Richard Jones

A Level Computer Science Coursework

Music Centre Program

Centre Number – 52423

Candidate Number - 2389

Module - H446/03~04

Authors : Richard Jones

Date : 7 February 2017

Version : V0.1

E-mail : richardj99@sky.com

Contents

1 Analysis

1.1 Suitability for Solubility by computational Methods

1.1.1 Description of Relevant Features of problem

From the late 1800s, music has been able to be recorded and distributed through many different media. For most of the 20th century, music was mainly released on either vinyl or cassette formats. However, as computers began to become more and more commonplace in households and society as a whole, digital solutions for storing recorded music have been released. With this advancement, the music needs to be organised (similar to a music library of physical music). A physical music library is normally sorted by a category, such as the record's artist name, album name or song name. Usually this order is done by hand and is a laborious task. This organisation is done to allow people to easily and quickly navigate their library in search for a specific record. Alongside their library, music listeners also keep a music player of some kind. This could be anything from a vinyl record turntable to a CD player. In order to cater for this feature of the problem, I will need to make sure that the solution can play this music.

However, the idea of a music player program is no new concept. Many other companies have managed to create monumentally successful programs that organise music libraries and play music. Later, I will conduct research into similar solutions to gain a further understanding of how I can make my proposed solution more complex and relatable to consumer requirements.

I have been approached by Samuel Barrett, one of my fellow students, who expressed his own opinions about a music player program. He felt that many of the other music playing programs on the market today have become too complex in hopes to achieve a wider target market. In doing so, the programs have become too confusing to use and are becoming over encumbering on the machines that they are being run on. From this conversation, I decided to aim to create a solution that is lightweight and simple in nature. I also made Samuel Barrett one of my end users for the solution and a stakeholder in the finished product.

Though I am taking ideas from several previous solutions to this problem. It is important to understand that I do not want this program to be a main competitor in this market. This solution is not meant to be a fully official and profitable attempt, it is just a prototyped solution that will aim to solve the basic problems mentioned earlier.

1.1.2 Why is this problem Suitable for solution by computational Methods?

The solution can be solved by computational methods for a multitude of reasons:

First, the library organisation. Using a database and tag reading, information about the songs (including their artist, and album) can be stored. Then, using a selection of methods, which are yet to be decided on (such as whether to use search queries or iterative table searching), a view similar to a physical music library can be customised. In fact, using this method, the library can be improved upon. If somebody wanted to change the view of their library, say from sorted by artist to sorted by album, they would have to physically reorder their records which could be physically strenuous and time consuming. Using a computer, the exact same task can be done by clicking a button.

Second, searching for a song, album or artist would be even quicker. In a physical music library, the user would have to manually scan the library for the record. While this could be quickened with an index, it would not reach the speed that it could be searched for using a computer. Using either an iterative search function or an SQL query, the record could be found in a matter or milliseconds.

Third, music can be played using computational methods. Audio files exist for computers and can be played very easily. Many operating systems even come built in with integrated music players. By simply recording the location of each music file, the file can be played instantly if the correct codec is used. This would be very efficient compared to the physical solution, where

the record would have to be taken from the library and manually placed in the CD player or turntable.

In conclusion, many improvements can be made through solving the problem by computational methods. Making the overall task of listening and organising music more enjoyable and less strenuous to the user.

1.2 Stakeholder Management

1.2.1 Who are the stakeholders and how were they chosen?

There could be many potential stakeholders for this solution as most of the population have a mutual enjoyment of music and would be interested in the media player program described.

Many members of my tutor group have expressed a deep interest in music and would be suitable stakeholders in this project. As I have already mentioned, Samuel Barrett has already expressed a deep interest in my program, and hopes to see it to be a successful, yet lightweight solution to the problem. I aim to choose stakeholders who are genuinely interested in the program's wellbeing and who will give detailed yet constructive criticism to ensure its success.

To collate as wide a target market as possible, I have randomly selected six students to be my stakeholders. These people will originate from my tutor group, my school's staff and my family. They will express their wants and needs out of a solution to this problem and will be consulted throughout the design, development and testing process of this program.

1.2.2 How will they make use of the proposed solution?

Several studies (and my own personal experience) have shown that people tend to listen to music while they are working and relaxing, either at home or at a workplace. I aim for my solution to be used in these situations.

The solution will be in the form of an application that people can keep open on their desktop, allowing them to concentrate on work or relax while using the solution.

Furthermore, I have conducted several interviews with the stakeholders to create a more specific picture of what my solution needs to satisfy their needs. Here is the survey that I gave to all of them:

Name:
Age:
Vocation:
Which program do you currently use to play music?
What, in your opinion, are the unique strengths of this solution?
What, in your opinion, are the limitations of this solution?
What feature would you personally want most out of a solution tailored to your needs?

The results from this survey were then collated to allow the data to be viewed and concluded more easily. Here are the results:

- **Which program do you currently use to play music?**
  - Windows Media Player **- 2**
  - ITunes – **3**
  - Other – **1**
- **What, in your opinion, are the unique strengths of this solution?**

  - Wide support for files – **2**
  - Online store – **1**
  - Video Playback – **1**
  - Smart playlists/Genius Playlists - 2
- **What, in your opinion, are the limitations of this solution?**
  - Slows down machine – **5**
  - Too many pointless features – **1**
- **What feature would you personally want most out of a solution tailored to your needs?**
  - Lightweight solution/won't interfere with machine performance – **3**
  - Playlists, both automatically created – **3**

Conclusion

Some conclusions can be made from the data gathered in this interview. First, my selected stakeholders show a great care in how the solution will affect the performance of their machine when it is running.

1.2.3 Why will the approach taken be suitable for stakeholder needs

As previously mentioned, my solution is going to be a lightweight, yet useful, attempt at the 'music player' problem. This will be suitable as many of the stakeholders mentioned how their preferred music players produced issues with the performance of their PC.

Furthermore, my stakeholder interviews have showed only a small number of users are interested in video playback. This could be because of the advancements in video streaming products such as Netflix and Amazon Prime Instant Video. These solutions do not require a player to play their files, it is all included in one system. However, my will to focus on music playback perfectly suits the stakeholders' needs.

1.2.4 Research into Alternative Solutions

The main competitors for this solution are Microsoft's Windows Media Player and Apple's ITunes.

Product Name:	Windows Media Player
Features:	1. Plays music (with music navigation buttons and progress bar). 2. Oorganises local media files into libraries 3. Allows video and photo playback 4. Allows connection to Network Accessible Devices 5. Provides plugins for cosmetic features such as player themes and visualizations for music.
Supported File Formats:	6. Windows Media Formats (.asf, .wma, .wmv, .wm) 7. Windows Media Metafiles (.asx, .wax, .wvx, .wmx) 8. Windows Media Metafiles (.wpl) 9. Microsoft Digital Video Recording (.dvr-ms) 10. Windows Media Download Package (.wmd) 11. Audio Visual Interleave (.avi) 12. Moving Pictures Experts Group

	(.mpg, .mpeg, .m1v, .mp2, .mp3, .mpa, .mpe, .m3u) 13. Musical Instrument Digital Interface (.mid, .midi, .rmi) 14. Audio Interchange File Format (.aif, .aifc, .aiff) 15. Sun Microsystems and NeXT (.au, .snd) 16. Audio for Windows (.wav) 17. CD Audio Track (.cda) 18. Indeo Video Technology (.ivf) 19. Windows Media Player Skins (.wmz, .wms) 20. QuickTime Movie file (.mov) 21. MP4 Audio file (.m4a) 22. MP4 Video file (.mp4, .m4v, .mp4v, .3g2, .3gp2, .3gp, .3gpp) 23. Windows audio file (.aac, .adt, .adts) 24. MPEG-2 TS Video file (.m2ts)
Support:	24/7 Help Centre provided by Microsoft

Product Name:	ITunes
Features:	Plays Music, organises local media files into libraries, allows video and photo playback, contains a store where users can buy music and video.
Supported Music File Formats:	.mp3, .aiff, .wav, .mpeg, .aac, .m4a, .mov, .aac, .ogg
Support:	24/7 Help Centre provided by Apple

Conclusion

While both of these solutions have strengths and weaknesses in areas, both provide exemplar solutions to the problem in discussion. The most prominent being Windows Media Player's unprecedented selection of supported file types and ITune's online store, providing original features to the problem in question.

However, through research, I discovered that both solutions have had many reports of slowing down the performance of machines, particularly ITunes. While the additional features that they provide are appreciated by many people, it seems they come at a cost in the usability of the program. The main point that my stakeholders brought to my attention in the interview is that they require a solution that has little to no impact on their machine. This means that some features of the solution (including some that are in these solutions) must be ignored to ensure its usability.

It is also appropriate to mention that both solutions are professional attempts, meant for a large-scale audience in a largely commercial environment. Because of this, these solutions are likely to have taken several years to perfect, requiring the help of a large team working on the solution at the same time. My solution is meant for a smaller audience and is scheduled to be done in less than a year, meaning that many of the features must be forgotten to meet the target by the required deadline.

It is also unlikely that my solution will be able to provide the 24/7 support, that these two solutions allow, in the immediate future. To address this, I have decided that the program

will be open-source. This will have a very small effect on the solution in question, just that the program will have to be well structured with clear variable names to accommodate third-party development.

1.2.5 Minimum Essential features required

1.2.5.1 Database

1. Form a normalized database to store information relating to the music.
2. Create a table for the songs.
3. Create a table for the albums.
4. Create a table for the artists.
5. Create a table for the playlists.
6. Create a table that stores information about the program's users.

1.2.5.2 User Interface

7. Provide a login window
8. Provide a create account window
9. Provide a Media Player window
10. Provide a settings window
11. Provide a playlist manager screen
12. Provide a dialog box to change a playlist's name.

1.2.5.3 Login Screen

13. Provide an input box where users can enter their username.
14. Provide an input box where users can enter their password.
15. Provide a button that users can use to log in to the program.
16. Validate the username and password against the program's database.
17. Provide access to the main program if validation is successful.
18. Provide a button that allow the user to navigate to the create account window.
19. Provide a label that notifies the user of any errors and gives instructions.

1.2.5.4 Create Account Screen

20. Provide a text box that allows the user to input their desired username.
21. Provide a text box that allows the user to input their desired password.
22. Provide a button that allows the user to create their account.
23. Validate the username and password using regex to make sure the inputs are long enough, contain numbers and contain letters.
24. Provided the validation is successful, append the details to the database alongside a 0 for the administrator field. Then close the window.

1.2.5.5 Media Player Screen

25. Provide a table that allows the user to navigate the library.
26. When an artist is clicked in the table, show all of the albums by the respective artist in the table.
27. When an album is clicked in the table, show all of the song from the respective album in the table.
28. When a song is clicked, play the respective song.
29. Create a queue of the next songs in the table when a song is clicked.
30. When a playlist is clicked, display all the song from the respective playlist.
31. Provide a button that allows the user to change the view of the library to scrutinize

artists.

32. Provide a button that allows the user to change the view of the library to scrutinize albums.
33. Provide a button that allows the user to change the view of the library to scrutinize songs.
34. Provide a button that allows the user to change the view of the library to scrutinize playlists. If the user isn't an administrator, only show the playlists that are owned by the user or are ownerless (such as the 'most played playlist'.
35. Provide a button that allows the user to navigate to the settings window.
36. Provide a button that allows the user to navigate to the playlist manager window.
37. Provide a text box that allows the user to search for an item in the database.
38. Provide a dropdown box that allows the user to select what item type they want to search for (e.g. artist, album, song).
39. Provide a button that allows the user to confirm the search.
40. Provide a button that allows the user to exit the application.
41. Provide a button that allows the user to pause the song that is currently being played.
42. Provide a button that allows the user to resume the song if a song is paused.
43. Provide a button that allows the user to skip to the next song in the queue when clicked.
44. Add the previously played song to a stack when said button is activated.
45. Provide a button that allows the user to play the previously played track when clicked.
46. Provide labels that notify the user what song is playing.
47. Provide labels that indicate what artist or album the user is currently viewing in the table.

1.2.5.6 Settings Screen

48. Provide an administrator section that only users with administrator permissions can see.
    a. Provide a table that shows all the program's users (excluding the current user).
    b. Allows users to be selected from the table.
    c. Provide a label that displays the selected user at a given time.
    d. Provide a button that deletes the selected user from the Users table in the database.
    e. Provide a button that toggles the selected user's administrator settings in the Users table.
49. Provide an import section:
    a. Provide a text box that allows the user to input a selected directory.
    b. Provide an import button that reads in the user's input when clicked.
    c. Validate the input to make sure the directory exists.
    d. Scan through the inputted directory and catch any .mp3 files.
    e. Read the relevant information from the ID3 tags of each .mp3 file, e.g. Song name, artist name, album name, genre, song length.
    f. Append the relevant information from the tags into the database, namely the Songs, Artists and Albums tables.
    g. Input N/A if any of the tags are not found.
50. Provide a button that allows the user to exit the settings window when clicked.
51. Provide a button that allows the user to log out of the program, taking them back to the login screen.

1.2.5.7 Playlist Manager Window

52. Provide a table that will display all of the user's playlists.
53. When a playlist in this table is clicked, songs from the respective playlist will be shown.
54. When a song in this table is clicked, the respective song will be removed from the playlist it belongs to (these changes will also be made in the database).

55. Provide a table that will display all of the songs in the database.
56. When a song in this table is clicked, add the song to the selected playlist provided one is selected (this change will also be made in the database).
57. Provide a button that will deselect the currently selected playlist and display all of the playlists in the playlist tables (see objective 52).
58. Provide a button that will open the new playlist dialog box when clicked.
59. Provide a button that will close this window when clicked.

1.2.5.8 Playlist Dialog Box

60. Provide a text box that will allow the user to input their desired name for the playlist.
61. Provide a button that will allow the user to create their playlist using their input when clicked.
62. Provide a button that will allow the user to close the dialog box when clicked.

1.2.6 Limitation of Minimum Features

There are several limitations to my proposed solution:

Firstly, my solution will only cater for music playback and not photo or video. This decision was made to reduce the strain that the solution had over the machine it was running on. Better suiting the stakeholder's requirements. I also feel that it would be an important decision to keep this project focused on music for the near future. This is to make sure that all efforts stays on perfecting the music playing problem and making a better all-round solution. I felt that this would be more suitable than diverting resources to incorporate phot and video.

Secondly, my solution will only support .mp3 files. MP3 files are the most used audio files to date, but there are many other widely used formats which could need support. However, many of these files do not contain the same tags as mp3 (ID3), meaning that my program will not be able to read them into the database. This decision was made to keep the solution as 'cut down' as possible as well as reducing the time of development to fit in with the deadline. Better suiting the stakeholder's needs.

I have personally had thoughts of incorporating access to an internet database of music. This could be utilized in many ways, including finding missing information for the local music files as well as recommending new music to the user depending on what music they listen to more. However, given the time pressures that this project has, I feel that it would be inappropriate to take on this responsibility.

1.2.7 Identification of proposed solution with reference to Research

Through the research into similar solutions, I discovered that both solutions have had many reports of slowing down the performance of machines, particularly ITunes. While the additional features that they provide are appreciated by many people, it seems they come at a cost in the usability of the program. The main point that my stakeholders brought to my attention in the interview is that they require a solution that has little to no impact on their machine. This means that some features of the solution (including some that are in these solutions) must be ignored to ensure its usability.

It is also appropriate to mention that both solutions are professional attempts, meant for a large-scale audience in a largely commercial environment. Because of this, these solutions are likely to have taken several years to perfect, requiring the help of a large team working on the solution at the same time. My solution is meant for a smaller audience and is scheduled to be done in less than a year, with minimal help from outside sources. This means that many of the features must be forgotten to meet the target by the required deadline.

1.2.8 Hardware and software requirements for solution

This solution will be programmed in Python 3.4. Python was chosen because it is an object-oriented language, allowing the solution to use models, methods and classes. These constructs

will aid the solution's development because it will allow the problem to be split up into more solvable sections, reducing space and producing a more efficient solution.

Python was also chosen due to its large collection of libraries. Libraries will be used in this solution to read the ID3 tags and allow the playback of songs. The use of these libraries will reduce the amount of time that the solution will be developed, as well as decrease the amount of code in the core program.

2 Design

2.1 Breakdown of Problem

2.1.1 Problem Elements suitable for computational solutions

I aim to decompose the problem down into smaller sub-problems in the hopes that this will make the problem easier to solve. These are the steps that will be taken:

1. First, the application's User Interface will be designed. The UI will comprise of several windows each serving different purposes. The windows will be the login screen, the create account screen, the media player screen and the settings window. Later in this section, the initial design of these screens will be established by the stakeholders. The UI will be developed in PyQt Designer.
2. After the UI has been designed, the database will be established. This database will hold all the data used by the program, including data relating to the music files (songs, artists, albums and playlists) and data related to the users (usernames and passwords).
3. Once the infrastructure for the application has been implemented, the solution will be coded. This process will start with the coding for the login section. This section will accept account information from the user and validate it in the program's database. If the details are accepted, the user is admitted into the main program.
4. After the login section, the import function will be programmed in the Settings Window. The function will accept a user specified directory and scan it for any mp3 files, inputting the details from the files into the database.
5. Then the view of the Music Player's library will be coded. Buttons will be implemented to allow the user to view all the tables in the database (artists, songs, albums etc.) and allow the user to traverse the music library (selecting an artist will retrieve the albums by that artist, selecting an album will retrieve the songs by that artist and selecting a song will play it).
6. The music playback functions will then be coded. This will play a song when it is selected from the view. A pause button will also allow a song to be stopped momentarily. A play button will also be provided to resume a paused song.
7. The queue function will then be coded. The function will add all of the songs under a selected song to a queue. Once the selected song has finished playing, the next song in the queue will be retrieved. This will continue until the queue is empty or a new song is selected.
8. After the queue function has been implemented into the program, the skip buttons will be programmed to traverse the queue backwards and forwards.
9. After the song queueing function has been programmed, the search function will be programmed. The search function will accept a string and use it to search the Music Library for any similar fields in the records. All those that match the User's input will be outputted in the view. The function will also be able to scrutinize what to search for (over artists, albums, songs and playlists).
10. Then the playlist feature will be coded. The feature will allow users to add new playlists into the database as well as access existing ones. A window can then be accessed that will allow the user to add and remove songs from the playlist.

2.1.2 Justifying the process

As the previous section has shown. The problem has been decomposed into sub-problems. The process has been done this way to ensure that the problem can be solved in a more simplistic nature. Furthermore, decomposing this problem also shows how it can be solved using computational methods such as modularity and object oriented programming. If each of the screens referred to in the success criteria (e.g. login screen, create account screen, main screen, playlist manager screen and settings screen) was assigned a separate class and therefore a separate object, the solution would become more efficient for a multitude of reasons. Some code could become reusable through the methods of a class and the variables from one class can be kept separate from another, ensuring the that no errors occur by mixing up variables.

2.2 Detailed Structure of Solution to be developed

2.2.1 Algorithms

In this section, pseudocode will be written for some of the main functions of the program. The pseudocode should give a basic understanding of how the program is meant to be designed and developed.

2.2.1.1 Login Screen Class

```
START LoginScreen

    class LoginScreenUI
        new btn_login //objects in login window defined
        new btn_crtacc
        new txt_username
        new txt_password

        public procedure new
            instance.loadUI
            instance.display()
        endprocedure

    endclass

    class LoginScreen inherits LoginScreenUI
        private userID
        private admin

        public procedure Login
            username = txt_username.text()
            password = txt_password.text()
            passwordHashed = password.hash
            IF username == "" or password == "":
                OUTPUT("Invalid username or password")
            ELSE
                users = openRead("tbl_users")
                i = 0 // counter variable
                flag = false
                while i != Length(users) or flag = true:
                    IF users[i][1] == username:
                        OUTPUT("Username Accepted")
                        IF users[i][2] == passwordHashed:
                            OUTPUT("Password Accepted")
                            userID = users[i][2]
                            admin = users[i][3]
                            Flag = True
                            MainScreenUI.show
                        ELSE:
                            OUTPUT("Password not accepted")
                        ENDIF
                    ELSE:
                        OUTPUT("Username not accepted")
                    ENDIF
    ENDIF

endprocedure

    public procedure CreateAccountTransistion
        CreateAccountScreen.show
```

2.2.1.2 Create Account Screen Class

```
START CreateAccountScreen

    class CreateAccountScreenUI
        new btn_crtacc
        new lbl_info
        new lbl_password
        new lbl_username
        new txt_password
        new txt_username

        public procedure new
            instance.loadUI
            instance.display()
        endprocedure
    endclass

    class CreateAccountScreen INHERITS CreateAccountScreenUI

        public procedure createAccount
            newUsername = txt_username.text()
            newPassword = txt_password.text()
            //regex
            newHashedPassword = newPassword.Hash
            users = openRead("tbl_users")
            i = 0
            highestUserID = 0
            WHILE i != LENGTH(users):
                currentUserID = users[i][0]
                IFcurrentUserID > highestUserID:
                    highestUserID = currentUserID
                ENDIF
                i = i + 1
            ENDWHILE
            userID = highestUserID + 1
            i = 0
            flag = FALSE
            WHILE i != LENGTH(users) + 1 or flag = TRUE:
                IF users[i][1] == newUsername:
                    OUTPUT("Username is already in use")
                    flag = TRUE
                ELSE:
                    i = i + 1
                ENDIF
            ENDWHILE
            IF flag != TRUE:
                users.APPEND[userID, newUsername, newHashedPassword, 0]
                CreateAccountScreenUI.hide
                LoginScreenUI.show
        endprocedure
END
```

2.2.1.3 Main Window Screen Class

```
START MAINWINDOW

    CLASS MainWindowUI  // User Interface for the Main Window
        new btn_back
        new btn_forward
        new btn_pause
        new btn_play
        new lbl_nowPlaying
        new btn_exit
        new btn_search
        new drp_search
        new lbl_album
        new lbl_artist
        new lbl_nowviewing
        new txt_search
        new btn_albums
        new btn_artists
        new btn_playlists
        new btn_plylstmngr
        new btn_settings
        new btn_songs
        new tbl_songs

        public procedure new()
            instance.loadUI
            instance.display()
        endprocedure

    ENDCLASS

    CLASS MainWindow INHERITS MainWindowUI
        private lst_data
        private queue_next
        private stack_prev
        private int_songSkip
        private bool_songsQueued
        private lst_nowPlaying

        public procedure init
            tbl_songs.init
        endprocedure

        public procedure loadData(lst_data)
            IF lst_data > 0:
```

```
            arrayI = [LENGTH(lst_data), -1, -1]
            i = 0
            WHILE i != 3:
                lst_data.pop[arrayI[0]]
                tbl_songs.remove[i]
                i = i + 1
            ENDWHILE
            i = 0
            WHILE i != LENGTH(lst_data):
                tbl_songs.append[lst_data[i]]
                i = i + 1
            ENDWHILE
        ENDIF
    endprocedure

    public function retrieveRow
        lst_rowdetails = []
        tbl_rowdetails = tbl_songs.rowSelected // row selected by user is assigned to this variable
        lst_details = lst_data[tbl_rowdetails]
        return lst_details
    endprocedure

    public procedure songs
        songs = openRead("tbl_songs")
        i = 0
        WHILE i != LENGTH(songs) + 1:
            tbl_songs.append(songs[i])
            i = i + 1
        ENDWHILE
    endprocedure

    public procedure albums
        albums = openRead("tbl_songs")
```

```
        WHILE i != LENGTH(albums) + 1:
            tbl_songs.append(albums[i])
            i =i + 1
        ENDWHILE
    endprocedure

    public procedure artists
        artists = openRead("tbl_songs")
        i = 0
        WHILE i != LENGTH(artists) + 1:
            tbl_songs.append(artists[i])
            i = i + 1
        ENDWHILE
    endprocedure


    public procedure playlists
        playlists = openRead("tbl_songs")
        i = 0
        WHILE i != LENGTH(playlists) + 1:
            IF playlists[i][-1] == LoginScreenInstance.userID:
                tbl_songs.append(playlists[i])
            ENDIF
            i = i + 1
        ENDWHILE
    endprocedure

    public procedure artistsToAlbums
        row = retrieveRow()
        int_artistID = row[0]
        albums = openRead("tbl_albums")
        i = 0
        WHILE i != LENGTH(albums) + 1:
            IF albums[i][-1] == int_artistID:
                tbl_songs.append(albums[i])
            ENDIF
            i = i + 1
        ENDWHILE

    public procedure albumsToSongs
        row = retrieveRow()
        int_albumID = row[0]
        songs = openRead("tbl_songs")
        i = 0
```

```
        WHILE i != LENGTH(songs) + 1:
            IF albums[i][-3] == int_albumID:
                tbl_songs.append(songs[i])
            ENDIF
        ENDWHILE

    public procedure playlistsToSongs
        row = retrieveRow()
        int_playlistID = row[0]
        playlistSongs = openRead("tbl_playlistsongs")
        songs = openRead("tbl_songs")
        i = 0
        WHILE i != LENGTH(playlistSongs) + 1:
            IF playlistSongs[i][1] = int_playlistID:
                int_songID = playlistSongs[i][2]
                j = 0
                flag = False
                WHILE j != LENGTH(songs) OR flag == FALSE:
                    IF songs[j][0] == int_songID:
                        tbl_songs.append(songs[j])
                        flag = TRUE
                    ELSE:
                        i = i + 1
                    ENDIF
                ENDWHILE
            ENDIF
        ENDWHILE
    endprocedure

    public procedure playSong
        row = retrieveRow
        nowPlaying = row
        str_songLocation = row[2]
        int_songID = row[0]
        int_songPlays = row[7]
        i = 0
        flag = FALSE
        WHILE i != LENGTH(songs) + 1 or flag = FALSE:
            IF songs[i][0] = int_songID:
                songs[i].APPEND(songs[i][0], songs[i][1], songs[i][2],
                songs[i][3], songs[i][4], songs[i][5], songs[i][7] + 1,
                songs[i][8])
```

```
                flag = TRUE
            ELSE:
            i = i + 1
            ENDIF
        ENDWHILE
        music.PLAY(str_songLocation)
    endprocedure

    public procedure play
        music.RESUME
    endprocedure

    public procedure pause
        music.PAUSE
    endprocedure

    public procedure settings
        IF LoginScreenInstance.int_admin == 0:
            SettingsScreen.hideAdminObjects
        ELSE:
            SettingsScreen.loadUserData
        ENDIF
        SettingsScreen.show
    endprocedure

    public procedure queueSongs
        queue_next = []
        flag = FALSE
        i = 1
        WHILE flag == FALSE:
            row = selectedRow + i
            row_details = tbl_songs[row]
            queue_next.append(row_details)
            IF row_details == tbl_songs[-1]
                flag == TRUE
            ENDIF
        ENDWHILE
    endprocedure

    public procedure skipForward
        int_songSkip = 1
    endprocedure
```

```
        public procedure skipBackward
            int_songSkip = -1
        endprocedure

        public procedure nextSong
            WHILE music.active() == TRUE and int_songSkip == 0:
                UI.processEvents()
            ENDWHILE
            IF int_songSkip == -1:
                int_songSkip = 0
                queue_next.append(0, nowPlaying)
                nextSong = stack_prev[0]
                stack_prev.pop[0]
                playSong(nextSong)
            ELSE:
                int_songSkip == 0
                stack_prev.append(0, nowPlaying)
                nextSong = queue_next[0]
                queue_next.drop[0]
                playSong(nextSong)
            ENDIF
        endprocedure

        public procedure playlistManager
            PlaylistScreenUI.show
            i = 0
            playlists = openRead("tbl_playlists")
            WHILE i != LENGTH(playlists):
                IF playlists[i][0] == LoginScreenInstance.userID:
                    tbl_playlists.append(playlists[i])
                ENDIF
                i = i + 1
            ENDWHILE
        endprocedure

    ENDCLASS

END
```

2.2.1.4 Settings Screen

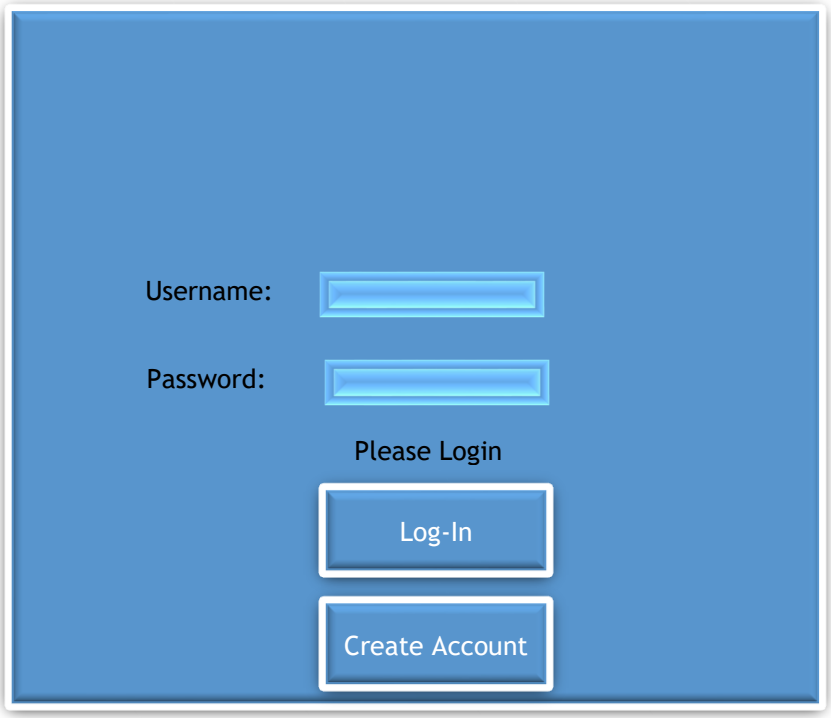
```
START SettingsScreen

    CLASS SettingsScreenUI
        new btn_exit
        new btn_logout
        new btn_admin
        new btn_delUser
        new lbl_user
        new tbl_users
        new btn_import
        new lbl_dir
        new lbl_import
        new txt_dir

        PUBLIC PROCEDURE new
            instance.loadUI
            instance.display()
        ENDPROCEDURE

    ENDCLASS

    CLASS SettingsScreen
        new str_selectedUser

        PUBLIC PROCEDURE loadUserData
            int_userID = LoginScreenInstance.int_userID
            users = openRead("tbl_users")
            i = 0
            WHILE i != LENGTH(users) + 1:
                IF users[i][0] != int_userID:
                    tbl_users.append(users[i])
                ENDIF
            ENDWHILE
        ENDPROCEDURE

        PUBLIC PROCEDURE selectUser
            row_details = tbl_users[selectedRow]
            selectedUser = row_details[1]
        ENDPROCEDURE

        PUBLIC PROCEDURE importing
            INPUT directory
            IF directory.isADirectory == FALSE:
                OUTPUT("Invalid Directory")
```

```
                Output("Invalid Directory")
            ELSE:
                directory = WALK(directory)
                WHILE i != LENGTH(directory.files) + 1:
                    mp3 = ID3(directory.files[i])
                    artist = mp3.artist
                    album = mp3.album
                    song = mp3.song
                    track = mp3.tracknumber
                    gnere = mp3.genre
                    length = mp3.length
                    artistsTable = openRead("tbl_artists")
                    i = 0
                    artistValid = 0
                    highestArtistID = 0
                    WHILE i != LENGTH(artistsTable):
                        IF artistsTable[i][1] == artist:
                            artistValid = 1
                            highestArtistID = artistsTable[i][0]
                        ENDIF
                        IF highestArtistID <= artistsTable[i][0] AND artistValid = 0:
                            highestArtistID = artistsTable[i][0]
                        ENDIF
                        i = i + 1
                    ENDWHILE
                    IF artistValid == 0:
                        artistID = highestArtistID + 1
                        tbl_artists.append[highestArtistID + 1, artist]
                    ENDIF
                    albumsTable = openRead("tbl_albums")
                    i = 0
                    albumValid = 0
                    highestAlbumID = 0
                    WHILE i !=LENGTH(albumsTable):
                        IF albumsTable[i][1] == album:
                            albumValid = 1
                            highestAlbumID = albumsTable[i][0]
                        ENDIF
                        IF highestAlbumID <=albumsTable[i][0] AND artistValid = 0:
                            highestAlbumID = albumsTable[i][0]
                    ENDIF
                    i = i + 1
                ENDWHILE
                IF albumValid == 0
                    tbl_albums.append[highestAlbumID + 1, album, artistID]
                ENDIF
                songsTable = openRead("tbl_songs")
                i = 0
                highestSongID = 0
                WHILE i != LENGTH(songsTable) + 1:
                    IF highestSongID <= songsTable[i][0]:
                        highestSongID = songsTable[i][0]
                ENDWHILE
                tbl_songs.append[highestSongID + 1, track, song, genre, highestAlbumID, length, 0]
            ENDWHILE
        ENDIF
    ENDPROCEDURE

ENDCLASS

END
```

2.2.2 User Interface Design

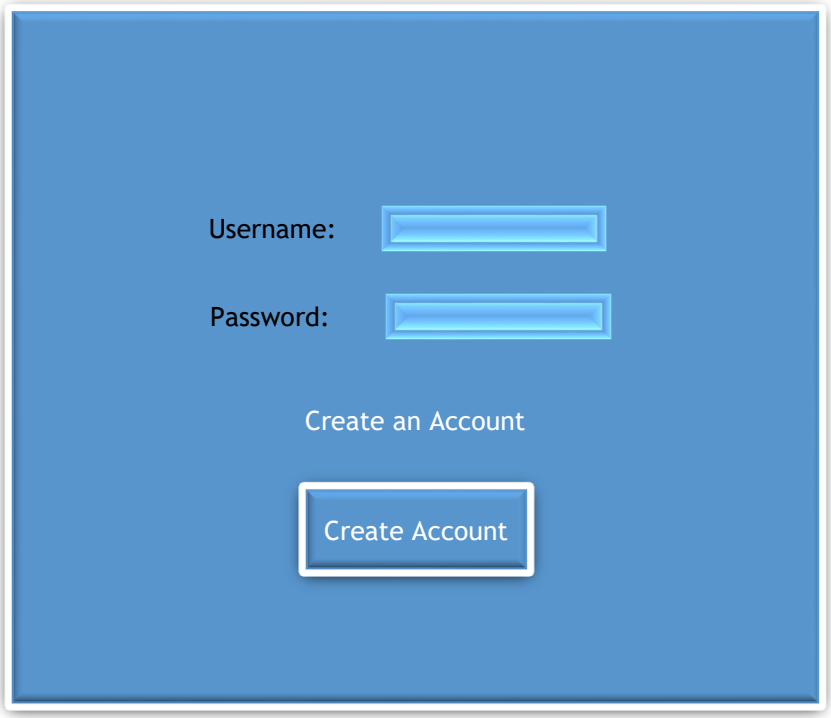
After having a discussion with the stakeholders for this product. I drew up a plan of the interface that everyone could agree on. Slight changes are likely to be made to these in order to improve functionality.

2.2.2.1 Login Window



Object	Number	Purpose	Example Input
txt_username	1	This is a textbox where the user can input their username.	"user1" → String
txt_pass	2	This is a textbox where the user can input their password.	"password1" → String
btn_login	3	This is a button that, once clicked, uses the information that the user has provided and attempts to gain access to the main program.	N/A
btn_crtacc	4	This is a button that, once clicked, will take the user to the 'create account' window. The user can then proceed to create a new account.	N/A
lbl_username	5	This label currently displays 'Please Login'. However, if any errors are brought up when checking the user's inputs (e.g. username doesn't exist in database), a suitable message will be displayed.	N/A
lbl_password	6	This label is used to indicate where the user's password should be entered.	N/A
lbl_info	7	This label is used to indicate where the user's username should be entered.	N/A

2.2.2.2 Create Account Window



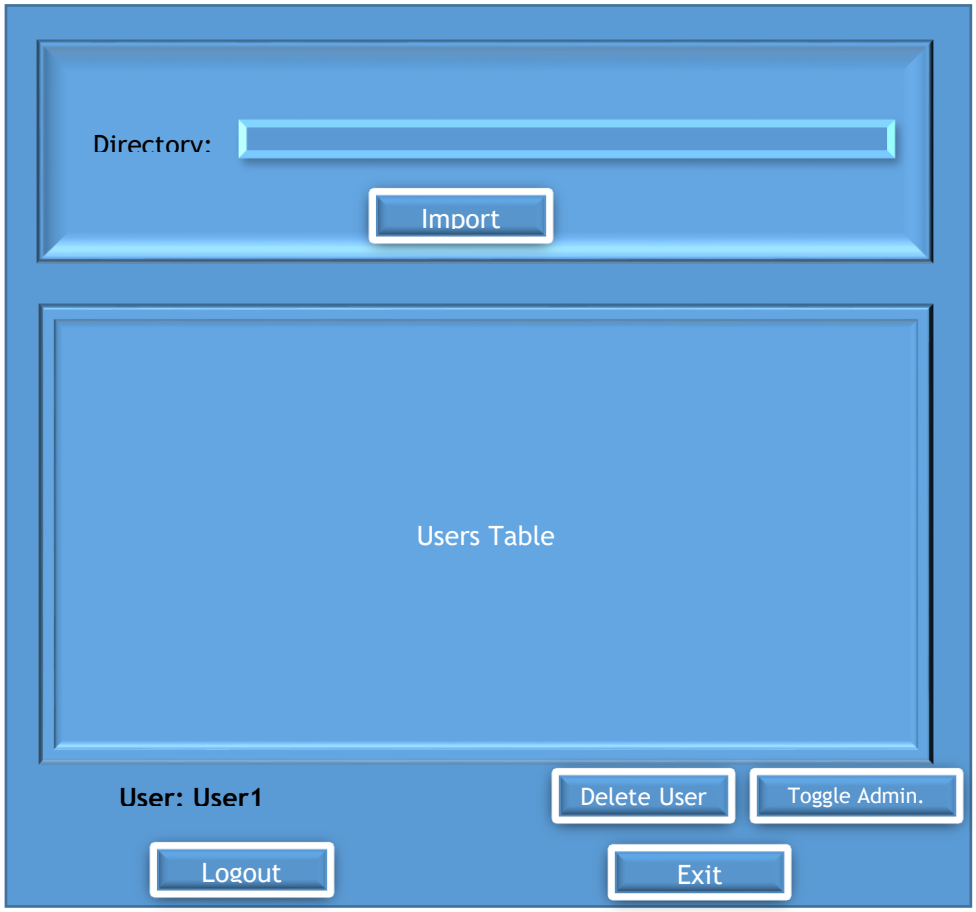
Object	Number	Purpose	Example Input
txt_username	1	This object allows the user to input their desired username.	"User1" → String
txt_password	2	This object allows the user to input their desired password.	"Password1" → String
lbl_username	3	This label indicates where the user should enter their username.	N/A
lbl_password	4	This label indicates where the user should enter their password.	N/A
lbl_info	5	This label initially displays "Create an Account". However, if any errors are raised when validating the user's inputs, the label will show a suitable message (e.g. Username has already been taken).	N/A
btn_crtacc	6	This button will activate the procedure that takes the user's inputs and creates an account using them (provided the inputs are successfully validated)	N/A

2.2.2.3 Media Player Window (Main Window)

Object	Number	Purpose	Example Input
Btn_back	1	When this button is activated, it will play the song at the top of the 'previously playing' stack.	N/A
Btn_forward	2	When this button is activated, it will play the song at the front of the "next song" queue.	N/A
Btn_pause	3	When this button is activated, the song that is currently playing will be paused.	N/A
Btn_play	4	When this button is activated, any song that has been paused will be resumed.	N/A
Lbl_nowplaying	5	Initially, this label will display: "Now Playing:". However, when a song begins playback, the name of the song will be added to the end of the label.	N/A
Btn_exit	6	When this button is activated. The entire program will shut down.	N/A
Btn_search	7	When this button is activated, The input from the drop-down menu and the search box will be collected and the music library will then be searched (using these inputs).	N/A
Drp_search	8	This drop-down menu allows the user to specify whether they want to search for Albums, Songs, Artists or Playlists	N/A
txt_search	9	This object allows the users to input the term that they want to search for in the Music Library.	"ArtistName" →String

Lbl_album	10	This label will show which album (if any) is being viewed in the Song View	N/A
Lbl_artist	11	This label will show which artist (if any) is being viewed in the Song View	N/A
Lbl_nowviewing	12	This label is to help the user understand what the lbl_artist and lbl_album objects are denoting	N/A
Btn_albums	13	When this button is activated, the song view (tbl_songs) will show all the albums in the music library	N/A
Btn_artists	14	When this button is activated, the song view (tbl_songs) will show all the artists in the music library.	N/A
Btn_playlists	15	When this button is activated, the song view (tbl_songs) will show all the playlists in the music library.	N/A
Btn_plylstmngr	16	When this button is activated, the 'Playlist Manager' window will be opened on top of the 'Main' Window.	N/A
Btn_settings	17	When this button is activated, the 'Settings' Window will be opened on top of the 'Main' Window	N/A
Tbl_songs	18	This table will show all the fields in the libraries database. At a basic level, it will show all of artists, artists, songs and playlists (depending on which button the user selects). However, it will also display albums belonging to a certain artist, songs belonging to a certain album and songs belonging to a certain playlist	N/A

2.2.2.4 Settings Window



Object	Number	Purpose	Example Input
Btn_exit	1	When this button is activated, this window will be closed.	N/A
Btn_logout	2	When this button is activated, the user will be taken back to the login screen. Any variables that identify the user (e.g. UserID and UserName) will be set to a null value.	N/A
Btn_admin	3	When this button is activated, the selected user (the field selected in tbl_users) will be made an admin or, if they are already an admin, will be made an ordinary user.	N/A
Btn_delUser	4	When this button is activated, the selected user (the field selected in tbl_users) will be deleted from the program and the database.	N/A
Lbl_user	5	This object has a null value of "User:". However, when a user is selected from tbl_users, the object will also show the name of the user.	N/A
Tbl_users	6	This table will show all of the users (from the users table in the database), excluding the user who is currently logged in.	N/A

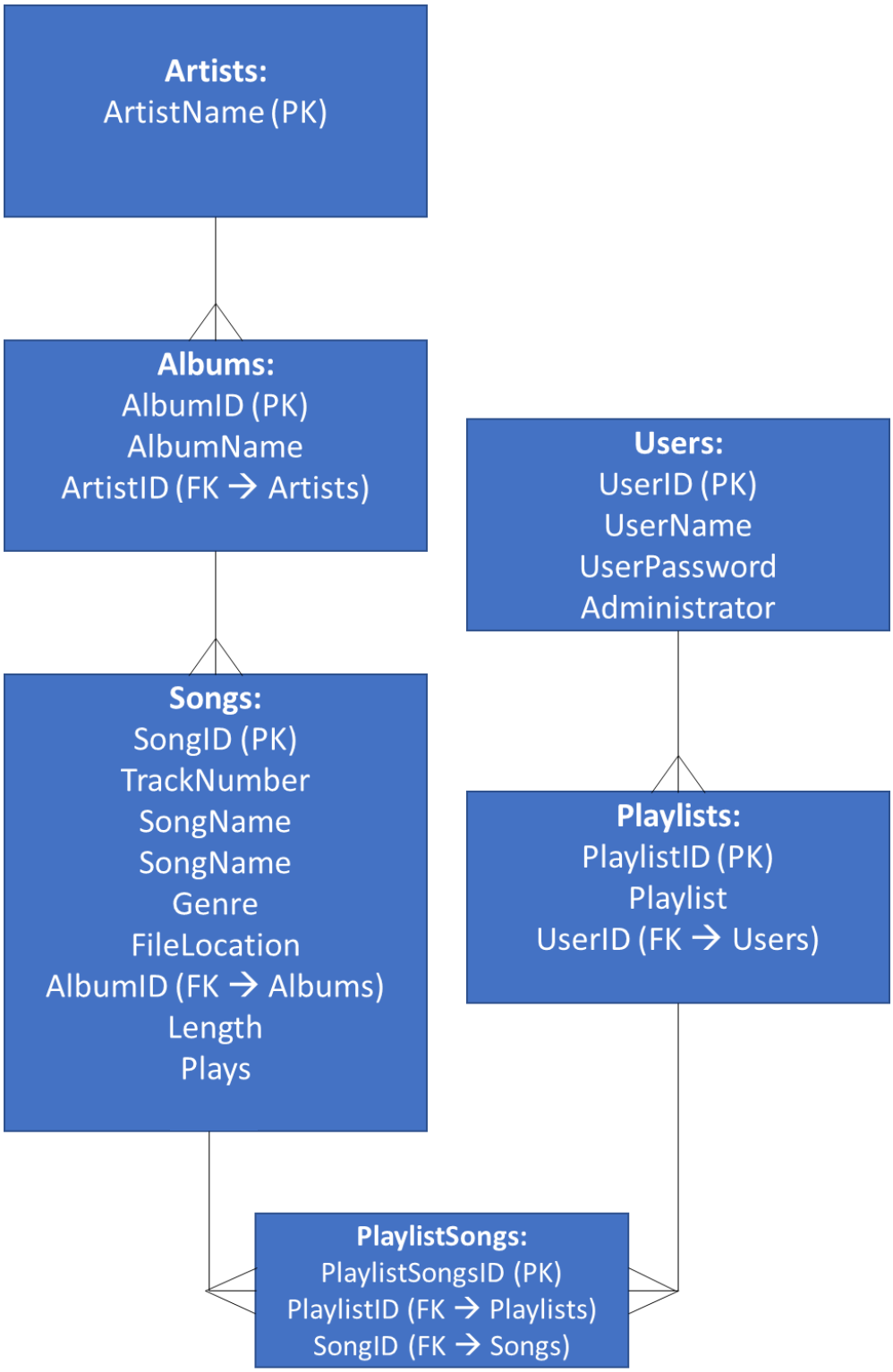
Btn_import	7	When this button is activated, the user's input (their selected directory) from txt_dir is taken. The music in this directory is then imported into the database.	N/A
Lbl_dir	8	This object will indicate where the user should enter their directory.	N/A
Txt_dir	9	This object will allow the user to input the directory from where they want their music to be imported from.	"C:/Users/User /Music/" → String

2.2.2.5 Playlist Manager Window

Object	Number	Purpose	Example Input
Btn_close	1	When this button is activated, the window is closed, taking the user back to the Main Window	
Btn_newplylst	2	When this button is activated, a dialog box will open. Here the user can input the new playlist's name.	
Btn_plylstreset	3	When this button is activated, th	
Tbl_playlists	4	This table will display all of the information relating to playlist. This includes the names of the playlists and the songs that each playlist contain.	
Tbl_songs	5	This table will show all of the songs that are currently in the music library.	

2.2.3 Database Design

In order to maintain complete and consistent efficiency in my program, I will need to have a normalised database, preferrably in Third Normal Form.
In order to do this, I have decided to isolate the three most independent fields to describe the songs (The artist, album and title).



The diagram also shows that there will be a PlaylistSongs table. This table was used to create a relation between the Songs and Playlists tables in a more normalized format (making sure the two tables did not have a many to many relationship).

2.2.4 UML Diagrams

2.2.4.1 UML Class Diagram

Here are the diagrams showing the individual classes as well as their attributes and methods:

MAIN WINDOW CLASS

ATTRIBUTES:
Bool_songsqueued
Int_songskip
Lst_data
Lst_nowPlaying
Lst_Model
Queue_next
Stack_prev
Str_searchTable
Str_searchTerm

METHODS:
Init
Load_data
Retrieve_row
Songs
Albums
Artists
Playlists
Artists_to_Albums
Albums_to_songs
Playlists_to_Songs
Play_song
Play
Pause
Settings
Hide_song_columns
Hide_album_columns
Hide_artist_colums
queueSongs
Next_song
Search
Skip_forward
Skip_backward
Refresh_most_played_playlists
playlistManager
Exit

SETTINGS WINDOW CLASS

ATTRIBUTES:
Int_userID
lst_data
Lst_Model
Selected_user

METHODS:
Init
loadUserData
selectUser
Importing
Logout
Exit
deleteUser
toggleAdmin

PLAYLIST MANAGER WINDOW CLASS

ATTRIBUTES:
Lst_playlistdata
Lst_Model
Int_Playlist_id
Str_Playlist_query
Lst_songData

METHODS:
Init
Retrieve_playlists_row
Add_to_playlist
Remove_from_playlist
Playlists_to_songs
Load_song_data
Load_playlist_data
Refresh_playlist
Close_window
Playlist_reset
New_playlist

CREATE ACCOUNT WINDOW CLASS

ATTRIBUTES:
Null
METHODS:
Init
Create_account
loginwindow

PLAYLIST NAME DIALOG BOX CLASS

ATTRIBUTES:
NULL

METHODS:
Init
Add_playlist
Close_window

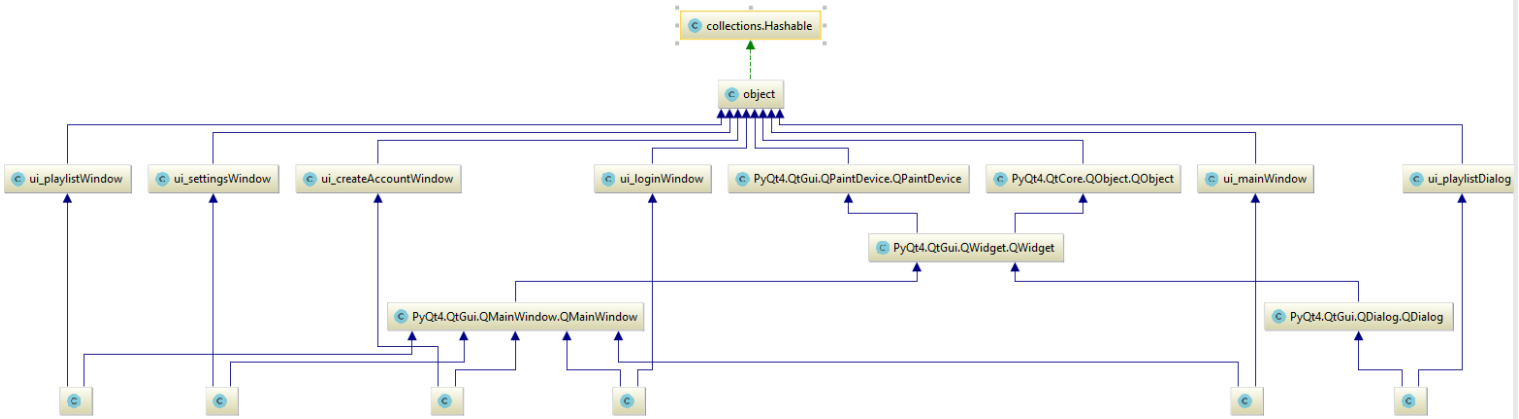
LOGIN WINDOW CLASS

ATTRIBUTES:
Int_userID
Int_Admin

METHODS:
Init
Login
createaccount

2.2.4.2 UML Program Stream Diagram

Here is a diagram showing the flow of the program stream and the connections between each of the classes:



2.3 Approach to Testing

An iterative approach will be taken to test the programming solution. Meaning that after some major changes to the code, it will be tested by myself. Any problems or weaknesses that arrive from this testing will be improved in a revised iteration of the code.

2.3.1 Test Data

This solution does not require much test data before testing begins because most of the data is inputted during the running of the program. For example, the data on the music files are appended when the physical music files are imported in the import function. Furthermore, user data can also be created in the create account screen of the program. The only data that must be inputted into the database is the Most Played Playlist record in the Playlist Table. However, an administrator user will also be added to the users table to ensure the ease of testing the solution.

However, for the solution to be tested, some mp3 files will need to exist. For my initial tests, I will use my own personal music library. However, in later sections of the program I will release my program to the initial stakeholders to gather their point of view and see how the code performs with different machines and music libraries.

2.3.2 Test Plan

2.3.2.1 Database

Test ID	Success Criteria Tested	Input	Expected Output
1	1	Open database file	All tables in the database will be related to one another and primary keys will exist for all tables.
2	2	Open database file	Songs table will exist
3	3	Open database file	Albums table will exist
4	4	Open database file	Artists table will exist
5	5	Open database file	Playlist table will exist
6	6	Open database file	Users table will exist

2.3.2.2 User Interface

Test ID	Success Criteria Tested	Input	Expected Output
7	7, 13, 14, 15, 18, 19	Open login window .ui file	User Interface will exist with all expected objects
8	8, 20, 21, 22	Open create account window .ui file	User Interface will exist with all expected objects
9	9, 25, 31, 32, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 45, 46, 47	Open Main Window .ui file	User Interface will exist with all expected objects
10	10	Open Settings Window.ui file	User Interface will exist with all expected objects
11	11	Open Playlist Manager .ui file	User Interface will exist with all expected objects
12	12	Open Playlist Name .ui file	User Interface will exist with all expected objects

2.3.2.3 Login Screen

Test ID	Success Criteria Tested	Input	Branches	Expected Output
13	16, 17	Login Button Pressed	a) Correct username, incorrect password. b) Incorrect username, correct password. c) No details inputted. d) Correct details inputted.	a) Label will display: "Wrong Password" b) Label will display: "Wrong Username" c) Label will display: "Fields Empty" d) User will be logged in & Main Window will be displayed.

2.3.2.4 Create Account Window

Test ID	Success Criteria Tested	Input	Branches	Expected Output
14	23, 24	Create Account Button pressed	a) Password has no letters. b) Password has no numbers c) Password is not between 6 and 16 characters. d) Username has already been used	a) Label displays: "Please use letters in the password" b) Label displays: "Please use numbers in the password" c) Label displays: "Password must be 6-16 characters"

2.3.2.5 Main Window

Test ID	Success Criteria Tested	Input	Branches	Expected Output
15	26	Artist row is clicked in the table		All of the albums by artist in library are shown.

16	27	Album row clicked in table		All songs in album are shown
17	28	Song row clicked in table		Song begins playback
18	29, 43	Skip forward button press		Next song in queue will begin playback
19	30	Playlist row clicked in table		All song in playlist are shown
20	31	Artist button pressed		All artists in music library are shown
21	32	Albums button pressed		All albums in music library are shown
22	33	Songs button pressed		All songs in music library are shown
23	34	Playlists button pressed		All playlists owned by the user and the 'Most Played' playlist are shown
24	35	Settings button press		Settings window opens
25	36	Playlist Manager button pressed		Playlist Manager window opens
26	39	Search button pressed with inputs for the search term and its criteria		Matches with the term and criteria will be displayed.
27	40	Exit button pressed		The application will close
28	41	Pause button pressed		Music playback stops
29	42	Resume button pressed		Music playback **resumes**
30	44, 45	Skip backward button pressed		Previously played song begins playback
31	46	Song is played		Label changes to the title of the song being played.
32	47	Table is navigated		Labels change depending on the artist/album they are looking at

2.3.2.6 Settings Screen

Test Id	Success Criteria Tested	Input	Branches	Expected Output
33	48	Settings button in Main Window pressed while logged in with an administrator account.	a) Select user from the table b) Delete user c) Toggle admin of user	Administrator section is shown a) User is selected and label changes to the name of the user. b) User is deleted from database. c) User's administrator permissions are successfully toggled.
34	49	Directory Inputted and Import button pressed	a) Invalid directory	All tagged files in folders are directly inputted into the library a) Label will show: "Directory does not exist"
35	50	Exit button pressed		Settings window is closed
36	51	Logout button pressed		User is returned to the logout screen

2.3.2.7 Playlist Manager Window

Test ID	Success Criteria Tested	Input	Branches	Expected Output
37	53	Playlist row selected in playlists table		Songs from selected playlists are shown in playlists table
38	54	Song in playlists table is selected		Selected song is removed from the playlist it is in
39	56	Song is songs table is selected		Selected song is added to the selected playlist
40	57	Deselect Playlist button clicked		Playlist is deselected and all playlists are shown again in

				the playlists table
41	58	New Playlist button clicked		Playlist Name dialog box opens
42	59	Close Window button clicked		Playlist Manager window will close

2.3.2.8 Playlist Dialog Box

Test ID	Success Criteria Tested	Input	Branches	Expected Output
43	61	OK button pressed	a) Playlist name is under 5 characters	Playlist Name dialog box closes and playlist is created. a) Label shows "Playlist name is too short"
44	62	Cancel button pressed		Playlist Name dialog box is closed

3 System Development

3.1 Database Initialisation

3.1.1 Problem Decomposition

The database will have to hold all the data on the songs in the user's library, particularly the artist's name, the album's name and the song's name. On top of this, the database must also store all the information on the playlists created in the program and the information of all the users who have access to the program.

To maintain efficiency in the database (removing data redundancy and duplication), the database will be put in third normal form. Doing so will also prevent problems in data management by the program.

3.1.2 Creating the Tables

As mentioned in the Design section of the documentation, the database will contain the following tables:

25. Albums
26. Artists
27. Songs
28. Playlists
29. Users
30. PlaylistSongs

The required columns for each of these tables will be created through SQLite code. The code will also have relations assigned to them through foreign keys. Primary keys will also be assigned to each table.

When the solution is distributed for the stakeholder's use, the Albums, Artists. Songs, Users and PlaylistSongs tables will be empty, as these are filled with information specific to each user. The Playlists table will contain one field, for the most played playlist. For the testing of the program, test data will be loaded into the

First, the tables and primary keys will be established. Then, the foreign keys relating the databases will be established.

3.1.2.1 Albums

The albums table will hold all of the information specifically related to the albums in general:

31. AlbumID (int) **(PK)**
32. AlbumName (text)
33. ArtistID (int) **(FK)**

Any relations to artists table will be established after the creation of the tables.

Creation Code:

```
CREATE TABLE `Albums` (
        `AlbumID`       INTEGER UNIQUE,
        `AlbumName`     TEXT,
        `ArtistID`      INTEGER,
        PRIMARY KEY(`AlbumID`),
);
```

3.1.2.2 Artists

The artists table will hold all the information specifically related to the artists in general:

34. ArtistsID (int) **(PK)**
35. ArtistName (text)

Creation Code:

```
CREATE TABLE `Artists` (
        `ArtistID`      INTEGER UNIQUE,
        `ArtistName`    TEXT,
        PRIMARY KEY(`ArtistID`)
);
```

3.1.2.3 Songs

The songs table will hold all of the information specifically on the songs in general:

36. SongID (int)
37. SongName (text)
38. TrackNumber (int)
39. Genre (text)
40. FileLocation (str)
41. AlbumID (int) **(FK)**
42. Length (int)
43. Plays (int)

Any relations to the albums and artists tables will be created after the initial creation of the tables.

Creation Code:

```
CREATE TABLE `Songs` (
        `SongID`        INTEGER UNIQUE,
        `TrackNumber` INTEGER,
        `SongName`      TEXT,
        `Genre`         TEXT,
        `FileLocation`  TEXT,
        `AlbumID`       INTEGER,
        `Length`        REAL,
        `Plays` INTEGER,
        PRIMARY KEY(`SongID`),
);
```

3.1.2.4 Playlists

The playlists table will hold all the information related to the playlists in the program. This will include all user-created playlists and the 'Most Played' playlist that will already be in the table. The table will include the following values:

44. PlaylistID (int) **(PK)**
45. Playlist (text)
46. UserID (int) **(FK)**

Creation Code:

```
CREATE TABLE `Playlists` (
        `PlaylistID`    INTEGER UNIQUE,
        `Playlist`      TEXT,
        `UserID`        INT,
        PRIMARY KEY(`PlaylistID`),
);
```

3.1.2.5 PlaylistSongs

In order to keep the database in third normal form, none of the tables can have a many-to-many relationship with another. This table is used as a go-between for the Playlists table and the Songs table. Instead of having a many-to-many relationship with one another, they now each have a one-to-many relationship with the PlaylistSongs table. The contents of this table are as follows:

47. PlaylistSongsID **(PK)**
48. PlaylistID **(FK)**
49. SongID **(FK)**

Creation Code:

```
CREATE TABLE `PlaylistSongs` (
        `PlaylistSongsID`       INTEGER,
        `PlaylistID`    INTEGER,
        `SongID`        INTEGER,
        PRIMARY KEY(`PlaylistSongsID`),
);
```

3.1.2.6 Users

This table will hold all of the information about the Users. The fields in the table are as follows:

50. UserID (int) **(PK)**
51. UserName (text)
52. UserPassword (text)
53. Administrator (int)

Creation Code:

```
CREATE TABLE `Users` (
        `UserID`        INTEGER UNIQUE,
        `UserName`      TEXT,
        `UserPassword`TEXT,
        `Administrator`INT DEFAULT 0,
        PRIMARY KEY(`UserID`)
);
```

3.1.3 Creating Relationships

After the initial tables were created, the relationships between them were established through the foreign keys. The foreign keys used in this database are as follows:

54. Albums.ArtistID → Artists.ArtistID
55. Playlists.UserID → Users.UserID
56. PlaylistSongs.PlaylistID → Playlists.PlaylistID
57. PlaylistSongs.SongID → Songs.SongID
58. Songs.AlbumID → Albums.AlbumID

These relationships will be made in **DB Browser for SQLite** (the Database Management System).

3.1.3.1 Reformed Code

Albums

```
CREATE TABLE `Albums` (
	`AlbumID`	INTEGER UNIQUE,
	`AlbumName`	TEXT,
	`ArtistID`	INTEGER,
	PRIMARY KEY(`AlbumID`),
	FOREIGN KEY(`ArtistID`) REFERENCES `Artists`(`ArtistID`)
);
```

Artists

```
CREATE TABLE `Artists` (
	`ArtistID`	INTEGER UNIQUE,
	`ArtistName`	TEXT,
	PRIMARY KEY(`ArtistID`)
);
```

PlaylistSongs

```
CREATE TABLE `PlaylistSongs` (
	`PlaylistSongsID`	INTEGER,
	`PlaylistID`	INTEGER,
	`SongID`	INTEGER,
	PRIMARY KEY(`PlaylistSongsID`),
	FOREIGN KEY(`PlaylistID`) REFERENCES `Playlists`(`PlaylistID`),
	FOREIGN KEY(`SongID`) REFERENCES `Songs`(`SongID`)
);
```

Playlists

```
CREATE TABLE `Playlists` (
	`PlaylistID`	INTEGER UNIQUE,
	`Playlist`	TEXT,
	`UserID`	INT,
	PRIMARY KEY(`PlaylistID`),
	FOREIGN KEY(`UserID`) REFERENCES `Users`(`UserID`)
);
```

Songs

```
CREATE TABLE `Songs` (
        `SongID`        INTEGER UNIQUE,
        `TrackNumber`INTEGER,
        `SongName`      TEXT,
        `Genre`         TEXT,
        `FileLocation`  TEXT,
        `AlbumID`       INTEGER,
        `Length`        REAL,
        `Plays` INTEGER,
        PRIMARY KEY(`SongID`),
        FOREIGN KEY(`AlbumID`) REFERENCES `Albums`(`AlbumID`)
);
```

Users

```
CREATE TABLE `Users` (
        `UserID`        INTEGER UNIQUE,
        `UserName`      TEXT,
        `UserPassword`TEXT,
        `Administrator`INT DEFAULT 0,
        PRIMARY KEY(`UserID`)
);
```

3.1.4 Loading the Test Data

As mentioned previously, the artists, albums, songs and PlaylistSongs tables take their data as inputs from the user. These tables will be empty as a result. The Playlist table will only contain a field for the 'Most Played' playlist.

3.1.4.1 Most Played Playlist

Here is the code used to insert 'Most Played' playlist into the Playlists database:

"INSERT INTO Playlists VALUES (1, "Most Played", 0)"

The playlist is given a UserID of 0 as it is available for all users to use.

3.1.5 Final View

Here is the final view of the database:

Name	Type	Schema
Tables (7)		
Albums		CREATE TABLE "Albums" (`AlbumID` INTEGER UNIQUE, `AlbumName` TEXT, `ArtistID` INTEGER, FOREIGN KEY(`ArtistID`) REFERENCES Artists(ArtistID
AlbumID	INTEGER	`AlbumID` INTEGER UNIQUE
AlbumName	TEXT	`AlbumName` TEXT
ArtistID	INTEGER	`ArtistID` INTEGER
Artists		CREATE TABLE "Artists" (`ArtistID` INTEGER UNIQUE, `ArtistName` TEXT, PRIMARY KEY(`ArtistID`))
ArtistID	INTEGER	`ArtistID` INTEGER UNIQUE
ArtistName	TEXT	`ArtistName` TEXT
PlaylistSongs		CREATE TABLE "PlaylistSongs" (`PlaylistID` INTEGER, `SongID` INTEGER, FOREIGN KEY(`PlaylistID`) REFERENCES `Playlists`(`PlaylistID`), FOREIGN KEY
PlaylistID	INTEGER	`PlaylistID` INTEGER
SongID	INTEGER	`SongID` INTEGER
Playlists		CREATE TABLE "Playlists" (`PlaylistID` INTEGER UNIQUE, `Playlist` TEXT, PRIMARY KEY(`PlaylistID`))
PlaylistID	INTEGER	`PlaylistID` INTEGER UNIQUE
Playlist	TEXT	`Playlist` TEXT
PlaylistsUsers		CREATE TABLE "PlaylistsUsers" (`PlaylistID` INTEGER, `UserID` INTEGER, FOREIGN KEY(`PlaylistID`) REFERENCES Playlists(PlaylistID), FOREIGN KEY(`Us
PlaylistID	INTEGER	`PlaylistID` INTEGER
UserID	INTEGER	`UserID` INTEGER
Songs		CREATE TABLE "Songs" (`SongID` INTEGER UNIQUE, `SongName` TEXT, `TrackNumber` TEXT, `SongRating` INTEGER, `Genre` TEXT, `FileLocation` TE
SongID	INTEGER	`SongID` INTEGER UNIQUE
SongName	TEXT	`SongName` TEXT
TrackNumber	TEXT	`TrackNumber` TEXT
SongRating	INTEGER	`SongRating` INTEGER
Genre	TEXT	`Genre` TEXT
FileLocation	TEXT	`FileLocation` TEXT
AlbumID	INTEGER	`AlbumID` INTEGER
Users		CREATE TABLE "Users" (`UserID` INTEGER UNIQUE, `UserName` TEXT, `UserPassword` TEXT, `Administrator` INT DEFAULT 0, PRIMARY KEY(`UserID`))
UserID	INTEGER	`UserID` INTEGER UNIQUE
UserName	TEXT	`UserName` TEXT
UserPassword	TEXT	`UserPassword` TEXT
Administrator	INT	`Administrator` INT DEFAULT 0

3.2 Class-less code

Some of the code to be used in the program does not belong in any specific class. Most of these lines are to initialise components of the program (such as the database and .ui files) or to create objects.

All the imported libraries also belong in this section.

3.2.1 Imported Libraries

Here is the code for the imported libraries that are used by the program. The lines have also been annotated to show what each library is used for:

```
import os  # os library imported, used for directory scanning and to exit the program
import sys  # sys library imported, used to create the application alongside PyQt
import sqlite3 as lite  # sqlite3 library imported, used for database access
import hashlib  # hashlib library imported, used for hashing passwords.
import re # re library imported, used for regex during registration
from mutagen.easyid3 import EasyID3  # EasyID3 library imported from mutagen library,
# used to read the ID3 tags in mp3 files
from mutagen.mp3 import MP3  # MP3 library imported from mutagen library, used to find the length of songs
from pygame import mixer  # mixer library imported from pygame, used for music playback
from PyQt4 import QtGui, uic, QtCore  # QtGui, uic and QtCore libraries imported from PyQt library.
# Used for user interface and application creation
```

3.2.2 Linking the Database with the code

The database created in 4.1 must be linked to the code before it can be used by the program.

Here is the code used to connect the database:

```
con = lite.connect('Music Library.db')  # Connection to database established
cur = con.cursor()  # Cursor created for database management
```

The first line connects the database file (Music Library.db) to the program, while the second line establishes the cursor to allow SQL queries to made for the database.

3.2.3 Connecting the User Interface Files

The User Interface files are created in QtDesigner through .ui files. The files are then imported into the program.

As the files were added at different points in the development cycle, evidence of iterative development will be shown in later sections. However, the final code of the UI imports are as follows:

```
ui_loginWindow = uic.loadUiType("Login UI.ui")[0]  # Login Window loaded in
ui_createAccountWindow = uic.loadUiType("Create Account.ui")[0]  # Create Account Window loaded in
ui_mainWindow = uic.loadUiType("Media Player UI.ui")[0]  # Main Window loaded in
ui_settingsWindow = uic.loadUiType("Settings.ui")[0]  # Settings Window loaded in
ui_playlistWindow = uic.loadUiType("Playlist Manager.ui")[0]  # Playlist Window loaded in
ui_playlistDialog = uic.loadUiType("Playlist Name.ui")[0]  # Playlist Name Dialog Box loaded in
```

3.3 Login Screen Class

3.3.1 Project Decomposition

This class will hold all the code required for the 'Log-In' section of the program. It will be the first screen/class used by the user when the program starts.

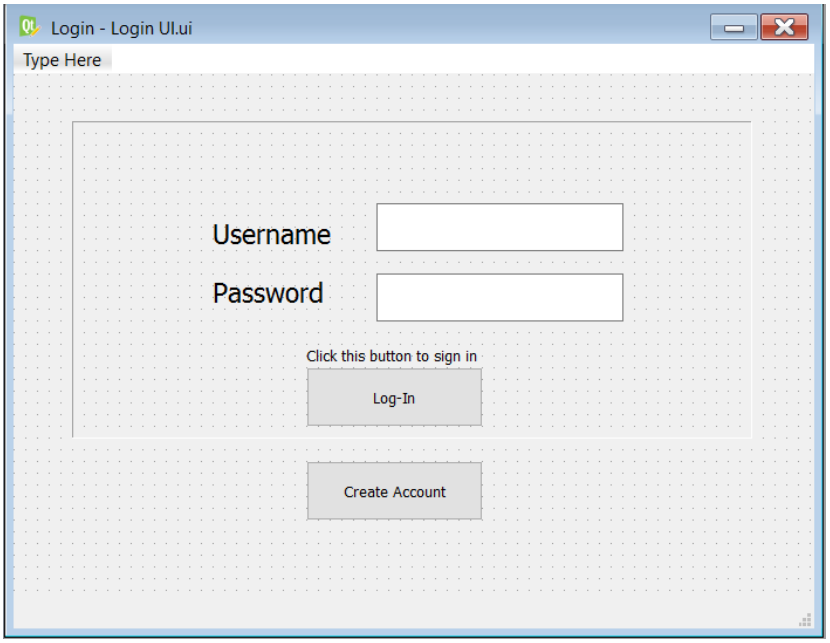
The code will have to take two inputs (the user's username and password) and validate them against the details in the Users table in the database. As the passwords in the database are

required to be hashed, the users input must also be hashed before it is compared to the password in the database. If the validation is not successful, an appropriate message will be displayed to the user. Else, the Login window will close and the Main Window will be opened.

3.3.2 1st Iteration

3.3.2.1 User Interface

Before I started to code the class, I created the User Interface for the Login Screen. Changes will be made throughout the development to accommodate more features. In this iteration, I aimed to only cater for the basic login section of this class. Here is the initial look of the login screen:



The Window uses several objects from the Qt Framework:

59. The textboxes use QLineEdit objects to allow the users to input their usernames and passwords.
60. QPushButton objects are used for the Log-In and Create Account buttons.
61. QLabel objects are used for the Username and Password labels as well as the sign-in instruction.

3.3.2.2 Code

For the first attempt at writing this class, I focused on writing just the validation process, with the hopes to implement the database and hashing function later. Here is the first iteration of the code.

First, the correct username and password were declared outside of the class, as well as the 'success' variable for the validation:

```
real_username = 'jonesra'
real_password = 'jones'
success = False
```

Then the shell of the class was created and the initialisation function was written. The initialisation functions in each class will hold all of the definitions of attributes (including the objects in the ui) and the initialisation code for the GUI:

```
class LoginWindowClass(QtGui.QMainWindow, login_window):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)  # Qt GUI initialisation script run
        self.txt_uname = ""  # self variable declared, used to store the user's input for their username
        self.txt_password = ""  # self variable declared, used to store the user's input for their password
        self.setupUi(self)  # User Interface set up using Qt function
        self.btn_login.clicked.connect(self.login)  # Login button from user interface programmed
        self.btn_crtacc.clicked.connect(self.createaccount)  # Create Account button from ui programmed
```

Then the algorithm for the login validation was written. The code retrieves the user's inputs and validates them against the 'real username' and the 'real password' that have already been declared. If the usernames or passwords don't match, an appropriate message will be displayed. Else, the current screen will be closed and the next screen will be opened:

```
def login(self):
    global success  # success value globalised
    self.txt_uname = self.txt_uname.text()  # draws variables from what is in the two line edit boxes
    self.txt_password = self.txt_pass.text()  # ^
    print(self.txt_uname, self.txt_password)  # Inputs are outputted as a test
    if self.txt_uname == real_username:  # validation to see if username is correct
        if self.txt_password == real_password:  # validation for correct password
            success = True  # validation has passed
            LoginWindow.hide()  # switches window from login to the main window
            MainWindow.show()  # ^
        else:
            self.lbl_info.setText("Wrong password")  # notifies user that the wrong password has been used
    else:
        self.lbl_info.setText("Wrong username")  # notifies user that the wrong username has been used
```

The code has been written using a **Hungarian notation** standard. This is to support maintainability of code.

This code does not represent the full functionality of the program. However, it will still need to be tested.

3.3.3 Testing for Iteration 01

3.3.3.1 Inputting the correct User Details

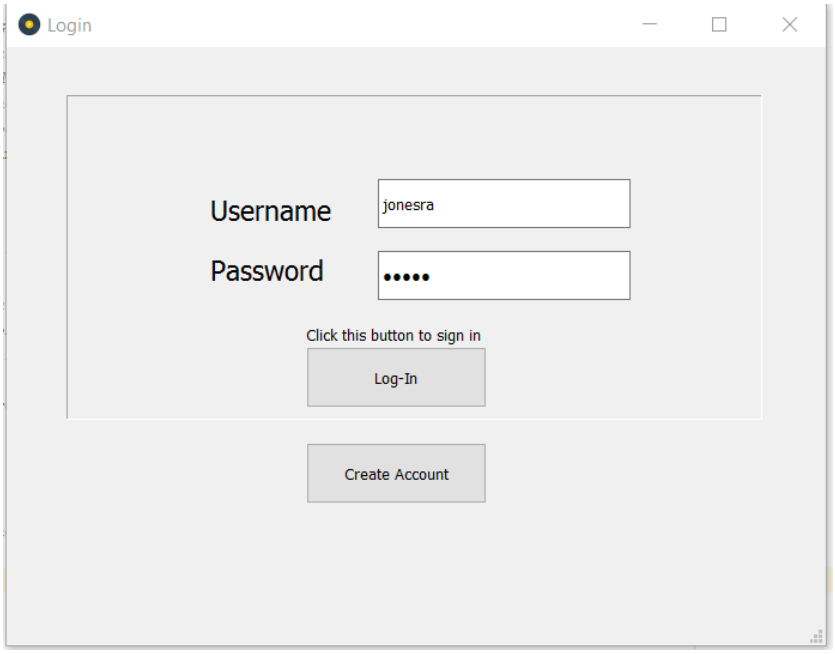
For this section, the inputs used will be:

Username = "jonesra"
Password = "jones"
And the Login button is activated

As the username and password variables match the 'real username' and 'real password' variables, the expected output will be that the login window will close and the media player window will open.

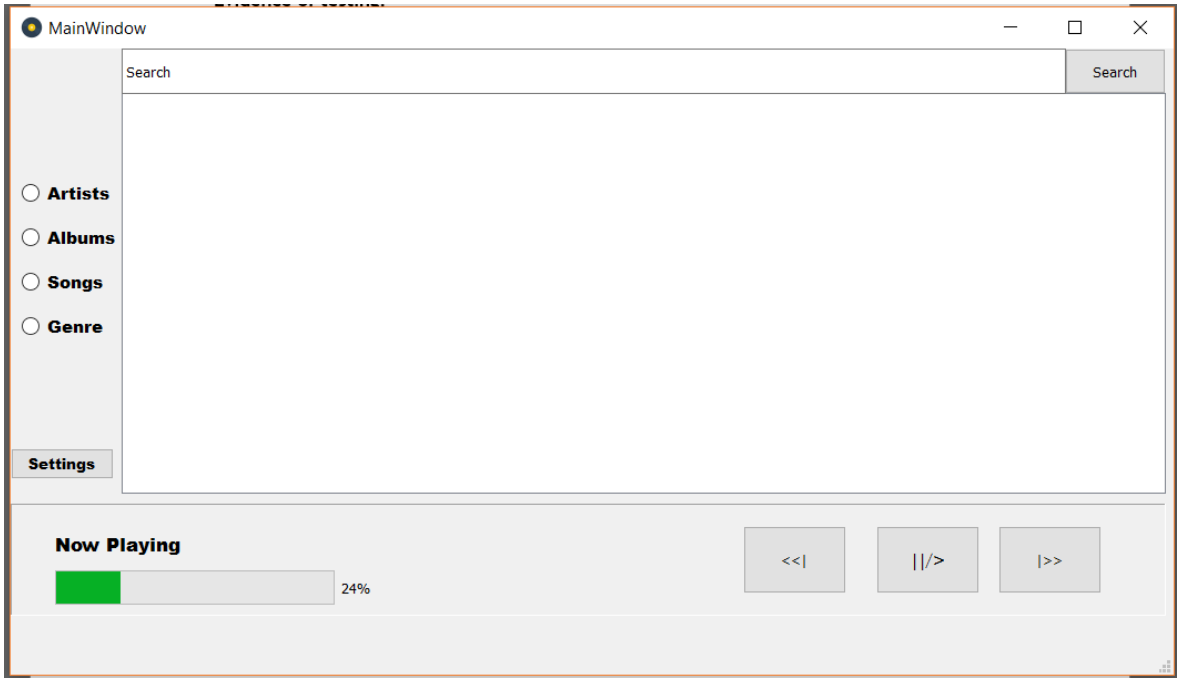
Evidence of testing:

Inputs:



Output:

```
C:\Python34\python.exe "C:/Coursework Snapshots/31-Jan-2017/Login Code.py"
jonesra jones
```

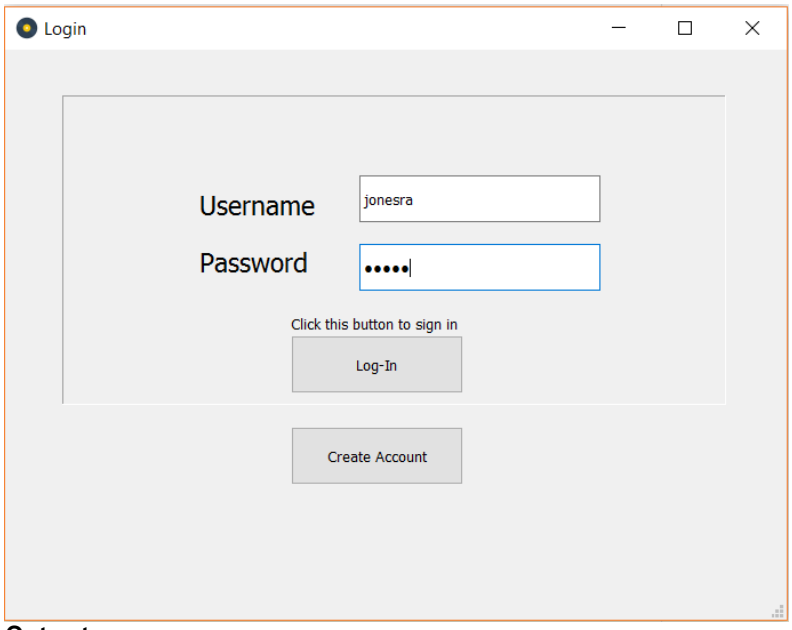


The two pictures show the inputs being inputted and the result of it. The fact that the window changed proves that the test is **Successful**

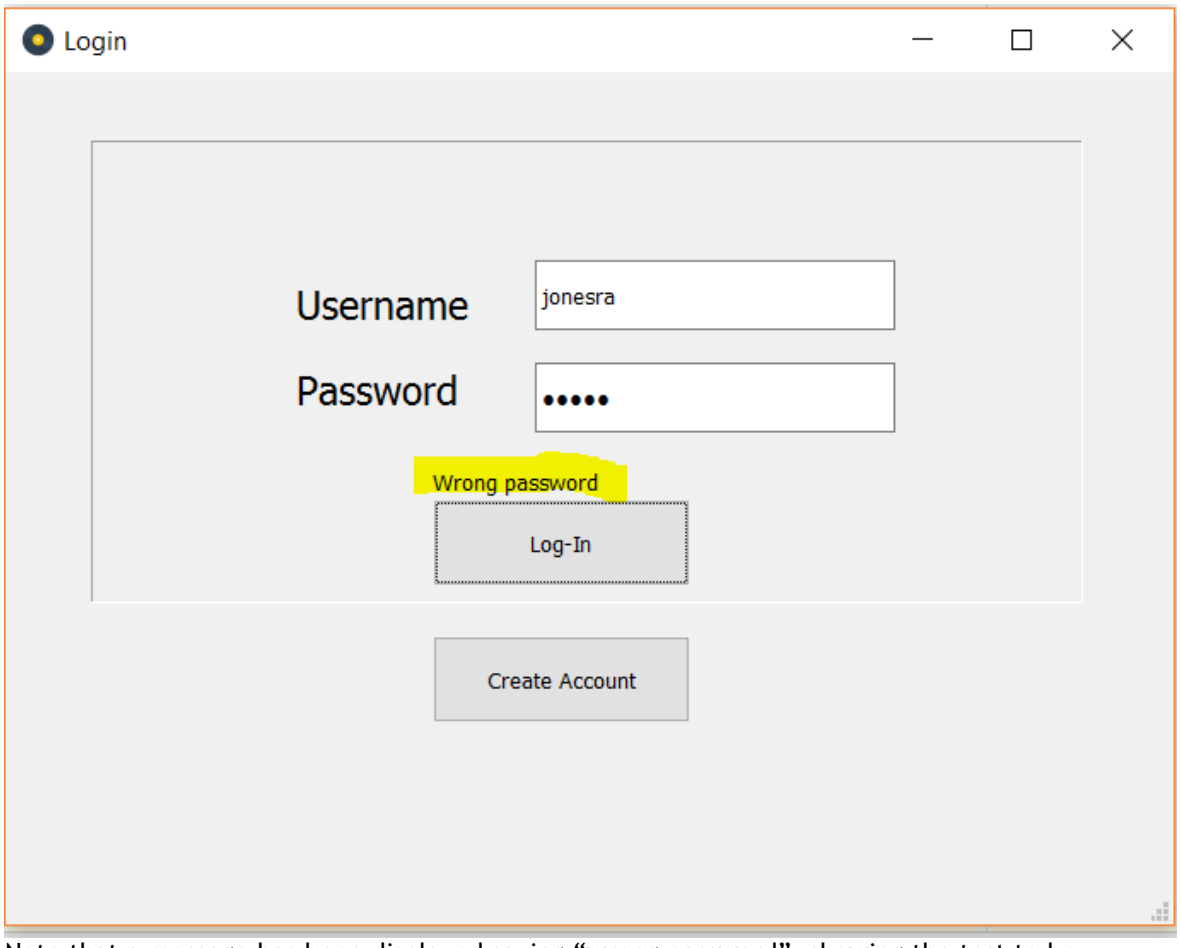
3.3.3.2 Inputting the correct Username and the incorrect Password

For this test, the username input will be “jonesra” and the password input will be “wrong”. The expected result is that the message “wrong password” will be displayed on the screen.

Inputs:



Outputs:

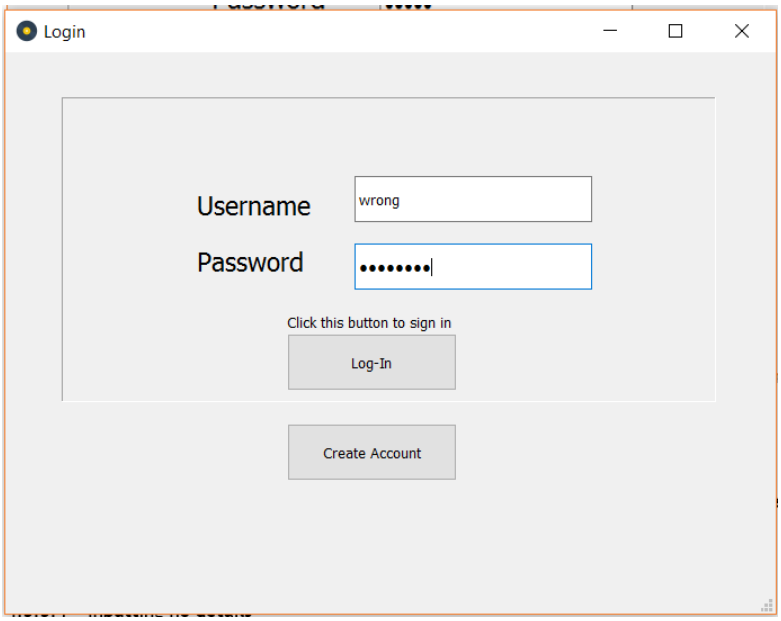


Note that a message has been displayed saying "wrong password", showing the test to be **Successful**.

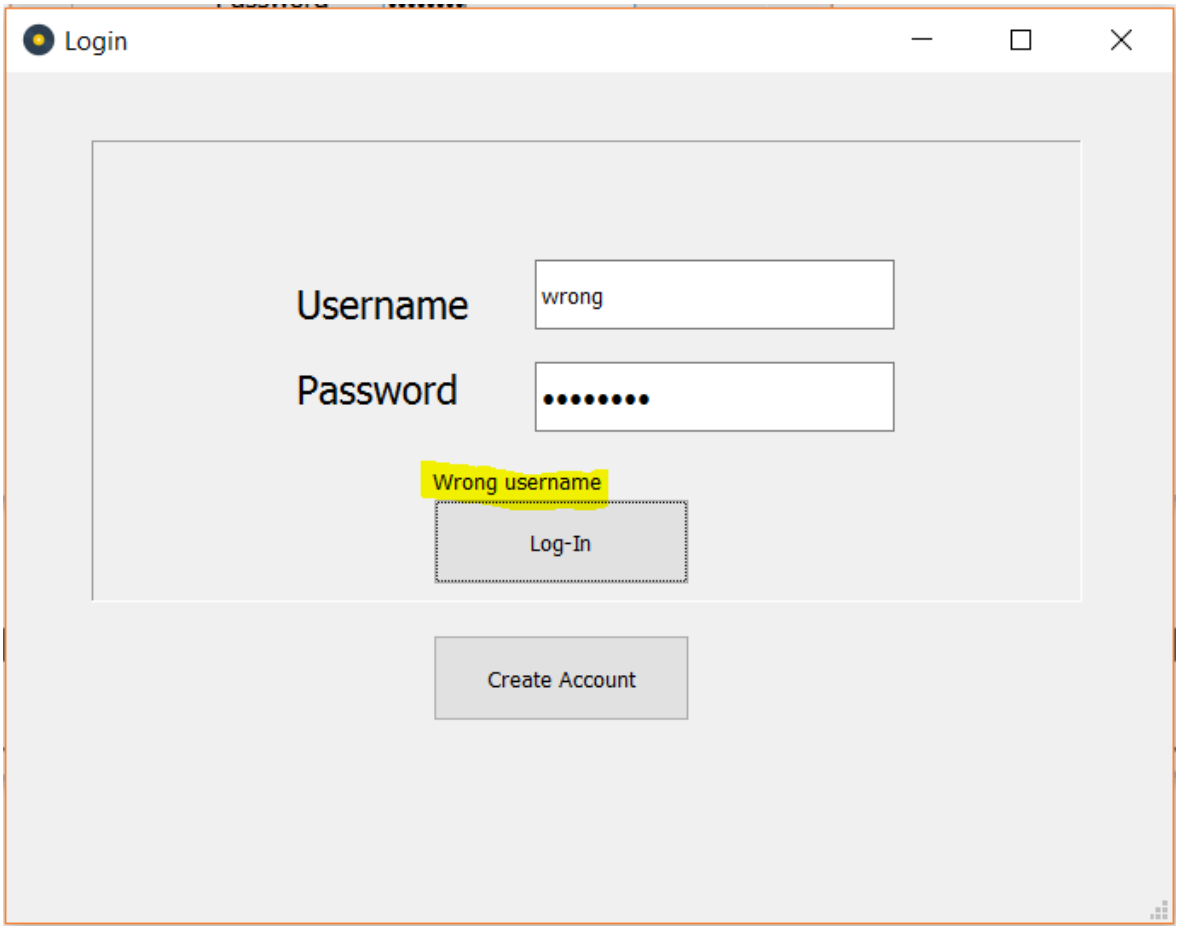
3.3.3.3 Inputting the correct Password and the incorrect Username

For this test, the username input will be "wrong" and the password will be "baritone". The expected result will be that the screen will display the "wrong username" message.

Inputs:



Outputs:

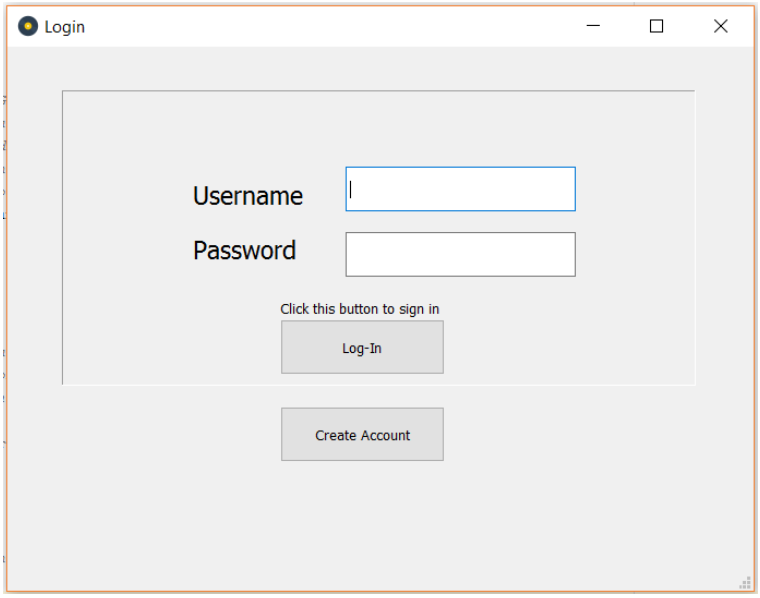


As the screenshot shows, the message “wrong username” was displayed, showing the test to be a **Success**

3.3.3.4 No username or password is inputted

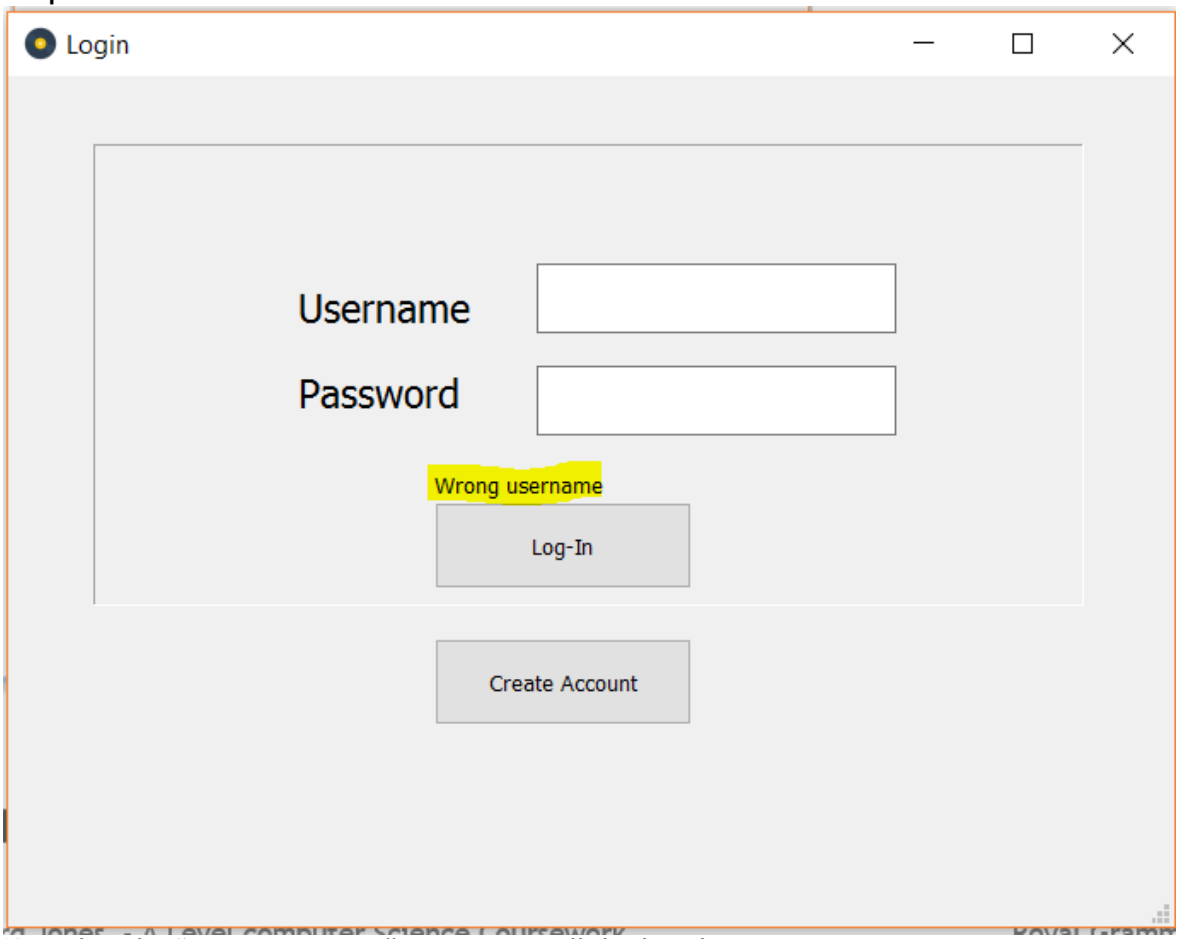
In this test, no username or password will be inputted. Just the login button will be activated. This section of the problem has not been coded yet, so the test is likely to show an error and no output. This will be fixed in the later iteration.

Inputs:



As the screenshot shows, there are no inputs in the text boxes.

Outputs:



Note that the "wrong username" message is still displayed.

However, there was no error message shown:

```
C:\Python34\python.exe "C:/Coursework Snapshots/31-Jan-2017/Login Code.py"
```

This test was obviously a **Failure.** Though no error was shown, a specific message will need to be shown in the final solution to meet all the objectives.

3.3.3.5 Conclusion

These tests show that the login problem is taking shape. However, the code still needs to check with details in the database and hash the password inputs. Furthermore, the testing showed that a specific message will need to be shown if the user doesn't input anything in the login screen. These changes will be addressed in the second iteration of the code.

3.3.4 2nd Iteration

The changes mentioned in the conclusion for the 1st Iteration need to be made in this iteration. These mainly focus around:

62. Checking the login inputs against the users table in the database, hashing the passwords to achieve this.
63. Display a specific message if the user doesn't input a username or password.

On top of this, the User's ID, admin permission, and username will have to be stored as variables to be accessed by later sections of code.

3.3.4.1 Code

```
def __init__(self, parent=None):
    QtGui.QMainWindow.__init__(self, parent)
    self.int_userID = 0
    self.int_admin = 0
    self.setupUi(self)
    self.btn_login.clicked.connect(self.login)
    self.txt_pass.returnPressed.connect(self.login)
    self.btn_crtacc.clicked.connect(self.createaccount)
```

The initialisation section of the class has been slightly improved. First, the username and password sections have been changed from 'self' variables to local variables in the login procedure. This was because the variables were not needed throughout the class or the rest of the code. Furthermore, the int_userID and int_admin variables were declared. These variables will store the id and administrator permissions of the user who is logged in for each session. The listener for the create account button was also set up to call the 'createaccount' procedure when it is clicked.

```
def login(self):
    str_username = self.txt_uname.text()  # draws variables from what is in the two line edit boxes
    str_password = self.txt_pass.text()
    str_passwordhash = hashlib.sha256(str_password.encode()).hexdigest()  # hashhes user's inputted password
    print(str_username, str_password)  # prints users inputs
    cur.execute("SELECT COUNT(Username) FROM Users WHERE UserName = ?", (str(str_username),))  # checks to see if
    # username is in the database
    if str_username == "" or str_password == "":  # Validates user's inputs to make sure that no fields
        # have been left empty
        self.lbl_info.setText("Field(s) Empty")  # Notifies user if any fields are empty
    elif cur.fetchone()[0] >= 0:  # Validates to see if username was found in database. Else, a message is displayed
        cur.execute("SELECT UserPassword FROM Users WHERE UserName = ?", (str(str_username),))  # retrieves password
        # belonging to username from database
        if cur.fetchone()[0] == str_passwordhash:  # validate user's password against password in database
            cur.execute("select UserID from Users where UserName=?", (str(str_username),)) # retrieves user's ID
            self.int_userID = int(cur.fetchone()[0]) # assigns user's ID to a variable
            cur.execute("SELECT Administrator FROM Users WHERE UserID = ?", (int(self.int_userID),))
            # retrieves user's administrator permissions from db
            self.int_admin = cur.fetchall()[0][0] # assigns user's admin. permissions to a variable
            print(self.int_userID)  # prints user's ID
            print(self.int_admin)  # prints user's admin. permissions
            LoginWindow.hide()  # closes login window
            MainWindow.showFullScreen()  # opens media player screen
        else:
            self.lbl_info.setText("Wrong Password")  # message displayed if password validation fails
    else:
        self.lbl_info.setText("Wrong Username")  # message displayed if username validation fails
```

This code has been extended to meet the changes mentioned earlier. As the annotations show, the code checks the user's login details against all of the login details in the database using SQL queries. The user's inputs are validated for three pieces of criteria: that the user has inputted both a username and a password, that the username is in the database and that the user's password corresponding to the inputted username is correct (using the hashing function to do so).

Furthermore, the code for opening the create account screen was coded:

```
def createaccount(self):
    CreateWindow.show()
```

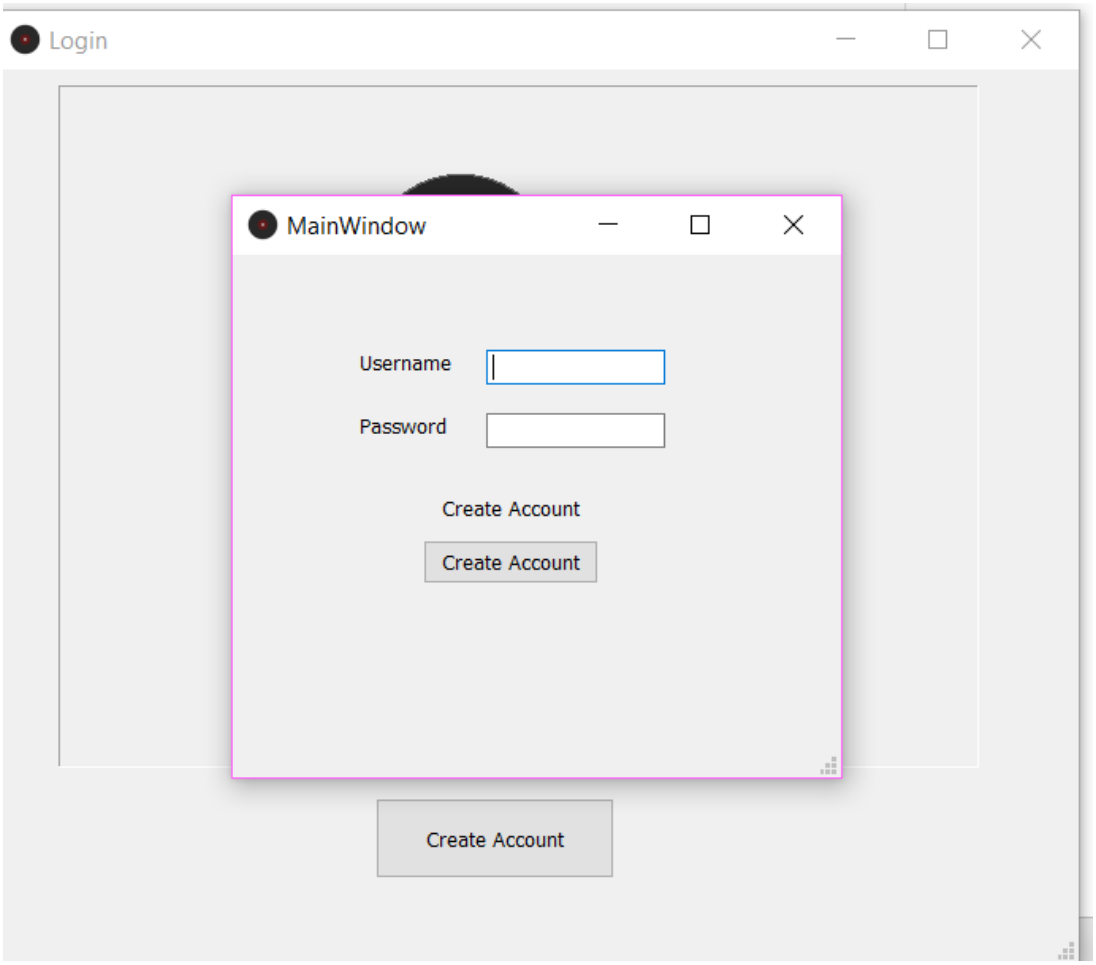
This procedure will simply open the create account window when it is run.

3.3.5 Testing for Iteration 02

3.3.5.1 Create Account Window

For this test, the create account button was activated. The expected output is that the create account window will show.

Output:

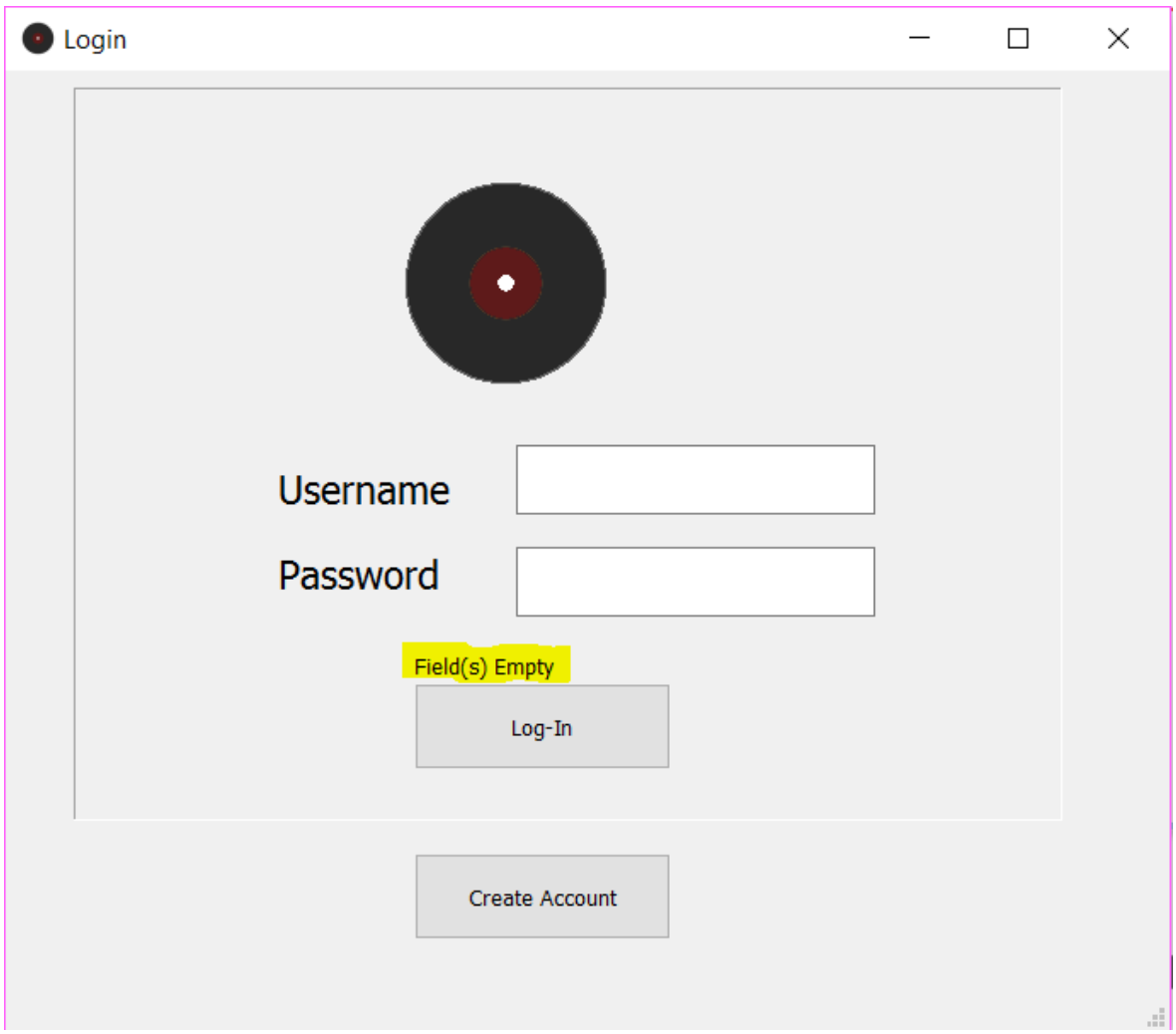


This screenshot shows that the create account window was opened. Therefore, the test was successful.

3.3.5.2 No username or password is inputted

For this test, the login button was activated without any inputs for the username or password. The expected output is that the message “Field(s) Empty” should be displayed in the lbl_info object.

Output:



As the screenshot shows, the message was shown with the given inputs. Therefore, the test was successful.

3.4 Create Account Window Class

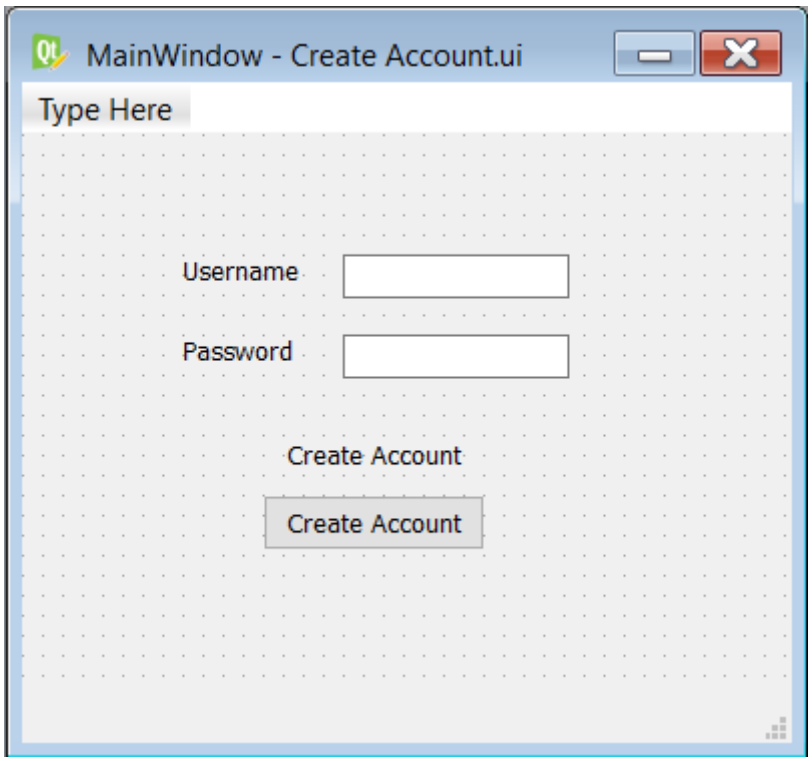
3.4.1 Project Decomposition

For this section of the code, the program will need to allow the user to input the username and password that they want for their new account. Then, through the click of a button, they can have their details validated for several criteria (length, whether or not it had letters and numbers in and whether the username already exists in the database). If the validation is passed, then the new account will be created in the database and the user will be taken back to the Login page. If the validation fails, an appropriate message will be shown.

3.4.2 Final Code

3.4.2.1 User Interface

Here is the final design of the 'Create Account' Window's user interface:



The interface contains all of the objects mentioned in the original mock-up of the design as well as some labels (for the username and password boxes) to increase accessibility to the user.

3.4.2.2 Initialisation Code

```
class CreateAccountWindowClass(QtGui.QMainWindow, ui_createAccountWindow):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)  # initialisation script run
        self.setupUi(self)  # user interface set up
        self.btn_crtacc.clicked.connect(self.create_account)  # runs create account routine if create account button
        # is pressed
```

As the annotations show, this initialisation script has three main functions. First, it runs initialisation script for PyQt. Second, it sets up the previously specified user interface file and third, it creates a listener for the create account button.

3.4.2.3 Create Account Function

```
def create_account(self):
    str_newusername = self.txt_username.text()  # retrieves user's input from textbox
    str_newpassword = self.txt_password.text()  # retrieves user's input from textbox
    if re.match(r"^.*[a-z]+.*$", str_newpassword) == None:
        self.lbl_info.setText("Please use letters in the password")
    elif re.match(r"^.*[0-9]+.*$", str_newpassword) == None:
        self.lbl_info.setText("Please use numbers in the password")
    elif re.match(r"^.{6,16}$", str_newpassword) == None:
        self.lbl_info.setText("Password must be 6-16 characters")
    else:
        print(str_newusername, str_newpassword)
        cur.execute("SELECT UserID FROM Users ORDER BY UserID DESC")
        # Retrieves the largest UserID from the Users table
        int_userid = int(cur.fetchone()[0]) + 1  # Forms the new userID by incrementing the largest UserID by 1
        print(int_userid)
        cur.execute("select COUNT(UserName) from Users where UserName=?", (str(str_newusername),))
        # Validates whether chosen username is already in the table
        int_usernamevalidate = int(cur.fetchone()[0])
        print(int_usernamevalidate)
        if int_usernamevalidate != 0:
            self.lbl_info.setText("Username is already in use")
        else:
            str_passwordhash = hashlib.sha256(str_newpassword.encode()).hexdigest()
            print(str_passwordhash)
            cur.execute("INSERT INTO Users VALUES(?,?,?,0)", (int(int_userid), str(str_newusername),
                                                              str(str_passwordhash)))
            con.commit()
            self.loginwindow()

def loginwindow(self):
```

This code is the main part of the create account class. The function takes the two inputs from the user and assigns them to variables in Hungarian notation. Then the password is passed through a set of validation conditions using regular expressions (regex). The validation checks to make sure the password is secure enough by making sure it contained letters and numbers and was between 6 and 16 characters. After the validation for the password, a new userID is created by incrementing the highest userID in the database by 1. Then the username is also validated to make sure there is not an identical username already in the database. This is done by counting number of fields in the Users tables where the username matches the user's input. If the outputted result is not 0, then an appropriate message is displayed.

3.5 Main Window Class

3.5.1 Project Decomposition

This class has a lot of different functions:

3.5.1.1 Songs Table

The code will need to provide a table object that can have multiple pieces of data placed into it. The nature of this data will be specified by the user (e.g. albums, artists, songs, playlists) using a button. The table will allow data to be selected and activated, an appropriate action will occur. If an album record is clicked, the songs in the album will be displayed. If an artist is clicked, the albums by that artists will be displayed. If a song is clicked, the song will be played.

Headings will also be used to help users to understand what they are looking at.

3.5.1.2 Playing Songs

When a song is selected from the table. The data of that song will be retrieved and assigned to separate variables. A label will assign the name of the song to itself to notify the user of the song that is currently playing. The plays number of the song will also be incremented in the database. The next songs in the table were then queued.

3.5.1.3 Playbar Buttons

When a user activates the pause button, the song that is currently playing will be paused. When a user activates the skip forward button, the next song in the queue will be played. When a user activates the skip backward button, the song that was previously playing in the queue will be played.

3.5.1.4 Search Function

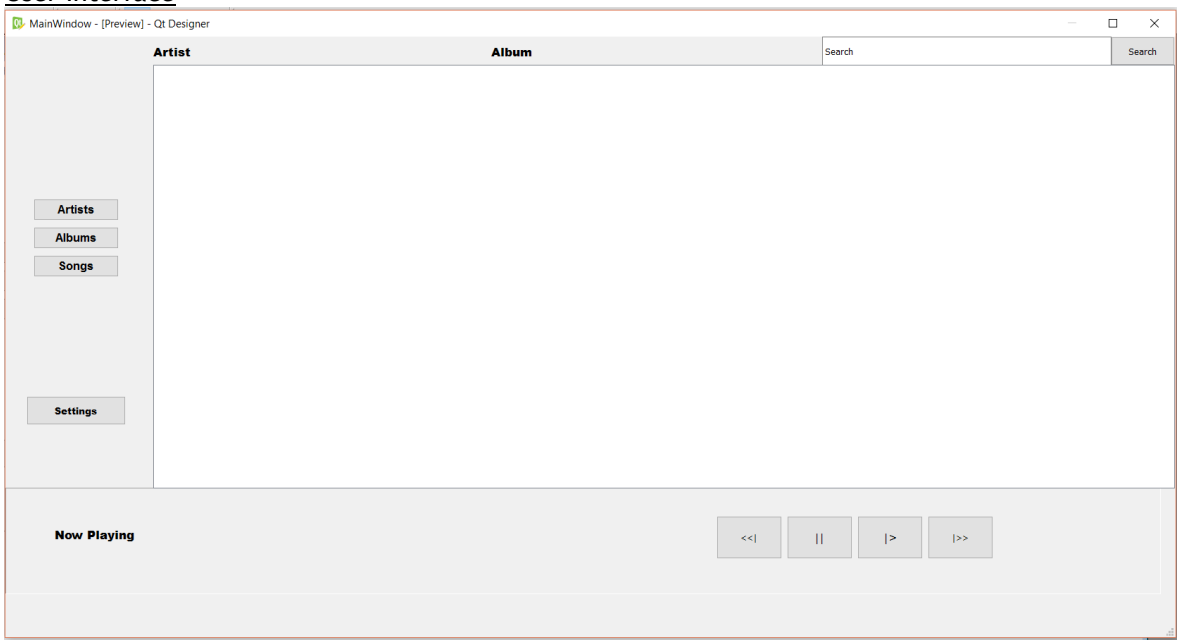
For this section. The user will be able to input a term that they want to search by and select what criteria they want to search (e.g. search by album, artist, song or playlist). Similar results should then be outputted in the song table.

3.5.2 Iterative Development

3.5.2.1 Table Navigation

To start the coding of the Main Window class, I attempted to implement the table navigation functionality. The table would need to be loaded with data from many different inputs, including when one of the navigation buttons were clicked by the user or when a record in the table was clicked. Using SQL, this problem can be simplified. Here is what the first few steps of the code looked like:

User Interface



Here is the first iteration of the user interface. The interface has been set up to include some of the objects required for the final solution; such as the library navigation buttons (labelled artists, albums, songs), the settings buttons, the playbar buttons and the library table. The final code will hopefully also have a more stylish design that would be more enticing to the end user.

Initialisation Code

```
def __init__(self, parent=None):
    mixer.init()
    QtGui.QMainWindow.__init__(self, parent)
    self.setupUi(self)
    self.data = []
    self.model = QtGui.QStandardItemModel(self)
    self.tbl_songs.setModel(self.model)
    self.btn_settings.clicked.connect(self.settings)  # settings button programmed
    self.btn_albums.clicked.connect(self.albums)
    self.btn_songs.clicked.connect(self.songs)
    self.btn_artists.clicked.connect(self.artists)
    self.btn_play.clicked.connect(self.play)
    self.btn_pause.clicked.connect(self.pause)
```

The initialisation code is similar to the other two classes I have already programmed. The code includes the initialisation of the user interface, the declaring of self variables and the setting up of listeners for the buttons of the user interface. The table navigation utilises the albums, songs and artists button listeners.

Table Navigation Buttons

Before the methods for the table navigation buttons could be implemented, the load_data method

```
def songs(self):
    s = "SELECT * FROM Songs"
    cur.execute(s)
    self.data = cur.fetchall()
    self.load_data()
    self.tbl_songs.doubleClicked.connect(self.play_song)

def albums(self):
    s = "SELECT * FROM Albums"
    cur.execute(s)
    self.data = cur.fetchall()
    self.load_data()
    self.tbl_songs.doubleClicked.connect(self.albums_to_songs)

def artists(self):
    s = "SELECT * FROM Artists"
    cur.execute(s)
    self.data = cur.fetchall()
    self.load_data()
    self.tbl_songs.doubleClicked.connect(self.artists_to_albums())
```

These three methods are each called when their respective buttons are clicked by the user. Each method starts by select all of the records from their respective tables through the use of an SQL query. The query is executed and the results are fetched and loaded into the songs table using the load data method. Then, each method contains a listener for the songs table in case its fields are double clicked. When a record is clicked, if the user is looking at songs (meaning they have clicked the song button and run the songs method), the play song method is run from that listener. If the user is looking at albums (meaning they have clicked the albums buttons and run the albums method), the albums to songs method is run.

This is continued by the albums_to_songs and artists_to_albums methods:

```
def artists_to_albums(self):
    self.retrieve_row()
    artist_id = row_details[0]
    cur.execute("SELECT * FROM Albums WHERE ArtistID=?", (str(artist_id)), )
    self.data = cur.fetchall()
    self.load_data()
    self.tbl_songs.doubleClicked.connect(self.albums_to_songs())

def albums_to_songs(self):
    self.retrieve_row()
    album_id = row_details[0]
    cur.execute("SELECT * FROM Songs WHERE AlbumID=?", (str(album_id),))
    self.data = cur.fetchall()
    self.load_data()
    self.tbl_songs.doubleClicked.connect(self.play_song)
```

These two methods start by retrieving the row that has been selected by the user. In the case of the artists_to_albums method, the albums by the artist selected by the user will be retrieved through the execution of an SQL query. The results of the query are (once again) loaded into the songs table. Separate listeners for interaction with the songs table.

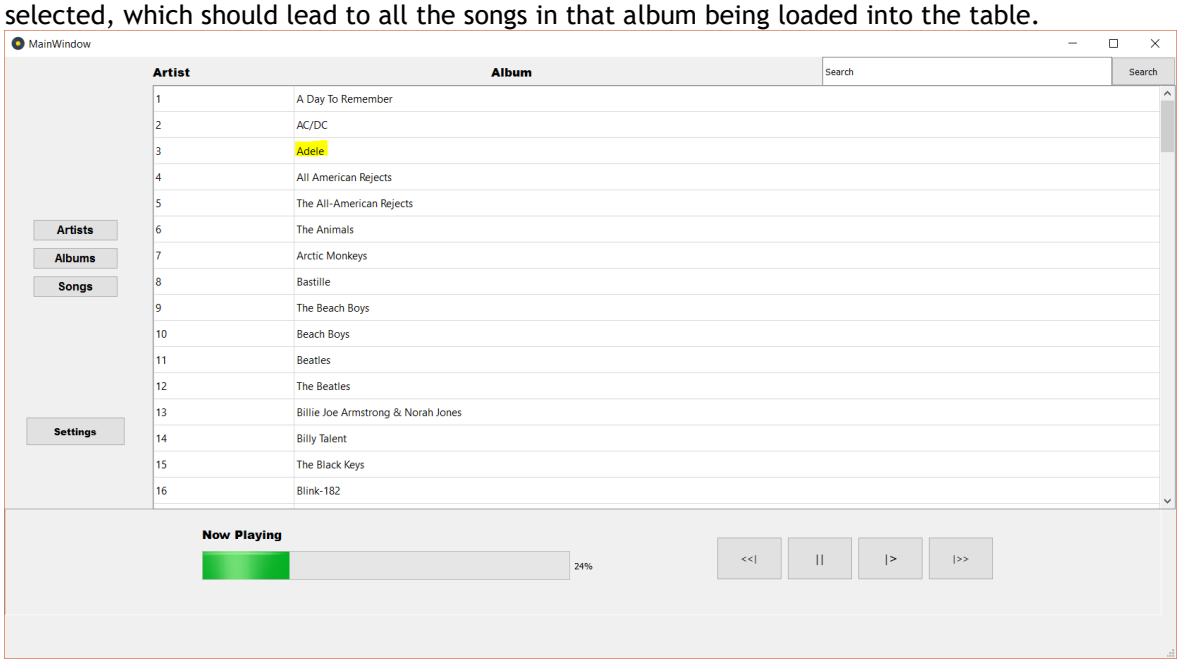
In conclusion, the methods constantly retrieve new data for the rows from SQL queries and set up specific listeners for interaction with the songs table depending on the user's inputs. This functionality will now be thoroughly tested

3.5.3 Iterative Testing

3.5.3.1 Navigate from an artist to an album to a song.

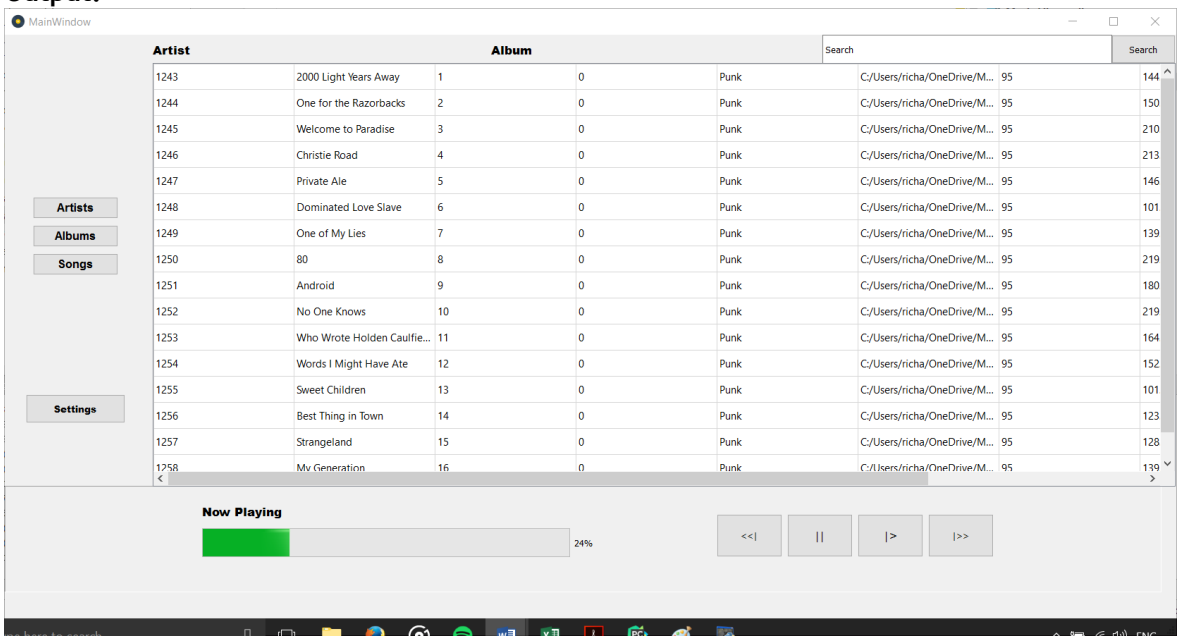
Input:

First, the artist button will be clicked, an artist will then be selected in the table which should lead to all of the albums by that artist being loaded into the table. Then, an album will be selected, which should lead to all the songs in that album being loaded into the table.



The Adele will be selected from the table.

Output:



This was not the expected result from the input. It appears that the results shown are all of the songs from an unknown album.

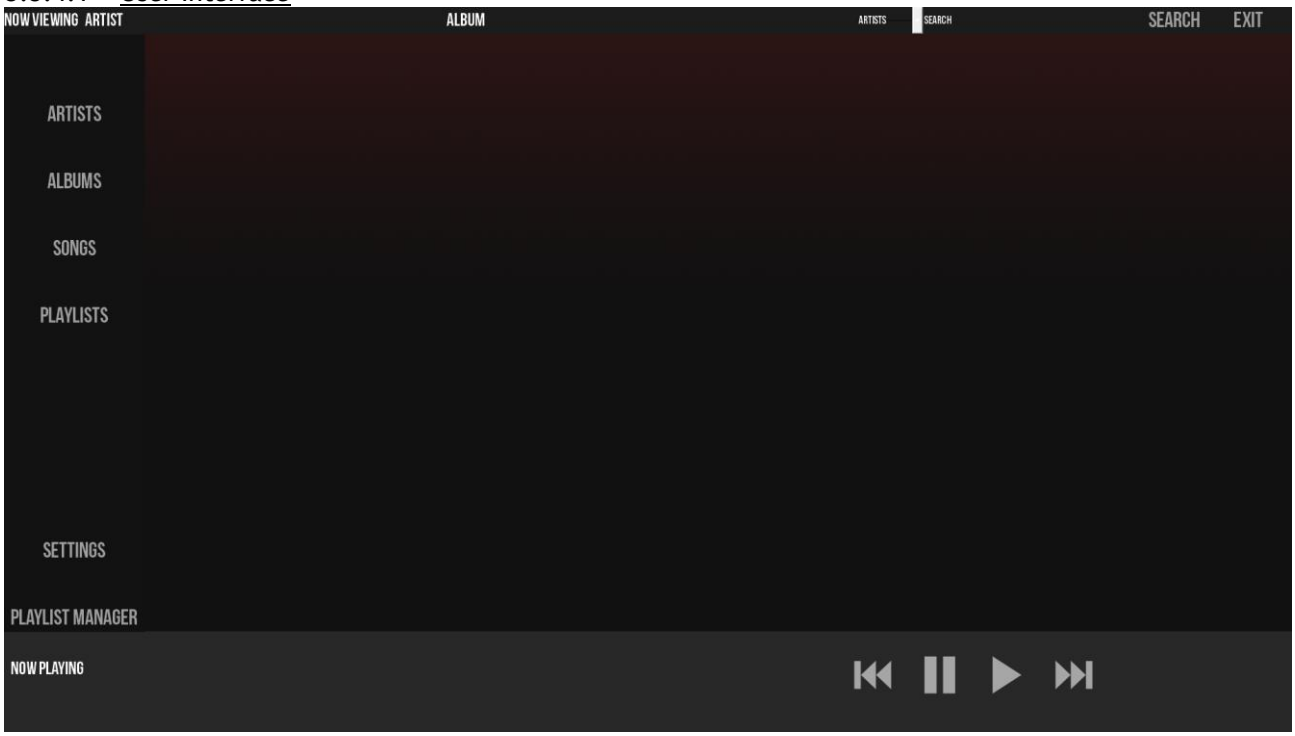
Upon researching the functions and the implementation in my code, I realised that only one listener can be set up for each PyQt object. The code was constantly calling the method from the first listener that was set up, which explains why the results are so different to what is expected.

In terms of how to fix this issue, I will create a new column in the songs, albums, artists and playlists tables in the database. Then only one listener will be set up for the songs table and will call a function containing conditions that discriminate between the values in these new columns

and run the appropriate method. On top of fixing the problem, this adaption will also make the code more efficient.

3.5.4 Final Code

3.5.4.1 User Interface



In order to make the solution more attractive and comparable to an End-User product. I took some advice on the design of Main Window Screen from an Art Designer (Oliver Munby). I was advised to consider flat textured buttons with shades of grey and red. He also recommended I include some gradient effects for the background. I made an attempt to include this in the final design.

3.5.4.2 Initialisation Code

Here is the final look of the initialisation code:

```
class MainWindowClass(QtGui.QMainWindow, ui_mainWindow):
    def __init__(self, parent=None):
        mixer.init()  # Starts mixer from pygame for music playback
        QtGui.QMainWindow.__init__(self, parent)  # pyqt initialisation code
        self.setupUi(self)  # pyqt user interface setup
        self.lst_data = []  # self variable declared, holds retrieved data from tbl_songs
        self.model = QtGui.QStandardItemModel(self)  # declares model used for tbl_songs
        self.tbl_songs.setModel(self.model)  # tbl_songs model set
        self.btn_settings.clicked.connect(self.settings)  # settings button programmed
        self.btn_albums.clicked.connect(self.albums)  # Albums button programmed
        self.btn_songs.clicked.connect(self.songs)  # Songs button programmed
        self.btn_artists.clicked.connect(self.artists)  # Artists button programmed
        self.btn_play.clicked.connect(self.play)  # Play button programmed
        self.btn_pause.clicked.connect(self.pause)  # Pause button programmed
        self.btn_playlists.clicked.connect(self.playlists)  # playlists button programmed
        self.btn_search.clicked.connect(self.search)  # search button programmed
        self.btn_forward.clicked.connect(self.skipForward)  # skip forward button programmed
        self.btn_back.clicked.connect(self.skipBackward)  # skip backward button programmed
        self.tbl_songs.doubleClicked.connect(self.retrieve_row)  # Table interaction programmed
        self.btn_plylstmngr.clicked.connect(self.playlistManager)  # playlist manager button programmed
        self.btn_exit.clicked.connect(self.exit)  # exit button programmed
        self.queue_next = []  # self list declared. Holds the details of the next songs in the queue
        self.stack_prev = []  # self list declared. Holds the details of the previous song played
        self.int_songSkip = 0  # self variable declared. Changes to one when the skip forward button is activated
        self.bool_songsQueued = False  # self variable declared. Changes when songs have been queued.
        self.lst_nowPlaying = []  # self list declared. Holds the details of the song that is being played.
```

3.5.4.3 Songs Table

For the table selection code, this code has been amended to fix an error found in the iterative testing. The error happened because there were too many tblsongs.doubleClicked commands which had different responses. To fix this, I introduced a new field in the artists, albums, songs and playlists table. A different value was held for each individual table. Now, only one connect function can be used with several if statements discriminating this new field. Here is the code:

First, the album, songs and artist buttons were programmed:

```
def songs(self):
    self.lbl_artist.setText("Artist:")
    self.lbl_album.setText("Album:")
    str_query = "SELECT * FROM Songs ORDER BY SongName"  # SQL statement to retrieve all of the songs in the Music Library
    cur.execute(str_query)  # Executes SQL statement
    self.lst_data = cur.fetchall()  # Results from statement are fetched
    self.load_data()  # Data found from statement is loaded into table widget
    self.hide_song_columns()

def albums(self):
    self.lbl_artist.setText("Artist:")
    self.lbl_album.setText("Album:")
    str_query = "SELECT * FROM Albums ORDER BY AlbumName ASC"  # SQL statement to retrieve all albums in the music library
    cur.execute(str_query)  # Statement is executed
    self.lst_data = cur.fetchall()  # Data from statement is fetched
    self.load_data()  # Data from statement is loaded into table widget
    self.hide_album_columns()

def artists(self):
    self.lbl_artist.setText("Artist:")
    self.lbl_album.setText("Album:")
    str_query = "SELECT * FROM Artists ORDER BY ArtistName ASC"  # SQL statement to retrieve all artists in the music library
    cur.execute(str_query)  # Statement is executed
    self.lst_data = cur.fetchall()  # Data from statement is fetched
    self.load_data()  # Data from statement is loaded into table widget
    self.hide_artist_columns()

def playlists(self):
    self.lbl_artist.setText("Artist:")
    self.lbl_album.setText("Album:")
    print(LoginWindow.int_userID)
    cur.execute("SELECT * FROM Playlists WHERE UserID=? OR UserID=0", (int(LoginWindow.int_userID),))
    self.lst_data = cur.fetchall()
    self.load_data()
```

Each of these procedures follow the same formula. First, the labels used to assist user navigation were reset. Then, a suitable SQL query was used to retrieve all of the items in each table. The data from the queries were then fetched assigned to a variable. The load_data function was then called. The hide_?_columns function was also used to tidy up the output to be seen by the user.

Here is the load_data procedure:

```
def load_data(self):
    if len(self.lst_data) > 0:  # If data table already has items in it, the table is cleaned
        for i in range(len(self.lst_data) - 1, -1):
            print("removing row", i)
            self.lst_data.pop(i)
            self.model.removeRow(index.row(self.lst_data[i]))
    self.model = QtGui.QStandardItemModel(self)  # new model declared from pyqt
    self.tbl_songs.setModel(self.model)  # model set for tbl_songs
    for row in self.lst_data:
        items = [QtGui.QStandardItem(str(field)) for field in row]
        self.model.appendRow(items)
```

And here are the hide_?_column procedures:

```
def hide_song_columns(self):
    for i in range(8):
        columns_to_hide = (0, 3, 4, 5, 6, 7, 8, 9)
        self.tbl_songs.hideColumn(columns_to_hide[i])
        i += 1

def hide_album_columns(self):
    for i in range(3):
        columns_to_hide = (0, 2, 3)
        self.tbl_songs.hideColumn(columns_to_hide[i])
        i += 1

def hide_artist_columns(self):
    for i in range(2):
        columns_to_hide = (0, 2)
        self.tbl_songs.hideColumn(columns_to_hide[i])
        i += 1
```

The procedures use iteration to hide all of the columns that the user does not need to see (such as the directory that the song are in). The specific columns are placed in an array.

The code must also allow the user to select certain fields in the table and perform a specific action, e.g. when an album is clicked in the table, the table will show all of the song in that album. This first piece of code is the first procedure used when the user double-clicks a field in the table:

```
def retrieve_row(self):
    lst_rowdetails = []  # variable declared, holds the information retrieved from the row.
    tbl_rowdetails = self.tbl_songs.selectionModel().selectedRows()  # row details retrieved from table model
    for index in tbl_rowdetails:  # iteration to convert unusable data in tbl_rowdetails to usable in lst_rowdetails
        row = index.row()  # row of table found
        print(row)
        lst_rowdetails = self.lst_data[row]  # Retrieves data from selected row
    print(lst_rowdetails[-1])
    if lst_rowdetails[-1] == 0:  # If the row bears the type of 0, it is a song and is played
        self.bool_songsQueued = False  # songs are no longer queued so variable is changed
        self.queue_next = []  # next queue is reset
        self.stack_prev = []  # previous stack is reset
        self.play_song(lst_rowdetails)
    elif lst_rowdetails[-1] == 1:  # If the row bears the type of 1, it is a artist and the artist's albums are found
        self.artists_to_albums(lst_rowdetails)
    elif lst_rowdetails[-1] == 2:  # if the row bears the type of 2, it is an album and the album's songs are retrieved
        self.albums_to_songs(lst_rowdetails)
    elif lst_rowdetails[-1] == 3:
        self.playlists_to_songs(lst_rowdetails)
```

This is the procedure which shows the new adjustments that were made to this class.

This code also points to four functions, artists_to_albums, albums_to_songs, playlist_to_songs and play_song, here are the first three:

```
def artists_to_albums(self, details):
    int_artist_id = details[0]  # Artist ID is retrieved from selected row
    str_artist_name = details[1]
    self.lbl_artist.setText("Artist: " + str_artist_name)
    cur.execute("SELECT * FROM Albums WHERE ArtistID=? ORDER BY AlbumName ASC", (str(int_artist_id),))
    # SQL statement to retrieve all albums
    # by selected artist
    self.lst_data = cur.fetchall()  # Data from statement is fetched
    self.load_data()  # Data from statement is loaded into table widget
    self.hide_album_columns()

def albums_to_songs(self, details):
    int_album_id = details[0]  # Album ID is retrieved from selected row
    str_album_name = details[1]
    self.lbl_album.setText("Album: " + str_album_name)
    cur.execute("SELECT * FROM Songs WHERE AlbumID=? ORDER BY TrackNumber", (str(int_album_id),))
    # SQL statement to retrieve all songs
    # from selected album
    self.lst_data = cur.fetchall()  # Data from statement is fetched
    self.load_data()  # Data is loaded into table widget
    self.hide_song_columns()

def playlists_to_songs(self, details):
    int_playlist_id = details[0]
    cur.execute("SELECT PlaylistSongs.SongID, Songs.TrackNumber, Songs.SongName, Songs.SongRating, Songs.Genre,"
                " Songs.FileLocation, Songs.AlbumID, Songs.Length, Songs.Plays, Songs.Type FROM PlaylistSongs "
                "INNER JOIN Songs ON PlaylistSongs.SongID = Songs.SongID WHERE PlaylistSongs.PlaylistID = ?"
                , (int(int_playlist_id),))
    self.lst_data = cur.fetchall()  # Data from statement is fetched
    self.load_data()  # Data is loaded into table widget
    self.hide_song_columns()
```

Once again, the functions follow the same formula to one another. First, the relevant information of the selected row is retrieved, typically the ID. For the first two functions, the navigation are labels are changed to either the artist or album name. Then an SQL query is used to fetch the relevant data where the ID number is identical. For the playlist_to_songs function, the sql query uses an inner join query to achieve a result similar to the other two functions. The data is then loaded into the table.

3.5.4.4 Playing Songs

The fourth function, play_song, differs slightly:

```
def play_song(self, details):
    print(details)
    self.lst_nowPlaying = details
    str_song_name = details[2]
    str_location = (details[4])  # Song location is retrieved from selected row
    print(str_location)
    int_song_id = details[0]  # Song ID is retrieved from selected row
    int_plays = details[7] + 1  # Number of song plats is retrieved from selected row and incremented
    cur.execute("UPDATE Songs SET Plays=? WHERE SongID=?", (int(int_plays), int(int_song_id)))
    # Updates number of plays of
    # the song in the database
    con.commit()  # Commits previous statement
    mixer.music.load(str_location)  # Selected music is loaded into pygame mixer
    mixer.music.play()  # Selected music is played through the mixer
    self.lbl_nowplaying.setText("Now Playing: " + str_song_name)
    if self.bool_songsQueued == False:
        self.queueSongs()
    self.refreshmostplayedplaylist()
    self.next_song()
```

This function retrieves the relative details from its parent procedures, including the song name, its location in storage, its ID and its number of plays. The song's number of plays is incremented and the record is updated with this new value. After this update query has been committed, the song's location is used to load the song into the mixer and then player. Then, the songsQueued variable is checked. If the songs have not been queued, the queuesongs procedure will be run.

After this if statement, the most played playlist is refreshed to include the updated statistics. Then the next song procedure is run.

Here is the code for the queuesongs procedure:

```
def queueSongs(self):
    self.queue_next = []  # next queue cleared
    flag = False  # iteration condition set
    i = 1  # counter set
    while flag == False:
        row_details = []  # row details is reset
        table_row_details = self.tbl_songs.selectionModel().selectedRows()  # selected row is retrieved
        for index in table_row_details:
            row = index.row() + i  # row is incremented by counter
            print(row)
            row_details = self.lst_data[row]  # data from retrieved row taken.
            print(row_details[2])
            self.queue_next.append(row_details)  # next song is queued
            if row_details[0] == self.lst_data[-1][0]:  # if the most recently retrieved row is the last row in the table
                flag = True  # loop is broken
            i = i + 1  # counter is incremented
    self.bool_songsQueued = True
```

This section of code is reused from the retrieve row procedure, with some changes. Iteration is used to retrieve every row under the row that the user clicked.

Here is the code to refresh the most played playlist:

```
def refreshmostplayedplaylist(self):
    cur.execute("DELETE FROM PlaylistSongs WHERE PlaylistID = 1")
    con.commit()
    cur.execute("SELECT * FROM Songs WHERE Plays > 0 ORDER BY Plays DESC")
    lst_playlistItems = cur.fetchall()
    print(lst_playlistItems)
    cur.execute("SELECT PlaylistSongsID FROM PlaylistSongs ORDER BY PlaylistSongsID DESC")
    latestPlaylistSongsID = cur.fetchone()[0] + 1
    for i in range(20):
        cur.execute("SELECT PlaylistSongsID FROM PlaylistSongs ORDER BY PlaylistSongsID DESC")
        PlaylistSongsID = latestPlaylistSongsID + i
        print(PlaylistSongsID)
        songID = lst_playlistItems[i][0]
        cur.execute("INSERT INTO PlaylistSongs VALUES(?,1,?)", (int(PlaylistSongsID), int(songID),))
        con.commit()
```

To start this procedure, the songs currently related to the most played playlist must be deleted. An SQL query is used to remove all fields in the PlaylistSongs link table where the playlist ID is that of the Most Played Playlist (which is 1). After the query is committed, another query is executed to select all songs that have been played (where plays>0), ordered by plays highest to lowest. The results of this query are then set to the lst_playlistItems variable. Then, the highest current PlaylistSongsID is retrieved from the database. An iterative loop is then performed to insert the first 20 fields in the lst_playlistItems. The new PlaylistSongsID for each iteration are created by adding the previously highest playlistsongID to the counter for the iteration. At each iteration, the changes are committed.

Here is the code for the nextsong procedure. This procedure is used to detect when the next song must be loaded into the mixer:

```
def next_song(self):
    while mixer.music.get_busy() == True and self.int_songSkip == 0:
        QtCore.QCoreApplication.processEvents()
    if self.int_songSkip == -1:
        self.int_songSkip = 0
        self.queue_next.insert(0, self.lst_nowPlaying)
        next_song = self.stack_prev[0]
        self.stack_prev.remove(next_song)
        print("Song changing to " + next_song[2])
        self.play_song(next_song)
    else:
        self.int_songSkip = 0
        self.stack_prev.insert(0, self.lst_nowPlaying)
        next_song = self.queue_next[0]
        self.queue_next.remove(next_song)
        print("Song changing to " + next_song[2])
        self.play_song(next_song)
```

Firstly, this procedure includes a constant while loop, meaning that the code will keep repeating until the mixer is no longer playing a song or until the user skips the song. As this loop spends most of its time activated, a line of code had to be used to allow the PyQt interface to function. This is because python is an interpretive language and (normally) only processes one line at a time. Without the line "QtCore.QCoreApplication.processEvents()", the program would crash as soon as a PyQt object was clicked or selected.

3.5.4.5 Playbar Buttons

As the initialisation code shows; the play, pause and skip buttons have been programmed. Here are the procedures that they lead to:

Pause and Play

```
def play(self):
    mixer.music.unpause()   # Music in mixer is paused

def pause(self):
    mixer.music.pause()   # Music in mixer is resumed
```

These two simple procedures allow the music to be played or paused, depending on which button is activated.

Skip Forward & Backward

```
def skipForward(self):
    self.int_songSkip = 1

def skipBackward(self):
    self.int_songSkip = -1
```

These two procedures allow either the next or previous song to be played, depending on which button was pressed. The procedures change the state of the songSkip variable. This change is detected by the nextsong procedure (which have already been mentioned) which actually skips the song backwards or forwards.

3.5.4.6 Search Function

```
def search(self):
    self.searchTerm = self.txt_search.text()  # retrieves user input to search with
    self.searchTerm = str(self.searchTerm + "%")  # adds wildcard symbol to string
    self.searchTable = self.drp_search.currentText()  # retrieves input from dropdown box
    if self.searchTable == "Songs":  # if the dropdown menu was on songs
        cur.execute("SELECT * FROM Songs WHERE SongName LIKE ?", (self.searchTerm,))  # selects all songs similar to the user's input
        self.lst_data = cur.fetchall()  # fetches result from query
        self.load_data()  # loads relevant data into table
        self.hide_song_columns()
    elif self.searchTable == "Albums":  # if the dropdown menu was on albums
        cur.execute("SELECT * FROM Albums WHERE AlbumName LIKE ?", (self.searchTerm,))  # selects all albums similar to the user's input
        self.lst_data = cur.fetchall()  # fetches result from query
        self.load_data()  # loads relevant data into table
        self.hide_album_columns()
    elif self.searchTable == "Artists":  # if the dropdown menu was on artists
        cur.execute("SELECT * FROM Artists WHERE ArtistName LIKE ?", (self.searchTerm,))  # selects all artists similar to the user's input
        self.lst_data = cur.fetchall()  # fetches result from query
        self.load_data()  # loads relevant data into table
        self.hide_artist_columns()
    else:  # therefore, the dropdown menu was on playlists
        cur.execute("SELECT * FROM Playlists WHERE Playlist LIKE ?", (self.searchTerm,))  # selects all playlists similar to the user's input
        self.lst_data = cur.fetchall()  # fetches result from query
        self.load_data()  # loads relevant data into table
```

This code starts by retrieving two of the user's inputs: first, the user's search term, in string and second, the search criteria (e.g. album, artist, song or playlist) from a dropdown box. Multiple if statements are then used depending on the inputted search criteria, each statement uses a query to retrieve items similar to the user's search term in the table corresponding to the user's search criteria. This procedure uses multiple wildcards in the LIKE queries to retrieve an estimated result.

3.5.4.7 Exit/Transition Codes

Settings Window

```
def settings(self):
    if LoginWindow.int_admin == 0:
        SettingsWindow.fr_admin.hide()
    else:
        SettingsWindow.loadUserData()
    SettingsWindow.show()  # Settings window is shown
```

This section of code performs two main tasks. First it validates whether the user who is currently using the program is an admin, using encapsulation from the LoginWindow object. If the user is an admin, the admin section of the settings window is shown and its user data is loaded. Otherwise, the admin section is hidden. After this validation, the Settings Window is shown to the user.

Playlist Manager Window

```
def playlistManager(self):
    PlaylistWindow.showFullScreen()
    print(LoginWindow.int_userID)
    cur.execute("SELECT * FROM Playlists WHERE UserID=?", (int(LoginWindow.int_userID),))
    PlaylistWindow.playlistData = cur.fetchall()
    PlaylistWindow.load_playlist_data(PlaylistWindow.playlistData)
    PlaylistWindow.load_song_data()
```

This section of code's main purpose is to show the playlist manager window. It set's the window's behaviour to take up all of the screen. Then, a query is performed to retrieve all of the playlists that the user owns (where the userID in the playlist table is identical to the current user's ID). The results of this query are then fetched and loaded into the playlist table.A procedure is then called to load the song data into the songs table.

Program Exit Code

```
def exit(self):
    os._exit(0)
```

This procedure is activated when the exit button is clicked (see the initialisation code for this class) and simply closes down the entire python program. This decision was made for two reasons: first, the program was made full screen and this button was required to allow the user to close the program and second, if the program was closed without any exit code, any music

playing would keep playing. Closing the window only closes the Qt or UI section of the program, the mixer must also be closed to fully end the program.

3.6 Settings Window Class

3.6.1 Project Decomposition

3.6.1.1 Import Settings

For this section of the class. The program must accept an directory as a string from the user. This input is then validated to make sure that the directory actually exists. If the validation If the validation is successful, then the selected directory will be scanned and all of the mp3 files in the directory will be located. For each of these files, the ID3 tags will be read and inputted into the database. If a new artist is found, it is added to the artists table, is a new album is found, it is added to the albums table and when a new song is found, it is added to the songs table. If any of the information cannot be found, it is added as a null value into the record.

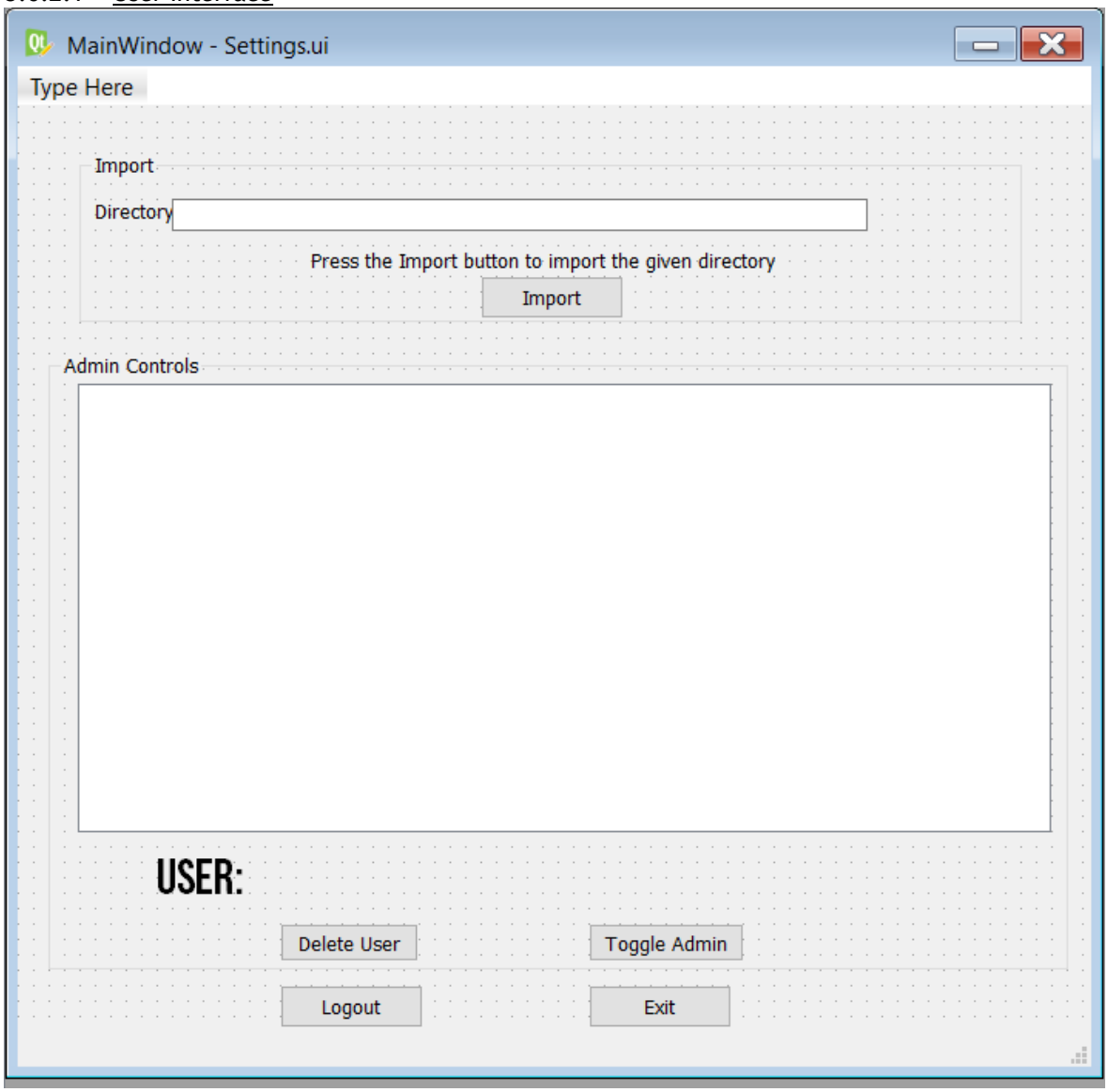
3.6.1.2 Admin Settings

For this section of the class. The code will provide a table that will contain all of the users (excluding the user currently using the program). The user will have the ability to select fields from this table and perform tasks. These tasks include: toggling the selected user's administrator privileges and deleting a selected user's account.

3.6.1.3 Other operations

3.6.2 Final Code

3.6.2.1 User Interface



3.6.2.2 Initialisation Code

Here is the initialisation code of the Settings Class:

```
class SettingsWindowClass(QtGui.QMainWindow, ui_settingsWindow):
        def __init__(self, parent=LoginWindowClass):
            QtGui.QMainWindow.__init__(self, parent)  # initialisation of pyqt UI
            self.setupUi(self)  # UI set up
            self.selectedUser = ""  # self variable declared, selected user
            self.tbl_users.clicked.connect(self.select_user)  # users table programmed
            self.btn_import.clicked.connect(self.importing)  # settings button pressed
            self.btn_logout.clicked.connect(self.logout)  # logout button programmd
            self.btn_delUser.clicked.connect(self.deleteUser)  # delete user button programmed
            self.btn_admin.clicked.connect(self.toggleAdmin)  # admin button programmed
            self.btn_exit.clicked.connect(self.exit)  # exit button programmed
```

3.6.2.3 Import Code

This code has been amended to validate that the directory inputted by the user actually exists. The code now also checks the format of the track number, if the track number is in the form “01/07”, only the first number in the fraction will be noted:

```
def importing(self):
    cur.execute("DELETE FROM Albums")
    cur.execute("DELETE FROM Artists")
    cur.execute("DELETE FROM SONGS")  # resets database
    print("scanning")
    counter = 1  # counter used to calculate song_id
    rawdirectory = self.txt_dir.text()  # sets directory specified from user for their music library
    if os.path.exists(rawdirectory) == False:
        self.lbl_import.setText("Directory doesn't exist")
    else:
        for dirName, subdirList, filelist in os.walk(rawdirectory):  # starts to walk through library
            for fname in filelist:  # scans filename in directory
                if fname[-3:] == 'mp3':  # validation for .mp3
                    song_location = str(dirName + '\\' + fname)  # forms file location from previous information
                    print(song_location)
                    song = EasyID3(song_location)  # retrieves tags from file
                    length_info = MP3(song_location)
                    length = length_info.info.length
                    artist_name = (song["artist"][0])  # sets artist tag to variable
                    album_name = (song["album"][0])  # sets album tag to variable
                    cur.execute("SELECT COUNT(ArtistName) FROM Artists where ArtistName= ?", (str(artist_name),))
                    # cheoks if artist name already exists
                    artist_validate = cur.fetchone()[0]  # fetches result from sql query in previous line
                    if artist_validate == 0:  # If artist name doesn't exist in artists table
                        cur.execute("select COUNT(ArtistName) FROM Artists")
                        # finds length of artists list, used to create artist_id
                        artists_length = cur.fetchone()[0]  # fetches result from sql query in previous line
                        artist_id = artists_length + 1  # creates artist_id for new artist
                        cur.execute("INSERT INTO Artists VALUES(?,?,1)", (int(artist_id), str(artist_name)))
                        # submits artist_id to new table
                        con.commit()  # commits sql query in previous line
                    else:  # if the artist is already in the table...
                        cur.execute("select ArtistID from Artists where ArtistName= ?", (str(artist_name),))
                        # retrieve artist_id of specific artist
                        artist_id = cur.fetchone()[0]  # fetches result from sql query on previous line
                    cur.execute("SELECT COUNT(AlbumName) FROM Albums WHERE AlbumName = ?", (str(album_name),))
                    # checks if album name already exists
                    album_validate = cur.fetchone()[0]  # fetches result from sql query in previous line
                    if album_validate == 0:  # If album name doesn't exist in album table
                        cur.execute("SELECT COUNT(AlbumName) FROM Albums")
                        # finds the length of Albums table, used to calculate album_id
                        albums_length = cur.fetchone()[0]  # fetches result from sql query on previous line
                        album_id = albums_length + 1  # calculates album_id
                        cur.execute("INSERT INTO Albums VALUES(?,?,?,2)", (int(album_id), str(album_name),
                                                                           int(artist_id)))
                        # adds collected data to table
                        con.commit()  # commits sql statement on previous line
                    else:  # if the album name already exists in Albums table
                        cur.execute("select AlbumID from Albums where AlbumName=?", (str(album_name),))
                        # retrieves album_id
                        album_id = cur.fetchone()[0]  # fetches result from sql statement on previous line
                    genre = (song["genre"][0])  # sets genre tag to variable
                    song_id = int(counter)  # forms song_id from counting the mp3 files
                    song_name = (song["title"][0])  # sets the song title tag to variable
                    track_number = (song["tracknumber"][0])  # sets track number tag to variable
                    slash_location = track_number.find("/")  # searches track number for a slash /
                    if slash_location != -1:  # if a slash is found in the
                        track_number = track_number[:slash_location]
                        print(track_number)
                    track_number = int(track_number)
                    cur.execute("INSERT INTO Songs VALUES(?,?,?,?,?,?,?,0,0)", (int(song_id), int(track_number),
                                                                                str(song_name), str(genre),
                                                                                str(song_location), int(album_id),
                                                                                       int(length)))
                    # appends data to table
                    counter += 1  # increments counter for next song
    self.lbl_import.setText("Import Successful")  # once all files have been added, 'Complete is printed'
```

The annotations for this code give a step by step guide through how the code works. The key line to point out is the first if statement. This statement is the first piece of validation and validates whether the directory actually exists. Furthermore, after the track number has been read from the ID3 tags, the location of the slash (if one exists) is found and the string before it is separated so it is only number. The track number variable is then set to an integer so that it can be inputted into the database.

3.6.2.4 Admin Settings

Here is the procedure that loads the user data into the users table:

```
def loadUserData(self):
    self.int_userID = LoginWindow.int_userID  # userID retrieved through encapsulation of LoginWindow object
    print(self.int_userID)
    cur.execute("SELECT * FROM Users WHERE UserID != ?", (int(self.int_userID),))  # query retrieves all information on users besides current user
    self.lst_data = cur.fetchall()  # fetches results of query and assigns result to a variable
    print(self.lst_data)
    if len(self.lst_data) > 0:  # If data table already has items in it, the table is cleaned
        for i in range(len(self.lst_data) - 1, -1):  # iteration to remove items from the list
            print("removing row", i)
            self.lst_data.pop(i)
            self.model.removeRow(index.row(self.lst_data[i]))
    self.model = QtGui.QStandardItemModel(self)
    self.tbl_users.setModel(self.model)
    for row in self.lst_data:
        items = [QtGui.QStandardItem(str(field)) for field in row]
        self.model.appendRow(items)
```

This method shows the use of polymorphism. This code is a slight manipulation of a previous procedure to load data into the songs table in the Main Window Class. The only differences are the query that is used and the name of the table that is being read to.

Then the code for when a user is selected in the table was written, this procedure is activated by the tbl_users.clicked.connect line in the initialisation:

```
def select_user(self):
    row_details = []
    table_row_details = self.tbl_users.selectionModel().selectedRows()
    for index in table_row_details:
        row = index.row()
        print(row)
        row_details = self.lst_data[row]  # Retrieves data from selected row
    self.selectedUser = row_details
    self.lbl_user.setText("User: " + row_details[1])
```

Once again, this code has been reused to better fit the problem to be dealt with. This procedure is identical to the retrieve row method in the Main Window class. However, this method finds the row in the users table rather than the songs table. Furthermore, after the row has been retrieved, the user's name is displayed on a label so that the User knows which user he is about to edit.

After the select user method, the toggle user method was written:

```
def toggleAdmin(self):
    if self.selectedUser[3] == 0:
        cur.execute("UPDATE Users SET Administrator = 1 WHERE UserID = ?", (int(self.selectedUser[0]),))
        con.commit()
    else:
        cur.execute("UPDATE Users SET Administrator = 0 WHERE UserID = ?", (int(self.selectedUser[0]),))
        con.commit()
    self.loadUserData()
    self.selectedUser = ""
```

This procedure starts by validating the details of the selected user. If the selected user is already an administrator (e.g. Administrator = 0), then their record in the Users table is updated to have an administrator value of 1. Else, the record is updated to have an administrator value of 0, the changes are then committed. After this, the user data is loaded again to update the information in the table. The selectedUser variable is also reset to avoid any mistakes that could be made by the user.

Then, the delete user method was written:

```
def deleteUser(self):
    cur.execute("DELETE FROM Users WHERE UserID = ?", (int(self.selectedUser[0]),))
    con.commit()
    self.loadUserData()
    self.selectedUser = ""
```

This method starts by using an SQL query to delete the user record corresponding to the selected user's userID. The query is committed and the user data is reloaded to show the new change that has been made. The selectedUser variable is then reset to avoid any accidental actions that the user could make.

3.6.2.5 Exit Codes

The first exit code to be programmed was the exit method. The purpose of the code is to close the settings window. The method is activated when the exit button in the Settings Screen UI is clicked:

```
def exit(self):
    SettingsWindow.hide()
```

The code simply hides the settings window, leaving only the Main Window Screen open.

The next exit code is the logout method. The purpose of the code is to log the user out of the program and return to the login screen:

```
def logout(self):
    LoginWindow.userID = ""
    MainWindow.hide()
    SettingsWindow.hide()
    LoginWindow.show()
```

This method starts by resetting the userID variable from the LoginWindow object. Then, all of the windows that are currently open (e.g. the settings window and the main window) are closed. Then the login window is shown.

3.7 Playlist Manager Window Class

3.7.1 Project Decomposition

3.7.1.1 User Interface

The user interface will include two tables. One for holding playlist data and one for holding song data. There will also be buttons to change the playlist that the user is viewing, to create a new playlist and to exit the Playlist Manager window.

3.7.1.2 Initialisation Code

The initialisation code for this class will have to perform several tasks. Similar to all of the other classes in this program, the user interface will be set up. Then, the code will have 'connect' functions added to it for the buttons on the screen. This means that when the buttons are activated, their respective methods will be run.

3.7.1.3 Loading data into the Playlist and Songs table

Before any other actions can be taken with this class, the tables on the screen must have data in them. Both methods will have to retrieve results from an SQL query:

64. The load playlists method's query will have already been executed in the playlistManager method of the Main Window Class. The query will retrieve all of the user's playlists.
65. The 'load songs' method's query will actually exist inside the method. The query will select all of the songs from the database.

The methods will then insert all of the data fetched from the database into their respective tables.

3.7.1.4 Retrieve Playlist Row

This method will need to be able to retrieve the contents of a row selected by the user in the Playlists table. Then, the code must check whether the row is a playlist or a song. If the row is a playlist, the playlist table will show all of the songs in that playlist. Else, the song will be removed from the selected playlist.

3.7.1.5 Add Song to Playlist

This method will work similarly to the retrieve playlist row section. However, once the row/song has been retrieved, the selected song is added to the selected playlist.

3.7.1.6 New Playlist

This method is activated when the new playlist button is pressed. The method must open the 'new playlist dialog box' screen.

3.7.1.7 Close Window

This method is activated when the close window button is pressed. This method closes the Playlist Manager window, leaving only the main window up.

3.7.1.8 Refresh Playlist

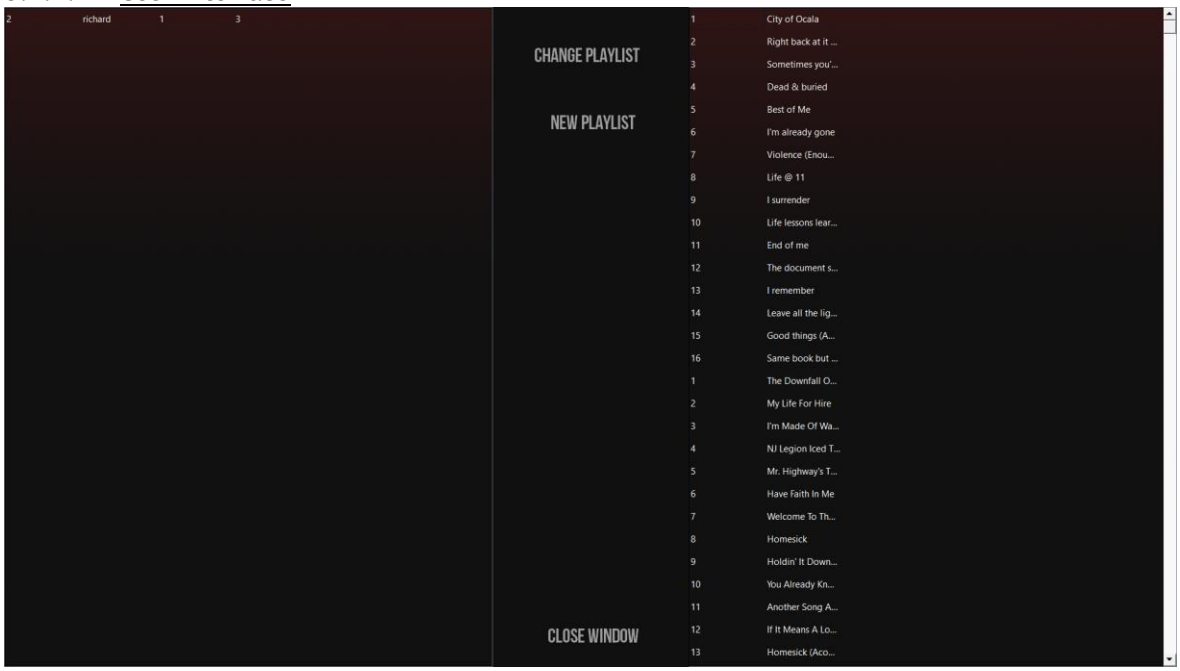
This method is activated when either a song is added or removed from a playlist. The code must refresh the playlist table to include the recent changes. This will be done be re-executing the previous SQL statement.

3.7.1.9 Playlist Reset

This method is activated when the 'Change Playlist' button is pressed. The method will execute a query that will retrieve all of the playlists, the 'load playlist data' method will then be activated with the retrieved data.

3.7.2 Final Code

3.7.2.1 User Interface



Here is the final release of the user interface.

3.7.2.2 Initialisation Code

```
def __init__(self, parent=None):
    QtGui.QMainWindow.__init__(self, parent)  # Initialisation code for PyQt UI run
    self.setupUi(self)  # User Interface set up
    self.lst_playlistData = []  # List declared. Used to hold data from queries about the playlist tables.
    self.btn_close.clicked.connect(self.close_window)  # close button programmed
    self.tbl_playlists.doubleClicked.connect(self.retrieve_playlists_row)  # playlist table interaction programmed
    self.tbl_songs.doubleClicked.connect(self.add_to_playlist)  # songs table interaction programmed
    self.btn_plylstreset.clicked.connect(self.playlist_reset)  # changed playlist button programmed
    self.btn_newplylst.clicked.connect(self.new_playlist)  # new playlist button programmed
```

First, the User Interface was set up. Then the playlist data list (a self variable) was declared. The list is used between methods and holds all the

3.7.2.3 Loading data into the Playlist and Songs table

Here is the final code for the load_playlist_data method:

```
def load_playlist_data(self, lst_playlistdata):
    print(lst_playlistdata)
    if len(lst_playlistdata) > 0:  # If data table already has items in it, the table is cleaned
        for i in range(len(lst_playlistdata) - 1, -1):
            print("removing row", i)
            lst_playlistdata.pop(i)
            self.model.removeRow(index.row(self.lst_data[i]))
    self.model = QtGui.QStandardItemModel(self)
    self.tbl_playlists.setModel(self.model)
    for row in lst_playlistdata:
        items = [QtGui.QStandardItem(str(field)) for field in row]
        self.model.appendRow(items)
```

This code is a slight adaption from the load_data method in the Main Window class. The changes made to this code were the names of the variables and the name of the table object.

Here is the final code for the load_song_data method:

```
def load_song_data(self):
    cur.execute("SELECT * FROM Songs")
    self.songData = cur.fetchall()
    if len(self.songData) > 0:  # If data table already has items in it, the table is cleaned
        for i in range(len(self.songData) - 1, -1):
            print("removing row", i)
            self.songData.pop(i)
            self.model.removeRow(index.row(self.lst_data[i]))
    self.model = QtGui.QStandardItemModel(self)
    self.tbl_songs.setModel(self.model)
    for row in self.songData:
        items = [QtGui.QStandardItem(str(field)) for field in row]
        self.model.appendRow(items)
    for i in range(8):
        columns_to_hide = (0, 3, 4, 5, 6, 7, 8, 9)
        self.tbl_songs.hideColumn(columns_to_hide[i])
        i += 1
```

Once again, this code is an adaption of the load_data method, showing how the solution uses **polymorphism** to ensure consistency and efficiency of the code. The code also includes another example of polymorphism at the end. The code uses an iteration loop that hides the selected columns (indicated by an array), this has been adapted from the hide_?_column methods in the Main Window class.

3.7.2.4 Retrieve Playlist Row

```
def retrieve_playlists_row(self):
    row_details = []
    table_row_details = self.tbl_playlists.selectionModel().selectedRows()
    for index in table_row_details:
        row = index.row()
        print(row)
        row_details = self.lst_playlistData[row]  # Retrieves data from selected row
        print(row_details)
    print(row_details[-1])
    if row_details[-1] == 3:  # If the row bears the type of 0, it is a song and is played
        self.playlists_to_songs(row_details)
    else:
        self.remove_from_playlist(row_details)
```

This section of code is a slightly adapted version of the retrieve row method in the Main Window Class. However, once the data has been retrieved, the details are validated using the type columns in the tables. If the type in the column is a 3, the row is a playlist and the playlists_to_song method is used. Else, the remove from playlist method is called.

Here is the code for the playlist to song method:

```
def playlists_to_songs(self, details):
    self.playlist_id = details[0]
    self.playlist_query = "SELECT PlaylistSongs.PlaylistID, Songs.SongID, Songs.SongName FROM PlaylistSongs " \
                          "INNER JOIN Songs ON PlaylistSongs.SongID = Songs.SongID " \
                          "WHERE PlaylistSongs.PlaylistID = ?"
    cur.execute(self.playlist_query, (int(self.playlist_id),))
    self.lst_playlistData = cur.fetchall()
    self.load_playlist_data(self.lst_playlistData)
```

This method uses an **Inner Join** SQL query to select all of the songs from the Songs table that correspond to the songID in the PlaylistSongs table that correspond to the playlistID of the playlist selected by the user. The query is then executed and fetched, the load_playlist_data method is run using the fetched data.

Here is the code for the remove_from_playlist method:

```
def remove_from_playlist(self, row_details):
    songID = row_details[1]
    print(songID, self.playlist_id)
    cur.execute("DELETE FROM PlaylistSongs WHERE PlaylistID = ? AND SongID = ?", (int(self.playlist_id),
                                                                                  int(songID),))
    print("Executed")
    con.commit()
    self.refresh_playlist()
```

This code first assigns a variable for the songID, retrieving the value from the row that was previously retrieved. Then, a SQL query is executed that will remove the corresponding field from the PlaylistSongs table (it will delete the record where the playlistID and songID match the selected playlist and song). The changes are committed and the refresh playlist method is run (this method is shown later in the session).

3.7.2.5 Add Song to Playlist

Here is the code for the add_to_playlist method:

```
def add_to_playlist(self):
    row_details = []
    table_row_details = self.tbl_songs.selectionModel().selectedRows()
    for index in table_row_details:
        row = index.row()
        print(row)
        row_details = self.songData[row]  # Retrieves data from selected row
        print(row_details)
    songID = row_details[0]
    cur.execute("SELECT PlaylistSongsID FROM PlaylistSongs ORDER BY PlaylistSongsID DESC")
    try:
        PlaylistSongsID = int(cur.fetchall()[0][0]) + 1
    except:
        PlaylistSongsID = int(cur.fetchone()[0]) + 1
        print(PlaylistSongsID)
    cur.execute("INSERT INTO PlaylistSongs VALUES (?,?,?)", (int(PlaylistSongsID), int(self.playlist_id),
                                                             int(songID),))
    con.commit()
    self.refresh_playlist()
```

The first section of the code is an adapted version of the retrieve_row method, showing how the code uses **polymorphism**. After the row has been retrieved, the songID (or first column) is retrieved from it. Then, a query selecting all of the PlaylistSongsIDs in the PlaylistSongs table (ordered by their PlaylistSongsID from highest lowest) is executed. A try and except clause is then used to retrieve the highest PlaylistSongsID and increment it by 1, creating the PlaylistSongsID for the new entry. An SQL query is then executed that inserts the a new record into the PlaylistSongsTable, using the selected SongID and PlaylistID and the newly created PlaylistSongID as inputs.

3.7.2.6 New Playlist

Here is the code for the New Playlist method:

```
def new_playlist(self):
    PlaylistNameWindow.show()
```

This method simply shows the Playlist Name Dialog Box. This box allows the user to enter the name of the new playlist and will be explained later in the development.

3.7.2.7 Close Window

Here is the code for the Close Window method:

```
def close_window(self):
    PlaylistWindow.hide()
```

This code simply closes the Playlist Manager Window, returning the user to the Main Window screen.

3.7.2.8 Refresh Playlist

Here is the code for the Refresh Playlist method:

```
def refresh_playlist(self):
    cur.execute(self.playlist_query, (int(self.playlist_id),))
    lst_playlistData = cur.fetchall()
    self.load_playlist_data(lst_playlistData)
```

This code starts by executing the SQL query denoted by the playlist_query variable, which is the most recent query executed by the playlist table. The data from the query is then fetched and the data is then loaded into the playlists table object (using the load_playlist_data method previously developed).

3.7.2.9 Playlist Reset

Here is the code for the Playlist Reset method:

```
def playlist_reset(self):
    cur.execute("SELECT * FROM Playlists WHERE UserID=?", (int(LoginWindow.userID),))
    self.lst_playlistData = cur.fetchall()
    self.load_playlist_data(self.lst_playlistData)
```

The code starts by executing an SQL query to select all the playlists that the user has access to (meaning the playlists that the user has created). The data from the query is then fetched and loaded into the playlist data using the Load Playlist Data method (which has been previously developed).

3.8 Playlist Name Dialog Box Class

3.8.1 Project Decomposition

3.8.1.1 User Interface

The user interface must include all of the objects described in the objectives and User Interface Design, these include:

66. An input box to allow the user to input their desired Playlist Name.
67. A Button that allows the user to confirm their input.
68. A button that allows the user to cancel the request and close the window.
69. A label to instruct the user and notify them of any problems with validation

3.8.1.2 Initialisation Code

Like all of the other initialisation methods, this code will include the initialisation code that the user interface requires. The code will also set up listeners for the two buttons to check when they are activated.

3.8.1.3 Add Playlist

This method is called when the OK button object is clicked by the user. When called, the method will need to retrieve the user's input from the input box and validate it to make sure the input is long enough (playlist names will need to be longer than 5 characters). If the validation fails, an error message will be shown from the label on the program. Else, a new playlistID will need to be created (by taking the highest current PlaylistID and incrementing it by 1). An SQL query will then need to be executed to insert a new record into the Playlist table (using the previously calculated playlistID, the playlistName entered by the user, userID of the current user and the value of 3 for the type (as the record is a playlist)). The changes to the database are then committed and the Playlist Reset method from the Playlist Manager screen is called. The Playlist Name dialog box is then closed.

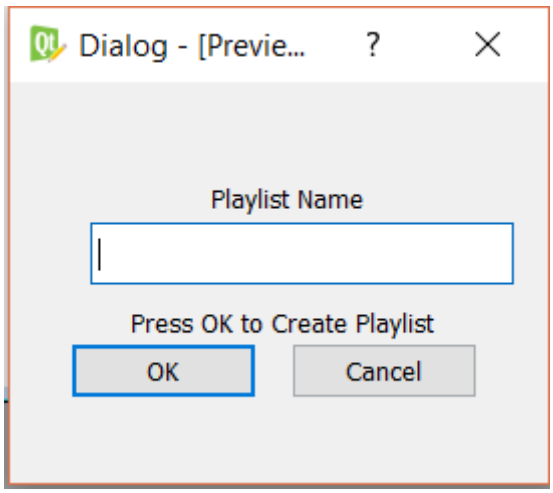
3.8.1.4 Close Window

This method is called when the cancel button object is clicked by the user. When called, the method will hide the Playlist Name dialog box, leaving only the Main Window screen.

3.8.2 Final Code

3.8.2.1 User Interface

Here is the final design of the user interface for the Playlist Name dialog box:



3.8.2.2 Initialisation Code

Here is the final code for the Initialisation method for the Playlist Name dialog box:

```
def __init__(self, parent=None):
    QtGui.QDialog.__init__(self, parent)
    self.setupUi(self)
    self.btn_ok.clicked.connect(self.add_playlist)
    self.btn_cancel.clicked.connect(self.close_window)
```

The code starts by running the initialisation code provided by PyQt to start the user interface. Then the listeners for the button objects in the user interface are set up to call their respective methods when they are clicked.

3.8.2.3 Add Playlist

Here is the final code for the Add Playlist method:

```
def add_playlist(self):
    playlistName = self.txt_plylstname.text()
    if len(playlistName) <= 5:
        self.lbl_plylst_create.setText("Longer Playlist Name Needed")
    else:
        cur.execute("SELECT PlaylistID FROM Playlists ORDER BY PlaylistID DESC")
        playlistID = cur.fetchall()[0][0] + 1
        cur.execute("INSERT INTO Playlists VALUES (?,?,?,3)", (int(playlistID), str(playlistName),
                                                               int(LoginWindow.userID)))
        con.commit()
        PlaylistWindow.playlist_reset()
        PlaylistNameWindow.hide()
```

The method starts by retrieving the input from the txt_plylstname object. The input is then validated to make sure its length is above 5 characters. If the validation fails, the text in the lbl_plylst_create object is altered to display an error to the user. Else, an SQL query is executed to retrieve all of the PlaylistIDs in the Playlists table, ordered from highest to lowest. The first record from this result is fetched and incremented by 1, creating the PlaylistID for the new Playlist record. Another SQL query is then executed to insert the values gathered (the PlaylistID, the Playlist name), the userID from the Login Window object and the value of 3 (for the type column, as the record is a playlist) into the Playlists table. After this query has been committed, the Playlist Reset method from the Playlist Window object is then called, in order to include the new changes that have been made. Finally, the Playlist Name dialog box is closed, returning the user to the Main Window screen.

4 Evaluation

4.1 Final Testing

In this section, I will test each section of the program to ensure that all objectives have been met.

4.1.1 Database

4.1.1.1 Objectives to be tested

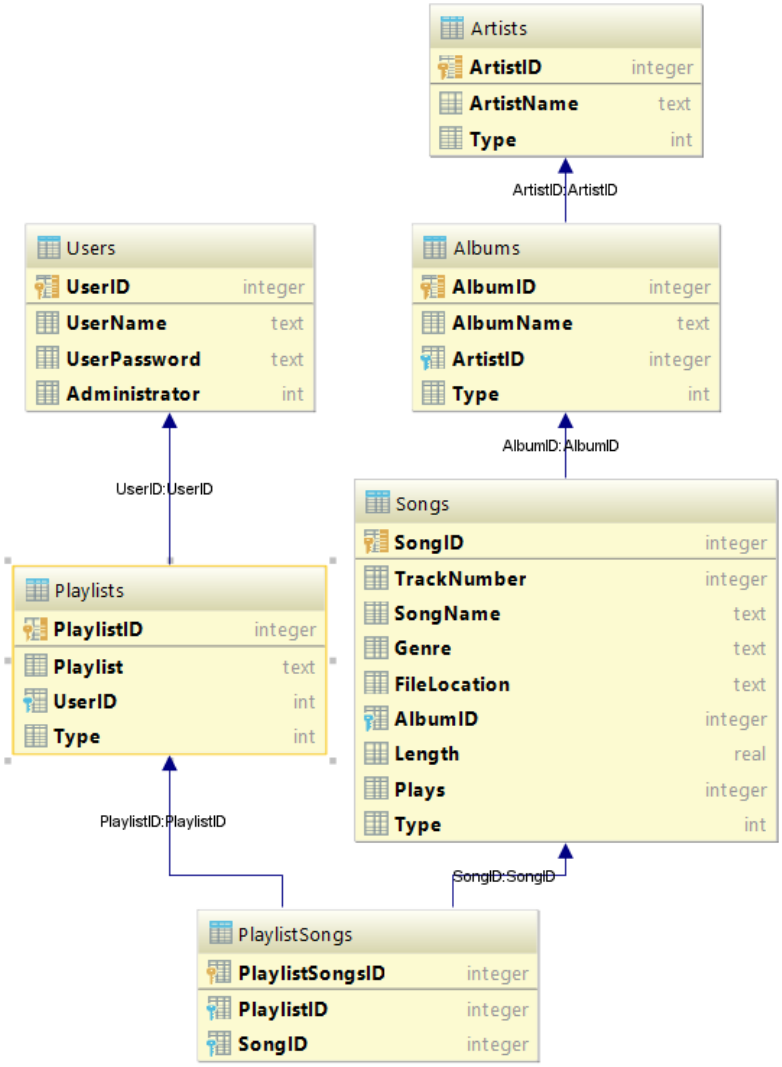
Test ID	Success Criteria Tested	Input	Expected Output
1	1	Open database file	All tables in the database will be related to one another and primary keys will exist for all tables.
	2	Open database file	Songs table will exist
	3	Open database file	Albums table will exist
	4	Open database file	Artists table will exist
	5	Open database file	Playlist table will exist
	6	Open database file	Users table will exist

4.1.1.2 Testing

Here are some screenshots showing proof of the database's existence:

Albums		CREATE TABLE "Albums" (`AlbumID` INTEGER UNIQUE, `AlbumName` TEXT, `ArtistID` INTEGER, `Type` INT, PRIMARY KEY(`AlbumID`), FOREIGN KEY(`ArtistID`
AlbumID	INTEGER	`AlbumID` INTEGER UNIQUE
AlbumName	TEXT	`AlbumName` TEXT
ArtistID	INTEGER	`ArtistID` INTEGER
Type	INT	`Type` INT
Artists		CREATE TABLE "Artists" (`ArtistID` INTEGER UNIQUE, `ArtistName` TEXT, Type INT NULL, PRIMARY KEY(`ArtistID`))
ArtistID	INTEGER	`ArtistID` INTEGER UNIQUE
ArtistName	TEXT	`ArtistName` TEXT
Type	INT	`Type` INT
PlaylistSongs		CREATE TABLE "PlaylistSongs" (`PlaylistSongsID` INTEGER, `PlaylistID` INTEGER, `SongID` INTEGER, PRIMARY KEY(`PlaylistSongsID`), FOREIGN KEY(`PlaylistID`
PlaylistSongsID	INTEGER	`PlaylistSongsID` INTEGER
PlaylistID	INTEGER	`PlaylistID` INTEGER
SongID	INTEGER	`SongID` INTEGER
Playlists		CREATE TABLE "Playlists" (`PlaylistID` INTEGER UNIQUE, `Playlist` TEXT, `UserID` INT, `Type` INT DEFAULT 3, PRIMARY KEY(`PlaylistID`), FOREIGN KEY(`UserID`
PlaylistID	INTEGER	`PlaylistID` INTEGER UNIQUE
Playlist	TEXT	`Playlist` TEXT
UserID	INT	`UserID` INT
Type	INT	`Type` INT DEFAULT 3
Songs		CREATE TABLE "Songs" (`SongID` INTEGER UNIQUE, `TrackNumber` INTEGER, `SongName` TEXT, `Genre` TEXT, `FileLocation` TEXT, `AlbumID` INTEGER, `Len
SongID	INTEGER	`SongID` INTEGER UNIQUE
TrackNumber	INTEGER	`TrackNumber` INTEGER
SongName	TEXT	`SongName` TEXT
Genre	TEXT	`Genre` TEXT
FileLocation	TEXT	`FileLocation` TEXT
AlbumID	INTEGER	`AlbumID` INTEGER
Length	REAL	`Length` REAL
Plays	INTEGER	`Plays` INTEGER
Type	INT	`Type` INT
Users		CREATE TABLE "Users" (`UserID` INTEGER UNIQUE, `UserName` TEXT, `UserPassword` TEXT, `Administrator` INT DEFAULT 0, PRIMARY KEY(`UserID`))
UserID	INTEGER	`UserID` INTEGER UNIQUE
UserName	TEXT	`UserName` TEXT
UserPassword	TEXT	`UserPassword` TEXT
Administrator	INT	`Administrator` INT DEFAULT 0

This diagram shows the actual existence of all of the tables in the database. Showing proof that objectives 2,3,4,5 and 6 have been met from the success criteria



This diagram shows the tables and the relationships between them (as well as the foreign keys that link them). The first entity of each table shown in the diagram shows the primary key. This overall proves that the solution has met objective 1 of the search criteria.

4.1.1.3 Conclusion

Test ID	Success Criteria Tested	Input	Expected Output	Output	Conclusion
1	1	Open database file	All tables in the database will be related to one another and primary keys will exist for all tables.	All tables in the database are related to one another and primary keys exist for all tables.	Success
1	2	Open database file	Songs table will exist	Songs table exists	Success
1	3	Open database file	Albums table will exist	Albums table exists	Success
1	4	Open database file	Artists table will exist	Artists table exists	Success

1	5	Open database file	Playlist table will exist	Playlist table exists	Success
1	6	Open database file	Users table will exist	Users table exists	Success

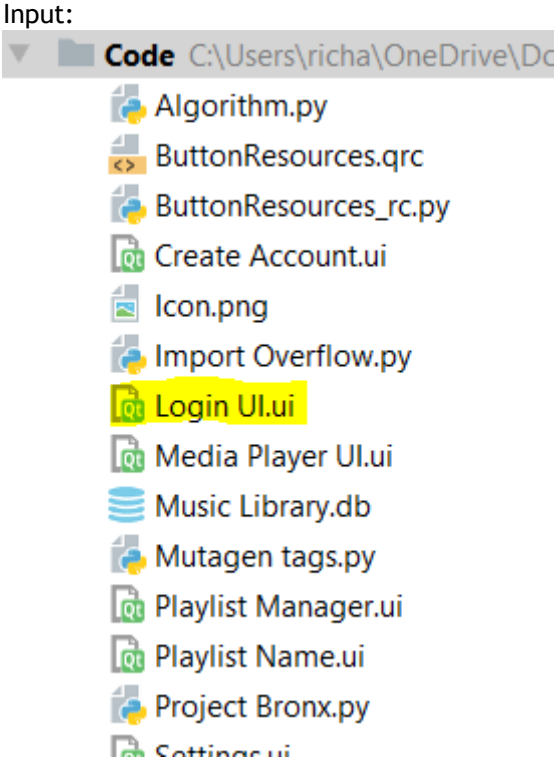
4.1.2 User Interface

4.1.2.1 Objectives to be tested

Test ID	Success Criteria Tested	Input	Expected Output
7	7, 13, 14, 15, 18, 19	Open login window .ui file	User Interface will exist with all expected objects
8	8, 20, 21, 22	Open create account window .ui file	User Interface will exist with all expected objects
9	9, 25, 31, 32, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 45, 46, 47	Open Main Window .ui file	User Interface will exist with all expected objects
10	10	Open Settings Window.ui file	User Interface will exist with all expected objects
11	11	Open Playlist Manager .ui file	User Interface will exist with all expected objects
12	12	Open Playlist Name .ui file	User Interface will exist with all expected objects

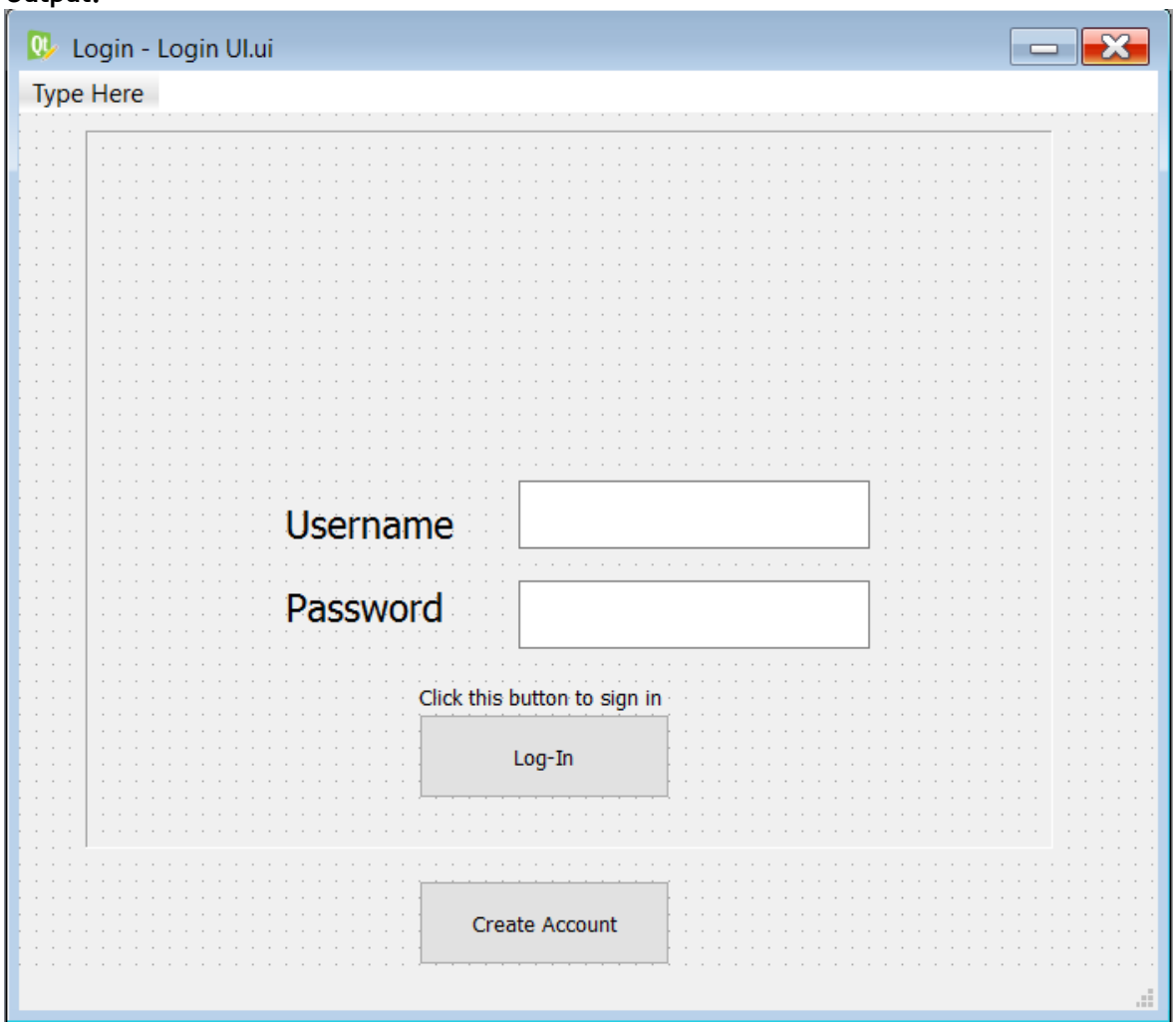
4.1.2.2 Test 07

Input:



The login ui interface was double clicked.

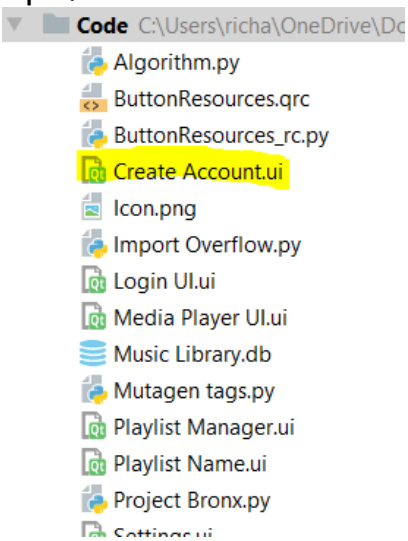
Output:



This screenshot shows the login window and all of the objects for the final program. Proving objectives 7, 13, 14, 15, 18 and 19 have been met.

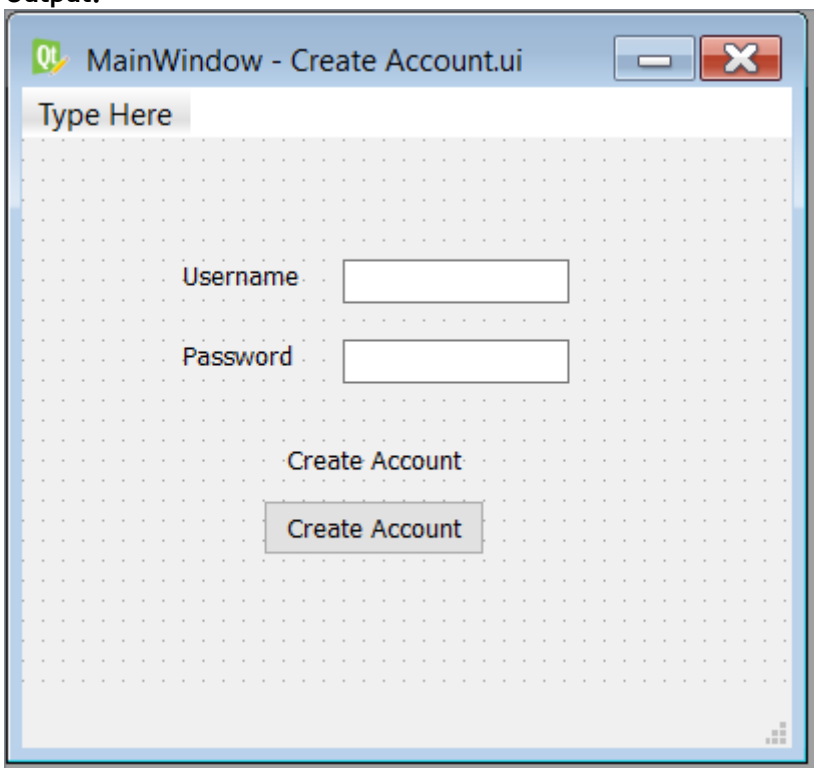
4.1.2.3 Test 08

Input:



The create account user interface file was double clicked.

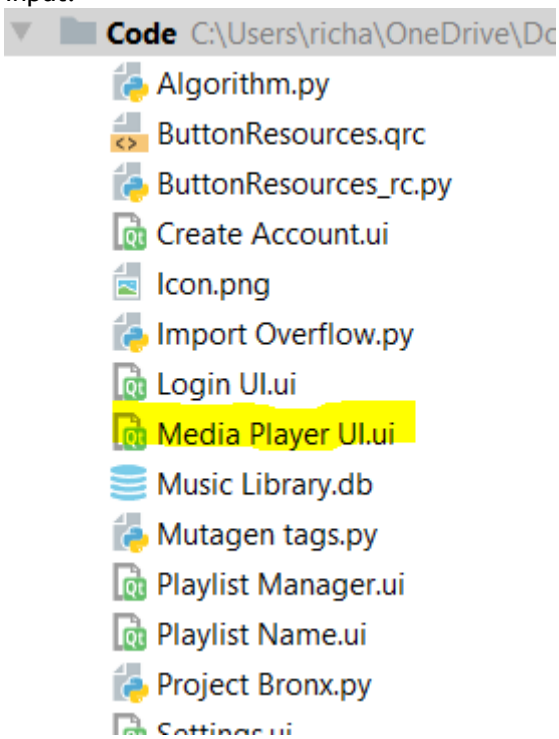
Output:



This screenshot shows the create account window, and its related objects, for the final solution. Proving that objectives 8, 20, 21 and 22 have been met.

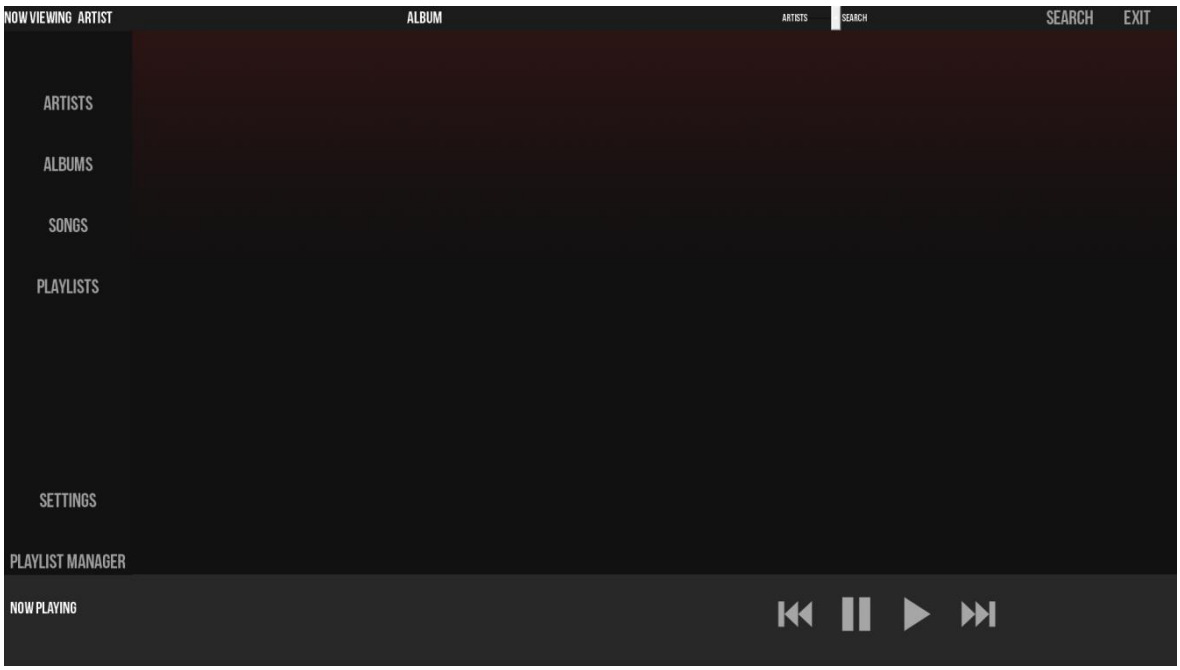
4.1.2.4 Test 09

Input:



The main window user interface file was double clicked.

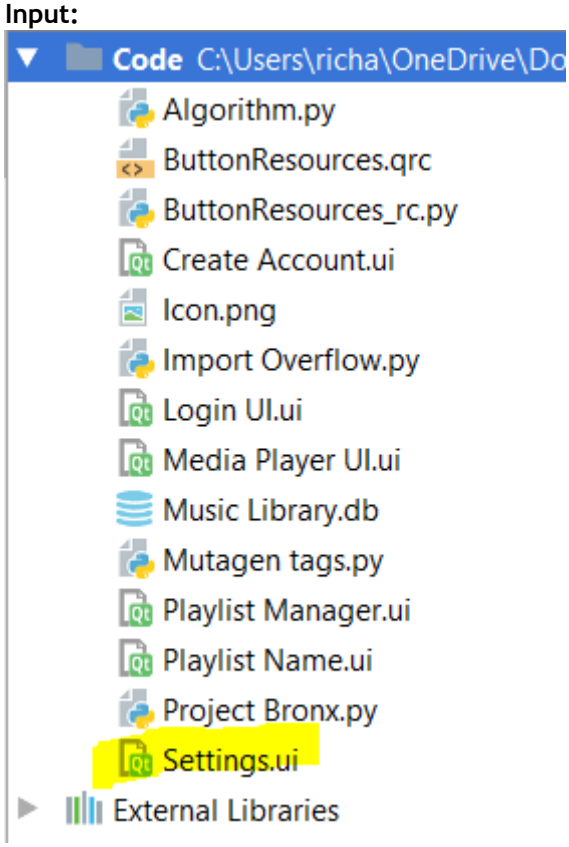
Output:



The output from this test shows the Main Window Screen and its related objects. Proving that objectives 9, 25, 31, 32, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 45, 46 and 47.

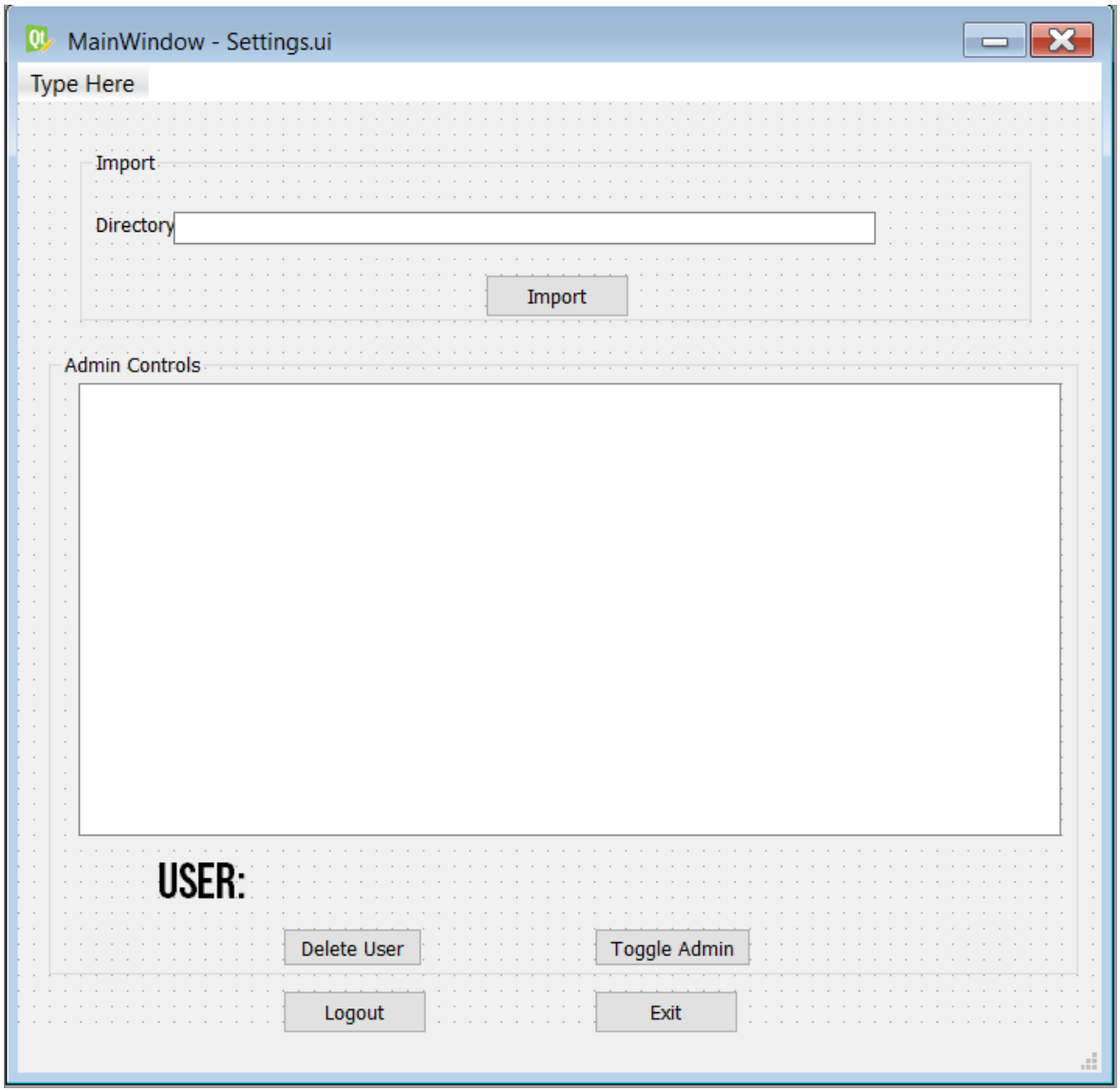
4.1.2.5 Test 10

Input:



The settings user interface file was double clicked.

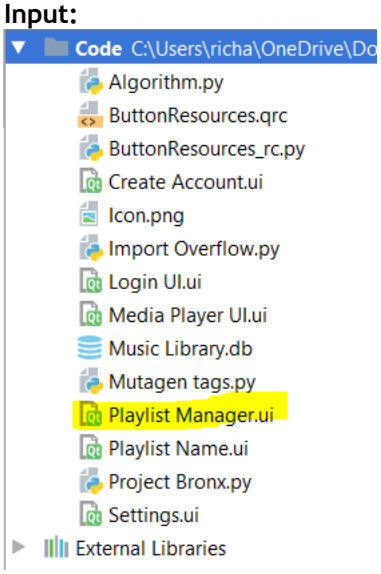
Output:



This screenshot shows the final screen for the settings window as well as its related objects. Proving that objective 10 of the search criteria has been met.

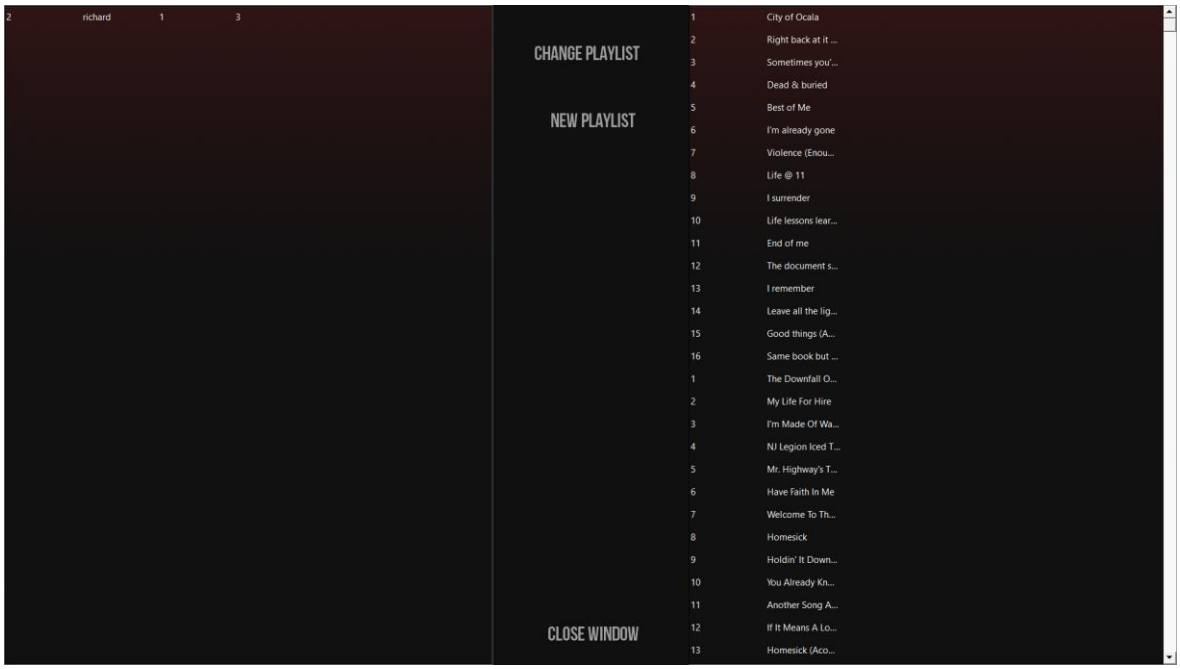
4.1.2.6 Test 11

Input:



The playlist manager user interface file was double clicked.

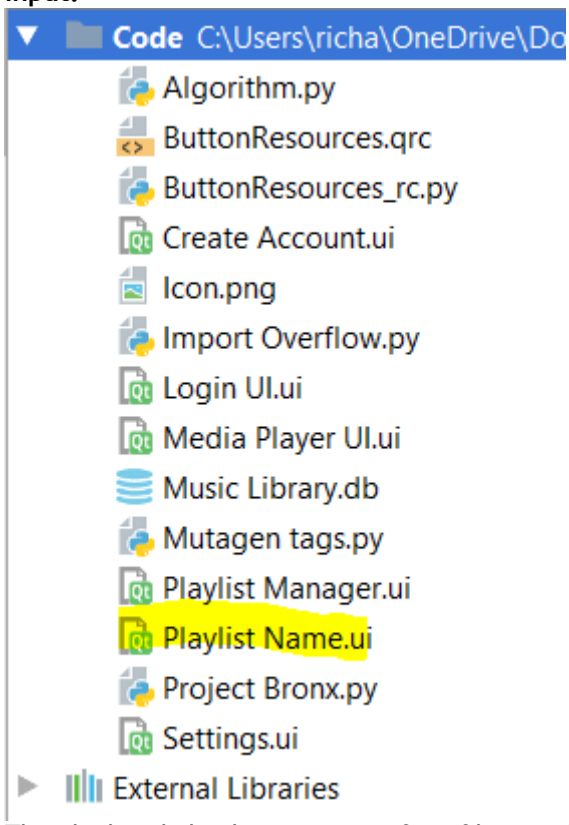
Output:



This screenshot shows the final screen for the playlist manager window, as well as the relevant objects needed for the user interface. This shows that objective 11 has been met.

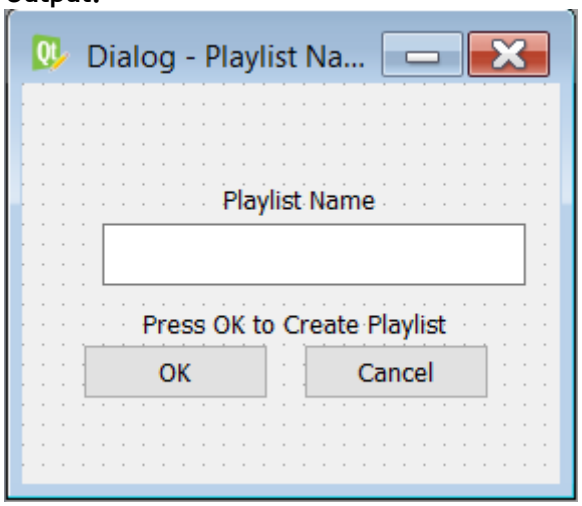
4.1.2.7 Test 12

Input:



The playlist dialog box user interface file was double clicked.

Output:



This screenshot shows the final screen for the Playlist 'name change' dialog box and its related objects. This shows the completion of objective 12 from the search criteria.

4.1.3 Login Screen

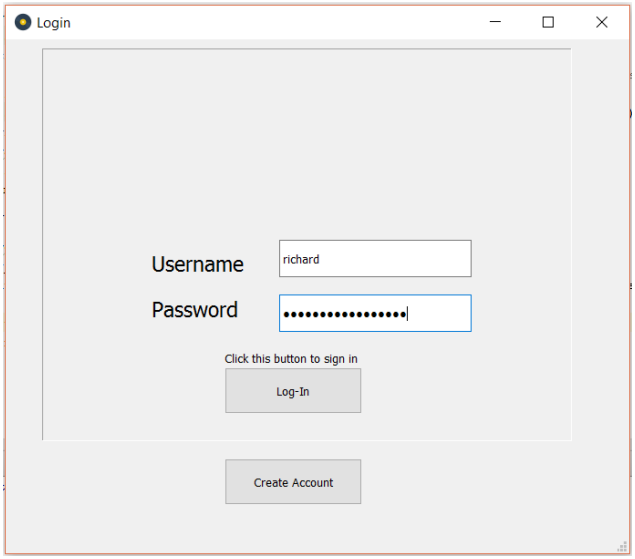
4.1.3.1 Objectives to be tested

Test ID	Success Criteria Tested	Input	Branches	Expected Output
13	16, 17	Login Button Pressed	a) Correct username, incorrect password. b) Incorrect username, correct password. c) No details inputted. d) Correct details inputted.	a) Label will display: "Wrong Password" b) Label will display: "Wrong Username" c) Label will display: "Fields Empty" d) User will be logged in & Main Window will be displayed.

4.1.3.2 Test 13

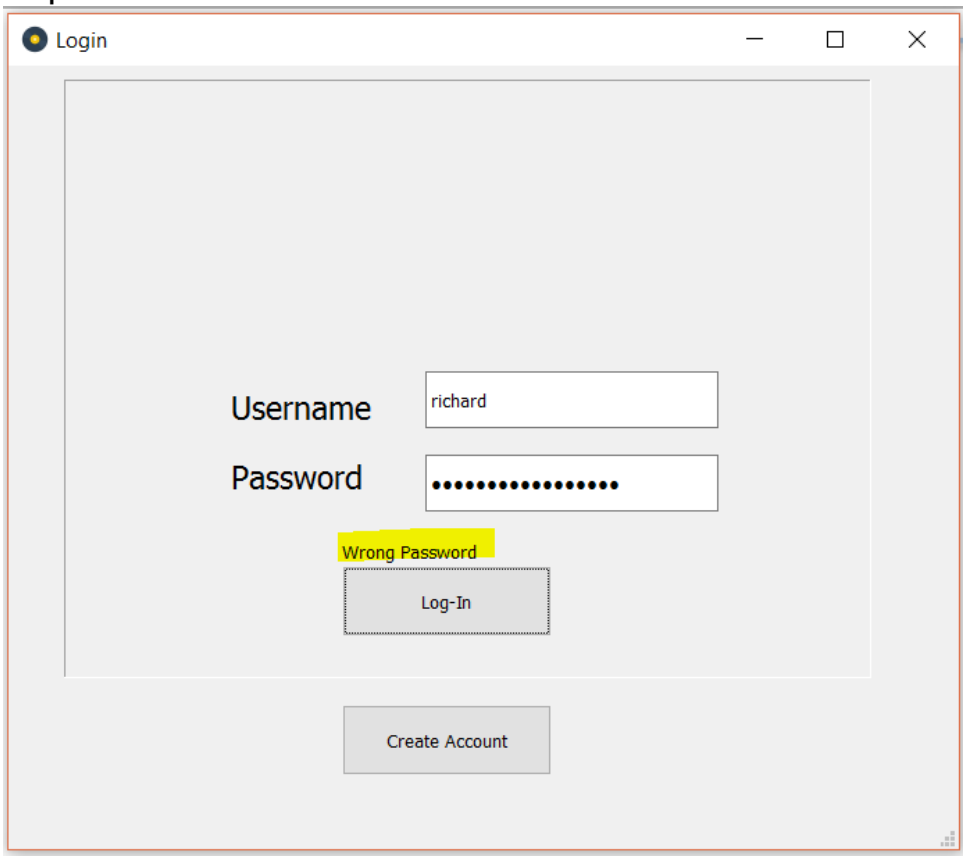
4.1.3.2.1 Section A

Input:



A username that exists in the database has been inputted, but the password does not match the password field of the user's record in the database.

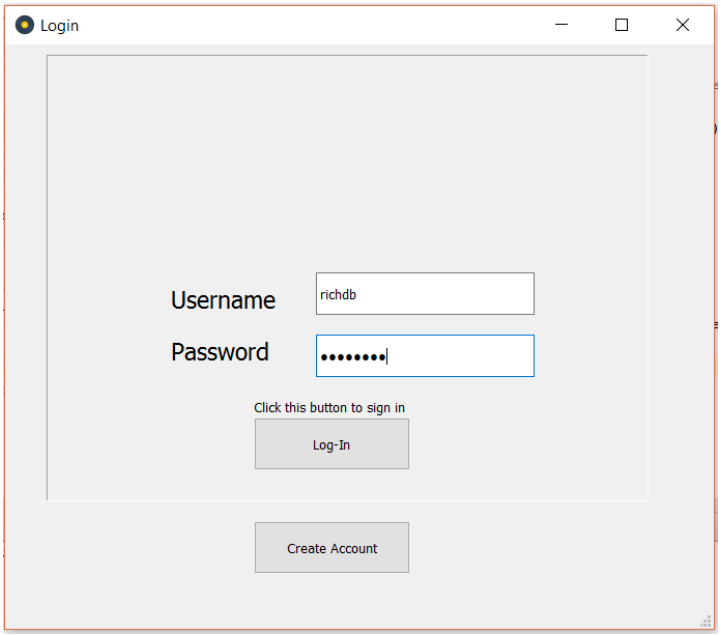
Output:



Using a label, the program has notified the user that the inputted password is incorrect.

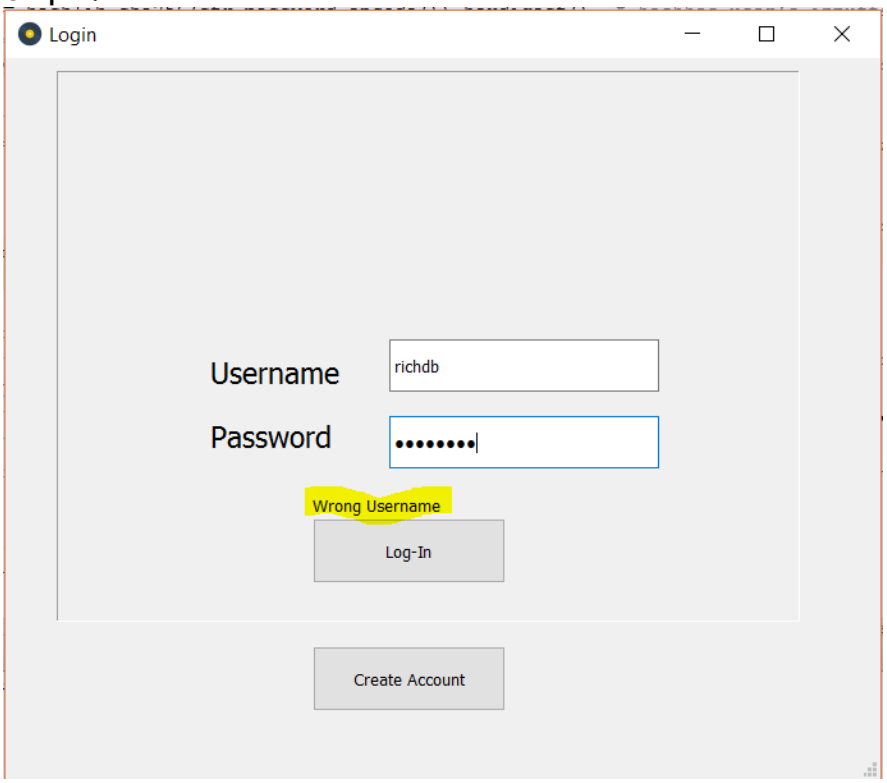
4.1.3.2.2 Section B

Input:



A password that exists in the database has been inputted. However, the username does not match the password.

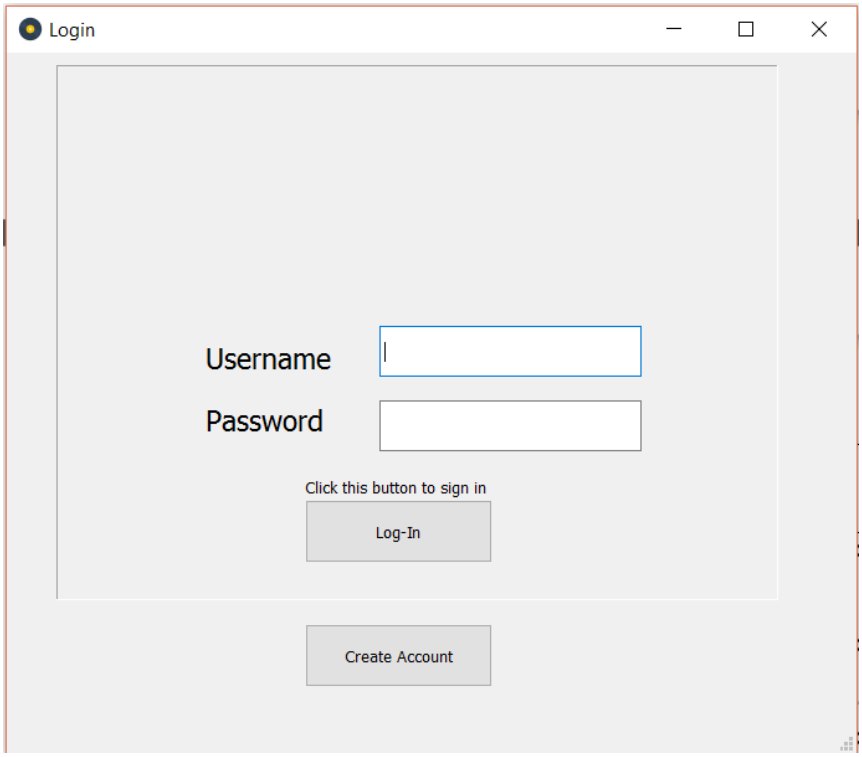
Output:



The application uses a label to notify the user that the username entered is incorrect.

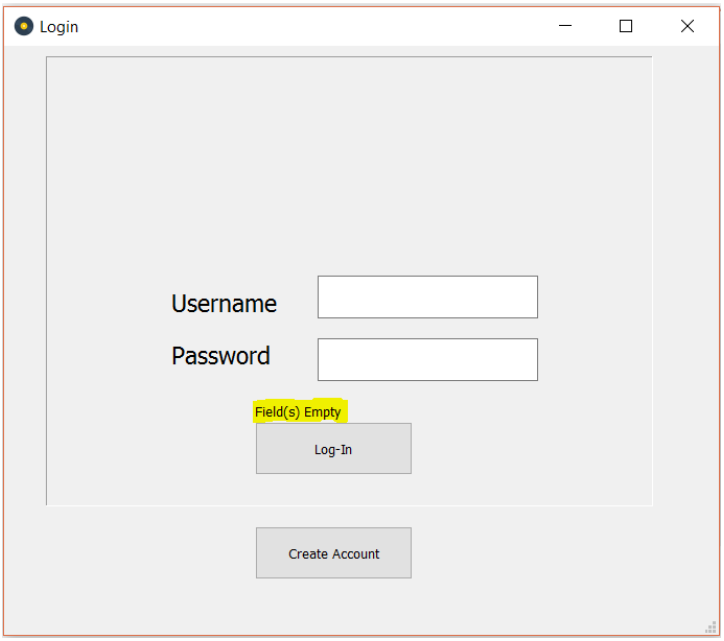
4.1.3.2.3 Section C

Input:



Neither a username or password has been inputted.

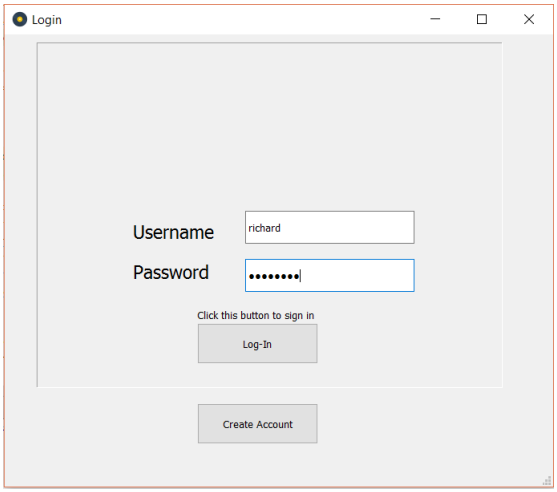
Output:



The program uses a label to notify the user that one of the fields is empty.

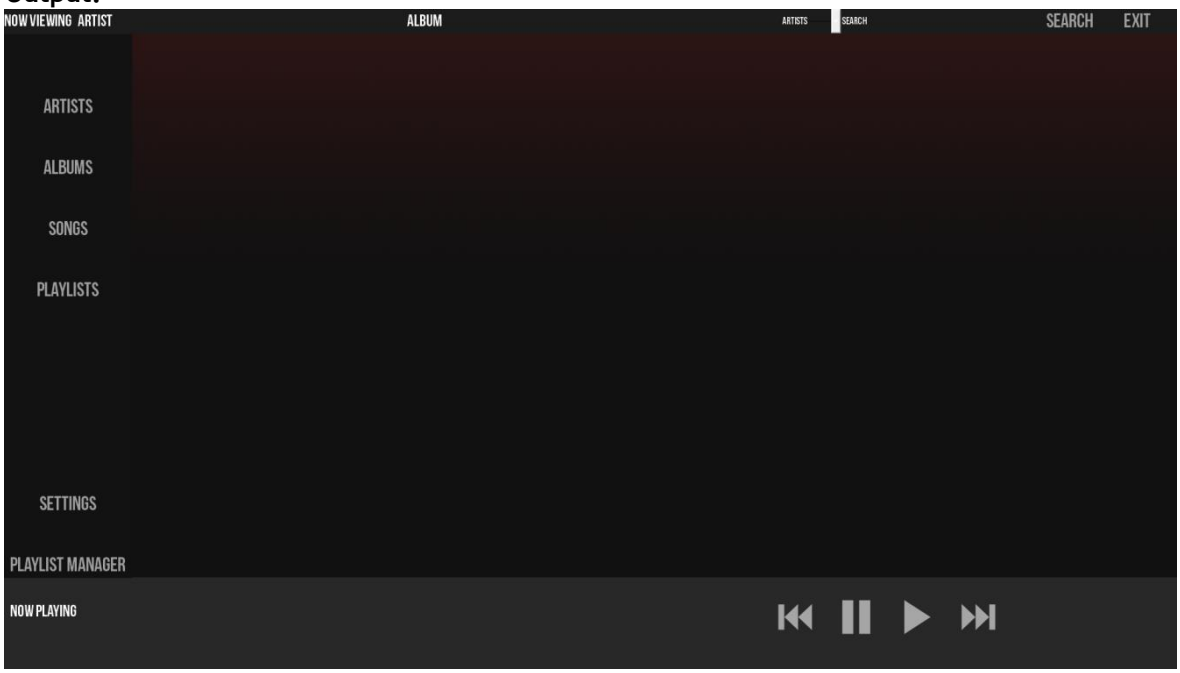
4.1.3.2.4 Section D

Input:



A username that exists in the database, alongside its corresponding password, is inputted into the fields.

Output:



The user is granted access to the program.

4.1.3.3 Conclusion

Test ID	Input	Branches	Expected Output	Output	Verdict
13	Login Button Pressed	a) Correct username, incorrect password. b) Incorrect username, correct password. c) No details inputted. d) Correct details	a) Label will display: "Wrong Password" b) Label will display: "Wrong Username" c) Label will display: "Fields Empty"	a) Label displayed "wrong password) b) Label displayed "Wrong Username" c) Label displayed "Fields Empty".	Success

		inputted.	d) User will be logged in & Main Window will be displayed.	d) User logged in and Main window was displayed.	

4.1.4 Create Account Screen

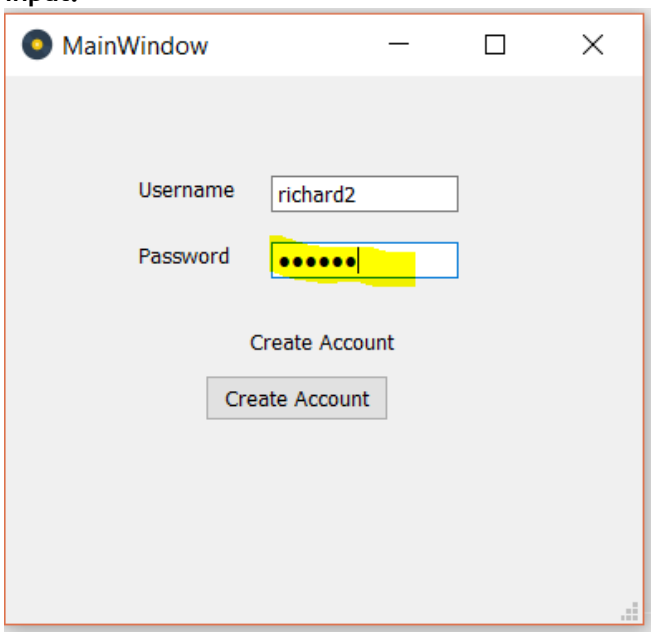
4.1.4.1 Objectives to be tested

Test ID	Success Criteria Tested	Input	Branches	Expected Output
14	23, 24	Create Account Button pressed	a) Password has no letters. b) Password has no numbers c) Password is not between 6 and 16 characters. d) Username has already been used	a) Label displays: "Please use letters in the password" b) Label displays: "Please use numbers in the password" c) Label displays: "Password must be 6-16 characters"

4.1.4.2 Test 14

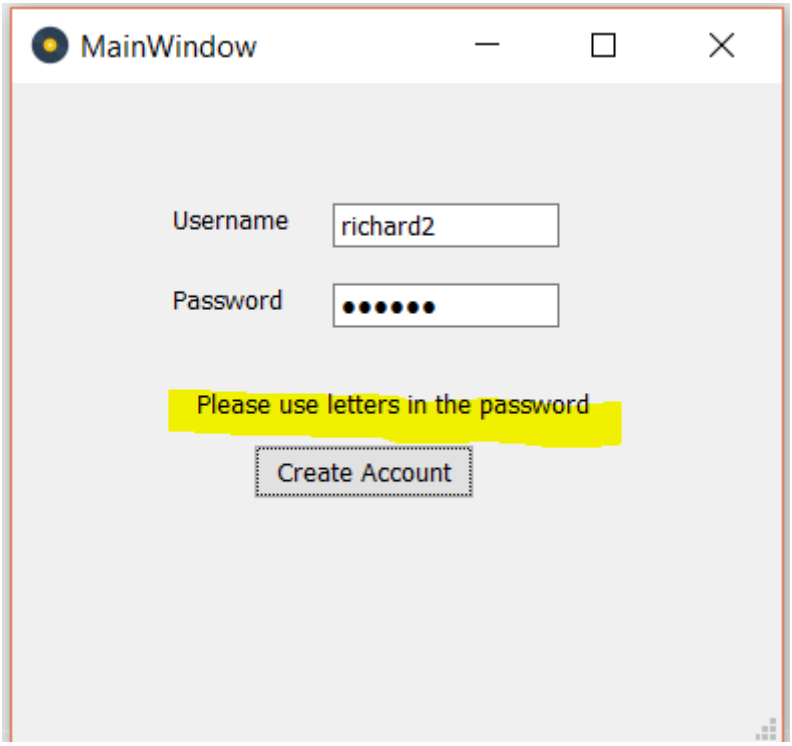
4.1.4.2.1 Section A

Input:



The password input is 123456,

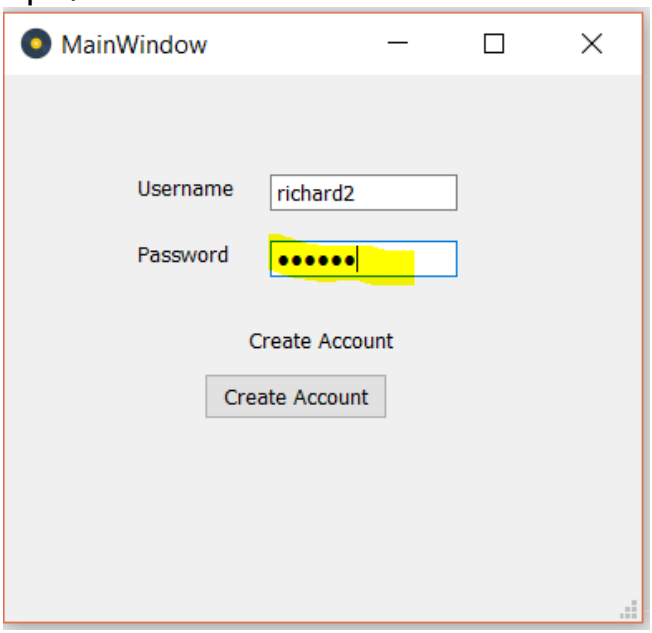
Output:



The program displays the highlighted message to the user.

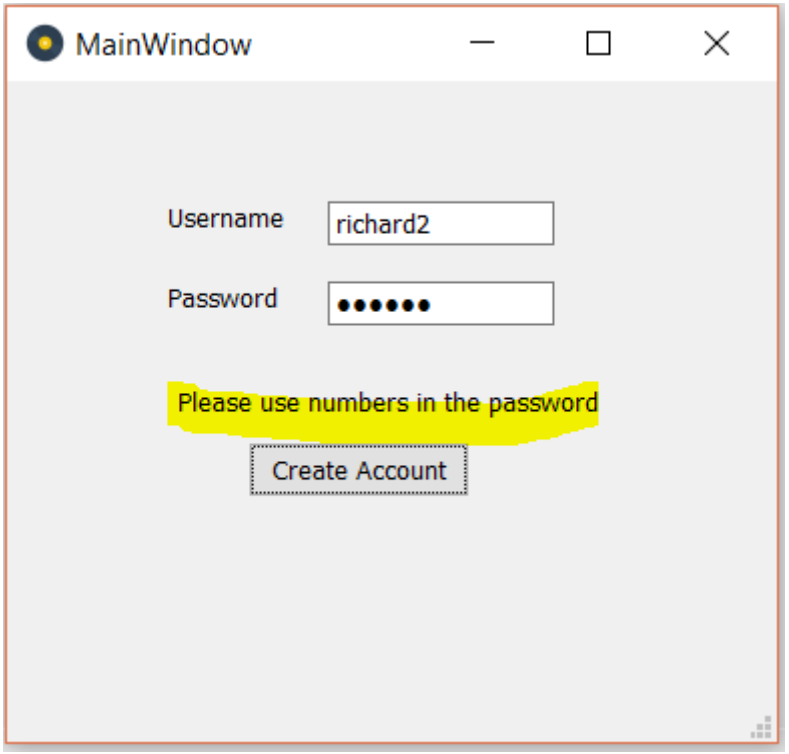
4.1.4.2.2 Section B

Input:



The password is inputted as abcdef

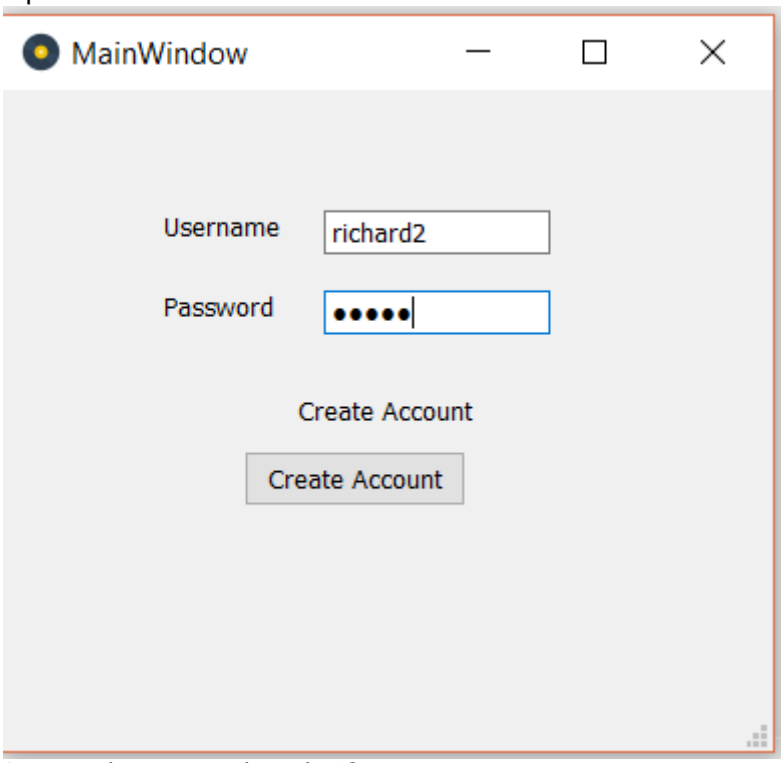
Output:



The highlighted message is shown to the user.

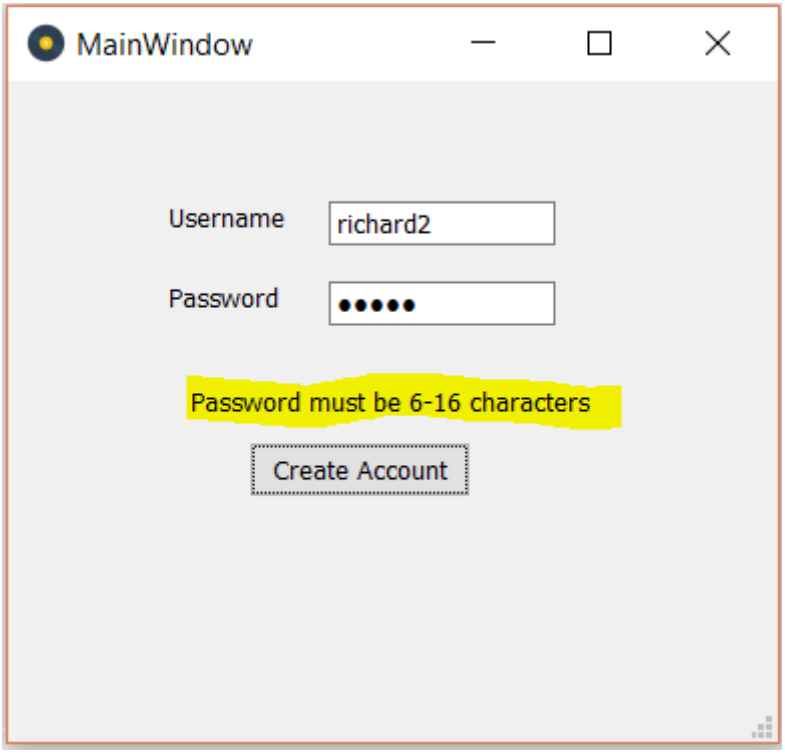
4.1.4.2.3 Section C

Input:



Password is inputted as abc12

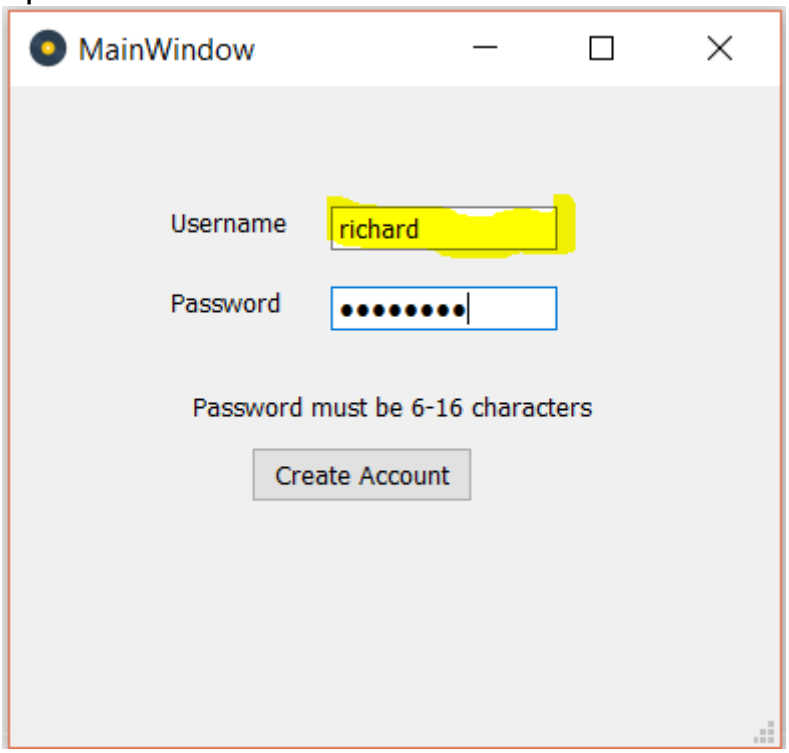
Output:



The highlighted message is shown to the user.

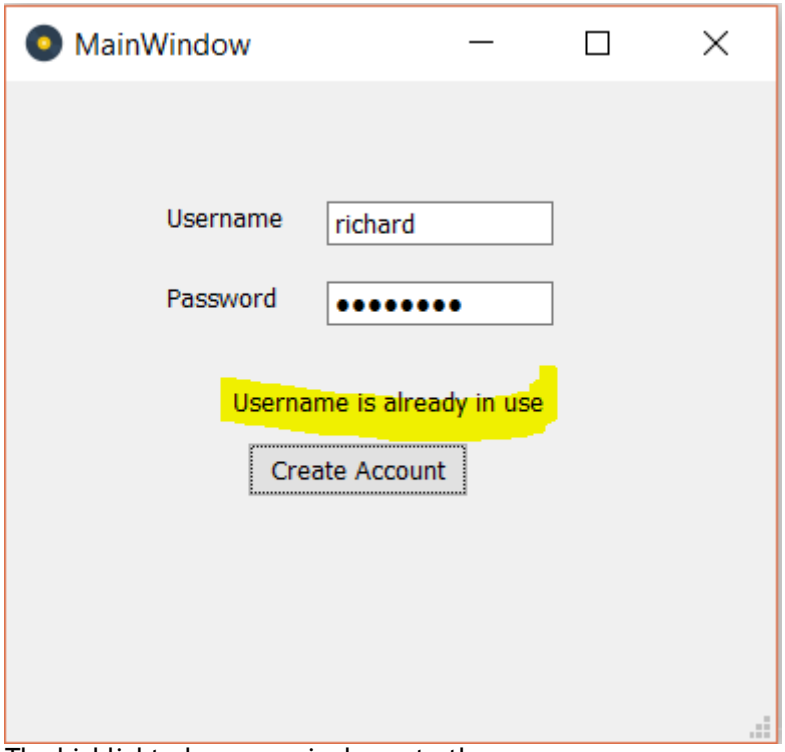
4.1.4.2.4 Section D

Input:



Username 'richard' is inputted (username already exists in database).

Output:



The highlighted message is shown to the user.

4.1.4.3 Conclusion

Test ID	Success Criteria Tested	Input	Branches	Expected Output	Verdict
14	23, 24	Create Account Button pressed	a) Password has no letters. b) Password has no numbers c) Password is not between 6 and 16 characters. d) Username has already been used	a) Label displays: "Please use letters in the password" b) Label displays: "Please use numbers in the password" c) Label displays: "Password must be 6-16 characters"	Success

4.1.5 Main Window Screen

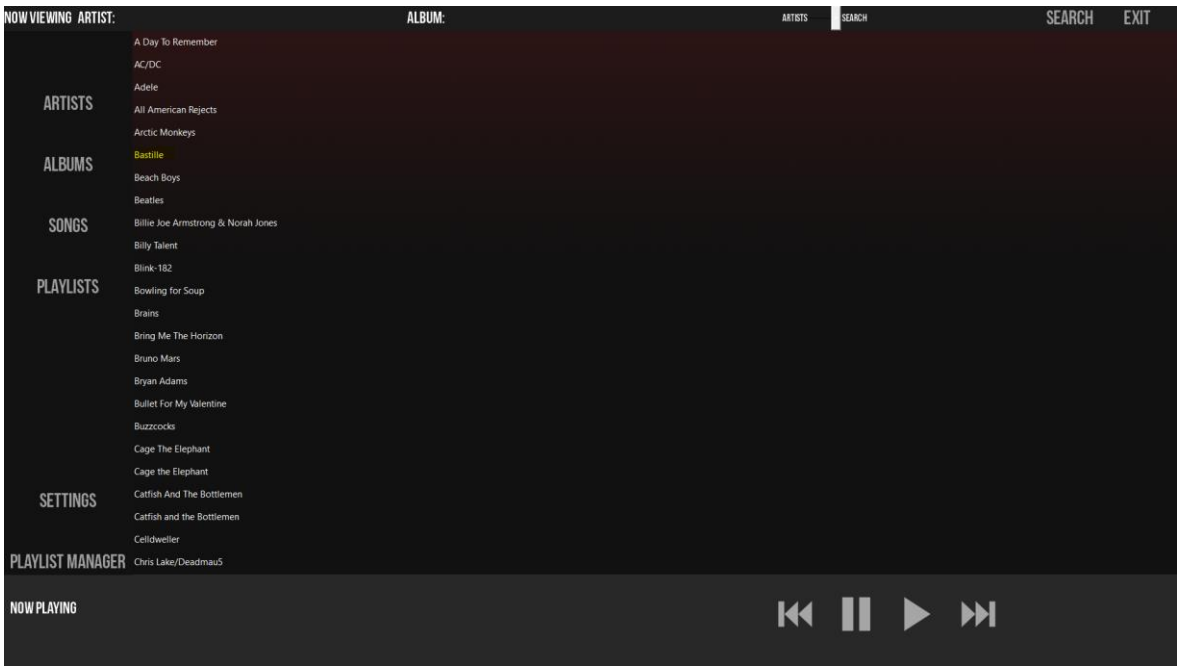
4.1.5.1 Objectives to be tested

Test ID	Success Criteria Tested	Input	Branches	Expected Output
15	26	Artist row is clicked in the table		All of the albums by artist in

				library are shown.
16	27	Album row clicked in table		All songs in album are shown
17	28	Song row clicked in table		Song begins playback
18	29, 43	Skip forward button press		Next song in queue will begin playback
19	30	Playlist row clicked in table		All song in playlist are shown
20	31	Artist button pressed		All artists in music library are shown
21	32	Albums button pressed		All albums in music library are shown
22	33	Songs button pressed		All songs in music library are shown
23	34	Playlists button pressed		All playlists owned by the user and the 'Most Played' playlist are shown
24	35	Settings button press		Settings window opens
25	36	Playlist Manager button pressed		Playlist Manager window opens
26	39	Search button pressed with inputs for the search term and its criteria		Matches with the term and criteria will be displayed.
27	40	Exit button pressed		The application will close
28	41	Pause button pressed		Music playback stops
29	42	Resume button pressed		Music playback **resumes**
30	44, 45	Skip backward button pressed		Previously played song begins playback
31	46	Song is played		Label changes to the title of the song being played.
32	47	Table is navigated		Labels change depending on the artist/album they are looking at

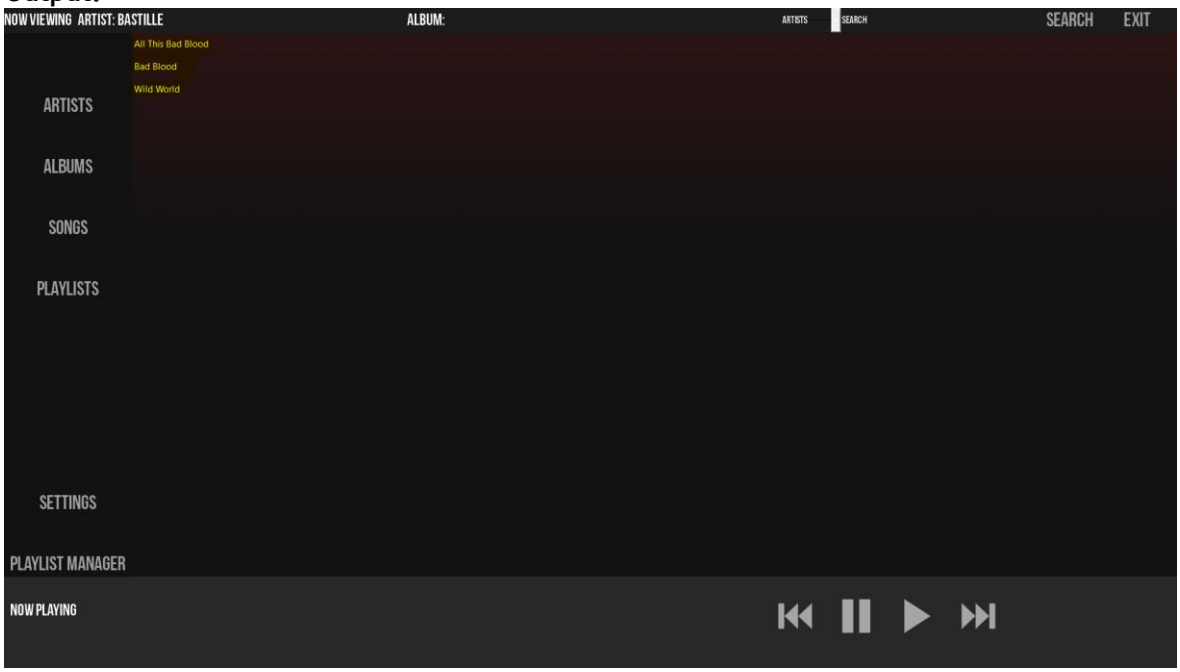
4.1.5.2 Test 15

Input:



The highlighted row ("Bastille") was double clicked.

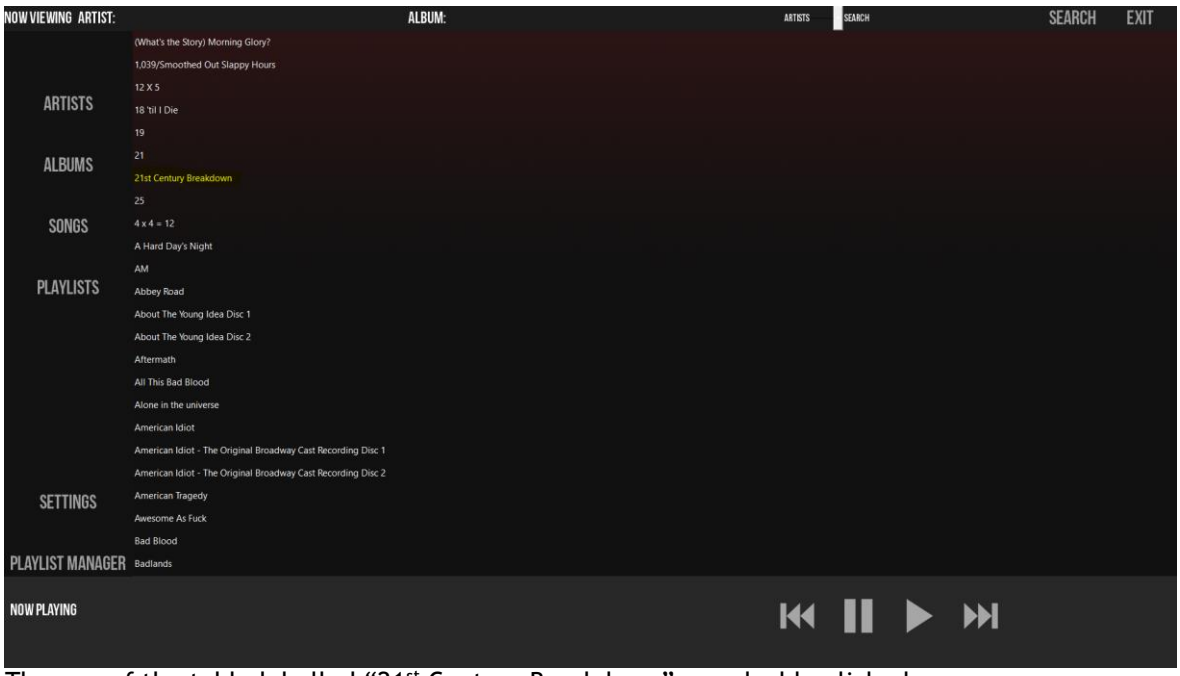
Output:



All of the albums by the selected artists ("Bastille") are shown in the songs table. Completing objective 26 of the success criteria.

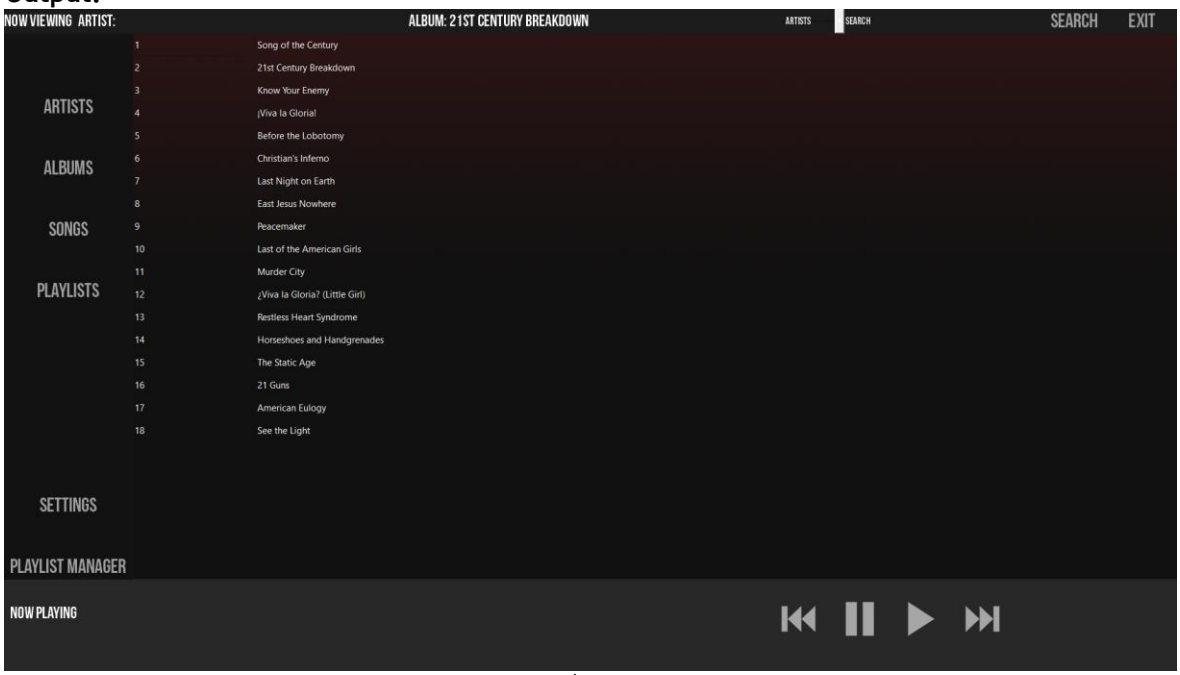
4.1.5.3 Test 16

Input:



The row of the table labelled "21st Century Breakdown" was double clicked.

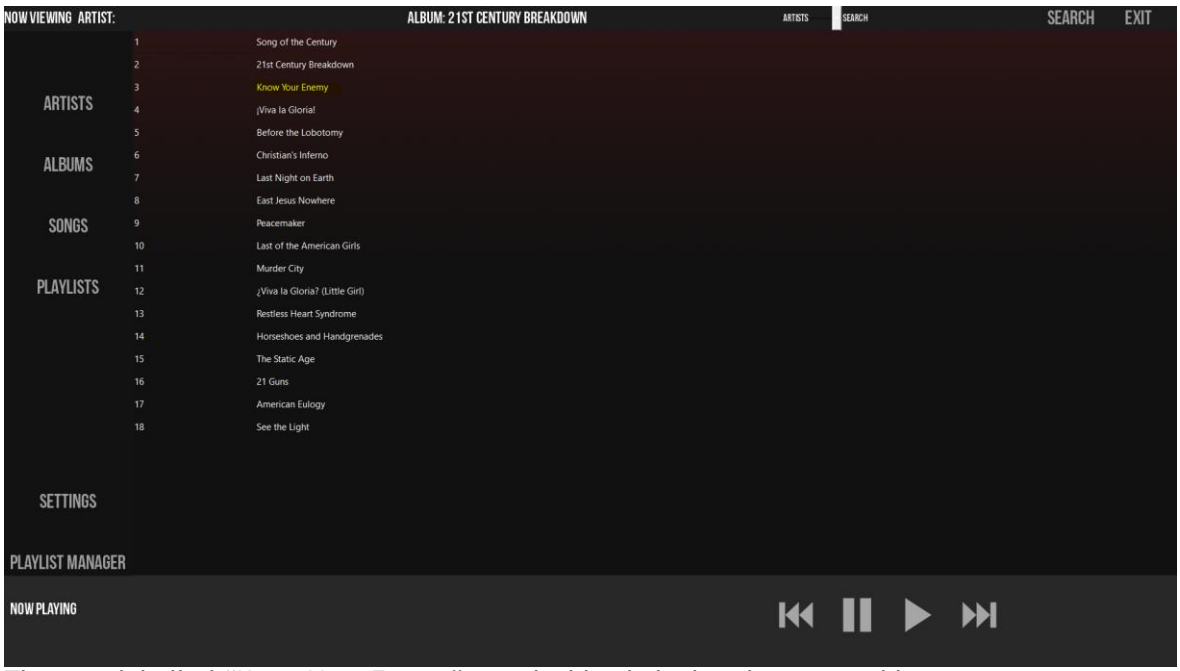
Output:



All of the songs in the album with the title "21st Century Breakdown" were displayed in the songs table. This proves the success of object 27 in the success criteria.

4.1.5.4 Test 17

Input:



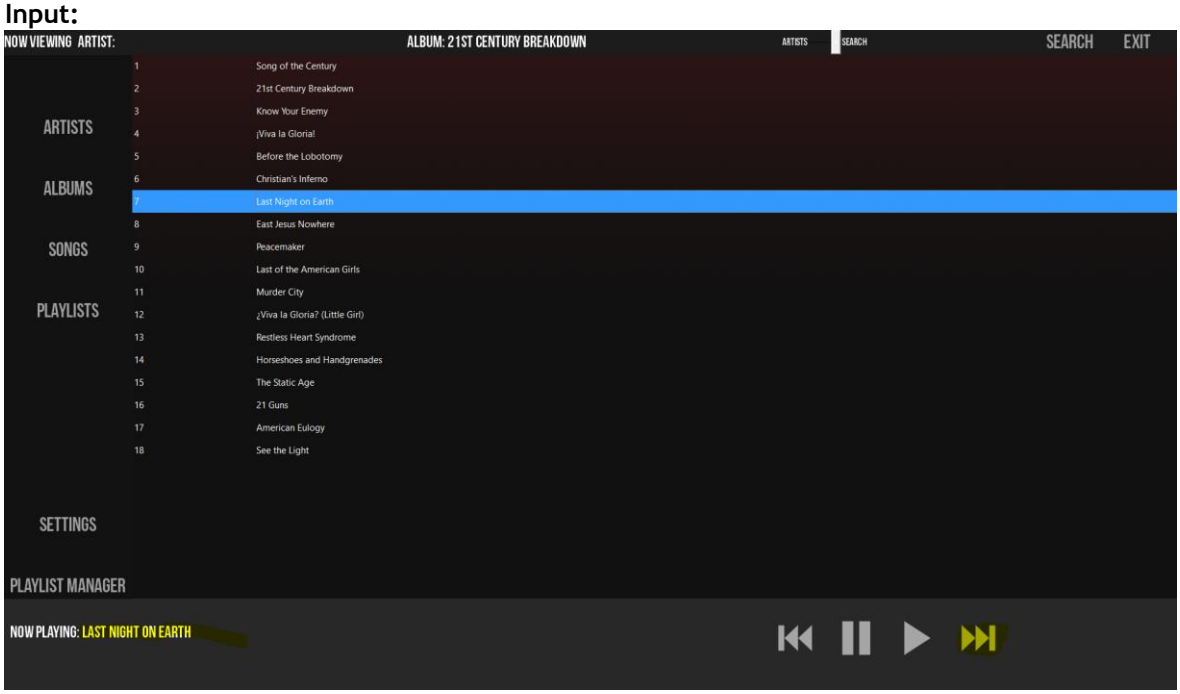
The song labelled “Know Your Enemy” was double clicked in the songs table.

Output:
To prove this test was successful (and prove that I met objective 28 of the success criteria), I found a stakeholder to sign off on the fact that music playback did occur.

Signed: Sam Barrett

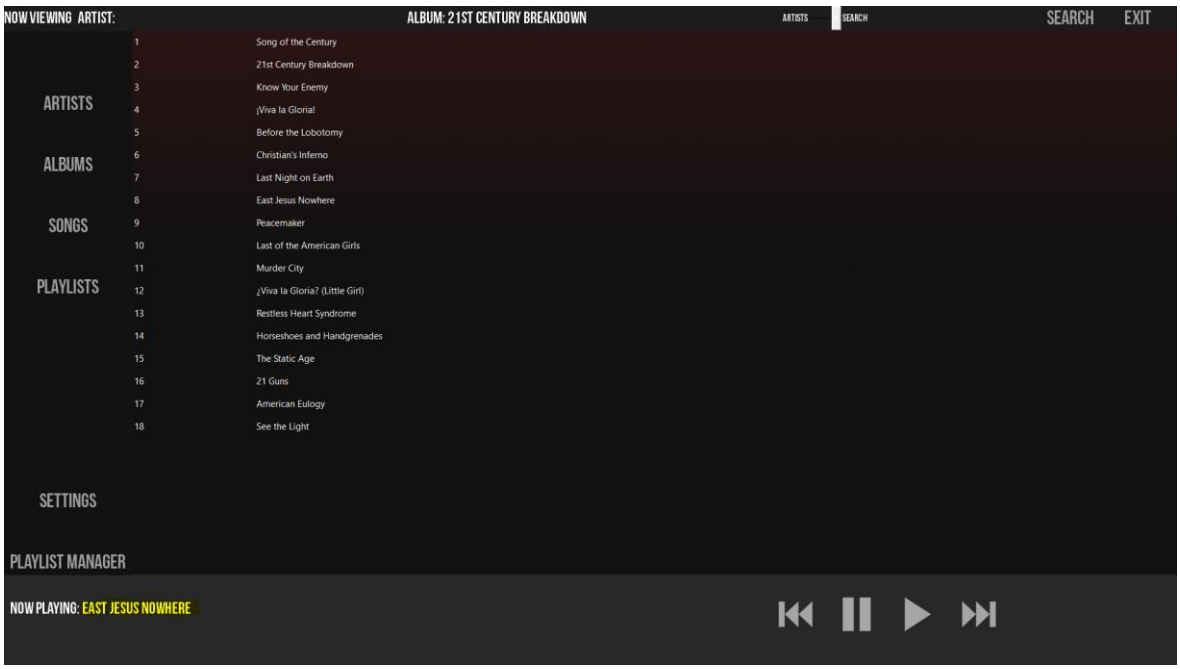
4.1.5.5 Test 18

Input:



The skip button was pressed while a song was currently playing (Last Night On Earth).

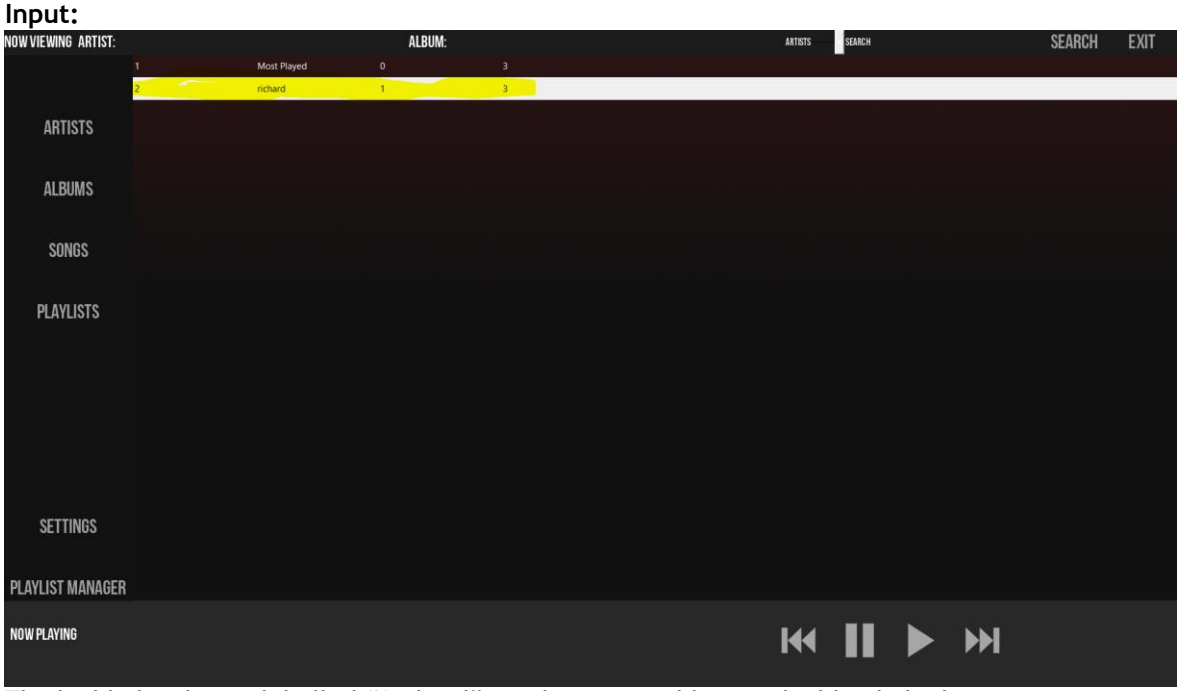
Output:



The next song in the table (East Jesus Nowhere) began to play. This shows that objective 29 and 43 of the success criteria have been met.

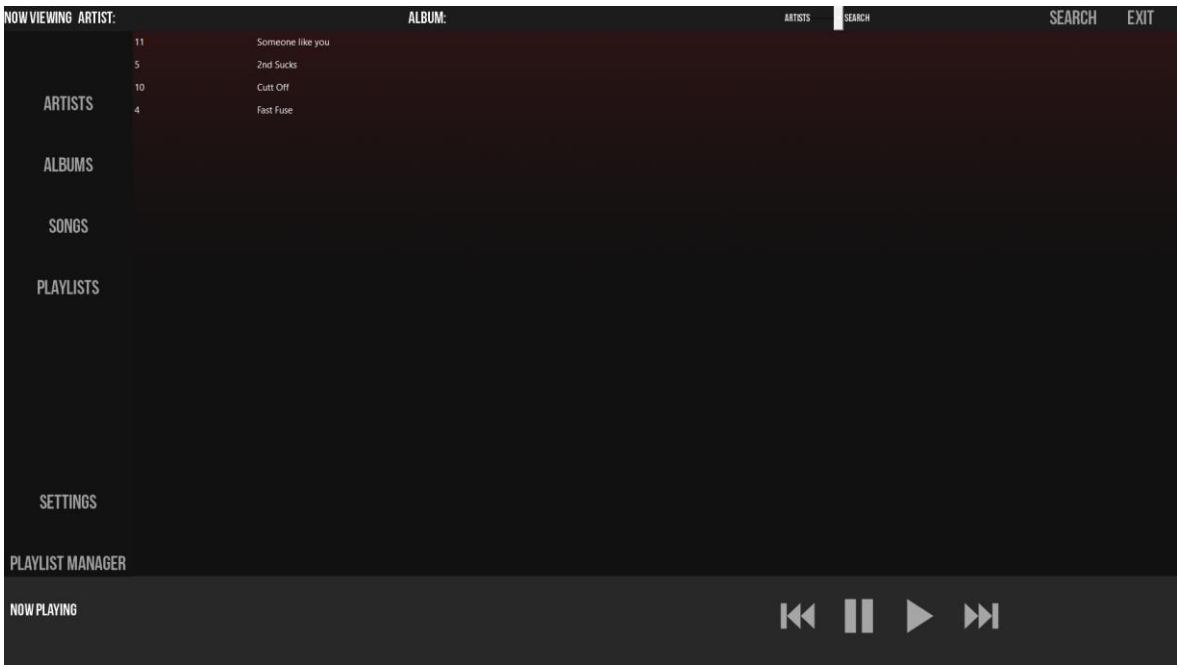
4.1.5.6 Test 19

Input:



The highlighted row (labelled “Richard”) in the songs table was double clicked.

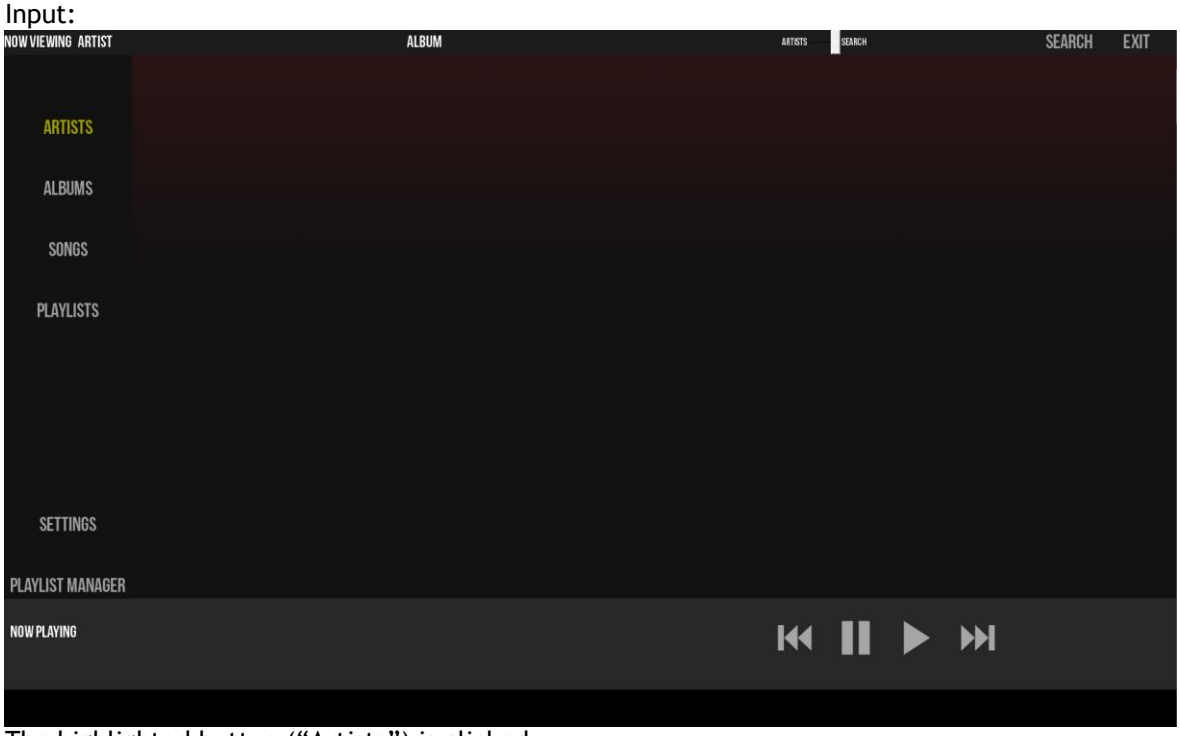
Output:



The songs from the "Richard" playlist are displayed in the songs table, showing that objective 30 from the success criteria has been met.

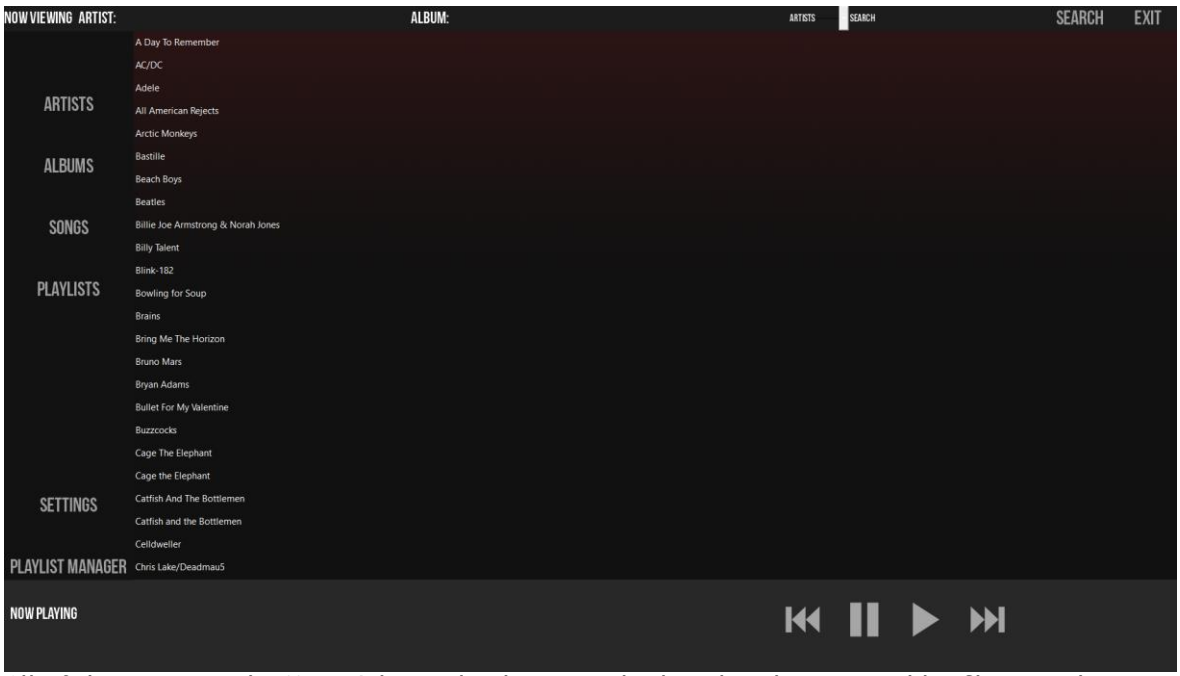
4.1.5.7 Test 20

Input:



The highlighted button ("Artists") is clicked.

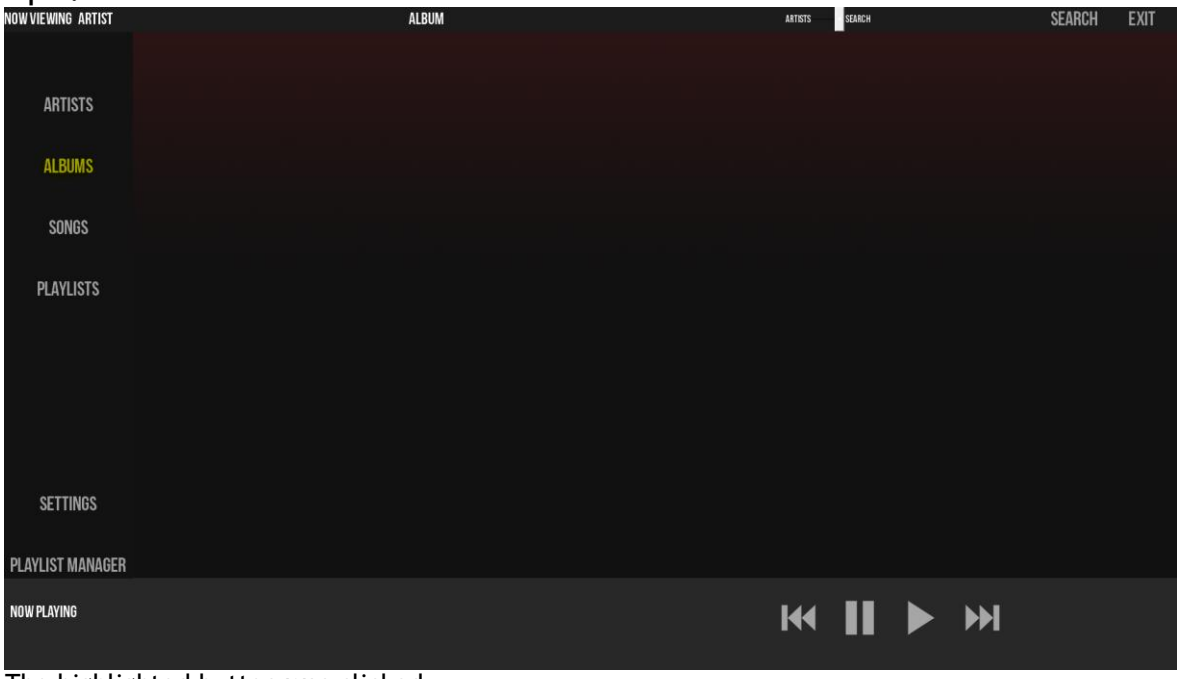
Output:



All of the artists in the Music Library database are displayed in the songs table. Showing that objective 32 has been met.

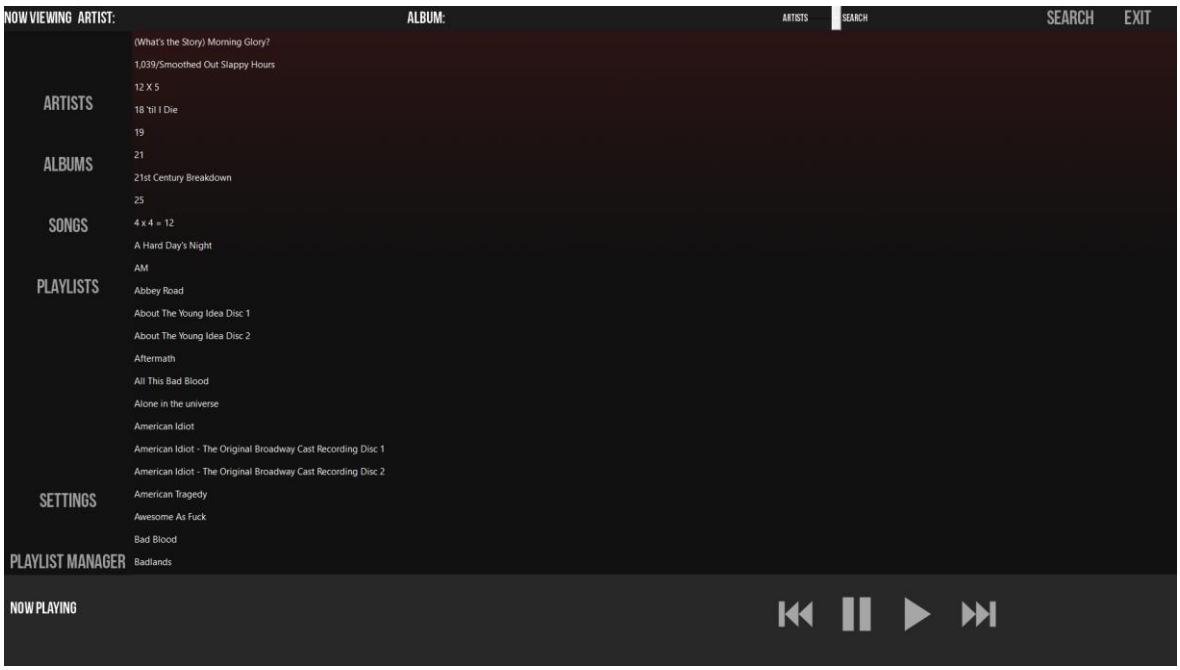
4.1.5.8 Test 21

Input:



The highlighted button was clicked.

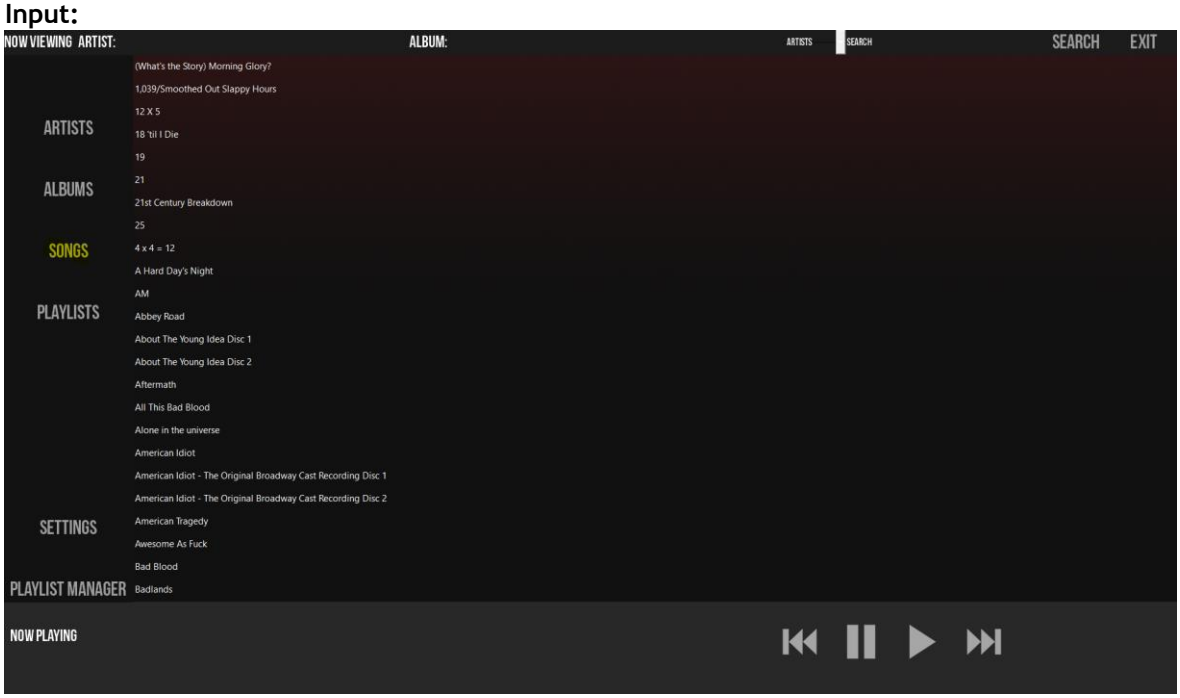
Output:



All of the albums in the music library database were displayed in the songs table. Showing that objective 32 of the success criteria has been met.

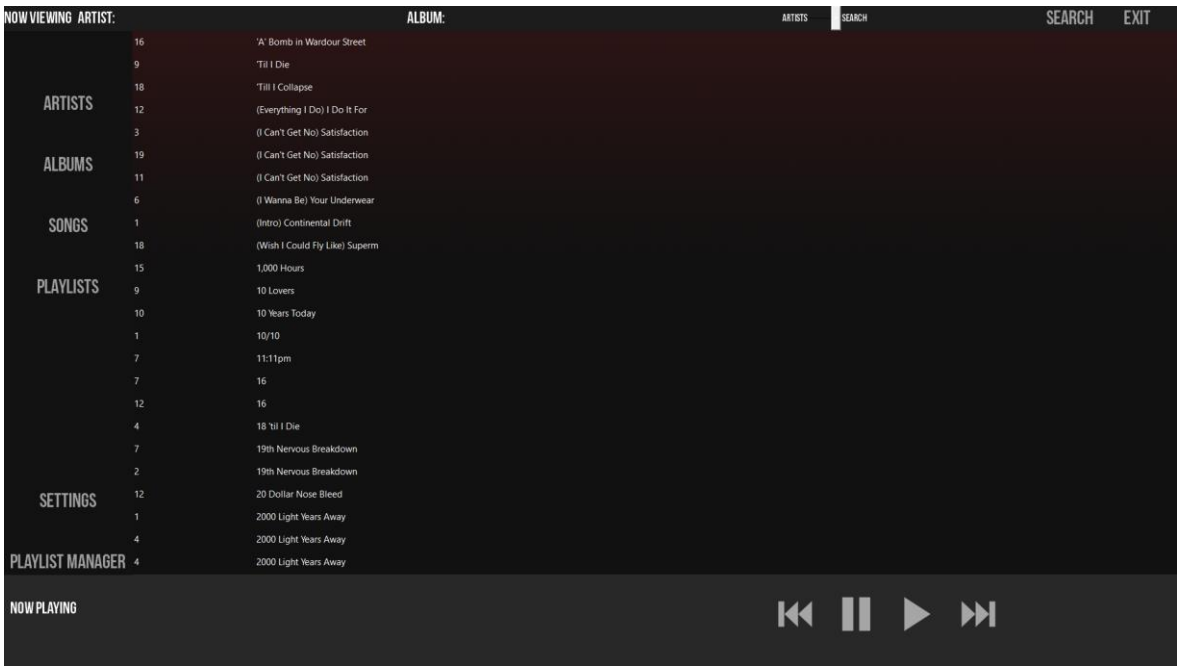
4.1.5.9 Test 22

Input:



The highlighted button was clicked.

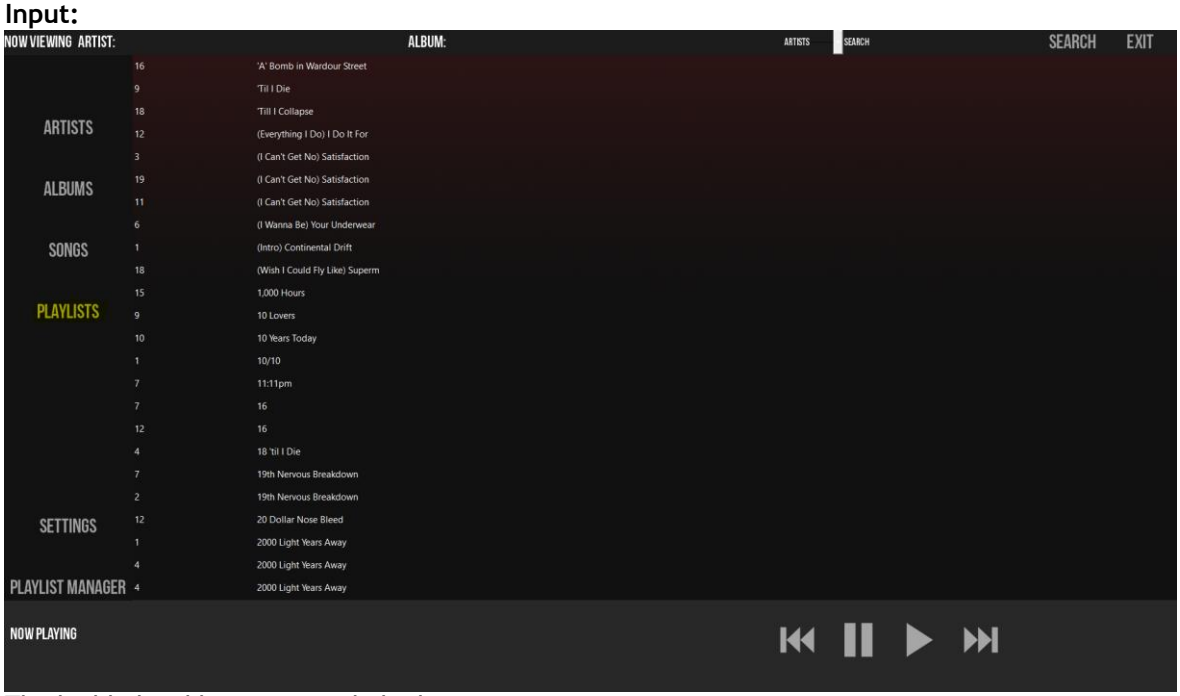
Output:



All of the songs in the music library database were displayed in the songs table. Showing that objective 33 of the success criteria has been met.

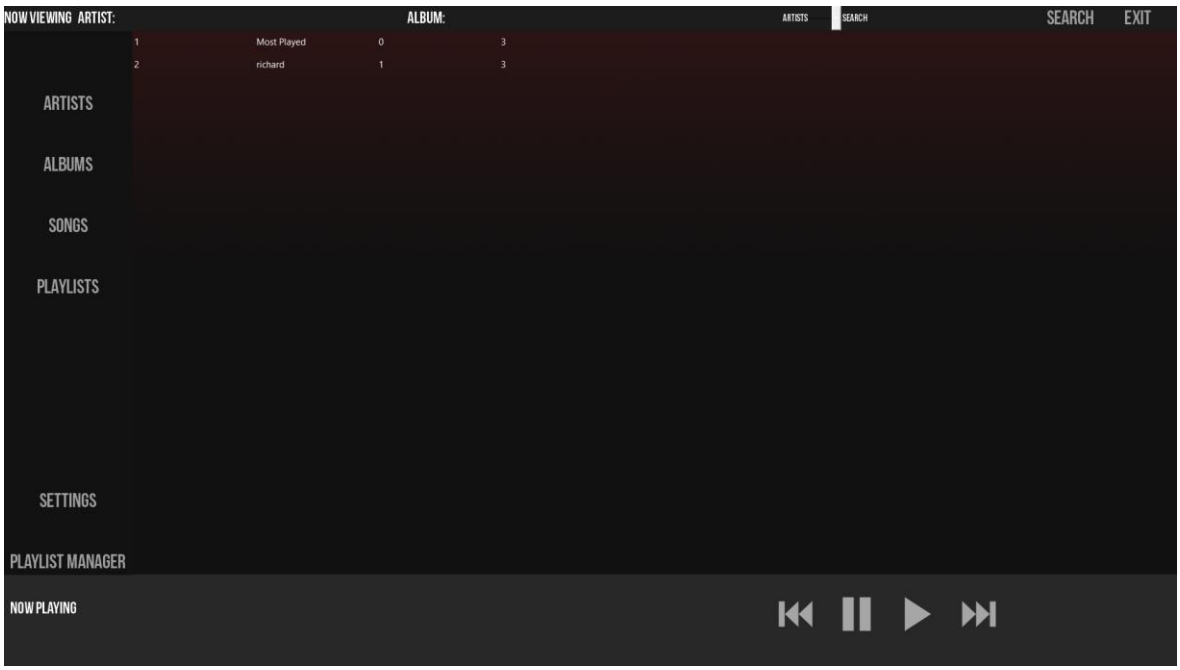
4.1.5.10 Test 23

Input:



The highlighted button was clicked.

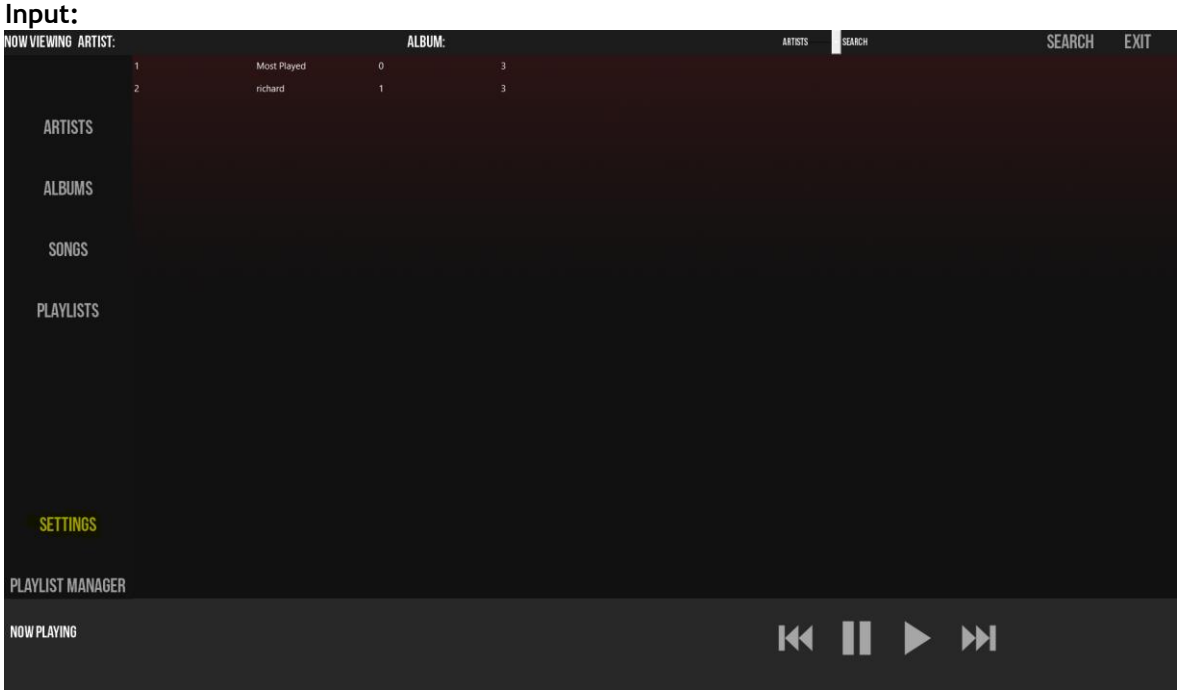
Output:



All of the playlists that the user has access to are shown in the table. Showing that objective 34 of the success criteria has been met.

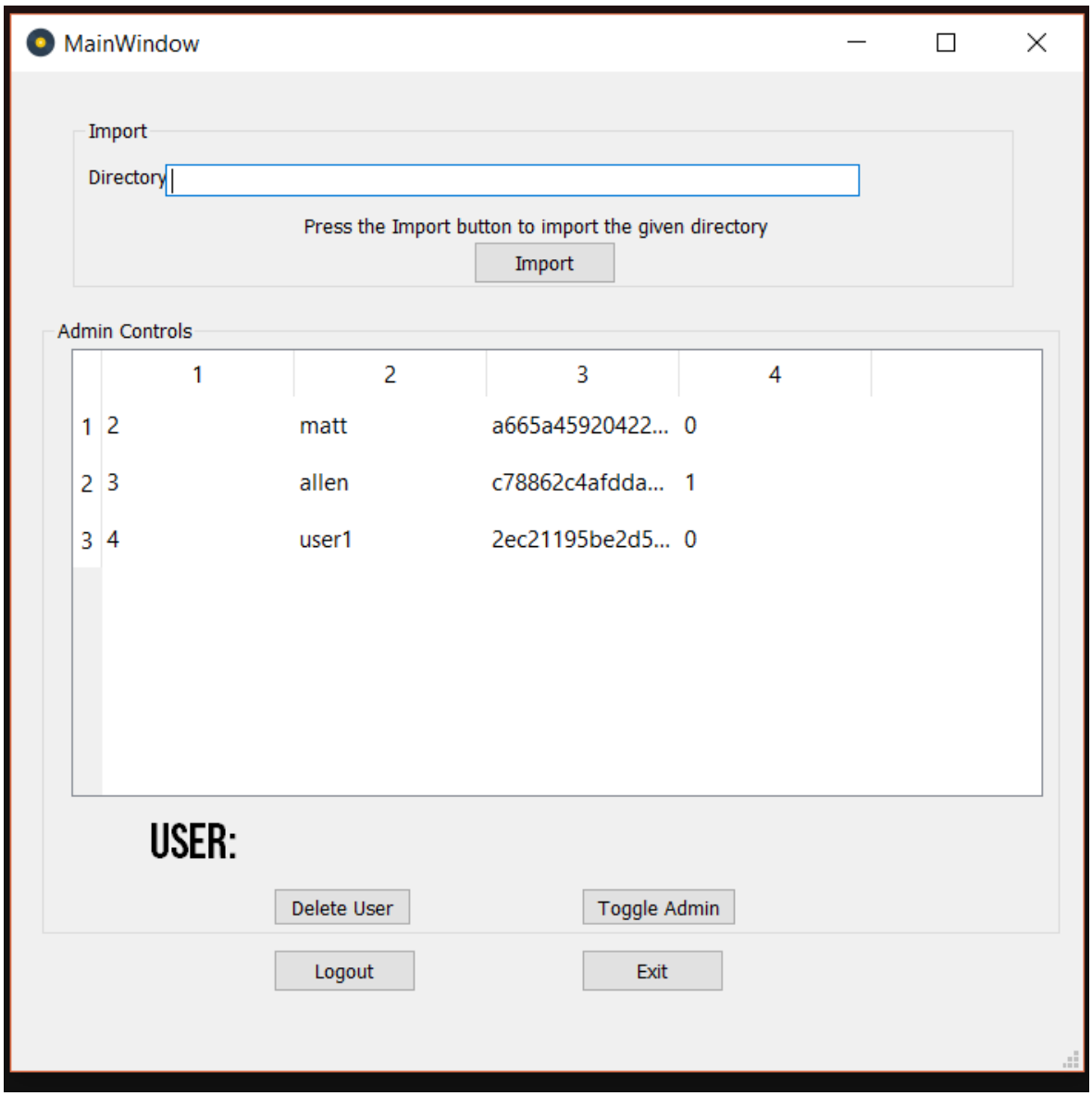
4.1.5.11 Test 24

Input:



The highlighted button was clicked.

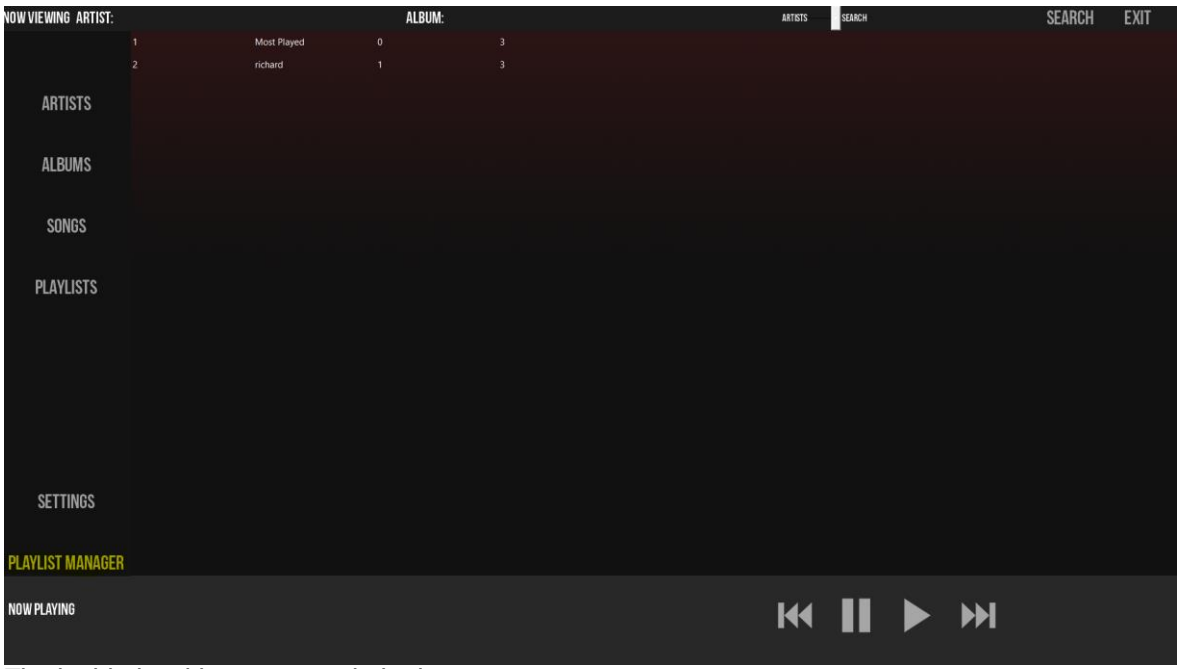
Output:



The settings window was displayed. Showing that objective 35 of the success criteria has been met.

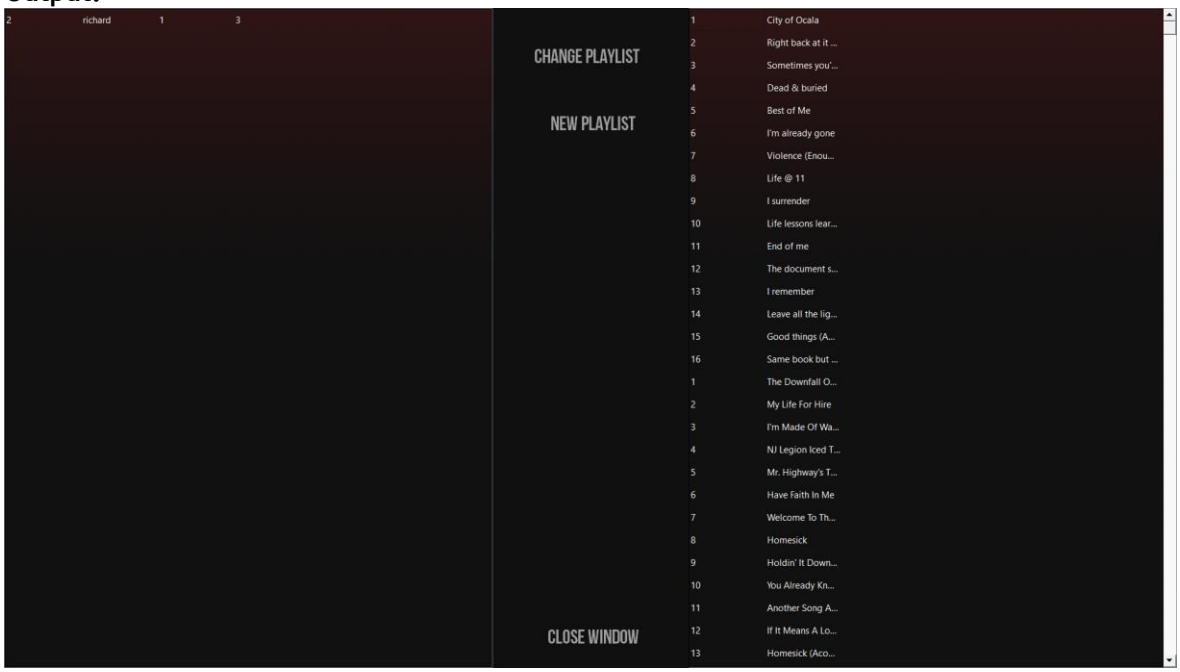
4.1.5.12 Test 25

Input:



The highlighted button was clicked.

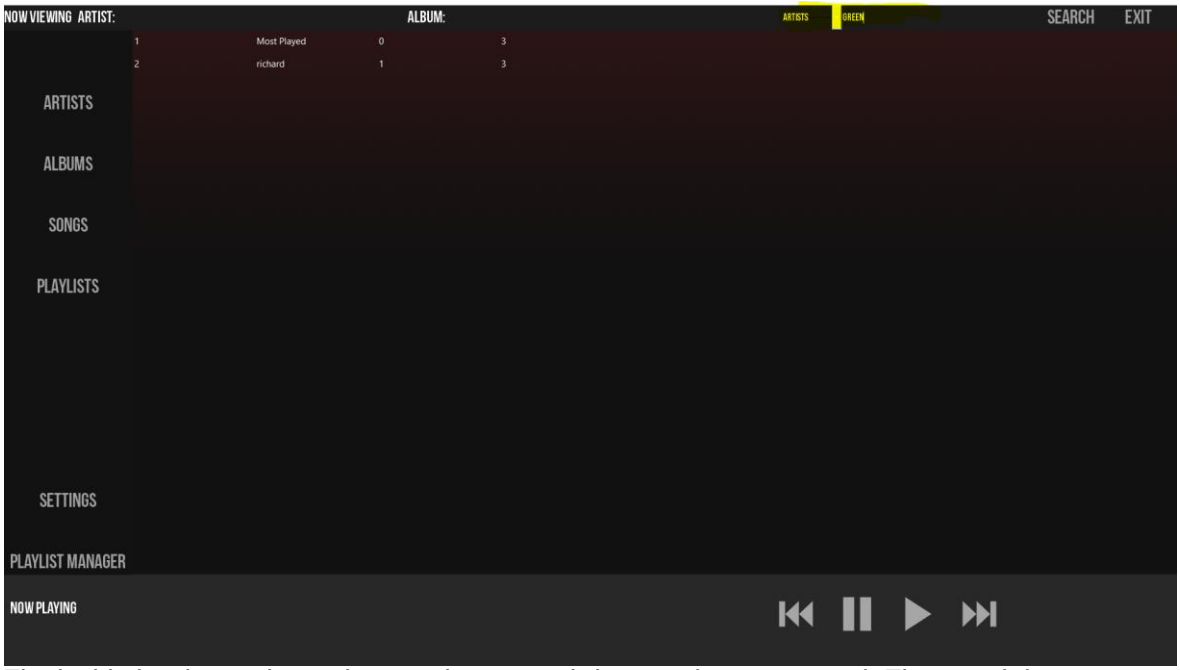
Output:



The playlist manager screen was shown, proving that objective 36 of the success criteria has been met.

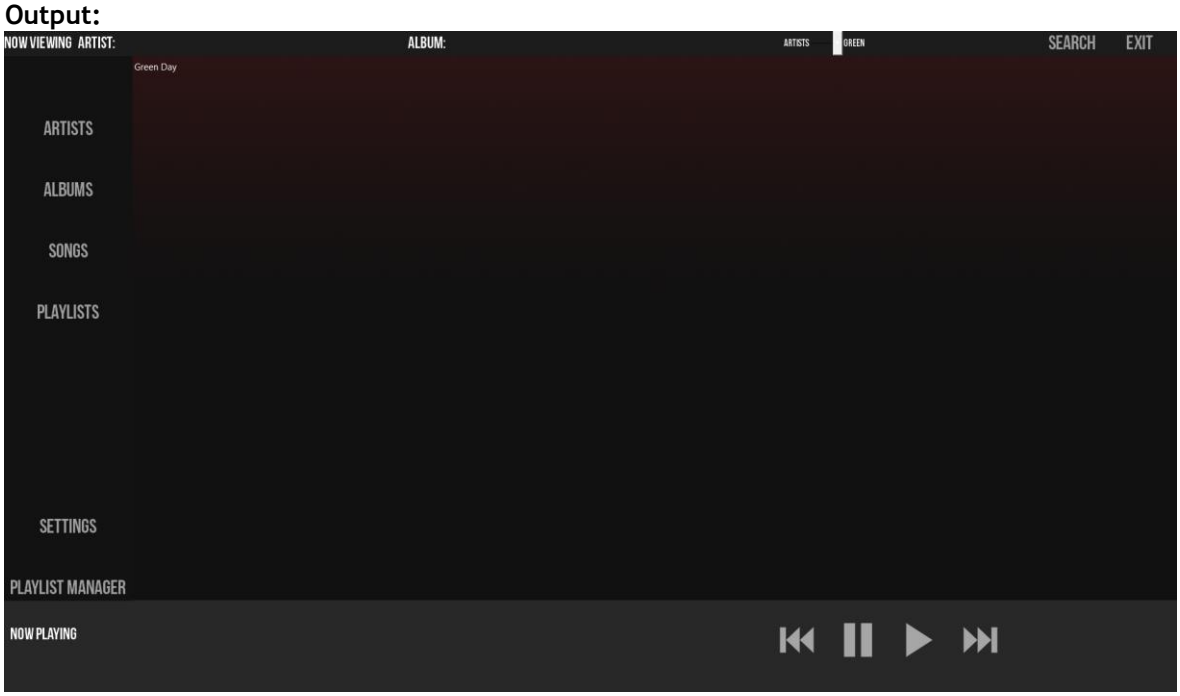
4.1.5.13 Test 26

Input:



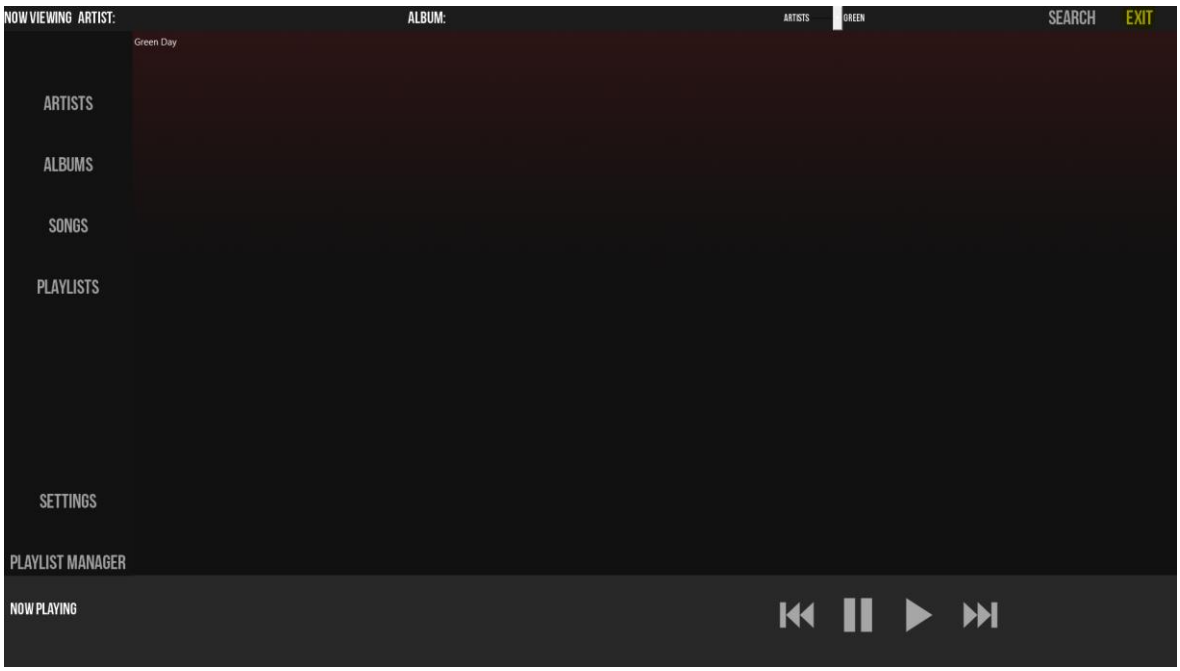
The highlighted text shows the search term and the search criteria used. The search button is then clicked.

Output:



The artists related to the search term (Green) are shown, in this case Green Day. This shows that objective 39 of the success criteria have been met.

4.1.5.14 Test 27
Input:

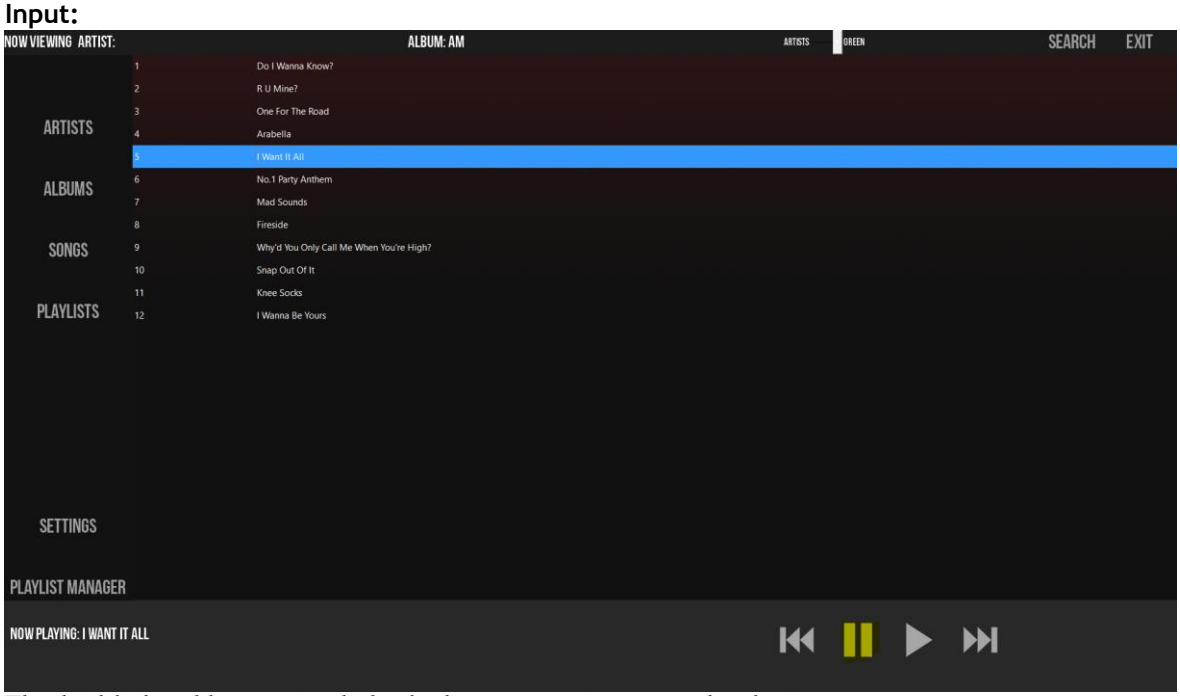


The highlighted button was clicked.

Output:
The main window was closed. Proving that objective 40 of the success criteria has been met.

4.1.5.15 Test 28

Input:



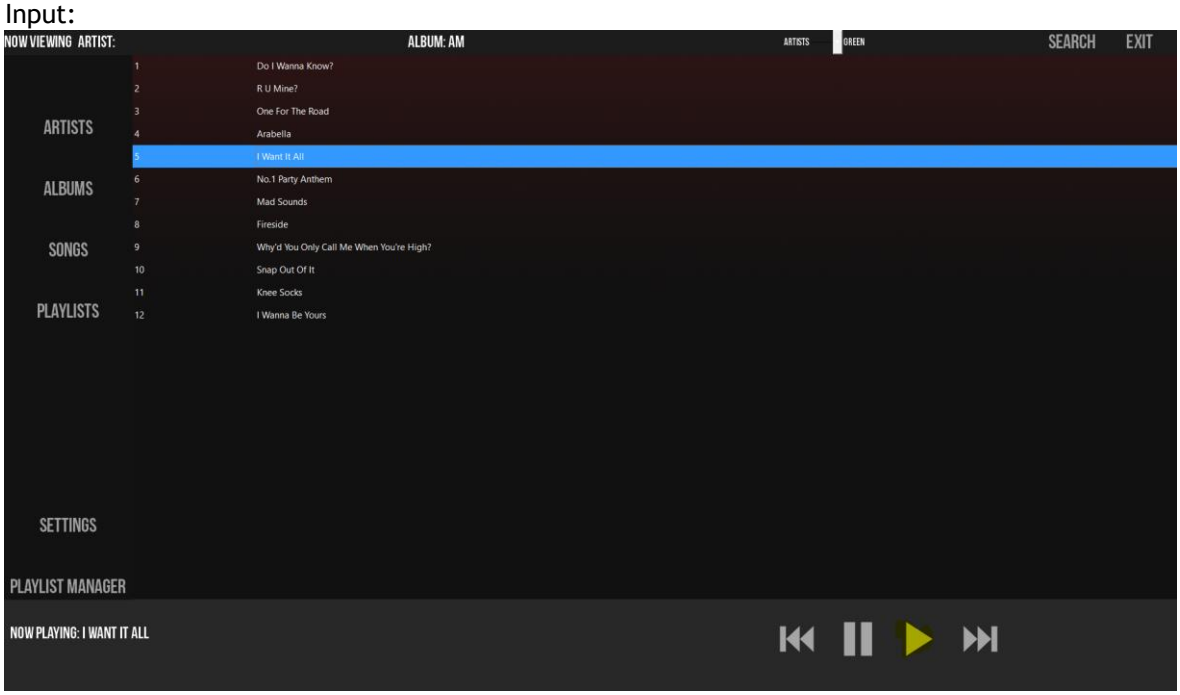
The highlighted button is clicked when a song is currently playing.

Output:
Once again, Samuel Barrett had to sign off on the fact that music playback did stop when the pause button was pressed.

Signed: Samuel Barrett

4.1.5.16 Test 29

Input:



The highlighted button was clicked.

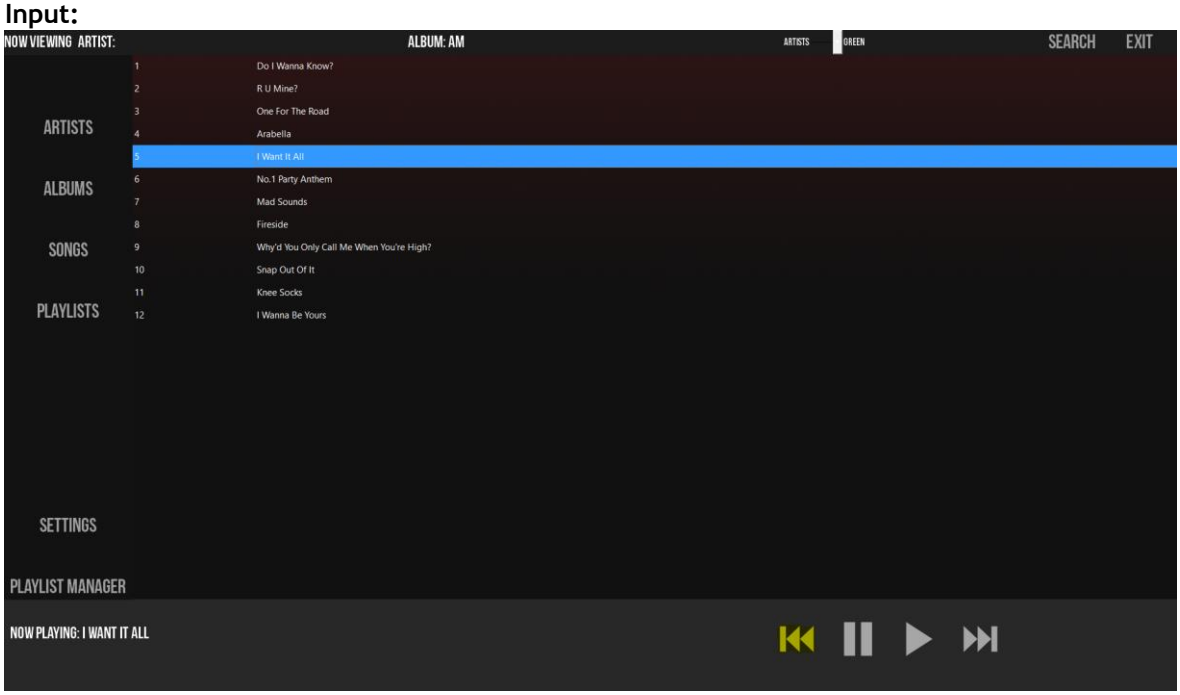
Output:

Samuel Barrett was used again to sign off on the fact that music playback of the song did resume, in the same place that it was paused.

Signed: Samuel Barrett

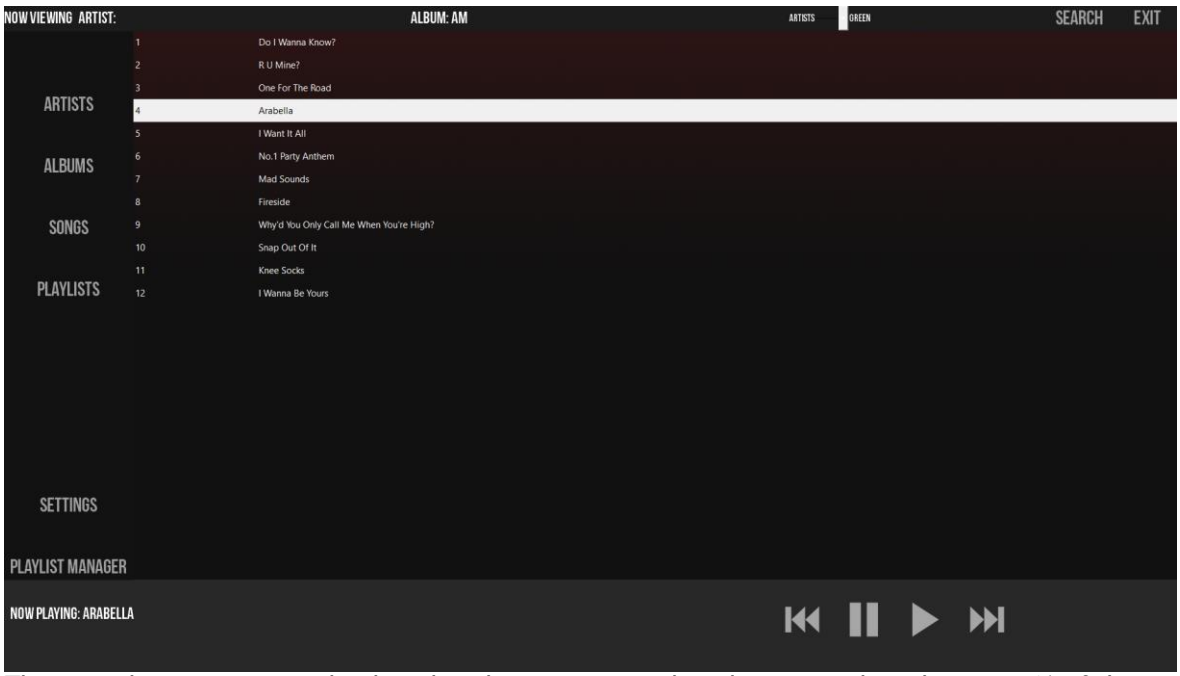
4.1.5.17 Test 30

Input:



The skip backward button is clicked.

Output:



The song that was previously played in the queue was played, proving that objective 46 of the success criteria was met.

4.1.5.18 Test 31 & 32

Throughout the testing, screenshots have shown the changing of the now playing label and now viewing labels. This proves that objectives 46 and 47 of the success criteria were met.

4.1.5.19 Conclusion

Test ID	Success Criteria Tested	Input	Branches	Expected Output	Test
15	26	Artist row is clicked in the table		All of the albums by artist in library are shown.	Success
16	27	Album row clicked in table		All songs in album are shown	Success
17	28	Song row clicked in table		Song begins playback	Success
18	29, 43	Skip forward button press		Next song in queue will begin playback	Success
19	30	Playlist row clicked in table		All song in playlist are shown	Success
20	31	Artist button pressed		All artists in music library are shown	Success
21	32	Albums button pressed		All albums in music library are shown	Success
22	33	Songs button pressed		All songs in music library are shown	Success

23	34	Playlists button pressed		All playlists owned by the user and the 'Most Played' playlist are shown	Success
24	35	Settings button press		Settings window opens	Success
25	36	Playlist Manager button pressed		Playlist Manager window opens	Success
26	39	Search button pressed with inputs for the search term and its criteria		Matches with the term and criteria will be displayed.	Success
27	40	Exit button pressed		The application will close	Success
28	41	Pause button pressed		Music playback stops	Success
29	42	Resume button pressed		Music playback **resumes**	Success
30	44, 45	Skip backward button pressed		Previously played song begins playback	Success
31	46	Song is played		Label changes to the title of the song being played.	Success
32	47	Table is navigated		Labels change depending on the artist/album they are looking at	Success

4.1.6 Settings Screen

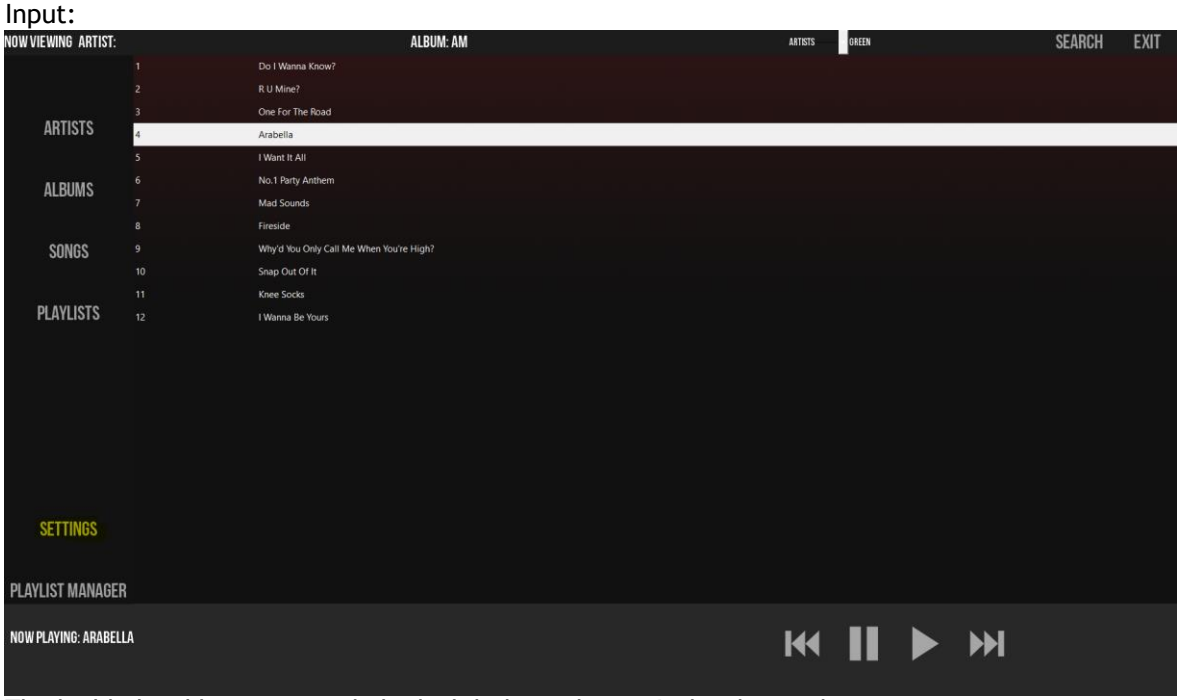
4.1.6.1 Objectives to be tested

Test Id	Success Criteria Tested	Input	Branches	Expected Output
33	48	Settings button in Main Window pressed while logged in with an administrator account.	a) Select user from the table b) Delete user c) Toggle admin of user	Administrator section is shown a) User is selected and label changes to the name of

				the user. b) User is deleted from database. c) User's administrator permissions are successfully toggled.
34	49	Directory Inputted and Import button pressed	b) Invalid directory	All tagged files in folders are directly inputted into the library b) Label will show: "Directory does not exist"
35	50	Exit button pressed		Settings window is closed
36	51	Logout button pressed		User is returned to the logout screen

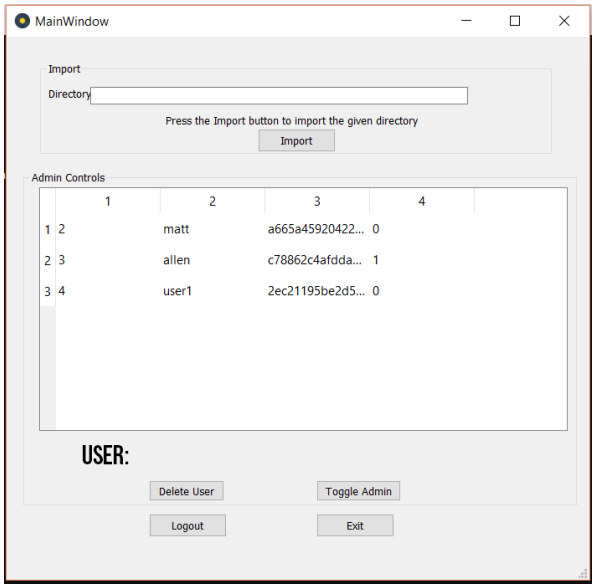
4.1.6.2 Test 33

Input:



The highlighted button was clicked while logged in as Richard (an administrator account).

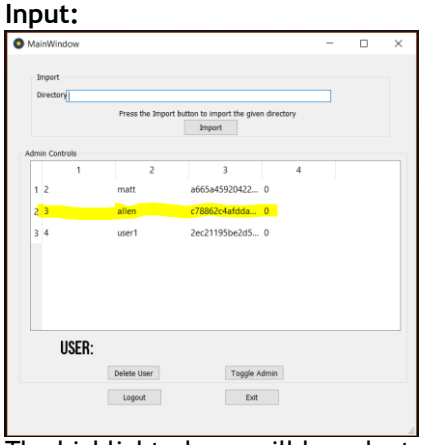
Output:



The settings window was opened, along with the administrator settings section.

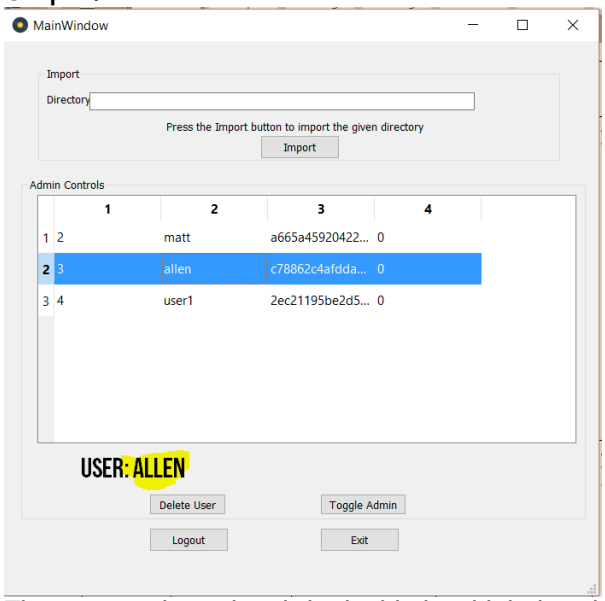
4.1.6.2.1 Section A

Input:



The highlighted row will be selected.

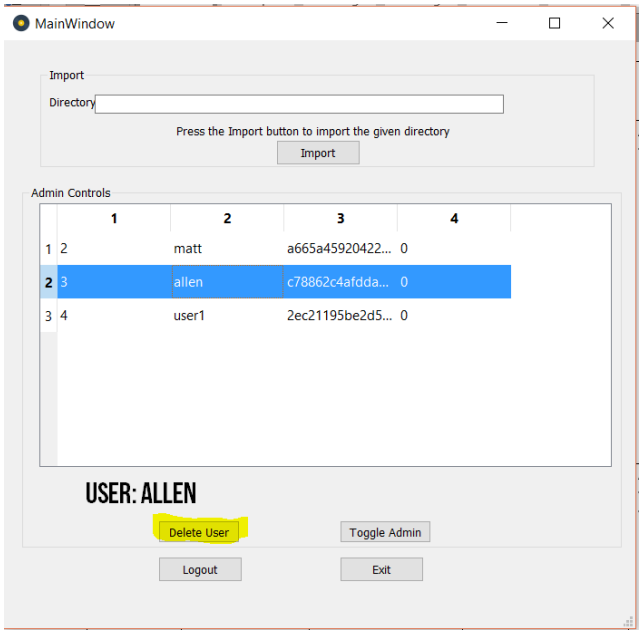
Output:



The row is selected and the highlighted label is changed to the name of the selected user.

4.1.6.2.2 Section B

Input:



The highlighted button is clicked.

Output:

The user is deleted from the list and the users table in the database.

4.1.6.2.3 Section C

Input:

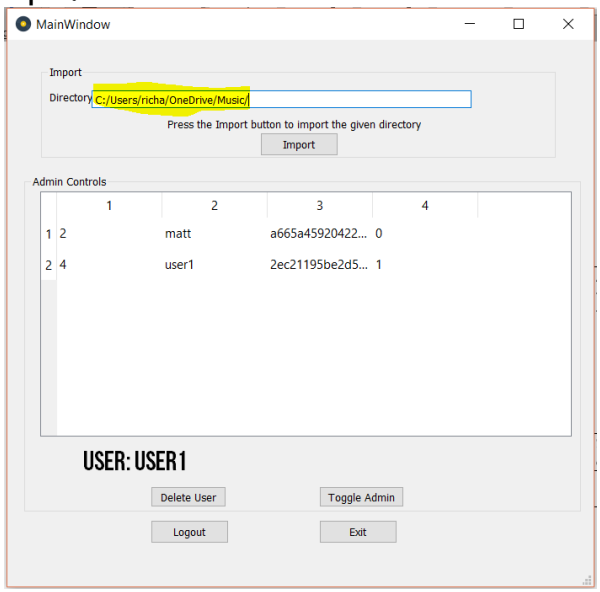
The toggle admin button is clicked with a user selected.

Output:

The admin permissions column is toggle from 0 to 1.

4.1.6.3 Test 34

Input:



A directory is inputted into the highlighted box.

Output:

The music library database is populated:

	SongID	TrackNumber	SongName	Genre	FileLocation	AlbumID	Length	Plays	Type
1	1	1	City of Ocala	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	209	0	0
2	2	2	Right back at it again	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	200	0	0
3	3	3	Sometimes you're the hammer, someti...	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	274	0	0
4	4	4	Dead & buried	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	193	0	0
5	5	5	Best of Me	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	207	0	0
6	6	6	I'm already gone	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	244	0	0
7	7	7	Violence (Enough is enough)	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	241	0	0
8	8	8	Life @ 11	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	202	0	0
9	9	9	I surrender	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	214	0	0
10	10	10	Life lessons learned the hard way	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	137	0	0
11	11	11	End of me	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	237	0	0
12	12	12	The document speaks for itself	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	283	0	0
13	13	13	I remember	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	544	0	0
14	14	14	Leave all the lights on (Additional...	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	211	0	0
15	15	15	Good things (Additional track)	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	179	0	0
16	16	16	Same book but never the same page (...	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	1	244	0	0
17	17	1	The Downfall Of Us All	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	2	209	0	0
18	18	2	My Life For Hire	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	2	213	0	0
19	19	3	I'm Made Of Wax Larry, What Are You...	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	2	180	0	0
20	20	4	NJ Legion Iced Tea	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	2	211	0	0
21	21	5	Mr. Highway's Thinking About The End	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	2	255	0	0
22	22	6	Have Faith In Me	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	2	188	0	0
23	23	7	Welcome To The Family	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	2	179	0	0
24	24	8	Homesick	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	2	236	0	0
25	25	9	Holdin' It Down For The Underground	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	2	203	0	0
26	26	10	You Already Know What You Are	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	2	87	0	0
27	27	11	Another Song About The Weekend	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	2	225	0	0
28	28	12	If It Means A Lot To You	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	2	243	0	0
29	29	13	Homesick (Acoustic - Bonus Track)	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	2	247	0	0
30	30	14	Another Song About The Weekend (Aco...	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	2	222	0	0
31	31	1	Sticks & Bricks	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	3	196	0	0
32	32	2	All I Want	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	3	202	0	0
33	33	3	It's Complicated	Hardcore	C:/Users/richa/OneDrive/Music/A Day...	3	177	0	0

	ArtistID	ArtistName	Type
1	1	A Day To Remember	1
2	2	AC/DC	1
3	3	Adele	1
4	4	All American Rejects	1
5	5	The All-American Rejects	1
6	6	The Animals	1
7	7	Arctic Monkeys	1
8	8	Bastille	1
9	9	The Beach Boys	1
10	10	Beach Boys	1
11	11	Beatles	1
12	12	The Beatles	1
13	13	Billie Joe Armstrong & Norah Jones	1
14	14	Billy Talent	1
15	15	The Black Keys	1
16	16	Blink-182	1
17	17	The Blues Brothers	1
18	18	James Brown	1
19	19	Bowling for Soup	1
20	20	Brains	1
21	21	The Brains	1
22	22	Bring Me The Horizon	1
23	23	Bruno Mars	1
24	24	Bryan Adams	1
25	25	Bullet For My Valentine	1
26	26	Buzzcocks	1
27	27	Cage the Elephant	1
28	28	Cage The Elephant	1
29	29	Catfish And The Bottlemen	1
30	30	Catfish and the Bottlemen	1
31	31	Celldweller	1
32	32	Cream	1
33	33	Creedence Clearwater Revival	1

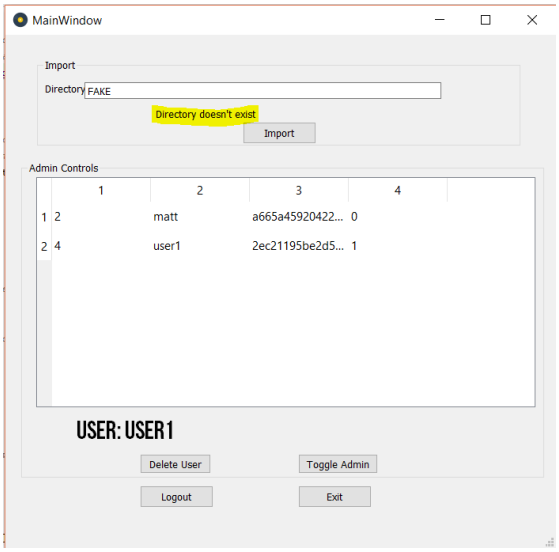
	AlbumID	AlbumName	ArtistID	Type
1	1	Common courtesy	1	2
2	2	Homesick	1	2
3	3	What Separates Me From You	1	2
4	4	Black Ice	2	2
5	5	19	3	2
6	6	21	3	2
7	7	25	3	2
8	8	Move Along	4	2
9	9	When The World Comes Down	5	2
10	10	The Most Of The Animals	6	2
11	11	AM	7	2
12	12	Favourite Worst Nightmare	7	2
13	13	Humbug	7	2
14	14	Suck It And See	7	2
15	15	Whatever People Say I Am, That's What I'm Not	7	2
16	16	Bad Blood	8	2
17	17	All This Bad Blood	8	2
18	18	Wild World	8	2
19	19	Pet Sounds	9	2
20	20	Surf's Up	10	2
21	21	A Hard Day's Night	11	2
22	22	Abbey Road	12	2
23	23	Beatles for Sale	12	2
24	24	Help!	12	2
25	25	Let It Be	11	2
26	26	Magical Mystery Tour	11	2
27	27	Past Masters Vol 1	11	2
28	28	Past Masters, Vol. 2	12	2
29	29	Please Please Me	11	2
30	30	Revolver	11	2
31	31	Rubber Soul	12	2
32	32	Sgt. Pepper's Lonely Hearts Club Band	12	2
33	33	The Beatles (White Album)	12	2

4.1.6.3.1 Section A

Input:

An invalid directory is inputted in the highlighted box and the import button is clicked.

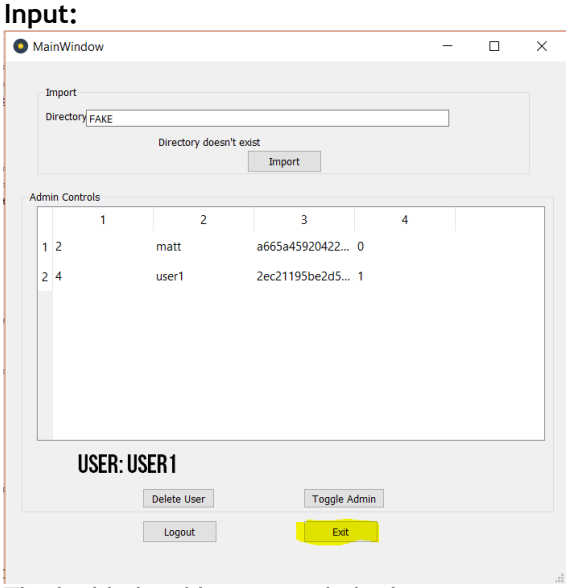
Output:



The highlighted label notifies the user that the inputted directory is invalid/does not exist.

4.1.6.4 Test 35

Input:

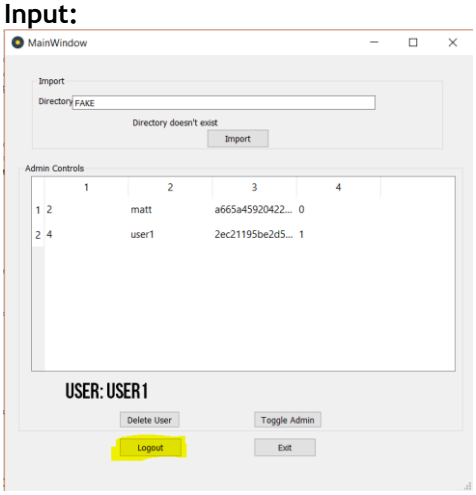


The highlighted button is clicked.

Output:
The settings window is closed, leaving only the main window showing.

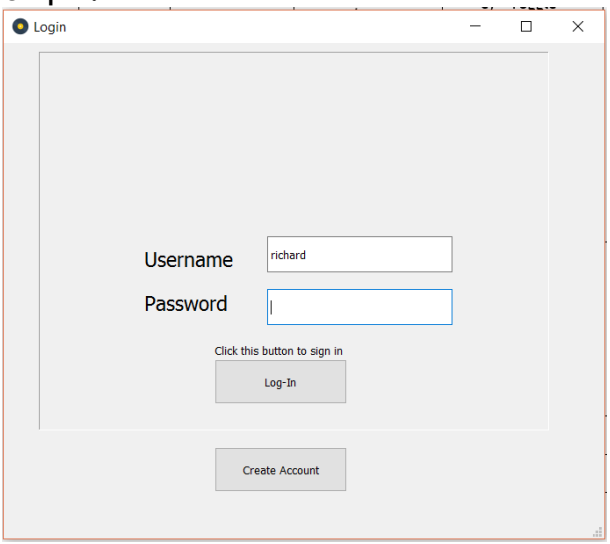
4.1.6.5 Test 36

Input:



The highlighted button is clicked.

Output:



The settings window and main window were closed. The login window was then shown.

4.1.6.6 Conclusion

Test Id	Success Criteria Tested	Input	Branches	Expected Output	Verdict
33	48	Settings button in Main Window pressed while logged in with an administrator account.	d) Select user from the table e) Delete user f) Toggle admin of user	Administrator section is shown d) User is selected and label changes to the name of the user. e) User is deleted from database. f) User's administrator permissions are successfully toggled.	Success
34	49	Directory Inputted and Import button pressed	c) Invalid directory	All tagged files in folders are directly inputted into the library c) Label will show: "Directory does not exist"	Success
35	50	Exit button pressed		Settings window is closed	Success
36	51	Logout button pressed		User is returned to the logout screen	Success

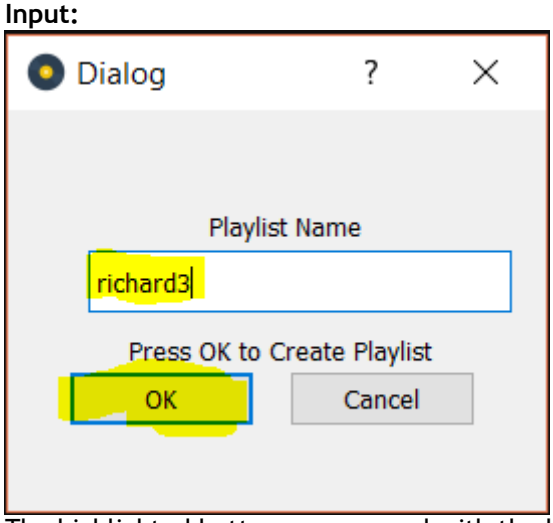
4.1.7 Playlist Dialog Box Screen

4.1.7.1 Objectives to be tested

Test ID	Success Criteria Tested	Input	Branches	Expected Output
43	61	OK button pressed	b) Playlist name is under 5 characters	Playlist Name dialog box closes and playlist is created. b) Label shows "Playlist name is too short"
44	62	Cancel button pressed		Playlist Name dialog box is closed

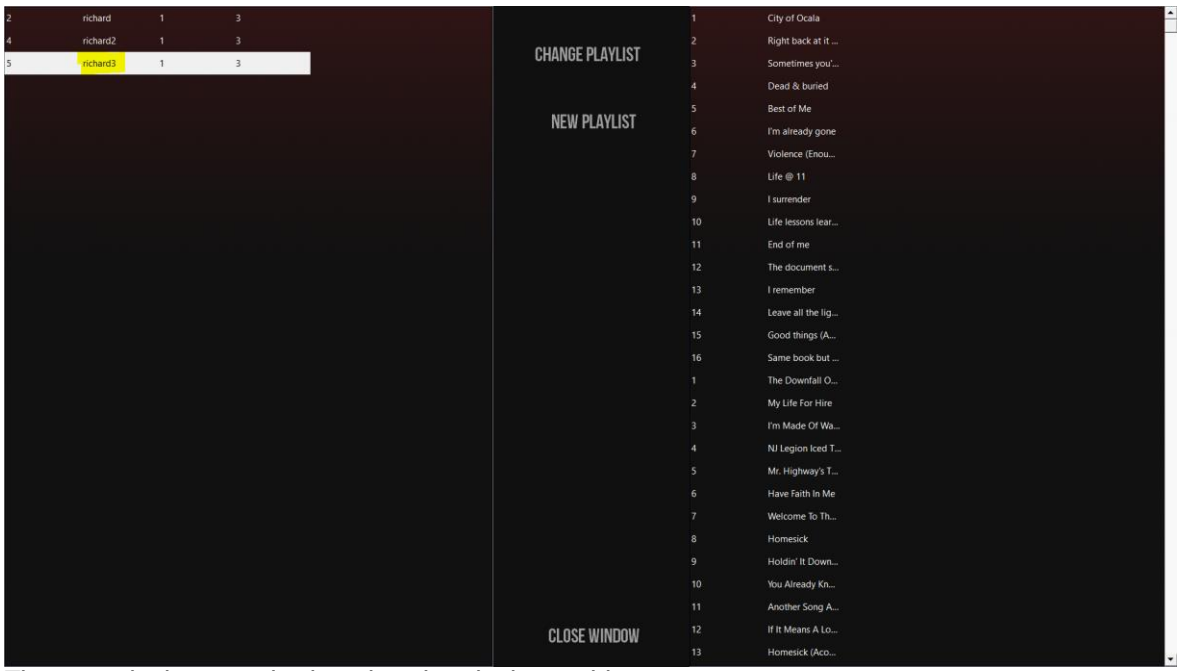
4.1.7.2 Test 43

Input:



The highlighted button was pressed with the highlighted input.

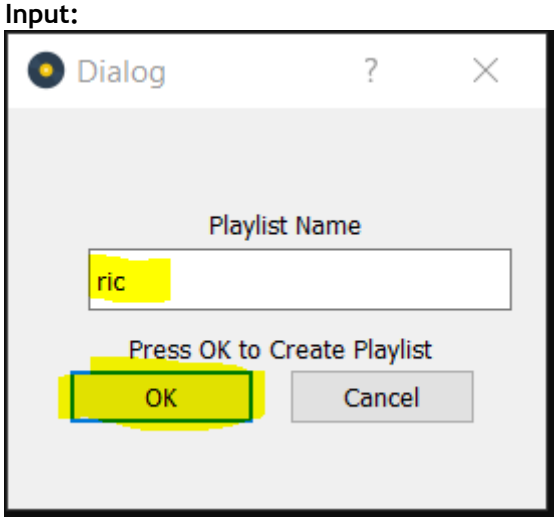
Output:



The new playlist was displayed in the playlists table.

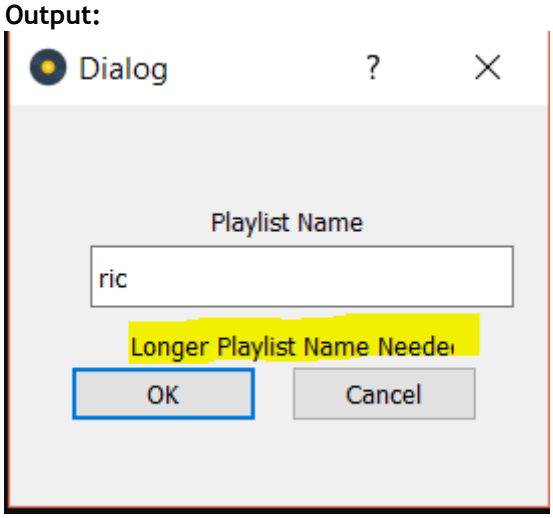
4.1.7.2.1 Section A

Input:



A three word playlist name was inputted and the OK button was clicked.

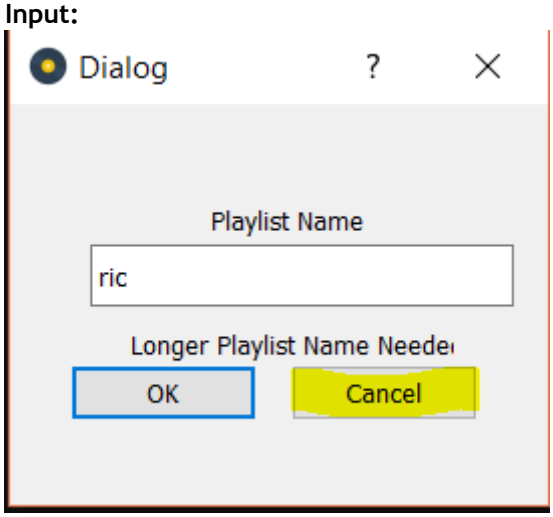
Output:



The highlighted label notified the user that the playlist name was too short.

4.1.7.3 Test 44

Input:



The highlighted button was clicked.

Output:
The playlist name dialog box was closed.

4.1.7.4 Conclusion

Test ID	Success Criteria Tested	Input	Branches	Expected Output	Test
43	61	OK button pressed	a) Playlist name is under 5 characters	Playlist Name dialog box closes and playlist is created. a) Label shows "Playlist name is too short"	Success
44	62	Cancel button pressed		Playlist Name dialog box is closed	Success

4.2 Conclusion of the Solution

In conclusion, all of the success criteria for the final solution have been met, meaning that the program is a full success. Despite this, I still believe that more features could have been added to the program to increase its functionality and its target market. However, in order to maintain the program's performance and the requirements of the stakeholders, as well as being able to meet the solutions time limit, these features had to be dropped. Despite this, more objectives could have been included in the final solution.

4.3 User Feedback

In order to gain a varied amount of feedback, the final solution was distributed to a selection of the stakeholders and their use of the program was overseen by myself. I found that this third-party testing to be very successful.

Samuel Barrett (one of the stakeholders) stated that the solution was "Exactly what I needed for my media player. It was lightweight and didn't have all of those useless features that Media Player and ITunes have. However, it still felt that it kept some key modern features that exist in those two programs."

Throughout this testing, no logical errors were produced from the running of the program and many users gave positive reviews of the application. Overall, 8 out of the 10 users said that they would use this application over the market leaders, and all the users believed that the solution was a capable market product.

However, this user testing did highlight some flaws in the programming. Many user found that they could not play music files with inverted question marks and exclamation points. Research into this problem showed that this is a problem of the character sets that the pygame library understands and therefore could not be fixed. In later iterations of the program, it may be possible to fix this in later iterations with the use of an independent music playback library for the program.

4.4 Performance Testing

The program does not have a lot of features that can be compared in terms of speed with other competitors to this solution. However, the import routine can:

Media Player Solution: **36.07 seconds**

Window Media Player: 1.5 minutes

ITunes: 2.75 minutes

So, the developed solution was the fastest at importing my personal music library. It is important to note that Windows Media Player and ITunes also retrieve artwork from the internet, making the process longer. However, I believe this achievement is still noteworthy in its own right.

4.5 Limitations of the Final Solution

First, the solution could be improved by using a compiled language such as C rather than and interpreted language such as python. Through the development, I ran into problems with the inability to run multiple iterations or functions at once. Though multithreading could be used to help this in python, the pressure that this puts on the processor is noteworthy. Furthermore, the increased functionality that professional languages such as C++ provide could aid in making the program more efficient and more competitive in the market.

Second, many similar solutions to the media player problem have internet functionality, meaning they can access online database to retrieve missing information on music and provide online marketplaces to buy songs. I feel that with more time developing the project, I could have implemented these features fairly easily. However, the time constraints and the various limits to my skills at this time would have made this job damaging to the integrity of the final solution.

I do also believe that some of the objectives were only partially met. The development was under time constraints, meaning that a lot of the code was implemented in a way that was not fully efficient. For example, Hungarian notation was only implemented in certain parts of the code, reducing its maintainability and hindering its modification.

4.6 Maintainability of the Solution

As well as the features mentioned in the limitations of the solution, I believe that several changes could be made in later iterations. First, the application could be modified to allow it to run on a mobile device. The fact that it is programmed in python would be suitable for this as python is a cross platform programming language. This modification would have to see slight changes to the user interface and the overall flow of the program, but it would have increased the programs target market and therefore consumption.