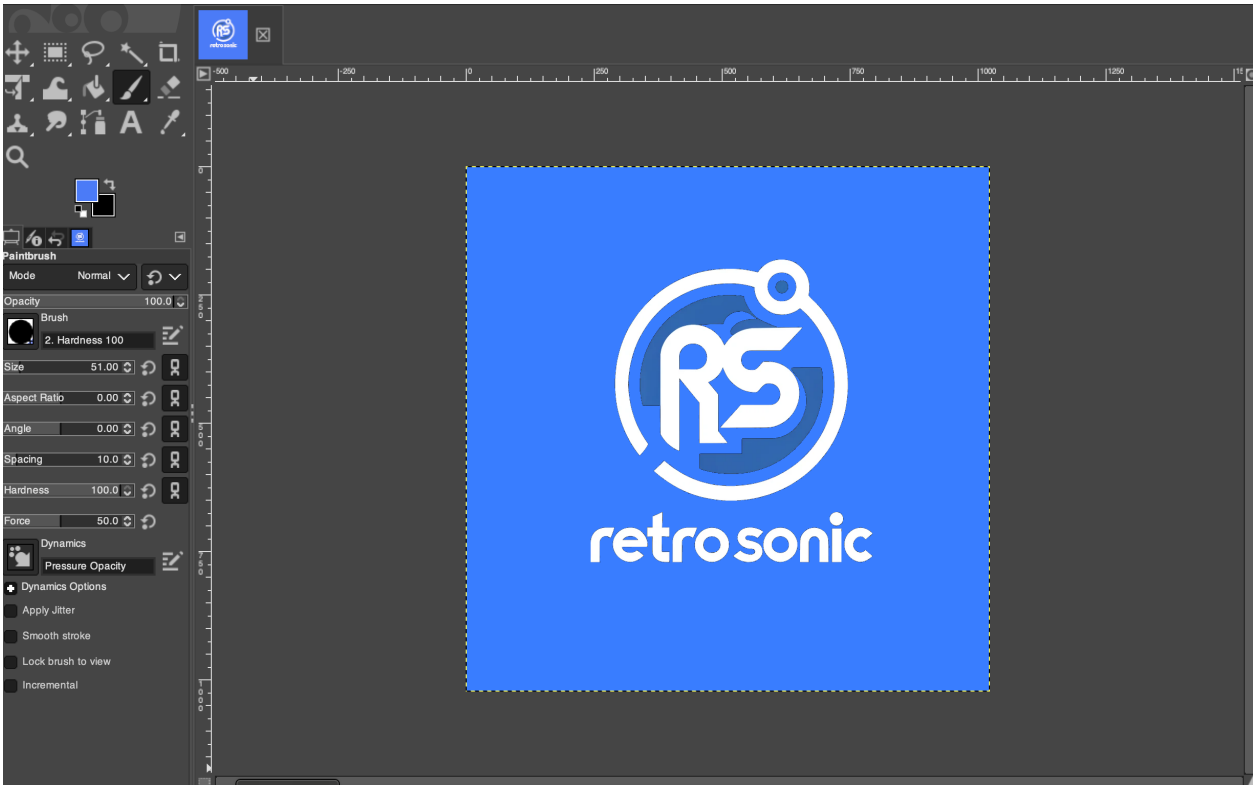
The Song Class also imports the MP3agic library to take advantage of its getter methods to extract the metadata and album arts of each song. This is done using the extractMetadata() and extractAlbumArt() methods which are activated in the constructor:

```
private void extractMetadata() {
    try {
        Mp3File mp3file = new Mp3File(this.file.getPath());
        String name = file.getName();//Incase file has no title
        name = name.substring(0, name.length() - 4);//remove .mp3

        if (mp3file.hasId3v2Tag()) {
            ID3v2 id3v2tag = mp3file.getId3v2Tag();
            title = id3v2tag.getTitle() != null ? id3v2tag.getTitle() : name;
            artist = id3v2tag.getArtist() != null ? id3v2tag.getArtist() : "Unknown Artist";
            album = id3v2tag.getAlbum() != null ? id3v2tag.getAlbum() : "Unknown Album";
            year = id3v2tag.getYear() != null ? id3v2tag.getYear() : "N/A";
            // ID3v2 might also contain more info such as comments, genre, etc.
        } else if (mp3file.hasId3v1Tag()) {
            // Some older files might only have ID3v1 tags
            ID3v1 id3v1tag = mp3file.getId3v1Tag();
            title = id3v1tag.getTitle() != null ? id3v1tag.getTitle() : name;
            artist = id3v1tag.getArtist() != null ? id3v1tag.getArtist() : "Unknown Artist";
            album = id3v1tag.getAlbum() != null ? id3v1tag.getAlbum() : "Unknown Album";
            year = id3v1tag.getYear() != null ? id3v1tag.getYear() : "N/A";
        }

        // Extract duration
        long durationSeconds = mp3file.getLengthInSeconds();
        duration = String.format("%02d:%02d", durationSeconds / 60, durationSeconds % 60);
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}


private void extractAlbumArt()
{
    try
    {
        Mp3File mp3file = new Mp3File(this.file.getPath());
        if (mp3file.hasId3v2Tag())
        {
            ID3v2 id3v2tag = mp3file.getId3v2Tag();
            byte[] imageData = id3v2tag.getAlbumImage();
            if (imageData != null) {
                this.albumArt = new Image(new ByteArrayInputStream(imageData));
            }
            else
            {
                this.albumArt = new Image(getClass().getResourceAsStream("/main/icons/Logo.png"));
            }
        }
    }
    catch (IOException | UnsupportedTagException | InvalidDataException e)
    {
        e.printStackTrace();
    }
}
```

These methods initialise a new Mp3File from the song file path then from the tag extracts the metadata from the song file and assigns it to the respective variable. If the tag extracted is null a default value is used in its place. If the artist, album and year variables is null ⇒ “Unknown Artist/Album” or “N/A” is used. If title is null, the last thing the user wants is to end up with 10 different songs called “Unknown Song”. Therefore, the extractMetaData() method gets the filename of the song and removes the last 4 letters to get the filename without the “.mp3” extension. This

filename lacking an extension is now used as the default title if the title tag is vacant.

The extractAlbumArt() method implements a default image from the icons folder, in the cases where the image is null. This logo was designed in Gimp:



The logo was made to be simple and eye-catching. It uses the same blue (#397DFF) colour as the upload music button, the tableview & the playlist buttons. This colour coding is to abide by a Simplified User Interface where minimum colours should be used and so that no one component stands out more than another on the music player.

GlobalMediaPlayer.Java

The GlobalMediaPlayer class loads in the songData, the songs are loaded in from the music directory through the loadSongs() method in the scenecontroller class as array<files> which then gets turned into an observablelist of song objects to be loaded in:

```java
public class GlobalMediaPlayer implements Serializable
{
    private static MediaPlayer mediaPlayer;
    private static Media media;

    private static String songName;
    private static int songIndex;
    private static ObservableList<Song> songs;

    private static boolean isPlaying = false;

    private static Map<Song, Integer> playCounts = new HashMap<>();// initialises a map wi
    private static final String PLAY_COUNTS_FILE = "play_counts.ser";//for frequently play


    //Imports the songs from SceneController.java
    public static void loadSongs(ObservableList<Song> songData)
    {
        songs = songData;
    }
```

The reason an observablelist is used is because it automatically updates the UI elements that are bound to it whenever added, removed or modified within the list. For example, a Song object has properties such as title, artist, album & year that are displayed and edited through the application's interface ***[Back-End⇒Scenecontroller⇒Editing Metadata]***. Changes to these properties can be automatically propagated through the application. I believe his variable type was the right choice, however, the choice to have both classes hold the song data was wrong. I believe next time, there are two choices:

- Initialise all the song data into the Global Media Player Class avoiding it being stored in the scenecontroller then create the instance the same way through the song number. This would save memory in the application causing it to run quicker. However, the change could not have been made at the late stage I noticed by, due to highly coupled code between the song data stored within the controller and the frequently played songs functionality.
- Initialise the song data & song number as global variables to allow access from all classes. I believe this option is the best, allowing to remove a lot of the complexity & stop having to track the song number between the Classes to know which song is being played. The reason this was not done was because the application was getting to the late implementation stage and this change would not bring a change in functionality in the front-end rather it would remove a layer of complexity from the back-end, which cannot be seen by a user of the app.

These are better options to keep in mind when building an application like this.

The GlobalMediaPlayer also implements the singleton design pattern, that was studied in COMP319-Software Engineering 2. It is used throughout it's initialisation to avoid multiple instances of the same object being created:

```
//Singleton instance of the controller for a specific song (Switching scenes)
public static MediaPlayer getInstance(int songNumber)
{
    songIndex = songNumber;
    if (mediaPlayer == null)
    {
        media = songs.get(songNumber).getMedia();//
        mediaPlayer = new MediaPlayer(media);
    }
    else
    {
          mediaPlayer.stop();
          mediaPlayer.dispose();
          Media media = new Media(songs.get(songNumber).toURI().toString());
          mediaPlayer = new MediaPlayer(media);
    }
    return mediaPlayer;
}

public static void setNewSong(File songFile)
     {
    media = new Media(songFile.toURI().toString());
    if (mediaPlayer != null)
    {
        mediaPlayer.stop();
        mediaPlayer.dispose();
    }
    mediaPlayer = new MediaPlayer(media);

}
```

The getInstance() & setNewSong() method implement the singleton pattern by checking whether the mediaPlayer is null before creation. This avoids multiple of the same GlobalMediaPlayer being created at the same time which was an error. At the beginning, before Singleton was implemented, when the scene was switched a new instance, it would start causing the media players to play songs over each other.

The media playback buttons also all check for the instance to be null, this is not singleton as it is not creating a new instance of the media player after the check:

```
public static void play()
{
    if (mediaPlayer != null)
    {
        mediaPlayer.play();
        isPlaying=true;
    }
}
public static void pause()
{
    System.out.println("pause clicked");
    if (mediaPlayer != null)
    {
        mediaPlayer.pause();
        isPlaying=false;
    }
}
public static void stop()
{
    if (mediaPlayer != null)
    {
        mediaPlayer.stop();
        isPlaying=false;
    }
}
```

However, like the singleton pattern, it does stops errors from occurring when the user tries to use the multimedia buttons on the front-end without an instance yet being created. This can be due to there being no songs being uploaded to the library.

Frequently Played Songs

The main portion of the functionality for creating the frequently played songs list exists within the GlobalMediaPlayer class. The class implements a hashmap of <Song,Integer> with the integer corresponding to how many times the song has been played.

```
    private static Map<Song, Integer> playCounts = new HashMap<>
```

The main functionality contained within the GlobalMediaPlayer Class in regards to frequently played songs is:

- incrementPlaycount() method, this is invoked every time the updatemediaplayer() method (every time a song changes) is invoked in the scenecontroller. It

```
public static void incrementPlayCount(Song song)
{
    int count = playCounts.getOrDefault(song, 0);/
    playCounts.put(song, count + 1);//Update Map
}
```

increments the Song's hashmap's integer by 1 every time it's played

- removeSongFromPlaycount() method to remove the song from the playcount list. This is used when songs are removed from the directory.

```
//Method to remove song from the playcount map
public static void removeSongFromPlayCount(Song song)
{
    playCounts.remove(song);
}
```

- getTopFrequentlyPlayedSongs(int n) - integer n refers to how many hboxs are in the front-end of main.fxml to be filled. There are currently 8 hboxs in the front-end so n=8. This method arranges the songs into an descending order based on playcounts then returns this list of 8 songs to be displayed. This method is invoked when the switchToMain() method is used to switch scenes to main.fxml

```
public static List<Song> getTopFrequentlyPlayedSongs(int N)
{
    List<Song> sortedSongs = new ArrayList<>(playCounts.keySet());// new li
    sortedSongs.sort((s1, s2) -> playCounts.get(s2) - playCounts.get(s1));/

    // Return the top N songs
    return sortedSongs.subList(0, Math.min(N, sortedSongs.size()));
}
```

- saveFrequentlyPlayedList() - this is invoked when the updateMediaPlayer() method is invoked. It allows for persistently saved data of this playcounts list. The method creates a bytestream to serialise the data into a serialisable file; "play_counts.ser", when this application is exited.

```
// Method to save playCounts map to a serialised file
public static void saveFrequentlyPlayedList()
{
    try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(PLAY_COUNTS_FILE)))
    {
        oos.writeObject(playCounts);//Serialise playcounts map into play_counts.ser
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
```

- THIS USES A LAMBDA EXPRESSION WHICH

CALCULATES IF song2 - song1 is positive if it is song2 is places before song1 because it has the higher play count. Lambda expressions are used[ref]

- loadFrequentlyPlayed list - if the user runs the app again, it will deserialise “play_counts.ser” then saves the method in the playcounts hashmap. If the file doesn’t exist a new playcounts hashmap is initialised. This method is invoked on start up.

```
@SuppressWarnings("unchecked")
public static void loadFrequentlyPlayedList()
{
    try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(PLAY_COUNTS_FILE)))
    {
        playCounts = (Map<Song, Integer>) ois.readObject();//deserialise playcount map from file

    //Iteration over all the entries in the map:
        Iterator<Map.Entry<Song, Integer>> iterator = playCounts.entrySet().iterator();
        while (iterator.hasNext())
        {
            Map.Entry<Song, Integer> entry = iterator.next();
            Song song = entry.getKey();

            // Check if the song is still exists
            if (!song.isValid())
            {
                // If the song is not valid, remove it from the playCounts map
                iterator.remove();
            }
        }

    }
    catch (IOException | ClassNotFoundException e)
    {
        // If the file doesn't exist or couldn't be read, initialise an empty map
        playCounts = new HashMap<>();
    }
}
```

The decision to use the serialised format over another format such as using JSON was because serialising is straightforward and efficient in Java. It only requires turning the data into bit stream then saving it in a file. This approach is highly efficient in terms of processing speed and storage space. It comes with the downside of the serialisable file not being user friendly to read, resulting in testing of the file being hard. However, this was not a concern that came to mind when creating the application as not many users are interested in the back-end of an application.

Scenecontroller.Java

The SceneController class’s main role is to control interaction with the front-end’s components and relay a response in the back-end to control the functionality of the application.

Updating UI

The controller implements initializable so when a scene is loaded, the intialize() method is invoked:

```
    //Used to show which scene is selected to choose what to populate
    private static boolean fromLibrary = false;
    private static boolean fromPlaylists = false;
//This method is run every time the scene is switched
    @Override
    public void initialize(URL location, ResourceBundle resources)
    {
        //Playlists loaded from a serialised file, or if the file doesn'
        Map<String, List<Song>> p = loadPlaylists();
        playlists = p != null ? p : new HashMap<>();
        validatePlayLists();//Remove deleted songs

        System.out.print(songNumber);
        //Import songData if it's empty (first run)

        if(songData.isEmpty())
        {
            loadSongs();
            for (File file : songs)
            {
                Song song = new Song(file);
                if(song.isValid()) {
                    songData.add(song);
                }
            }
            GlobalMediaPlayer.loadSongs(songData);
        }

//Choosing which part of UI to populate:
        if(!fromLibrary && !fromPlaylists)
        {
            populateFrequentlyPlayedSongs();
        }

        if(fromPlaylists)
        {
            populatePlaylistButtons(this);
        }

        if(!songs.isEmpty())
        {
            GlobalMediaPlayer.getInstance(songNumber);
            //songLabel.setText(songs.get(GlobalMediaPlayer.getSongIndex
        }
        System.out.println(GlobalMediaPlayer.getSongIndex());//TEST
        songProgressBar.setStyle("-fx-accent: #397DFF;");//progress bar
        updateSongDetails();
```

This method is ran every time the scene is switched, if the song data is empty (first run) it will load in the songs from the music directory then load them into the GlobalMediaPlayer class allowing the GlobalMediaPlayer class to have access to the songs***[refer to GlobalMediaPlayer.java above].*** The context is decided from the boolean values at the top, which are set to true when the “switchTo...” methods are invoked to switch scenes. This allows the application to know which

the current FXML file is by using if statements to provide context on which components to populate. E.g -

```
 if(!fromLibrary && !fromPlaylist)
 {
     populateFrequentlyPlayedSongs();
}
```

This gives the scenecontroller the context that the initialisation is not from the library or from the playlist therefore it's from the dashboard, so initialise the frequently played songs. The same is true for the other if statements.

The songs details are updated every time the scene is initialised using the updateSongDetails() method, this method is invoked every time the updateMediaPlayer() method (every time the song changes) is invoked:

```
//Update song details
    public void updateSongDetails()
    {
        Media media = GlobalMediaPlayer.getMedia();
        if (media != null)
        {
            Song song = GlobalMediaPlayer.getSong(songNumber);

            songLabel.setText(song.getTitle());
            artistLabel.setText(song.getArtist());
            albumLabel.setText(song.getAlbum());

            //Update time and bar
            songProgressBar.setStyle("-fx-accent: #397DFF;");
            beginTimer();
            Image songimage = song.getAlbumArt();

            if (songimage != null)
            {
               songArt.setImage(songimage); // Update album art.
            }
            else
            {
                // Set default image if there's no album art.
                songArt.setImage(new Image(getClass().getResourceAsStream("/main/icons/Logo.png")));
            }
            System.out.println("Testa");
        }
    }
```

This method checks to see if there is any media playing. If there is it extracts the details of the song using the Song class's getter methods and assigning the to the respective label. The time bar is then updated using the beginTimer() method and the song art is set. The beginTimer() method takes the current time of the song over the duration to set the progress of the bar:

```java
//Method for updating the progress bar on the song
    public void beginTimer() {
        if(timer == null)
        {
            timer = new Timer();
        }
        System.out.println("Test");
        task = new TimerTask() {
            public void run() {
                Platform.runLater(() -> {//Updates UI on the JavaFX application thread
                        double current = GlobalMediaPlayer.getCurrentTime().toSeconds();
                        double end = GlobalMediaPlayer.getMedia().getDuration().toSeconds();
                        String formattedCurrent = String.format("%d:%02d", (int) current / 60, (int) current % 60);
                        String formattedEnd = String.format("%d:%02d", (int) end / 60, (int) end % 60);
                        //Update UI:
                        songLength.setText(formattedEnd);
                        timeElapsed.setText(formattedCurrent);
                        songProgressBar.setProgress(current/end);

                        //System.out.println("TEST"+current/end);
                        if(current/end == 1) { //Song is finished when current and end = 1
                            cancelTimer();
                            nextMedia();
                            beginTimer();
                        }
                    }
                );
            }
        };
        timer.scheduleAtFixedRate(task, 0, 1000);        // Schedule the task to run every second
    }
```

The method checks if the timer is currently running, if it is it'll carry on using it. It activates a task which allows for the run() method to be activated every 1000ms. Inside the TimerTask run() method, Platform.runLater is ran every second to update the songLength label, the timeElapsed label, and the songProgressBar. These three components get updated in the UI, how regularly is based on the number on the bottom line currently it's 1000ms. When current/end = 1 it means the song is over so the song cancels the timer then begins the next song then starts the progress bar again.

The updateMediaPlayer() method is invoked every time the song number changes, causing a new song to be set based on this value:

```java
//Sets song then plays:
    private void updateMediaPlayer()
    {
        GlobalMediaPlayer.setNewSong(songs.get(songNumber));//Song set to current songNumber
        GlobalMediaPlayer.incrementPlayCount(songData.get(songNumber));//For frequently played count
        GlobalMediaPlayer.saveFrequentlyPlayedList();//Save list

        System.out.println("fromLibrary = " + fromLibrary);//Test

        //Update icon in scene.
        if (fromLibrary)
        {
            playPauseImage.setImage(new Image(getClass().getResourceAsStream("/main/icons/pause.png")));
        }
        else if(fromPlaylists && playPauseImagePlaylist != null)
        {
            playPauseImagePlaylist.setImage(new Image(getClass().getResourceAsStream("/main/icons/pause.png")));
        }
        else if(playPauseImageMain != null)
        {
            playPauseImageMain.setImage(new Image(getClass().getResourceAsStream("/main/icons/pause.png")));
        }
        updateSongDetails();
        GlobalMediaPlayer.play();//Play set song
    }
```

It sets the new song instance then increments the current song's play count for the frequently played list, then saves that same frequently played list by serialising it into play_counts.ser ***[refer to GlobalMediaPlayer.java].*** The playPauseimage is changed to "pause.png" ⇒ the song details are updated ⇒ then the newly set song is played.

Music Playback Buttons

The play/pause buttons main functionality is embedded within the GlobalMediaPlayer class, however, the image for the button is controlled within the SceneController class:

```
//Play & Pause Button:
    public void playMedia()
    {
        System.out.println("play button clicked " + fromLibrary);


        if (GlobalMediaPlayer.getPlaying()) //If media player is playing
        {//Update Icons:
            if (fromLibrary)
            {
                playPauseImage.setImage(new Image(getClass().getResourceAsStream("/main/icons/playbutton.png")));
            }
            else if (fromPlaylists)
            {
                playPauseImagePlaylist.setImage(new Image(getClass().getResourceAsStream("/main/icons/playbutton.png")));
            }
            else
            {
                playPauseImageMain.setImage(new Image(getClass().getResourceAsStream("/main/icons/playbutton.png")));
            }
            GlobalMediaPlayer.pause();//If playing n button clicked, pause
        }
        else //Not playing
        {//Update Icons:
            if (fromLibrary)
            {
                playPauseImage.setImage(new Image(getClass().getResourceAsStream("/main/icons/pause.png")));
            }
            else if (fromPlaylists)
            {
                playPauseImagePlaylist.setImage(new Image(getClass().getResourceAsStream("/main/icons/pause.png")));
            }
            else
            {
                playPauseImageMain.setImage(new Image(getClass().getResourceAsStream("/main/icons/pause.png")));
            }
            beginTimer();

            if (GlobalMediaPlayer.getMedia() != null)
            {
                GlobalMediaPlayer.play();
            }
            else
            {
                updateMediaPlayer();//Both methods will play when button clicked
            }
        }
    }
```

It's basic logic is that if the media player is playing it will present the "playbutton.png" and it its not playing it shows the "pause.png". The playButtonImage ImageView's fx:id has been varied slightly across fxml files. However, if I was to redo the project it would be beneficial to reducing the lines of code by keeping the fx:id the same then using the same logic:

```
if (GlobalMediaPlayer.getPlaying())
{
    playPauseImage.setImage(new Image(getClass().getResourceAsSt
}
else
{
    playPauseImage.setImage(new Image(getClass().getResourceAsSt
}
```

Evidently, this method is much shorter now, a lot of redundancy has been removed and the functionality remains the same.

The previousMedia() and nextMedia() methods work within the same logic:

```
//Previous Song Button:
    public void previousMedia()
    {
        System.out.println("songN in previousMedia " + songNumber);

        if(fromPlaylists)
        {
            int index = tableView2.getSelectionModel().getSelectedIndex();//Current index
            if (index > 0) //Check if it's first song
            {
                index--;
            }
            else //If it's first song, loop back round
            {
                index = playListData.size() - 1;
            }
            songNumber = getSongNumberOf(playListData.get(index));//Update songNumber
            tableView2.getSelectionModel().select(index);//select in UI

        }
        else
        {
            if (songNumber > 0) //Check first song
            {
                songNumber--;
            }
            else //if it is loop back
            {
                songNumber = songs.size() - 1;
            }

        }

        updateMediaPlayer();//plays new song
```

```
//Next Song Button:
    public void nextMedia() //Same context as the previousMedia
    {
        if(fromPlaylists)
        {
            int index = tableView2.getSelectionModel().getSelectedIndex();
            if (index < playListData.size() - 1)
            {
                index++;
            } else
            {
                index = 0;
            }
            tableView2.getSelectionModel().select(index);

            songNumber = getSongNumberOf(playListData.get(index));
        }
        else
        {
            if (songNumber < songs.size() - 1)
            {
                songNumber++;
            }
            else
            {
                songNumber = 0;
            }
        }

        updateMediaPlayer();
    }
```

The methods allows for the previous & next songs to be played but the key feature is the loop. The controller checks the current index against the size of the list of songs or if it's from the playlist scene against the size of the playlist. If the index is the last song of the playlist and the nextMedia() method is invoked it will loop round to the first song so index = 0. If its the first song and the previousMedia() method is invoked then it loops round to the last song of the playlist, therefore, songs.size()-1 or playlistData.size()-1 depending on the scene context.

Songs can also be double clicked in the library or playlist tableview to be played:

```
//If song is double clicked start playing
    public void doubleClickOnTableView(MouseEvent event)
    {
        if(event.getClickCount() == 2)
        {
            System.out.println("clicked twice");
            songNumber = tableView != null ? tableView.getSelectionModel().getSelectedIndex() :
                getSongNumberOf(tableView2.getSelectionModel().getSelectedItem());
            updateMediaPlayer();
        }
    }
```

When a song is double clicked the song number of the song from the selected row will be assigned as the song number and the updateMediaPlayer() method is invoked to set the new song, update details and increment the song play count.

Upload Music Button

The copyFileToMusicFolder() and copyFile() methods allow a file to be taken in as a parameter and copied to the “music” directory:

```
    private void copyFileToMusicFolder(File file)
    {
        File dest = new File("music/" + file.getName());
        copyFile(file, dest);
        if (dest.exists()) //Tests
        {
            System.out.println("Uploaded: " + dest.getAbsolutePath());
        }
        else
        {
            System.out.println("Failed to upload: " + file.getAbsolutePath());
        }
    }

    private void copyFile(File source, File dest)
    {
        try (FileInputStream is = new FileInputStream(source); //Both streams close at same tim
            FileOutputStream os = new FileOutputStream(dest))
        {
            byte[] buffer = new byte[1024];
            int length;

            while ((length = is.read(buffer)) > 0) //Length = exact number of bytes to dest fil
            {
                os.write(buffer, 0, length);
            }
        }
        catch (IOException e) //Can't copy
        {
            e.printStackTrace();
        }
    }
```

copyFile() gives the main copying functionality by creating a source and destination stream then writing the file to the destination.

copyFileToMusicFolder() takes advantage of the copyFile() method’s abilities by taking a file to be copied as a parameter then setting “***music/+ file.getName()***” as the destination for the file.

Then the uploadMusicAction() is bound to the “Upload Music” button on the Front-End so this is invoked when the button is clicked:

```
@FXML
private void uploadMusicAction(ActionEvent event)
{
    FileChooser fileChooser = new FileChooser();

    // Set extension filter for .mp3 files and allow directories
    fileChooser.getExtensionFilters().add(new FileChooser.ExtensionFilter("MP3 files (*.mp3)", "*.mp3"));
    fileChooser.setTitle("Select Music Files or Directory");

    List<File> selectedFiles = fileChooser.showOpenMultipleDialog(null); // Enable multiple selection

    if (selectedFiles != null) //If file selected
    {
        for (File file : selectedFiles) //Every file in selected files
        {
            if (file.isDirectory()) //If Directory
            {
                File[] filesInDir = file.listFiles((dir, name) -> name.toLowerCase().endsWith(".mp3"));//E
                if (filesInDir != null) //If not empty
                {
                    for (File mp3File : filesInDir) //Every MP3 File in fileInDir
                    {
                        copyFileToMusicFolder(mp3File);//Copy them over to music folder
                    }
                }
            }
            else if (file.getName().toLowerCase().endsWith(".mp3")) //Selected is one MP3
            {
                copyFileToMusicFolder(file);//Copy over to music folder
            }
        }
    }
}
```

The invoked method opens up the file directory by initialising a FileChooser Object, then setting the extension to accept MP3 files only, the selected MP3 file will be copied to the music folder. If multiple files(or a directory) are selected then the files are copied one by one to the music folder through a for loop. If a directory is embedded within the directory then it will copy every file within that embedded directory too.***[Refer to Front-End⇒Final Product of Front-End⇒Dialog Screens⇒Upload Music Button to see what the action does]***

Scene Switch Buttons

Switching to the Dashboard (Main.FXML):

When the Dashboard button is clicked on the front-end, the switchToMain() method is invoked:

```
/Switch to Main.FXML
   public void switchToMain(ActionEvent event) throws IOException
   {
       fromPlaylists = false;
       fromLibrary = false;

       System.out.println("switchToMain");

       Parent root = FXMLLoader.load(getClass().getResource("Main.fxml"));//Load main.fxml
       stage = (Stage)((Node)event.getSource()).getScene().getWindow();//Current Stage from button click
       scene = new Scene(root);//New scene with root loaded from Main.FXML
       stage.setScene(scene);//Set new scene on current stage
       stage.show();//Show the main view

   }
```

The context is first set to show that the scene is neither from the playlist nor from the library therefore they're both false. This will be used in the initialisation method [refer to Updating UI] to invoke the populateFrequentlyPlayed() method.

The main.fxml file is loaded in ⇒ the current stage is set as the stage ⇒ the scene is set on the stage ⇒ the stage is shown. This process is how a developer would regularly switch a scene in JavaFX. The other scenes cover the same concept but the population of the table view and cells is done within the switch methods as will be seen in switchToLibrary() in the section below.

The populateFrequentlyPlayed() when invoked populates the eight hboxs displayed in Main.FXML:

```
//Populating the frequently played songs list:
    private void populateFrequentlyPlayedSongs()
    {
        System.out.println("populateFrequentlyPlayedSongs one");//TEST

        GlobalMediaPlayer.loadFrequentlyPlayedList();//load the playcount data

        List<Song> topPlayedList = GlobalMediaPlayer.getTopFrequentlyPlayedSongs(8);/


        System.out.println("topPlayedList: ");//TEST
        System.out.println("freqPlayedSongs = " + freqPlayedSongs);//TEST

        // Iterate over the list and update the UI to display the songs.
        for (int i = 0; i < topPlayedList.size(); i++)
        {//Bob the builder
            Song song = topPlayedList.get(i);

            HBox hBox = (HBox) freqPlayedSongs.getChildren().get(i);//Hbox index for
            hBox.setStyle("-fx-background-color: #397DFF;"); //same as colour as uplo

            //Updates art,song title,artist & song duration within each hbox
            ImageView imageLabel = (ImageView) hBox.getChildren().get(0);
            imageLabel.setImage(song.getAlbumArt());

            Label titleLabel = (Label) hBox.getChildren().get(1);
            titleLabel.setStyle("-fx-text-fill: white;");
            titleLabel.setText(song.getTitle());

            Label artistLabel = (Label) hBox.getChildren().get(2);
            artistLabel.setStyle("-fx-text-fill: white;");
            artistLabel.setText(song.getArtist());

            Label durationLabel = (Label) hBox.getChildren().get(3);
            durationLabel.setStyle("-fx-text-fill: white;");
            durationLabel.setText(song.getDuration());


            //Playbutton function next to each song so they can be played
            ImageView playBtn = (ImageView) hBox.getChildren().get(4);
            int finalI = i;
            playBtn.setOnMouseClicked(new EventHandler<MouseEvent>() {
                @Override
                public void handle(MouseEvent event) {
                    songNumber = getNumberOf(topPlayedList.get(finalI));
                    updateMediaPlayer();//Plays song selected
                    updateSongDetails();
                }
            });

        }

    }
    @FXML
```

The main notion of this method is to invoke the getTopFrequentlyPlayedSongs(8) method, the 8 refers to the 8 hboxs and this method returns a list of songs ordered by play count by decreasing order.

It then goes through each frequently played song to add their details to each hbox child. These hbox children can be referred to like this :

- 0 - ImageView imageLabel - used to display song art in frequently played list. However, this does not work at the moment and should be noted to be fixed in future work. The hypothesis behind it not functioning is because the image view in the song method is transient so avoids serialisation then deserialisation into the songData variable, therefore it's not present, however, this hypothesis was never tested.
- 1 - Label titleLabel - Displays the title of the song using the Song class's getter methods
- 2 - Label artistLabel - Displays the artist of the song using the same getter methods
- 3 - Label durationLabel - Displays the duration of the song
- 4 - ImageView playBtn - the image view has a play button set upon it, when this image is clicked the song number of the song in the hbox is extracted using the getNumberOf() method then updating the Media Player and song details. The updateMediaPlayer() method ***[refer to Updating UI]*** starts a song instance with this selected song then plays it. The updateSongDetails() method is used thereafter to, consequently, update the labels.

Switching to the Library (Library.FXML):

Clicking the Library button will invoke the switchToLibrary() Method:

```
//Switch to Library.FXML
    public void switchToLibrary(ActionEvent event) throws IOException
    {
        fromLibrary = true;//Context for initialisation
        fromPlaylists = false;

        System.out.println("switchToLibrary");

        FXMLLoader loader = new FXMLLoader(getClass().getResource("Library.fxml"));
        Parent root = loader.load();
        SceneController sceneController = loader.getController();
```

It works with the same premise as switchToMain(), the context is assigned at the start of the method so fromLibrary is set to true. However, it creates an instance of

FXMLLoaded to extract the controller instance using the getController method(), this controller instance is used to set up the table for the library.

Using the table getter methods from the scenecontroller class:

```
//Table Getters:
    public TableView<Song> getTableView() {
        return tableView;
    }
    public TableView<Song> getTableView2() {
        return tableView2;
    }
    public TableColumn<Song, String> getSongTable() {
        return songTable;
    }
    public TableColumn<Song, String> getArtistTable() {
        return artistTable;
    }
    public TableColumn<Song, String> getAlbumTable() {
        return albumTable;
    }
    public TableColumn<Song, String> getYearTable() {
        return yearTable;
    }
    public TableColumn<Song, String> getActionTC() {
        return actionTC;
    }
    public TableColumn<Song, String> getActionTC2() {
        return actionTC2;
    }
    public VBox getFreqPlayedSongs() {
        return freqPlayedSongs;
    }
    public FlowPane getPlaylists_container() {
        return playlists_container;
    }

    public Label getSelectedPL() {
        return selectedPL;
    }
```

These getter are used so all the table view cells can be initialised:

```
sceneController.getSongTable().setCellValueFactory(new PropertyValueFactory<>("title"));
sceneController.getArtistTable().setCellValueFactory(new PropertyValueFactory<>("artist"));
sceneController.getAlbumTable().setCellValueFactory(new PropertyValueFactory<>("album"));
sceneController.getYearTable().setCellValueFactory(new PropertyValueFactory<>("year"));

// Cell for adding to the playlist
sceneController.getActionTC().setCellFactory(new Callback<TableColumn<Song, String>, TableCell<Song, String>>()
{
    @Override
    public TableCell<Song, String> call(TableColumn<Song, String> param)
    {
        return new TableCell<Song, String>()
        {
            final Button btn = new Button("+");
            {
                btn.setStyle("-fx-background-color: green; -fx-text-fill: white; -fx-font-size: 20px; -fx-padding: 0 10 0 10; -fx-margin: 0;");
                btn.setOnAction(event -> {
                    Song song = getTableView().getItems().get(getIndex());//Get index of song to add to playlist
                    System.out.println("Button clicked for song: " + song.getTitle());
                    showAddToPlaylistDialog(song);
                });
```

This comes with the inclusion of ActionTC which is the “Add to Playlist” button ***[refer to Front-End⇒ Library].*** When this button is clicked the table view index for

that row is assigned to song value, stating the song to be added to playlist. Then the showAddToPlaylistDialog() method is invoked, showing the user the Playlist Dialog Screen ***[Refer To Front-End⇒Dialog Screens⇒Playlist & SceneController.java⇒Playlist Functionality]*** where the user will then choose which playlist to send the Song to.

The updateItem method is used to check if the cell in the table is empty:

```
            @Override
            public void updateItem(String item, boolean empty) {
                super.updateItem(item, empty);
                if (empty)
                {
                    setGraphic(null);
                    setText(null);
                }
                else
                {
                    setGraphic(btn);
                    setText(null);
                }
            }
        };
    }
});
```

If the cell is empty, the application will make sure no graphics are set to it. If there is a cell present then the btn variable that was defined above will be used.

The songData is then extracted then set upon the table view:

```
        //Update table view with songData
        songData = sceneController.songData;
        sceneController.getTableView().setItems(songData);

        if (!songData.isEmpty()) {
            Song playingSong = songData.get(songNumber);
            sceneController.getTableView().getSelectionModel().select(playingSo
        }

        System.out.println(songNumber);

        //Set and show new  scene:
        stage = (Stage)((Node)event.getSource()).getScene().getWindow();
        scene = new Scene(root);
        stage.setScene(scene);
        stage.show();
    }
```

The if statement, gets the currently playing song then highlights it in the table view. The rest of the method is identical to how switchToMain() operates to show the scene.

Switching to the Playlists (Playlist.FXML):

The switchToPlaylist() method which is invoked by the Playlist button works the same as the two switch methods above, however, it requires initialisation of another cell ActionTC2:

```
sceneController.getActionTC2().setCellFactory(new Callback<TableColumn<Song, String>, TableCell<Song, String>>() {
    @Override
    public TableCell<Song, String> call(TableColumn<Song, String> param) {
        return new TableCell<Song, String>() {
            final Button btn = new Button("-");

            {
                btn.setStyle("-fx-background-color: red; -fx-text-fill: white; -fx-font-size: 20px; -fx-padding: 0 10 0 10; -fx-margin: 0;");
                btn.setOnAction(event -> {
                    Song song = getTableView().getItems().get(getIndex());
                    try {
                        deleteFromPlayList(song, sceneController, event);
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                });
            }

            @Override
            public void updateItem(String item, boolean empty) {
                super.updateItem(item, empty);
                if (empty) {
                    setGraphic(null);
                    setText(null);
                } else {
                    setGraphic(btn);
                    setText(null);
                }
            }
        };
    }
});
```

This is the “Remove” button [refer to Front-End⇒Playlists], the initialisation of this button works with the same concept as ActionTC but it uses the deleteFromPlaylist() method instead to remove the song from the playlist.[refer to Playlist Functionality]

The context of the player is also set up like this:

```
fromLibrary = false;
fromPlaylist = true;
```

This invokes the populatePlaylistButtons() method in the initialisation method ***[refer to Updating UI].***

Playlist Functionality

The populatePlaylistButtons() method initialises all the playlist buttons then starts playing the first playlist automatically:

```
private void populatePlaylistButtons(SceneController sceneController)
{
    getPlaylists_container().getChildren().clear();

    if (playlists != null) //If playlists exist
    {
        for (String playlistName : playlists.keySet()) //for each playlist name
        {
            Button playlistButton = new Button(playlistName);//Create new button with playlist
            playlistButton.setStyle("-fx-background-color:  #397DFF; -fx-text-fill: white;");
            playlistButton.setOnAction(event -> {
                // Selecting the playlist
                System.out.println("Selected playlist: " + playlistName);
                viewSongsOf(sceneController, playlists.get(playlistName));
                selectedPL.setText(playlistName);
            });

            playlists_container.getChildren().add(playlistButton);// Add the newly created butt
        }
        // Automatically start the first playlist button if exists
        if(!playlists_container.getChildren().isEmpty())
        {
            Button firstButtonPlayList = (Button) playlists_container.getChildren().get(0);

            if(firstButtonPlayList != null)
            {
                firstButtonPlayList.fire();
            }
        }

    }
}
```

This method check if playlists exist and for each playlist creates a new button within the FlowPane container ***[refer to Front-End⇒Playlist.FXML].*** When a playlist button is clicked the set action event is to invoke the viewSongsOf() method which populates the song within the playlist. The label text above the tableview is set to show the name of the currently playing playlist.

The viewSongsOf() method populates the table view with the selected songs in a subset within the playlist:

```
/Display songs within a playlist
   private void viewSongsOf(SceneController sceneController, List<Song> songs)
   {
       if(!songs.isEmpty()) //If playlist isn't empty
       {
           playListData.setAll(songs);//set pldata to selected pl
           sceneController.getTableView2().setItems(playListData);
           sceneController.getTableView2().getSelectionModel().selectFirst();//Select first
           songNumber = getSongNumberOf(tableView2.getSelectionModel().getSelectedItem());//
           updateMediaPlayer();
           updateSongDetails();
           //GlobalMediaPlayer.pause();//Player always plays after update so pause
       }
       else
       {
           playlists.clear();
           sceneController.getTableView2().setItems(playListData);
           GlobalMediaPlayer.pause();
       }
   }
```

The thesis behind this method is that it takes the subset of songs then displays them in the tableview. The first songs in this table view is selected and it's index is set as the current song number. The updateMediaPlayer() method starts playing the currently selected song which is the first song. Furthermore, when the scene is switched to the playlists the first song of the latest playlist will start playing automatically. If the playlists are empty, it clears all playlists then clears and resets the table view.

The playlist is loaded and saved with the same premises as the GlobalMediaPlayer class. The playlist data is serialised into "playlists.ser" using the savePlaylists() method, so that the playlists are kept persistent. Then when the application is run again the loadPlaylists() methods is ran so that "playlists.ser" is deserialised and the data reassigned to the playlists variable:

```java
//The serialisable file "playlists.ser" is deserialised and contains playlist data, persistently
    public Map<String, List<Song>> loadPlaylists()
    {
        Map<String, List<Song>> playlists = null;
        try (FileInputStream fileIn = new FileInputStream("playlists.ser");
             ObjectInputStream objectIn = new ObjectInputStream(fileIn))
        {
            playlists = (Map<String, List<Song>>) objectIn.readObject();
        }
        catch (IOException | ClassNotFoundException e) {
            //e.printStackTrace();
            System.out.println("playlists file is not found");
        }
        return playlists;
    }

//Takes playlist data serialises it into playlists.ser so that it is saved for next run
    private void savePlaylists()
    {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("playlists.ser")))
        {
            oos.writeObject(playlists);
            System.out.println("Playlists saved successfully.");
        }
        catch (IOException e)
        {
            e.printStackTrace();
            System.out.println("Error saving playlists: " + e.getMessage());
        }
    }
```

The concept is the same: the savePlaylists() method turns the playlists data into a byte stream to save into the serialised file like saveFrequentlyPlayedList() method does, the loadPlaylists() method works a lot simpler than loadFrequentlyPlayedList() as it doesn't implement an iterator, it's process only requires deserialising the file. This is because the playlists are validated in the validatePlaylists() method instead:

```java
//Remove songs that no longer exist in playlist
    public void validatePlayLists()
    {
        Iterator<Map.Entry<String, List<Song>>> iterator = playlists.entrySet().iterator();//to go throu
        while (iterator.hasNext())
        {
            Map.Entry<String, List<Song>> entry = iterator.next();
            List<Song> songs = entry.getValue();
            List<Song> validSongs = new ArrayList<>();

            // Iterate over songs in the playlist and check if they are valid
            for (Song song : songs)
            {
                if (song.isValid()) //if Song is valid
                {
                    validSongs.add(song);//Add to list of valid songs
                }

            }

            if (validSongs.isEmpty())
            {
                iterator.remove();//If no valid songs remove the playlist
            }
            else
            {
                playlists.put(entry.getKey(), validSongs);// Update the playlist with only valid songs
            }
        }
    }
```

This method uses the same concept as the loadFrequentlyPlayedList() method, it implements an iterator to check whether each song still exists when the application is loaded. It takes the songs that still exist within a playlist and puts them in an ArrayList, once it's gone through all the songs in the playlist it appends all the songs back to the playlist apart from ones which were removed or don't exist. If the list of valid songs is empty the playlist is removed entirely. This method is invoked every time a scene is initialised.

ActionTC - Adding Songs To Playlists (Used in switchToLibrary() & switchToPlaylist())

The showAddToPlaylistDialog() is invoked when the "add to playlist" actionTC buttons are clicked:

```java
//Add to Playlist Window
    private void showAddToPlaylistDialog(Song song)
    {
        Stage dialogStage = new Stage();
        dialogStage.initModality(Modality.APPLICATION_MODAL);//Block interaction with main window until
        dialogStage.setTitle("Add Song to Playlist");

        ComboBox<String> playlistComboBox = new ComboBox<>();

        ObservableList<String> playlistNames = FXCollections.observableArrayList(playlists.keySet());
        playlistComboBox.setItems(playlistNames);//Existing playlists

        TextField newPlaylistTextField = new TextField();//User Input
        newPlaylistTextField.setPromptText("Enter new playlist name");

        Button addButton = new Button("Add");
        addButton.setOnAction(event -> {
            String selectedPlaylist = playlistComboBox.getSelectionModel().getSelectedItem();
            String newPlaylist = newPlaylistTextField.getText().trim();

            if (selectedPlaylist != null || !newPlaylist.isEmpty())
            {
                String playlistName = newPlaylist.isEmpty() ? selectedPlaylist : newPlaylist;
                System.out.println("Adding song to playlist: " + playlistName);

                // Get the playlist from the map or create a new one if it doesn't exist
                List<Song> playlist = playlists.getOrDefault(playlistName, new ArrayList<>());
                // Add the song to the playlist
                playlist.add(song);
                // Update the map with the modified playlist
                playlists.put(playlistName, playlist);

                // Save then close:
                savePlaylists();
                dialogStage.close();
            }
        });

        //UI components in a VBox:
        VBox vBox = new VBox(10);
        vBox.setPadding(new Insets(10));

        Label playlistLabel = new Label("Select Playlist:");
        HBox playlistBox = new HBox(10);
        playlistBox.getChildren().addAll(playlistLabel, playlistComboBox);

        vBox.getChildren().addAll(playlistBox, newPlaylistTextField, addButton);

        Scene dialogScene = new Scene(vBox);
        dialogStage.setWidth(300);
        dialogStage.setScene(dialogScene);
        dialogStage.showAndWait();
    }
```

Modality:

The method implements a dialog stage, which blocks all interaction with the main window until the dialog box is closed. This is done through modality: there are different types of modality for this application we used “application modality”- which blocks interaction with the any application window- but there is also “none” - user can interact with every window - and “Window Modality” - blocks interaction with the owner window (useful with applications with many different windows but because RetroSonic only has one window it ends up having the same functionality as Application Modality).

Playlist Selection:

The ComboBox is initialised to show the existing playlists, allowing the user to select anyone of these playlists to add to.

The ObservableList holds the keys of the playlists map (which contains playlist names as keys and lists of songs as values) then populates the ComboBox

The TextField allows users to enter a new playlist name. It has placeholder text "Enter new playlist name" to guide users.

The Button labeled "Add" is set up with an event handler for its action event. When clicked, it performs several functions:

- It checks if a playlist is selected from the ComboBox or if a new playlist name is entered in the TextField
- It determines which playlist name to use - if new playlist TextField is empty use selected playlist name, if not use new playlist name
- It fetches the playlist from the playlists map using the chosen name or initialises a new list if the playlist does not exist.
- It adds the song to the specified playlist.
- It updates the playlists map with the modified playlist list.

UI Layout [Refer to Front-End⇒Final Product of Front-End⇒Dialog Screens⇒Playlists]

The VBox layout is used to arrange the UI components vertically with a spacing of 10 pixels. A Label and the ComboBox are wrapped together in an HBox for horizontal arrangement.These components are added to the VBox, which also includes the TextField and the Button.

Scene and Window Setup

The VBox is set as the root node of a new Scene, which is then set on the dialogStage.

dialogStage.showAndWait() is called to display the window and block any further user action on other application windows until this dialog is closed.

ActionTC2 - Used for removing songs from a playlist (Used in switchToPlaylist())

The deleteFromPlaylist() is invoked when the “Remove” actionTC2 button is clicked:

```
private void deleteFromPlayList(Song song, SceneController sceneController, ActionEvent event) throws IOException {
    String plName = sceneController.getSelectedPL().getText();//Current playlist name
    TableView<Song> tv = sceneController.getTableView2();// Table view for songs in the pl
    List<Song> pl = playlists.get(plName);//List of songs in the pl
    if(pl.size() == 1) //If playlist = 1 song
    {
        playlists.remove(plName);//Delete this playlist
        GlobalMediaPlayer.pause();
        //Find the button and delete it from the UI:
        Node node = null;
        for (Node child : sceneController.getPlaylists_container().getChildren())
        {
            if(((Button) child).getText() == plName)
            {
                node = child;
                break;
            }
        }
        sceneController.getPlaylists_container().getChildren().remove(node);
        //Update Changes and Switch to another existing playlist
        savePlaylists();
        switchToPlaylist(event);
        GlobalMediaPlayer.pause();

    }
    else //If playlist contains more than one song just remove the song
    {
        pl.remove(song);
        playlists.put(plName, pl);
    }
    savePlaylists();
    System.out.println(song.getTitle() + " is deleted from " + plName);
/Update table view:
    if(tv.getSelectionModel().getSelectedItem() != null && tv.getSelectionModel().getSelectedItem().equals(song))   // Check if c
    {
        int index = tv.getSelectionModel().getSelectedIndex();//index of removed song
        // Determine new index. If the removed song was not the last one, move to the next song;
        // otherwise, wrap around and select the first song in the playlist.
        if (index < pl.size() - 1)
        {
            index++;
        } else
        {
            index = 0;
        }
        tv.getSelectionModel().select(index);//Select song in tableview

        pl = playlists.get(plName);

        songNumber = getSongNumberOf(pl.get(index));//Update songNumber to reflect newly selected song
        updateMediaPlayer();//Play new song
    }

    // Update the TableView's items to reflect the current state of the playlist.
    if (pl != null)
    {
      // Check if playlist exists (it should not be null after deletion unless it was the only playlist and got removed)
        playListData.setAll(pl);// Update playlistData that backs the TableView with the modified playlist.
        tv.setItems(playListData);// Set the modified list of songs as the new item list for the TableView.
    }

}
```

This method begins by identifying the current playlist and the song to be removed. It fetches the playlist name and the respective TableView containing the songs from the SceneController, which manages the user interface interactions.

Playlist and Song Management

The method handles different scenarios based on the playlist's content:

- **Single Song Playlists:** If the playlist contains only one song, the entire playlist is removed from the application's playlist map. This also involves updating the

user interface to reflect the removal, handling the deletion of UI elements like buttons.

- **Multiple Song Playlists:** If more than one song exists, only the selected song is removed from the list. The playlists map is then updated to reflect this change.

UI and Playback Updates

Upon successfully updating the playlist data:

- **Table View Update -** If the deleted song was the currently selected song in the UI, the selection is moved to the next song, or loops back to the first song if the deleted song was the last in the list.
- **Playback Adjustment -** If necessary, the media player's state is updated; either pausing playback if the playlist is removed or starting the next song.

Persistent Data Management

After modifying the playlist, the savePlaylists() method is called to serialise the updated playlists map to persistent storage, ensuring that changes persist across application sessions. [refer to Figure Above]

Final Operations:

The method concludes by ensuring that any changes are not only reflected in the UI but also in the back-end data, maintaining integrity and consistency of the application's state.

Editing Metadata

The showEditSongMetadataDialog() is invoked when the settingButton is clicked:

```
    private void showEditSongMetadataDialog() {
        Song selectedSong = GlobalMediaPlayer.getSong(songNumber);

        if (selectedSong == null) {
            System.out.println("No song selected.");
            return;
        }

        Stage dialogStage = new Stage();
        dialogStage.initModality(Modality.APPLICATION_MODAL);
        dialogStage.setTitle("Edit Song Metadata");

        GridPane grid = new GridPane();
        grid.setHgap(10);
        grid.setVgap(10);
        grid.setPadding(new Insets(20, 150, 10, 10));

        TextField titleField = new TextField(selectedSong.getTitle());
        TextField artistField = new TextField(selectedSong.getArtist());
        TextField albumField = new TextField(selectedSong.getAlbum());
        TextField yearField = new TextField(selectedSong.getYear());

        grid.add(new Label("Title:"), 0, 0);
        grid.add(titleField, 3, 0);
        grid.add(new Label("Artist:"), 0, 1);
        grid.add(artistField, 3, 1);
        grid.add(new Label("Album:"), 0, 2);
        grid.add(albumField, 3, 2);
        grid.add(new Label("Year:"), 0, 3);
        grid.add(yearField, 3, 3);

        Button doneButton = new Button("Done");
        doneButton.setOnAction(event -> {
            // Apply changes and close dialog
            try {
                MP3TagWriter.setTitle(selectedSong, titleField.getText());
            } catch (NotSupportedException | IOException e) {

                System.out.println("Song contains Obselete Frames, Cannot be Saved!");
            }
            try {
                MP3TagWriter.setArtist(selectedSong, artistField.getText());
            } catch (NotSupportedException | IOException e) {
                System.out.println("Song contains Obselete Frames, Cannot be Saved!");
            }
            try {
                MP3TagWriter.setAlbum(selectedSong, albumField.getText());
            } catch (NotSupportedException | IOException e) {
                System.out.println("Song contains Obselete Frames, Cannot be Saved!");
            }
            try {
                MP3TagWriter.setYear(selectedSong, yearField.getText());
            } catch (NotSupportedException | IOException e) {
                System.out.println("Song contains Obselete Frames, Cannot be Saved!");
            }
            dialogStage.close();
        });

        grid.add(doneButton, 1, 4);

        Scene dialogScene = new Scene(grid, 500, 250);
        dialogStage.setScene(dialogScene);
        dialogStage.showAndWait();
    }
}
```

This method works in-hand with the MP3TagWriter class which handles all the editing of metadata in the back-end. This method provides the dialog screen with four text fields displaying the current title, artist, album & year tags which can be edited to provide the MP3TagWriter class with fields to edit.***[Refer to Front-End⇒Finished Product⇒Dialog Screens⇒ Edit Metadata]***

Modality

This method sets up a modal dialog window, which freezes all other user interactions with the application until the metadata editing is complete. This is accomplished through Modality.APPLICATION_MODAL, which is used in the showAddToPlaylistDialog(). The same logic applies here.

UI Layout and User Input

A GridPane layout is used to organise labels and text fields linearly for editing the song's title, artist, album, and year. These fields are pre-populated with the current metadata values of the selected song, allowing users to see and modify the existing information easily.

The grid is structured with consistent gaps and padding to ensure the dialog is visually appealing and easy to use. Each metadata attribute (title, artist, album, year) is paired with a corresponding TextField where users can input new values.

Applying Changes:

The 'Done' button at the bottom of the dialog triggers the update process. This button is wired to an event handler that attempt to write the updated metadata back to the song file using the MP3TagWriter class methods. If the metadata contains obsolete frames that cannot be saved, it catches exceptions.

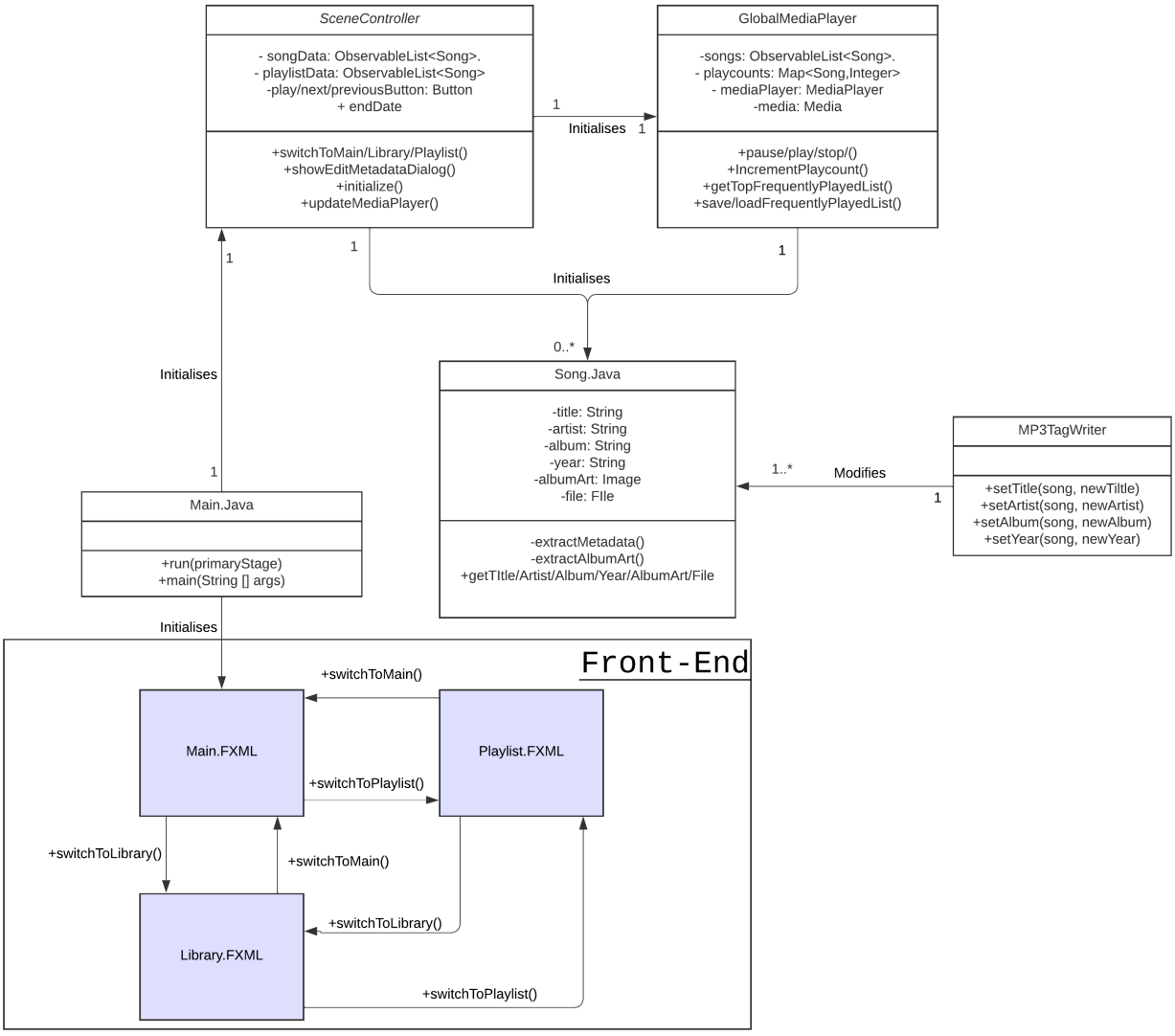
This is used incase of this error:

```
com.mpatric.mp3agic.NotSupportedException: Packing Obselete frames is not supported
        at com.mpatric.mp3agic.ID3v2ObseleteFrame.packFrame(ID3v2ObseleteFrame.java:32)
        at com.mpatric.mp3agic.ID3v2Frame.toBytes(ID3v2Frame.java:83)
        at com.mpatric.mp3agic.AbstractID3v2Tag.packSpecifiedFrames(AbstractID3v2Tag.java:275)
        at com.mpatric.mp3agic.AbstractID3v2Tag.packFrames(AbstractID3v2Tag.java:261)
        at com.mpatric.mp3agic.AbstractID3v2Tag.packTag(AbstractID3v2Tag.java:227)
        at com.mpatric.mp3agic.AbstractID3v2Tag.toBytes(AbstractID3v2Tag.java:218)
        at com.mpatric.mp3agic.Mp3File.save(Mp3File.java:450)
        at application.MP3TagWriter.saveMp3File(MP3TagWriter.java:69)
        at application.MP3TagWriter.setYear(MP3TagWriter.java:52)
        at application.SceneController.lambda$3(SceneController.java:1085)
```

This error occurs only with certain MP3 files, usually those which haven't been formatted correctly or contain unsupported tags. In these cases like these, this catch exception is used.

UML-Design

This UML design was created after the application was created. Thus, since doing this project the importance of a good back-end design was reinstated in my head due to seeing how it affects the development time of a project.



Evaluating The Design

The scenecontroller is used as the controller for all three fxml files ***[refer to Front-End]***. This is a SOLID design flaw destined for this application due to a lack of experience in building JavaFX applications. Instead, using three controllers, one per scene, it would have helped the application's implementation process tremendously due to making the logic separate for each part of the application. This is error in judgement, caused bugs to be hard to find due to a lack of modularity, having to look through lots of code to find errors. I remember attending COMP208-App Development and using the approach of one controller on each scene if the application was particularly complex like RetroSonic is.

This approach of one controller per scene would lead to a single responsibility for each controller; main.fxml controller would focus on populating frequently played songs, while library.fxml controller would focus on populating the library & playlist.fxml's controller would focus on populating playlists. Resulting in code that abides by the SOLID principle of Single Responsibility taught in COMP315-Software Engineering 2.

Currently, the controller has all three responsibilities but that's not to say it doesn't also have its benefits, the fxml files, due to being built upon on another's templates ***[refer to Front-End]*** share a lot of similar components - sidebar buttons, multimedia buttons, songs art & labels. If the application would abide by SOLID principles it would lead to lots of redundancy. For example, all three controllers have a play/pause, next or previous button, this means all three controllers would have to handle the scene's media control buttons using the same code.

Albeit, this code should have been shortened by handling most the logic in the GlobalMediaPlayer class so that each controller only has to use methods like this:

```
public void playMedia()
{
GlobalMediaPlayer.play()
}
```

The same for the settings button; the logic should be moved to the MP3TagWriter class so that if the controllers were separated, the dialog box from the

showEditMetaDataDialog() method doesn’t have get implemented three times across each controller instead this can be bind to the settings button:

```
public void showEditMetadataDialog()
{
MP3TagWriter.showSettingsDialogBox()
}
```

Overall, this shortens the amount of redundant code, if three controllers were to be used.

Testing & Evaluation-test song limit too

Log Testing

Most of the testing in this project was done through small print statements; through the console to check when methods are being invoked or if they’re being invoked at all:

Class	Method & Log	Why it’s used
GlobalMediaPlayer	pause() - System.out.print(”pause button clicked”)	Checking if fx:id is connected properly
SceneController	initialize() - System.out.print(songNumber);	Check what the songNumber is when switching scenes and initialising, was useful when having errors with the correct song showing up when switching scenes
SceneController	beginTimer() - System.out.println("Test");	Check if the song progress bar is being initialised. Was having a problem with the progress bar appearing so initialised this to see if this method was being invoked

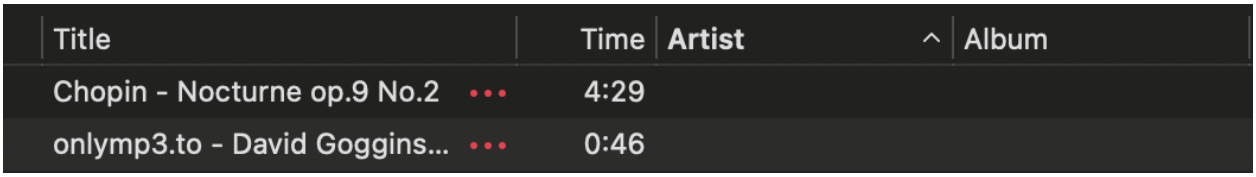
Console logs like these are planted throughout the SceneController to express variables at certain points of the applications to make sure the application is

functioning correctly.

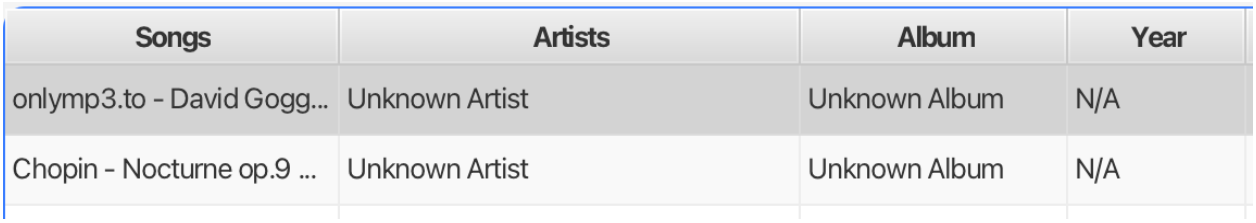
Testing Metadata Editing Functionality Works in Other Libraries

With a clear run of the application, I will upload 2 songs to the library, which are in need of tag altering.

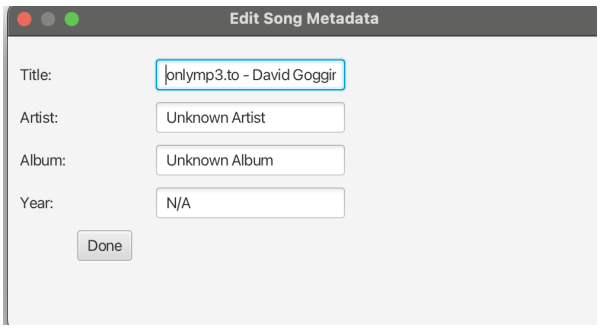
Here are the pre-edited songs in an iTunes library:

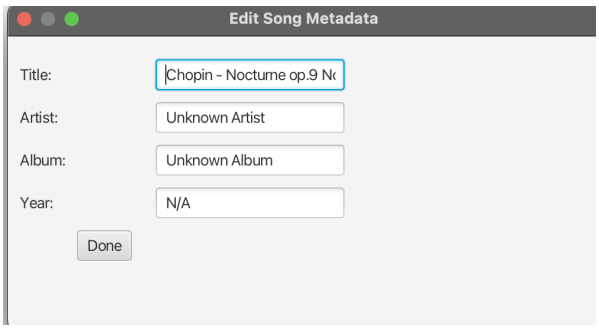


Here are the pre-edited songs in the RetroSonic library:

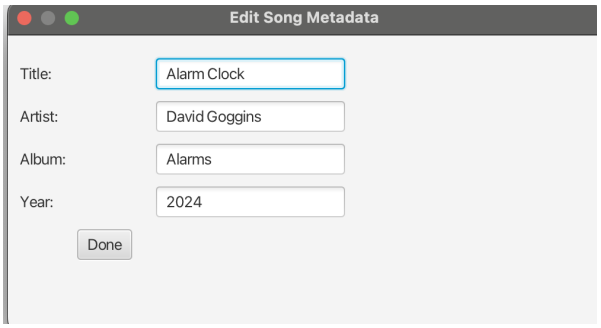


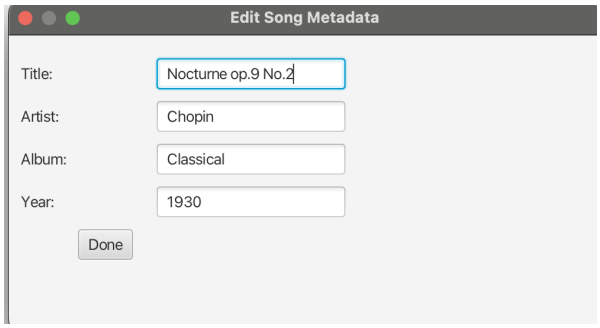
These two songs will be selected and the settings button will be clicked to reveal edit metadata dialog box.





The metadata tags of these songs will now be adjusted to the correct tags.





Here are the songs now with the edited tags in the RetroSonic library:

Songs	Artists	Album	Year
Alarm Clock	David Goggins	Alarms	2024
Nocturne op.9 No.2	Chopin	Classical	1930

When we export the songs out of the application, the tags are permanently edited so if we view these songs now in the iTunes library again:

Title	Time	Artist	Album
Nocturne op.9 No.2 •••	4:29	Chopin	Classical
Alarm Clock •••	0:46	David Goggins	Alarms

As expressed in the figure above, the metadata tag writing feature of this application works correctly, making permanent tag changes onto songs.

Project Ethics

Conclusion & Future Work

To conclude, the RetroSonic Music Player project represents a significant step forward in the personal management and playback of MP3 files, offering a user-friendly interface for song playback and metadata editing. By leveraging the

powerful JavaFX framework and integrating the MP3agic library for metadata manipulation, this project successfully delivers a standalone music management solution that caters to the specific needs of users who require a more hands-on approach to their music collections.

Throughout the development process, from requirement analysis to implementation and testing, the project adhered to a rigorous design and development methodology. The user-centric design, highlighted by the Simplified User Interface (SUI), ensures that even individuals with minimal technological expertise can navigate and utilise the RetroSonic Music Player with ease. The application's ability to edit and manage metadata directly within the UI adds a layer of functionality that sets it apart from traditional music players like iTunes and Windows Media Player, which separate media playback and metadata management into distinct functionalities.

However, the project is not without its areas for improvement and future development:

Refactoring - As explained in the Evaluating Scenecontroller Design Section, the current implementation utilises a single controller for multiple views, which, while functional, complicates the codebase and deviates from best practices in software architecture. Future iterations should aim to refactor the code into a more modular approach, using separate controllers for different functionalities to enhance maintainability and scalability.

Enhanced Metadata Support - Expanding the range of editable metadata fields and supporting more file formats would make RetroSonic a more versatile tool. MP3agic has many more built-in setter methods, which can edit fields such as album art & genre. For example another method that could be add:

```
public static void setGenre(Song song, String newGenre) throws N
        try
        {
            Mp3File mp3file = new Mp3File(song.getFilePath());
            ID3v2 tag = getOrCreateId3v2Tag(mp3file);
            tag.setGenre(newYear);
            saveMp3File(mp3file, song.getFilePath(), "temp_genre
        }
```

User Feedback & Bugs Fixes - Continued user testing and feedback are essential to refine the user interface and functionality. Implementing the feedback given about the application could provide ongoing insights into user needs and preferences, driving iterative improvements.

Mobile Compatibility - Extending the application to mobile platforms could significantly increase its accessibility and utility, allowing users to manage their music libraries on the go this would require scalable cloud servers to hold individual libraries and playlists. To accomplish this, I would use JavaFXPorts which has the ability to port javaFX applications to iPhone, iPad & Android Phones/Tablets.

Cloud Integration - Introducing cloud storage options would enable users to back up their music libraries and metadata edits online, ensuring data integrity and availability across multiple devices.

Shuffle play and Remove Songs Feature:

With the addition of these features, RetroSonic would be equipped to rival leading music players developed by major corporations.

BCS Criteria & Self-Reflection

This project is required to satisfy the BCS requirements for my degree program, below are the six criteria required:

1. An ability to apply practical and analytical skills gained during the degree programme.
2. Innovation and/or creativity.
3. Synthesis of information, ideas and practices to provide a quality solution together with an evaluation of that solution.
4. That your project meets a real need in a wider context.
5. An ability to self-manage a significant piece of work.
6. Critical self-evaluation of the process.

This is how my project satisfies these criteria below:

1. Practical and analytical skills have been showcased by the architecture of my music player, the programming knowledge I have polished over my degree has been displayed as well as software development methodologies learnt in my modules, which have been referenced in this report. I have been able to demonstrate I am able to use different libraries and frameworks which are new to me (JavaFX & MP3agic) and mastering them to implement more efficient algorithms. My analytical skills have be conveyed through in-depth analysis of audio-playback mechanisms, efficient library management and advanced playlist manipulation algorithms. Thus, being able to alter my code to incorporate new features and analyse which lines of code can be excluded or altered to speed up my algorithms.
2. Innovation and creativity has been embodied in the User Interface, being built on scene builder, which aims to be minimalistic and intuitive, eliminating "clutter features" which are found on lot of mainstream music players. My creativity will be on display with themes and layout of the User Interface.
3. My project synthesises the requirement information taken from my survey to see what features users find necessary within a music player. I have considered how users interact with the UI and apply best practices in the User Experience (UX). Software Engineering practices gained from my modules, such as modular coding, version control and AGILE development methodologies, have been implemented to create robust, high-quality code.
4. The project aims to address the need for a music player that combines music playback with the ability to edit metadata of songs, while also providing a streamlined Simplified User Interface that focuses on these core features. This means users that may not be as computer literate can navigate the User Interface without issue. User surveys have been conducted to validate the requirements for the minimalist music player and gather feedback on desired features.
5. Self-management has been crucial in overseeing the various stages of development, including project planning, design, coding, and testing. Project management tools like Gantt charts have been utilised to track progress and followed strictly to meet milestones efficiently. This allowed the project to be completed ahead of schedule to make time for bug fixes, creation of the video and writing of the dissertation. I believe the Gantt Chart's use was extremely

understated in this report, it was vital in being able to keep up with not only the implementation of the project but also the other assessments such as the project video and this report. I recommend this to developers who find it hard to keep up with deadlines, as splitting the work up into sections and giving these sections deadlines instead works a lot more efficiently, as it makes it easier to hit checkpoints more regularly in your work. Overall, this has been instrumental in ensuring the timely and enhanced completion of my project.

6. Critical self-evaluation has involved regular code reviews, testing and debugging. It has also encompassed addressing challenges such as code/design enhancements and improvements within this report, which can been illustrated when reading this report, I have been extremely critical on my mistakes and making sure they're known to the reader so that they do not make the same errors & I have also provided the reader with solutions for what I would do if I had to do this project from scratch again, so that each of my errors has a given solution within the report. For example, this could be seen in the Evaluating SceneController Design section where I discuss how the use of three controllers would have made the controller better abide to the Single Responsibility, SOLID principles that I learnt in about in COMP319-Software Engineering 2. I discuss my reasons for not doing this was due to my inexperience and if I remembered the work I did in COMP208-App Development with my Swift Applications, I wouldn't have made the same mistakes as I have experience with using a controller per scene. I have also discussed the way I would implement 3 controllers to reduce the redundancy of code caused by repetitive implementation of the multimedia buttons which improves modularity and decreases coupling of code. I have also discussed future features that I didn't prioritise in implementing in the Conclusion & Future Work section, while also portraying how they can be implemented so a reader wanting to make a music app can implement the extra methods for themselves. Therefore, I believe I have been extremely critical of myself during the writing of this report & it's apparent now how I could have implemented my work better.

References

https://gluonhq.com/products/mobile/javafxports/

Appendices