Object Oriented Programming: Otodecks

1. Requirement 1

1.1. R1A: Can load audio files into audio players

```
if (button == &loadButton) {
    auto fileChooserFlags = FileBrowserComponent::canSelectFiles;
    fChooser.launchAsync(fileChooserFlags, [this](const FileChooser& chooser) {
        URL track = URL(chooser.getResult());
        player->loadURL(track);
        waveformDisplay.loadURL(track);
        titleLabel.setText("Now Playing: " + track.getFileName(), dontSendNotification);
    });
}
```

The screenshot above shows the load button and the codes for the load button. For the fileChooserFlags, I decided to go with canSelectFiles instead of canSelectMultipleItems as this load function only loads to DeckGUI, so there isn't any need to be able to load multiple tracks. The file retrieved from launchAsync will be converted to a URL and used to load into the player and display the waveform.

```
void DJAudioPlayer::loadURL(URL audioURL)
{
    auto* reader = formatManager.createReaderFor(audioURL.createInputStream(false));
    if (reader != nullptr) // good file!
    {
        std::unique_ptr<AudioFormatReaderSource> newSource(new AudioFormatReaderSource(reader, true));
        transportSource.setSource(newSource.get(), 0, nullptr, reader->sampleRate);
        readerSource.reset(newSource.release());
        DBG("File loaded");
    }
}
```

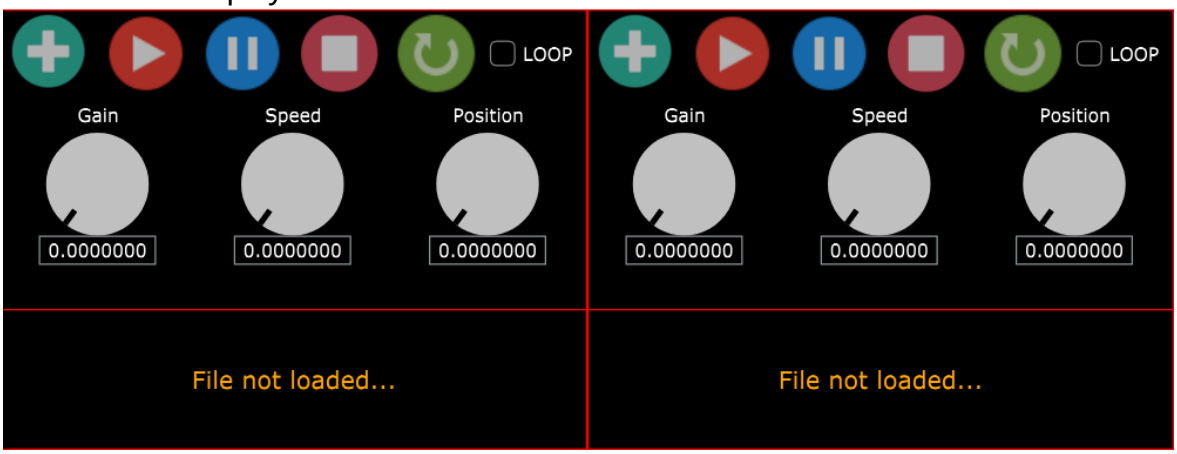
For the player->loadURL, firstly, the URL in the parameter will be used to create a reader. Then the reader will be used to create newSource and used by transport and reader source. This will load the file into the player.

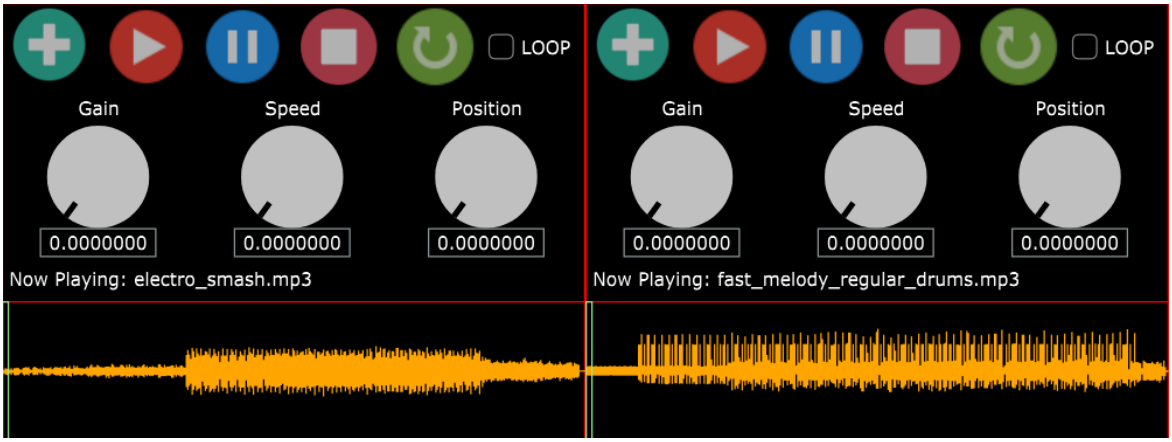
```
void WaveformDisplay::loadURL(URL audioURL)
{
    audioThumb.clear();
    fileLoaded = audioThumb.setSource(new URLInputSource(audioURL));
    if (fileLoaded)
    {
        std::cout << "wfd: loaded! " << std::endl;
        repaint();
    }
    else {
        std::cout << "wfd: not loaded! " << std::endl;
    }

}
```

For the waveform.loadURL, once the file has successfully been loaded the bool fileLoaded will be true and the waveform compartment will be repainted and displayed to the user.

1.2. R1B: Can play two or more tracks





The screenshot above shows the DeckGUI before and after loading the tracks.

```
DJAudioPlayer player1{ formatManager };
DeckGUI deckGUI1{ &player1, formatManager, thumbCache };

DJAudioPlayer player2{ formatManager };
DeckGUI deckGUI2{ &player2, formatManager, thumbCache };
```

In the MainComponent 2 audio players is created and assigned to the different DeckGUIs. This allows the user to play 2 different tracks at the same time.

1.3. R1C: Can mix the track by varying each of their volumes


```
if (slider == &volSlider)
{
    player->setGain(slider->getValue());
}
```

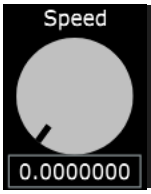
The screenshot above shows the function which adjusts the volume of the players in DeckGUI based on the slider value.

```cpp
void DJAudioPlayer::setGain(double gain)
{
    if (gain < 0 || gain > 1.0)
    {
        std::cout << "DJAudioPlayer::setGain gain should be between 0 and 1" << std::endl;
    }
    else {
        transportSource.setGain(gain);
    }

}
```

Based on the slider value the transportSource setGain will be called and the volume will be adjusted.

1.4. R1D: Can speed up and slow down the tracks




```cpp
if (slider == &speedSlider)
{
    player->setSpeed(slider->getValue());
}
```

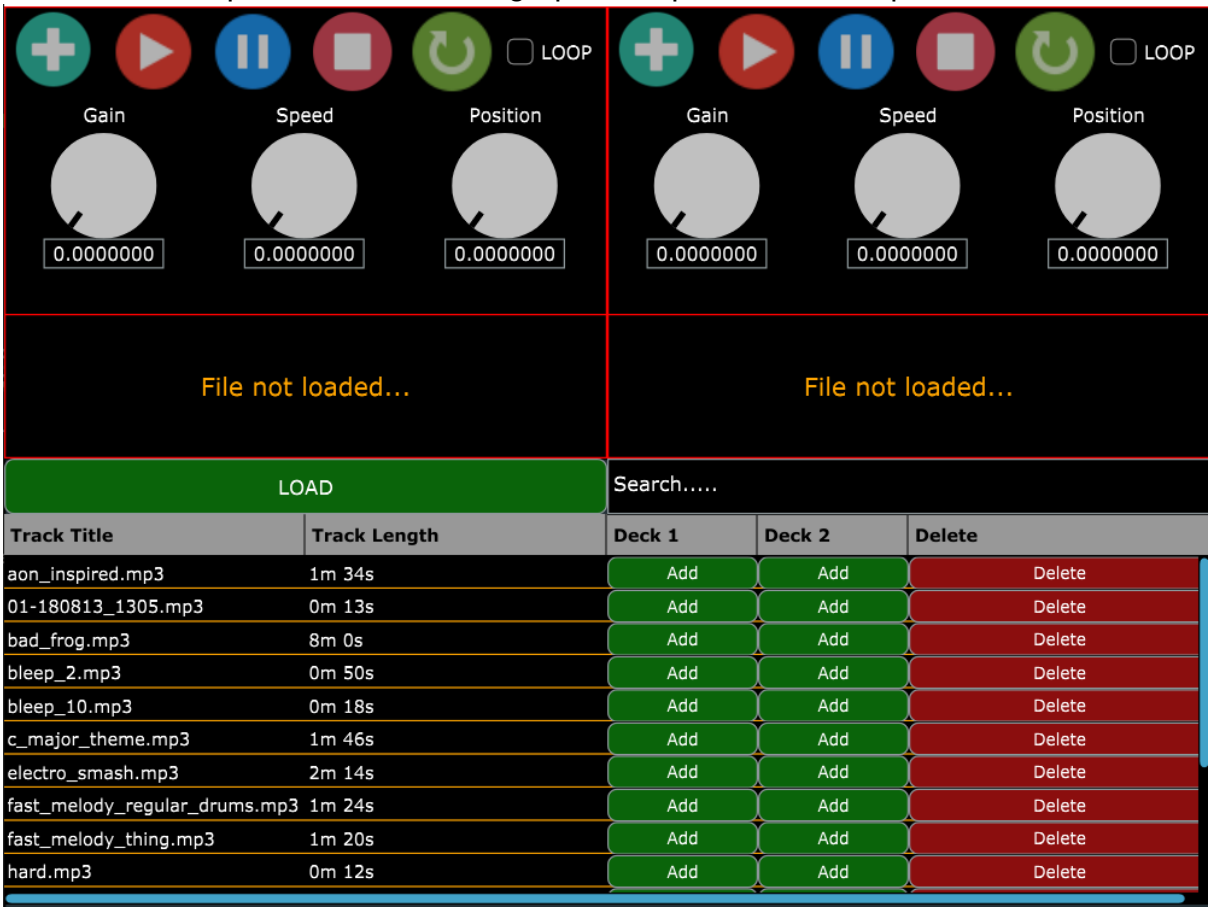
The screenshot above shows the function which adjusts the speed of the players in DeckGUI based on the slider value.

```cpp
void DJAudioPlayer::setSpeed(double ratio)
{
    if (ratio < 0 || ratio > 100.0)
    {
        std::cout << "DJAudioPlayer::setSpeed ratio should be between 0 and 100" << std::endl;
    }
    else {
        resampleSource.setResamplingRatio(ratio);
    }
}
```

Based on the slider value the setResamplingRatio function will be called on the resampleSource. This will adjust the speed of the playback.

2. Requirement 2

2.1. R2A: Component has custom graphics implemented in a paint function



The screenshot above shows the OtoDecks application that I have created.

```
void DeckGUI::paint (juce::Graphics& g)
{
    g.fillAll(Colours::black);   // clear the background

    g.setColour(Colours::red);
    g.drawRect(getLocalBounds(), 1);   // draw an outline around the component

    g.setColour(Colours::white);
    g.setFont(14.0f);
    // g.drawText ("DeckGUI", getLocalBounds(),Justification::centred, true);   // draw some placeholder text
}
```

This screenshot shows the paint function in DeckGUI. Most of the paint function in the other components also fills the background as black and the border as red.

```cpp
void PlaylistComponent::paint (juce::Graphics& g)
{
    g.fillAll(findColour(juce::ListBox::backgroundColourId));   // clear the background

    g.setColour(Colours::black);
    g.drawRect(getLocalBounds(), 1);   // draw an outline around the component

    g.setColour(juce::Colours::white);
    g.setFont(15.0f);

    // Text Editor Styles
    findFile.setColour(TextEditor::highlightedTextColourId, Colours::white);
    findFile.setColour(TextEditor::highlightColourId, Colours::blue);
    findFile.setColour(TextEditor::textColourId, Colours::white);
    findFile.setColour(TextEditor::backgroundColourId, Colours::black);
    findFile.setIndents(4, 8);
    findFile.setFont(16.0f);
}
```

The screenshot above shows the PlaylistComponent's paint function. The background is also set to black, but I decided not to continue with the red border. I changed the text editor text colour to white, the highlight colour to blue, the text colour to white, the background colour to black to complement the application, and the indents to 4 from the left and 8 from the top. I decided to go with the default button color as it complements the application.

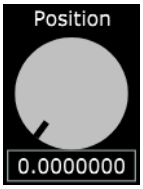
```cpp
void PlaylistComponent::paintRowBackground(Graphics& g, int rowNumber, int width, int height, bool rowIsSelected)
{
    if (rowIsSelected) {
        g.fillAll(Colours::orange);
    }
    else {
        g.fillAll(Colours::black);
    }
    g.setColour(Colours::orange);
    g.drawLine(0, height - 1, width, height - 1);
}

void PlaylistComponent::paintCell(Graphics& g, int rowNumber, int columnId, int width, int height, bool rowIsSelected)
{
    if (columnId == 1) {
        g.setColour(juce::Colours::white);
        g.drawText(trackTitles[rowNumber], 2, 0, width - 4, height, Justification::centredLeft, true);

    }
    if (columnId == 2) {
        g.setColour(Colours::white);
        g.drawText(convert(trackLength[rowNumber]), 2, 0, width - 4, height, Justification::centredLeft, true);
    }
}
```

The screenshot above shows the styling for the tablelistbox in my PlaylistComponent. When a row is selected the colour is set to orange or else it will appear as black. I also decided to draw an orange line between each row in the table.

2.2. R2B: Component enables the user to control the playback of a deck somehow




```
if (slider == &posSlider)
{
    player->setPositionRelative(slider->getValue());
}
```

The screenshot above shows the function which allows the user to adjust the position of the track based on the slider value.

```
void DJAudioPlayer::setPosition(double posInSecs)
{
    transportSource.setPosition(posInSecs);
}
```

Based on the slider value the setPosition will be called on the transportsource which will change the current position of the track.

3. Requirement 3

3.1. R3A: Component allows the user to add files to their library

The screenshots above shows the before and after loading a file to the library. By clicking the load button, the user is able to either select a single or multiple files to be added to the library.

```
if (button == &loadButton) {
    auto fileChooserFlags = FileBrowserComponent::canSelectMultipleItems;

    fChooser.launchAsync(fileChooserFlags, [this](const FileChooser& chooser) {
        Array<File> fileChosen = chooser.getResults();

    for (int i = 0; i < fileChosen.size(); i++) {
        String trackPath = fileChosen[i].getFullPathName().toStdString();
        String name = title(fileChosen[i]);
        double length = duration(fileChosen[i]);

        if (trackPlaylist.indexOf(trackPath) != -1) {
            DBG("File is already in playlist");
        }
        else {
            trackPlaylist.add(trackPath);
            trackTitles.add(name);
            trackLength.add(length);
        }
    }
    updateFile(trackPlaylist);
    tableComponent.updateContent();
    });
}
```

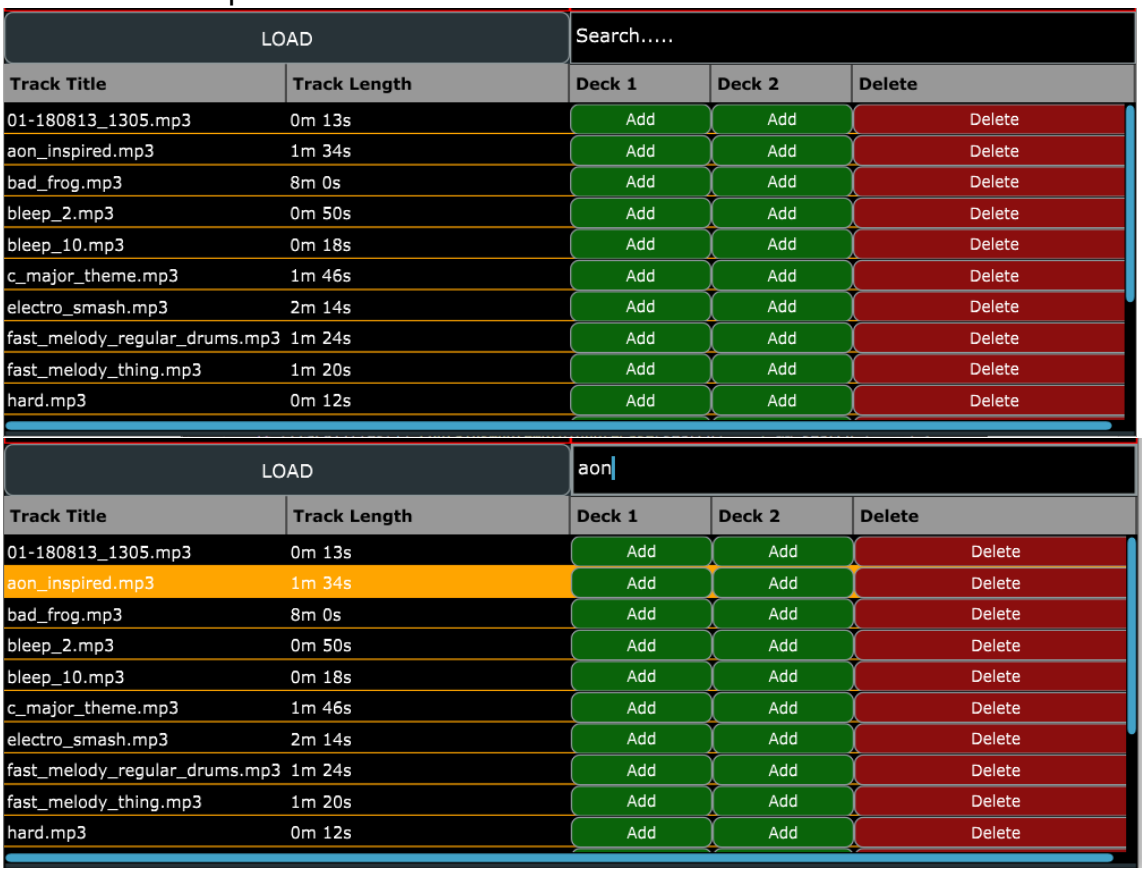
The screenshot above shows the codes needed to load the file/files into the library. I set the flags as FileBrowserComponent::canSelectMultipleItems to allow the user to select multiple files at once to import them into the library. Then the launchAsync function would be used to open the user's default loading program. Once the file/files are chosen the function chooser.getResults return the file/files in an array. The data such as track path, track title, and track length will then be pushed into a JUCE array. The file/files will then be checked to prevent adding duplicates into the library. Then the function updateFile will be called to update a txt file storing all the paths of the tracks loaded into the library. Lastly, the function tableComponent.updateContent is called to display the tracks added to the library.

3.2. R3B: Component parses and displays meta data such as filename and song length

```
String PlaylistComponent::title(File chosen)
{
    String tracktitle = juce::URL::removeEscapeChars(juce::URL{ chosen }.getFileName());
    return tracktitle;
}
double PlaylistComponent::duration(File chosen)
{
    AudioFormatReader* reader = formatManager.createReaderFor(chosen);
    double length = reader->lengthInSamples / reader->sampleRate;
    return length;
}
```

The screenshots above show the title and duration function I created. Both of the function takes in a JUCE file. The title function returns the filename by using the getFileName function while the duration function returns the length of the track by dividing the lengthInSamples and the sampleRate. These functions are called and displayed during the start-up of the application and also when loading the files into the library.

3.3. R3C: Component allows the user to search for files



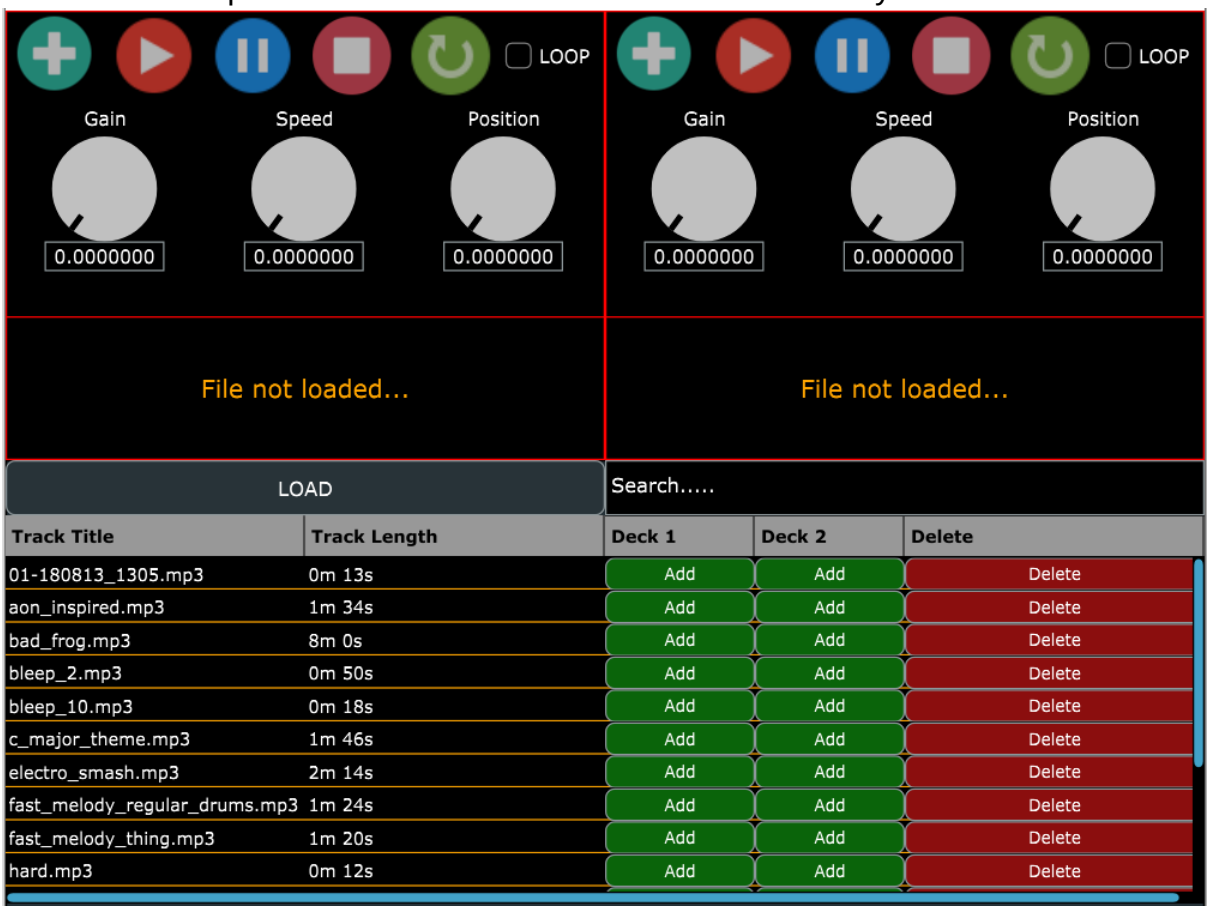
The screenshots above show the before and after searching for a file in the library. As seen by the screenshot when the user type aon the file with aon in its title will be highlighted.

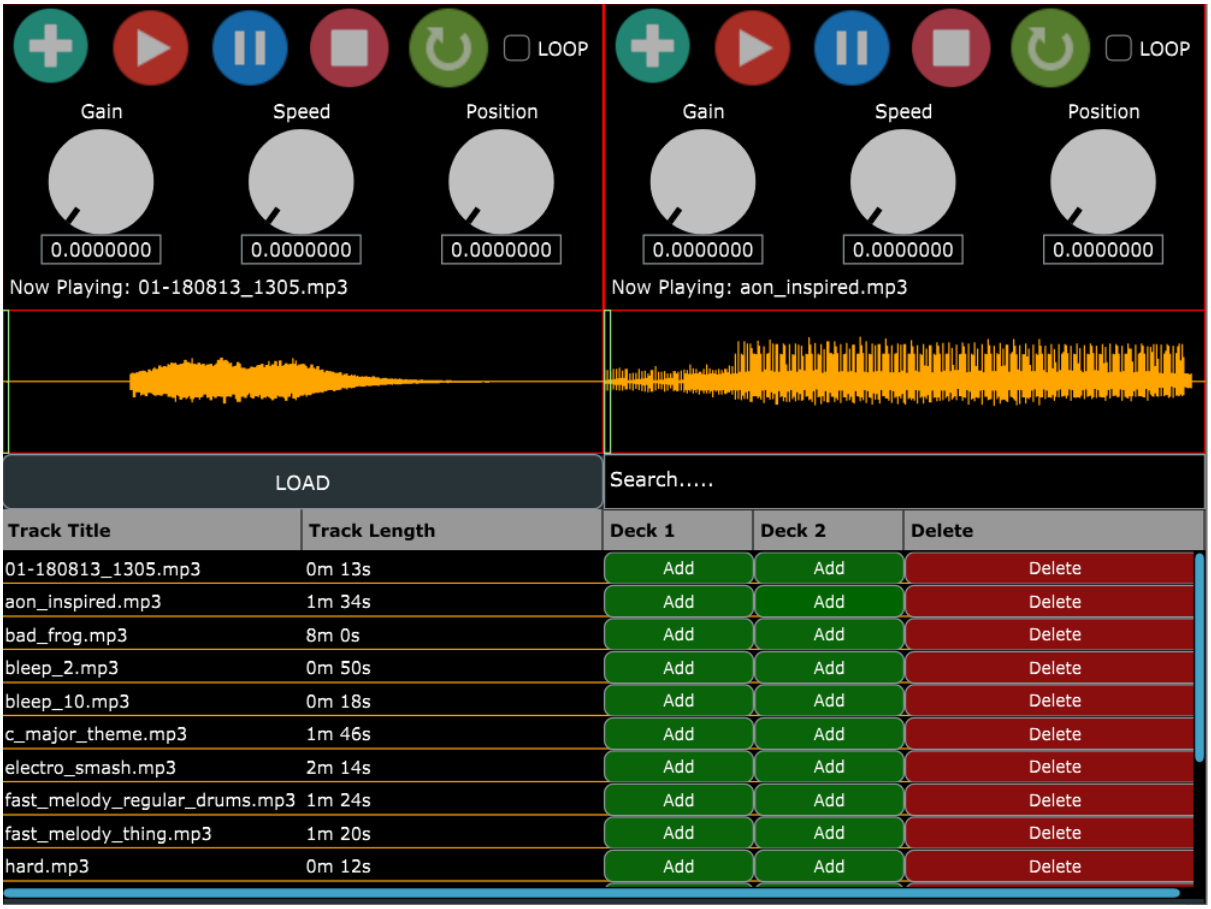
```
void PlaylistComponent::textEditorTextChanged(juce::TextEditor& editor)
{
    search = findFile.getText();

    for (int i = 0; i < trackTitles.size(); ++i) {
        if (trackTitles[i].containsWholeWordIgnoreCase(search) == 1 && search != "") {
            tableComponent.selectRow(i, false, true);
        }
        else if (search == "" || trackTitles[i].containsWholeWordIgnoreCase(search) == -1) {
            tableComponent.deselectAllRows();
        }
    }
}
```

The screenshot shown above shows the codes needed for the search function. It uses the function containsWholeWord to search for the track. If there is a title found based on the user's input the row will be highlighted, else all the rows are deselected.

3.4. R3D: Component allows the user to load files from library into a deck





The screenshots above shows the application before and after loading a track to DeckGUI.

```
if (columnId == 3) {
    if (existingComponentToUpdate == nullptr) {
        TextButton* btn = new TextButton("Add");
        btn->setColour(TextButton::buttonColourId, Colours::darkgreen);
        String id{ std::to_string(rowNumber) };
        btn->setComponentID(id);
        btn->addListener(this);
        existingComponentToUpdate = btn;
        btn->onClick = [this] {addDeck1(); };
    }
}
if (columnId == 4) {
    if (existingComponentToUpdate == nullptr) {
        TextButton* btn = new TextButton("Add");
        btn->setColour(TextButton::buttonColourId, Colours::darkgreen);
        String id{ std::to_string(rowNumber) };
        btn->setComponentID(id);
        btn->addListener(this);
        existingComponentToUpdate = btn;
        btn->onClick = [this] {addDeck2(); };
    }
}
```

The codes above adds an onclick function to columns 3 and 4 which calls the function addDeck1 and addDeck2 respectively.

```
else {
    rowIndex = std::stoi(button->getComponentID().toStdString());
}
```

```
void PlaylistComponent::addDeck1()
{
    String track = trackPlaylist[rowIndex];
    File trackChosen = File(track);
    URL fileURL = URL{ trackChosen };
    mainComponent->deckGUIPlay(fileURL, 1);
}

void PlaylistComponent::addDeck2()
{
    String track = trackPlaylist[rowIndex];
    File trackChosen = File(track);
    URL fileURL = URL{ trackChosen };
    mainComponent->deckGUIPlay(fileURL, 2);
}
```

When the user clicks a row the index will be returned and used in the functions addDeck1 and addDeck2. The track path of the track chosen based on the rowIndex will then be converted into a file then a URL. The URL will be passed to a function in the mainComponent deckGUIPlay.

```
void MainComponent::deckGUIPlay(URL track, int deck)
{
    if (deck == 1) {
        deckGUI1.play(track);
    }
    if (deck == 2) {
        deckGUI2.play(track);
    }
}
```

```
void DeckGUI::play(URL track)
{
    player->loadURL(track);
    waveformDisplay.loadURL(track);
    titleLabel.setText("Now Playing: " + track.getFileName(), dontSendNotification);
}
```

Based on the function in PlaylistComponent the track is either added to deckGUI1 or deckGUI2. Lastly, the function in deckGUI play will be called to add the track to a player and the waveform and title will be displayed.

3.5. R3E: The music library persists so that it is restored when the user exits then restarts the application

```
std::ifstream playlist("Playlist.txt");
std::string string;
if (is_empty(playlist)) {
    playlist.close();
    DBG("No tracks");
}
else {
    while (std::getline(playlist, string)) {
        trackPlaylist.add(string);
        String file_path = string;
        juce::File file(file_path);

        if (!file.existsAsFile()) {
            DBG("File does not exists");
        }
        trackTitles.add(title(file));
        trackLength.add(duration(file));
    }
    playlist.close();
}
```

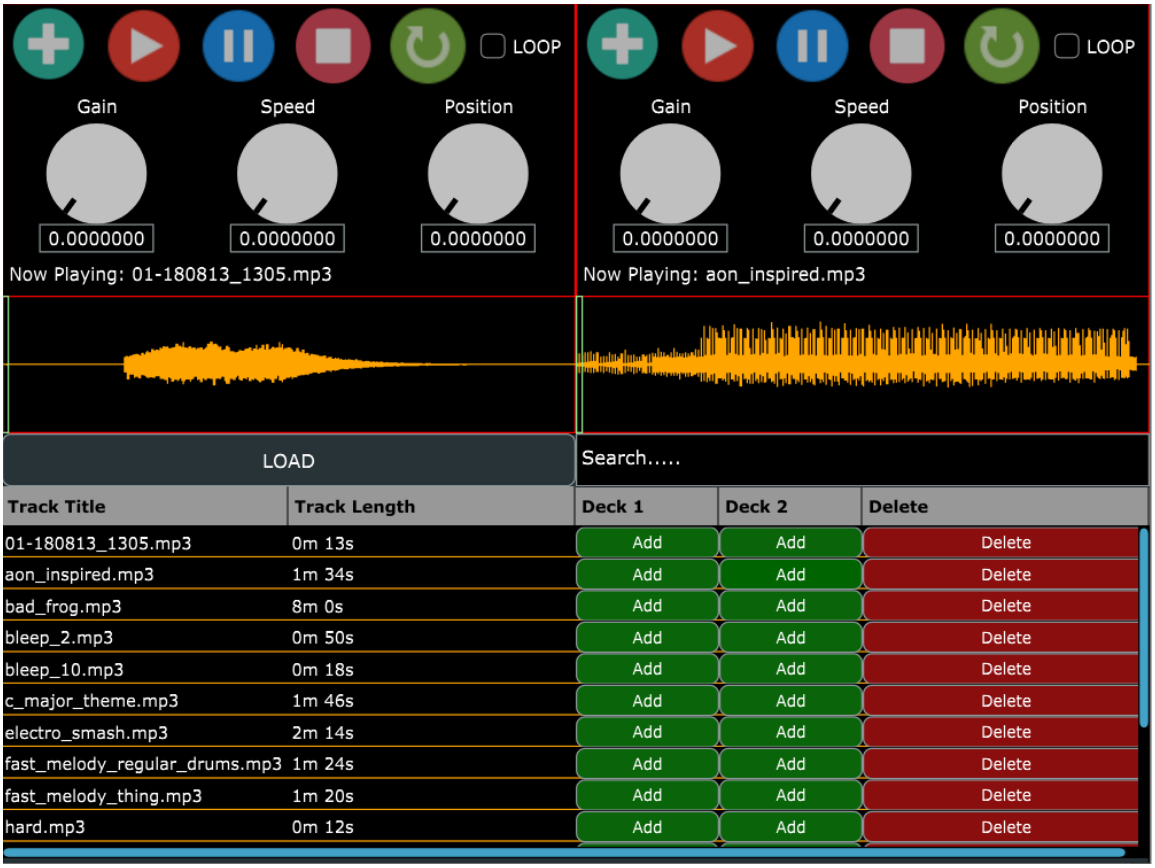
The screenshot above shows the codes needed to ensure that the music library persists when the user restarts the application. When the application is started up it will check for a txt file named Playlist.txt. If the file is present the contents will be converted and pushed row by row into the JUCE arrays trackPlaylist, trackTitles, and trackLength and displayed.

```
void PlaylistComponent::updateFile(Array<String> trackPlaylist)
{
    std::ofstream playlist("Playlist.txt");
    for (int i = 0; i < trackPlaylist.size(); i++) {
        playlist << trackPlaylist[i] << std::endl;
    }
    playlist.close();
}
```

The screenshot above shows the updateFile function used when the user loads files into the music library as seen in R3A. The function pushes the Array trackPlaylist row by row into the txt file named Playlist.txt. This is for the application to remember which files were added to the library even after the user closes the application.

4. Requirement 4

4.1. R4A, R4B and R4C



The screenshot above shows my final Otodecks application.

```
// Getting the Image from JUCE binary data and assigning it to a JUCE Image
Image playImage = ImageFileFormat::loadFrom(BinaryData::play_png, BinaryData::play_pngSize);
Image pauseImage = ImageFileFormat::loadFrom(BinaryData::pause_png, BinaryData::pause_pngSize);
Image restartImage = ImageFileFormat::loadFrom(BinaryData::restart_png, BinaryData::restart_pngSize);
Image stopImage = ImageFileFormat::loadFrom(BinaryData::stop_png, BinaryData::stop_pngSize);
Image loadImage = ImageFileFormat::loadFrom(BinaryData::load_png, BinaryData::load_pngSize);

// Setting the images of the image buttons
playButton.setImages(true, true, true, playImage, 0.7f, juce::Colours::transparentBlack,
                     playImage, 1.0f, juce::Colours::transparentBlack, playImage, 0.7f, juce::Colours::transparentBlack, 0.0f);
pauseButton.setImages(true, true, true, pauseImage, 0.7f, juce::Colours::transparentBlack,
                     pauseImage, 1.0f, juce::Colours::transparentBlack, pauseImage, 0.7f, juce::Colours::transparentBlack, 0.0f);
restartButton.setImages(true, true, true, restartImage, 0.7f, juce::Colours::transparentBlack,
                     restartImage, 1.0f, juce::Colours::transparentBlack, restartImage, 0.7f, juce::Colours::transparentBlack, 0.0f);
stopButton.setImages(true, true, true, stopImage, 0.7f, juce::Colours::transparentBlack,
                     stopImage, 1.0f, juce::Colours::transparentBlack, stopImage, 0.7f, juce::Colours::transparentBlack, 0.0f);
loadButton.setImages(true, true, true, loadImage, 0.7f, juce::Colours::transparentBlack,
                     loadImage, 1.0f, juce::Colours::transparentBlack, loadImage, 0.7f, juce::Colours::transparentBlack, 0.0f);
```

The screenshot above shows the codes needed to set an image button. I loaded the images from JUCE binary data after I dropped them into ProJucer. Then the images are used in the setImages function.

```
// Setting the text of the labels
volumeLabel.setText("Gain", dontSendNotification);
speedLabel.setText("Speed", dontSendNotification);
positionLabel.setText("Position", dontSendNotification);
titleLabel.setText("", dontSendNotification);

// Attaching the label to its component
volumeLabel.attachToComponent(&volSlider, false);
speedLabel.attachToComponent(&speedSlider, false);
positionLabel.attachToComponent(&posSlider, false);
titleLabel.attachToComponent(&waveformDisplay, false);

// Setting the label's justification type
volumeLabel.setJustificationType(Justification::centred);
speedLabel.setJustificationType(Justification::centred);
positionLabel.setJustificationType(Justification::centred);
titleLabel.setJustificationType(Justification::topLeft);
```

The screenshot above shows the codes I used to create the labels. Firstly, I set the text of the labels with dontSendNotification. Secondly, I attached each label to its component. Lastly, I set the justification type of the labels.

```
// Custom LookAndFeel for the 3 sliders
class OtherLookAndFeel : public LookAndFeel_V4
{
public:
    void drawRotarySlider(Graphics& g, int x, int y, int width, int height, float sliderPos, float rotaryStartAngle, float rotaryEndAngle, Slider& slider) override {
        // get the lesser of the 2 values as the height and width is different
        float diameter = jmin(width, height);
        float radius = diameter / 2;
        float centerX = x + width / 2;
        float centerY = y + height / 2;
        float rx = centerX - radius;
        float ry = centerY - radius;

        // scale the angle down
        float angle = rotaryStartAngle + (sliderPos * (rotaryEndAngle - rotaryStartAngle));

        Rectangle<float> dialArea(rx, ry, diameter, diameter);

        g.setColour(Colours::silver);
        g.fillEllipse(dialArea);

        g.setColour(Colours::black);

        // sequence of lines and curves that may either form a closed shape or be open-ended.
        Path dialTick;
        dialTick.addRectangle(0, -radius, 4.0f, radius * 0.33);

        // AffineTransform represents a 2d matrix
        g.fillPath(dialTick, AffineTransform::rotation(angle).translated(centerX, centerY));
    }
};
```

The screenshot above shows the codes needed to draw the rotary slider which I used for my application. I created another class with LookAndFeel_V4. Firstly, I had to draw the circle which is done by calculating the jmin of the width and height. This is to ensure that it would draw a perfect circle. Secondly, I calculated the angles based on the slider position. Lastly, I used the fillPath function to draw the line from the center of the circle to its edges to represent the current position of the slider.

```
// Setting the style of the slider to rotary
volSlider.setSliderStyle(Slider::SliderStyle::Rotary);
speedSlider.setSliderStyle(Slider::SliderStyle::Rotary);
posSlider.setSliderStyle(Slider::SliderStyle::Rotary);

// Set the text box to appear below the slider
volSlider.setTextBoxStyle(Slider::TextBoxBelow, false, 80, 20);
speedSlider.setTextBoxStyle(Slider::TextBoxBelow, false, 80, 20);
posSlider.setTextBoxStyle(Slider::TextBoxBelow, false, 80, 20);

// Setting the slider Look and Feel using the custom look and feel
otherLookAndFeel.setColour(juce::Slider::thumbColourId, juce::Colours::blue);
volSlider.setLookAndFeel(&otherLookAndFeel);
speedSlider.setLookAndFeel(&otherLookAndFeel);
posSlider.setLookAndFeel(&otherLookAndFeel);
```

The screenshot above shows the styles I used to create the sliders. I set all the sliders to rotary and also the text box to appear below the slider. Then I set the LookAndFeel of the sliders to use my custom LookAndFeel.

```
// Restart Button
if (button == &restartButton) {
    player->setPosition(0);
}
```

```
// Stop Button
if (button == &stopButton) {
    player->stop();
    player->setPosition(0);
}
```

The screenshot above shows the additional functions I created to restart and stop the track by setting the position of the player to 0 and also stop playing if the stop button is clicked. I also changed the stop button done in class to a pause button instead.

```
// Ensure that the track will be looped when it is finished
if (loopButton.getToggleState() == 1) {
    if (player->getPositionRelative() >= 1) {
        player->setPosition(0);
        player->start();
    }
}
else {
    if (player->getPositionRelative() >= 1) {
        player->stop();
    }
}
```

The screenshot above shows the additional function I created to loop the track. This is done by checking the toggle button state and also the current position of the track. If the track is more than or equal to 1 the track will be restarted.