

## R1:

### R1A:

The DJAudioPlayer is focusing on manipulate and manage the resource of audio.

Firstly, the DJAudioPlayer.loadURL takes responsible for loading audio files into audio players, and it comes from format manager to recognise the format of the audio files and read them.

Secondly, the format manager will create a reader and pass resource to reader.

Thirdly, the transport Source will take care of the resource and the audio is read by the unique pointer and cut by the sample rate.

Lastly, transport Source will deal with reset, new source, and start to play the audio.

The class DJAudioPlayer will have the object players, and they can use the same format manager to load different audio files.

The two players will be created together at the MainComponent

```
MainComponent();
~MainComponent() override;

//=====
void prepareToPlay (int samplesPerBlockExpected, double sampleRate) override;
void getNextAudioBlock (const juce::AudioSourceChannelInfo& bufferToFill) override;
void releaseResources() override;
```

and MainComponent's function will call the player to do the loading and playing functions.

```
DJAudioPlayer::DJAudioPlayer(juce::AudioFormatManager& _formatManager)
: formatManager(_formatManager)
{
}

DJAudioPlayer::~DJAudioPlayer(){
}

void DJAudioPlayer::prepareToPlay (int samplesPerBlockExpected, double sampleRate)
{
    transportSource.prepareToPlay(samplesPerBlockExpected,sampleRate);
    resampleSource.prepareToPlay(samplesPerBlockExpected,sampleRate);
}

void DJAudioPlayer::getNextAudioBlock(const juce::AudioSourceChannelInfo& bufferToFill)
{
    transportSource.getNextAudioBlock(bufferToFill);
    resampleSource.getNextAudioBlock(bufferToFill);
}

void DJAudioPlayer::releaseResources()
{
    transportSource.releaseResources();
    resampleSource.releaseResources();
}

void DJAudioPlayer::loadURL(juce::URL audioURL)
{
    auto* reader =formatManager.createReaderFor(audioURL.createInputStream(false)); ⚠️ 'createInputStream' is deprecated: New code shoul...
    //if it goes wrong it will erase the memoery
    if(reader !=nullptr) //good file !
    {
        std::unique_ptr<juce::AudioFormatReaderSource> newSource (new juce::AudioFormatReaderSource (reader,true));

        //pass reader to transport source
        transportSource.setSource(newSource.get(),
                                0,
                                nullptr,
                                reader->sampleRate);

        /* because it is unique pointer smart only one ownership, one pointer , this is class scope variable which means we
           can retain acces to it because we need to play */
        readerSource.reset(newSource.release());
        transportSource.start();
    }
}
```

### R1B:

Two or more players to play more tracks because they can share the format manager and each player has their own transport source prepared. Furthermore, mixersource inherited from mixerAudiosource helps the main component to manage input sources from different players.

Those are the core logic of the Audioplayer, and we still need buttons to control the functions, and users can interact with the media player by clicking the buttons created in DeckGUI.

DeckGui will create, paint, and resize those buttons and sliders including play, stop, load, and volume, position, and speed.

Lastly, the two decks created in the main component, and the main component will paint them in a good way with two players of the DJAudioPlayer, and they will share one format manager.

```

class DeckGUI : public juce::Component,
    public juce::Button::Listener,
    public juce::Slider::Listener,
    public juce::FileDragAndDropTarget,
    public juce::Timer,
    public juce::LookAndFeel_V4
{
//abstract class, pure virtual function in it , cannot use
// you have to use pure virtual functions
public:
    DeckGUI(int _id,
        DJAudioPlayer* _player,
        juce::AudioFormatManager & formatManagerToUse,
        juce::AudioThumbnailCache & cacheToUse );

    ~DeckGUI() override;
}

DJAudioPlayer player1{formatManager};
DJAudioPlayer player2{formatManager};
DJAudioPlayer playerForPlaylist{formatManager};

DeckGUI deckGUI1{1, &player1,formatManager, thumbCache};
DeckGUI deckGUI2{2, &player2,formatManager, thumbCache};

```

### R1C:

```

void MainComponent::prepareToPlay (int samplesPerBlockExpected, double sampleRate)
{
    player1.prepareToPlay(samplesPerBlockExpected, sampleRate);
    player2.prepareToPlay(samplesPerBlockExpected, sampleRate);

    mixerSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
    mixerSource.addInputSource(&player1, false);
    mixerSource.addInputSource(&player2, false);
}

void DeckGUI::sliderValueChanged (juce::Slider *slider)
{
    if(slider == &volSlider )
    {
        player->setGain(slider->getValue());
    }

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

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

```

The music is mixed by the mixersource in the main component, and we mix the music by implementing mixer source in prepare to play, get next audio block, and release Resources. Furthermore, their volumes are controlled by the vol sliders. Sliders will ask player pointer to access the set gain function, and the set gain function get the setgain from transportsource.

### R1D:

The tracks can be speed up and slow down by the speed slider, and sliders created by the juce technically and I kept the position of the song slider same way and changed other two sliders into arch forms, and it looks like what we used to use on the traditional radio machine and easier for user to understand and interact with.

The speed range is 0-5, and the player will get the slider value from the slider we are using, and the value we change will pass into DJAudioPlayer.cpp::setSpeed() function, the function received the changed ratio and use this ratio to multiple a resampleSource function, called setResamplingRatio, and the speed of the function will be changed by the slider's value, hence, if the slider's value is smaller than 1, and it will be slower, and the speed will be faster if the slider's value is greater than 1.

```

1 DeckGUI::DeckGUI(DJAudioPlayer* _player,
2     juce::AudioFormatManager & formatManagerToUse,
3     juce::AudioThumbnailCache & cacheToUse
4 ) : player(_player),
5     waveformDisplay(formatManagerToUse, cacheToUse)
6 {
7     setLookAndFeel (&otherLookAndFeel);
8
9     volSlider.setLookAndFeel (&otherLookAndFeel);
10    speedSlider.setLookAndFeel (&otherLookAndFeel);
11
12    volSlider.setSliderStyle (juce::Slider::Rotary);
13    volSlider.setTextBoxStyle (juce::Slider::NoTextBox, false, 0, 0);
14
15    speedSlider.setSliderStyle (juce::Slider::Rotary);
16    speedSlider.setTextBoxStyle (juce::Slider::NoTextBox, false, 0, 0);
17
18    addAndMakeVisible(playButton);
19    addAndMakeVisible(volSlider);
20    addAndMakeVisible(speedSlider);
21    addAndMakeVisible(stopButton);
22    addAndMakeVisible(loadButton);
23
24    addAndMakeVisible(posSlider);
25
26    addAndMakeVisible(waveformDisplay);
27
28    playButton.addListener(this);
29    stopButton.addListener(this);
30    loadButton.addListener(this);
31    speedSlider.addListener(this);
32    volSlider.addListener(this);
33    posSlider.addListener(this);
34
35    addAndMakeVisible(speedLabel);
36    speedLabel.setText ("Speed", juce::dontSendNotification);
37    speedLabel.attachToComponent (&speedSlider, true);
38
39    addAndMakeVisible(posLabel);
40
41    posLabel.setText ("Position", juce::dontSendNotification);
42    posLabel.attachToComponent (&posSlider, true);
43
44    addAndMakeVisible(vollabel);
45
46    vollabel.setText ("Volumn", juce::dontSendNotification);
47    vollabel.attachToComponent (&volSlider, true);
48
49    //volSlider.setrange(0.0,1, 1,10);
50    volSlider.setRange(0.0, 1.0);
51    speedSlider.setRange(0, 5);
52    posSlider.setRange(0.0, 1.0);
53
54    startTimer(500);
55
56    // player->registerBasicFormats();
57 }

```

## R2:

The deck control component with graphics could playback controlled by the slider, and I think the position control is reasonable with the length of the song and you can slide it with your finger and control it with this advanced way.

It has three columns of different colours just after launch, and the colour will help users to recognise what is happening and signal the user if any status is changed, and they can quickly recognise it. The playback function is composited by the set position relative function, slider, and labels. Consequently, you can pull the slider or change the number value of the slider to get the position of the track, and that is convenient, and it helps the user to control the playback in multiple ways.

```

1 if (slider == &posSlider)
2 {
3     player->setPositionRelative(slider->getValue());
4 }
5
6 double DJAudioPlayer::getPositionRelative()
7 {
8     return transportSource.getCurrentPosition() / transportSource.getLengthInSeconds();
9 }

```

## R2A:

The DeckGUI component is painted with different colours for different functions. For example, the background of the player will turn red once the song is stopped, and turn hot pink when the song is playing, and turn light pink when loading songs.

In the middle of the DeckGUI we have the waveform display, and it will show text “feed me music”, and once the music is loaded and it will start to show the waveform and length of the song.

Moreover, I added more functions to show the status of the player. For example, the flag to indicate that we clicked the button, and we can see the colour change of the player.

```

void DeckGUI::paint (juce::Graphics& g)
{
    /* This demo code just fills the component's background and
    draws some placeholder text to get you started.

    You should replace everything in this method with your own
    drawing code..

    */
    g.fillAll (getLookAndFeel().findColour (juce::ResizableWindow::backgroundColourId)); // clear the background
    g.setColour (juce::Colours::yellow);
    g.drawRect (getLocalBounds(), 1); // draw an outline around the component

    g.setColour (juce::Colours::darkred);
    g.setFont (14.0f);
    g.drawText ("Volume & Speed & Position ", getLocalBounds(),
        juce::Justification::centred, true); // draw some placeholder text

    g.setColour (juce::Colours::hotpink);
    auto centralArea = getLocalBounds().toFloat().reduced(10.0f);
    g.drawRoundedRectangle (centralArea, 5.0f, 3.0f);

    juce::Array<juce::Colour> colours{ juce::Colours::lightpink, juce::Colours::hotpink, juce::Colours::deeppink
    };

    auto colourBarArea = centralArea.reduced(4.0f).withHeight(275.0f);
    auto colourArea= colourBarArea.withWidth(colourBarArea.getWidth()/colours.size());

    for (auto colour : colours)
    {
        g.setColour (colour);
        g.fillRect (colourArea);
        colourArea.translate (colourArea.getWidth(), 0.0f);
    }

    if (colorFlag == 1)
    {
        g.fillAll(juce::Colours::palevioletred);
    }

    if (colorFlag == 2)
    {
        g.fillAll(juce::Colours::red);
    }

    if (colorFlag == 3)
    {
        g.fillAll(juce::Colours::hotpink);
    }
}

```

## R2B:

The main part to control the playback of a deck is In the DJAudioPlayer, and we have the format Manager to manage the format, and we have the set position relative, get position relative to control the position in a music we are playing. Furthermore, the transport source takes responsible for resampling the audio source into different pieces and we get the length of the music, and we get the position in seconds, which means the function set position relative enables the playback function to get the seconds of the music, and we can jump to the seconds in a music as long as we get the position and set the position.

Furthermore, the length of the track will show in minutes in the library, and it is due to the get length function in the playlist component.

```

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

void DJAudioPlayer::setPositionRelative(double pos)
{
    if (pos < 0 || pos > 1.0)
    {
        std::cout<<"DJAudio player :: setPosition Relative
        erro"<<std::endl;
    }
    else {
        double posInSecs =
            transportSource.getLengthInSeconds()* pos;
        setPosition(posInSecs);
    }
}

void DJAudioPlayer::loadURL(juce::URL audioURL)
{
    auto* reader =formatManager.createReaderFor(audioURL.createInputStream(false)); // 'createInputStream' is deprecated. New code should...
    //if it goes wrong it will erase the memory
    if(reader !=nullptr) //good file !
    {
        std::unique_ptr<juce::AudioFormatReaderSource> newSource (new juce::AudioFormatReaderSource (reader,true));

        //pass reader to transport source
        transportSource.setSource(newSource.get(),
            0,
            nullptr,
            reader->sampleRate);

        /* because it is unique pointer smart only one ownership, one pointer , this is class scope variable which means we
        can retain acces to it because we need to play */
        readerSource.reset(newSource.release());
        transportSource.start();
    }
}

juce::String PlaylistComponent::getLength(juce::URL audioURL)
{
    playerForPlaylist->loadURL(audioURL);
    double seconds{playerForPlaylist->getLengthInSeconds() };
    juce::String minutes{secondsToMinutes(seconds)} ;
    return minutes;
}

```

## • R3:



I am lost in the middle and feel hard to understand how to implement xml play list. There is difficulty to read the tutorial of the list box class and difficult to understand how to combine them together. Furthermore, I found music library on the JUCE library, and I start to use library inherited from the table list box, and vector of tracks to store music list, and need to implement the importation function to library function. In simple words, the table list box will contain the library, and the library will show the vector of those tracks name by the file chooser component, and once the button is clicked, and we will load those files as vectors in the library.

```

PlaylistComponent::PlaylistComponent(DJAudioPlayer*
    _playerForPlaylist,
                                   DeckGUI* _deckGUI1,
                                   DeckGUI* _deckGUI2
                                   ):playerForPlaylist
    (_playerForPlaylist),
    deckGUI1(_deckGUI1),
    deckGUI2(_deckGUI2)
{
    // In your constructor, you should add any child
    // components, and
    // initialise any special settings that your component
    // needs.

    //register the playlist component with the table list box as a
    // table list model

    addAndMakeVisible(tableComponent);
    addAndMakeVisible(importButton);
    addAndMakeVisible(library);
    addAndMakeVisible(addToPlayer1Button);
    addAndMakeVisible(addToPlayer2Button);
    addAndMakeVisible(searchField);

    importButton.addListener(this);
    tableComponent.setModel(this);
    searchField.addListener(this);
    addToPlayer1Button.addListener(this);
    addToPlayer2Button.addListener(this);

    searchField.setTextToShowWhenEmpty("search music name here
        and press ENTER !",juce::Colours::blue);

    searchField.onReturnKey = [this]{searchLibrary
        (searchField.getText());};

void PlaylistComponent::buttonClicked(juce::Button* button)
{
    if(button == & importButton)
    {
        DBG("Load button clicked ");
        importToLibrary();
        library.updateContent();
    }
    else if (button == & addToPlayer1Button)
    {
        DBG("Add to player 1 clicked");
        loadInPlayer(deckGUI1);
    }
    else if(button == & addToPlayer2Button)
    {
        DBG("add to player 2 clicked");
        loadInPlayer(deckGUI2);
    }
    else {
        int id = std::stoi(button->getComponentID().toStdString());
        DBG(tracks[id].title + "removed from library ");
        deleteFromTracks(id);
        library.updateContent();
    }
}

void PlaylistComponent::paint (juce::Graphics& g)
{
    /* This demo code just fills the component's background and
    draws some placeholder text to get you started.
    You should replace everything in this method with your own
    drawing code..
    */

    g.fillAll (getLookAndFeel().findColour (juce::ResizableWindow::backgroundColourId)); // clear the background

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

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

```

add and make library visible, and resized the library, and library needs header and set model as well, because it focuses on the music library.

Button click takes responsibility for the function load.

- **R3A:**

This component of function allows the user to import files from the file system to the library. We will need to use file chooser again to choose the music documents, and get the name of those music documents, and check if this name existed in the library, and get length of the music document, and push the new name to the tracks vector, so we can see those names in the library file.

```
void PlaylistComponent::importToLibrary()
{
    DBG("PlaylistComponent::importToLibrary called");

    //initialize file chooser
    juce::FileChooser chooser{ "Select files" };
    if (chooser.browseForMultipleFilesToOpen())
    {
        for (const juce::File& file : chooser.getResults())
        {
            juce::String fileNameWithoutExtension{
                file.getFileNameWithoutExtension() };
            if (!isInTracks(fileNameWithoutExtension)) // if not already loaded
            {
                Track newTrack{ file };
                juce::URL audioURL{ file };
                newTrack.length = getLength(audioURL);
                tracks.push_back(newTrack);
                DBG("loaded file: " << newTrack.title);
            }
            else // display info message
            {
                juce::AlertWindow::showMessageBox
                (juce::AlertWindow::AlertIconType::InfoIcon,
                 "Load information:",
                 fileNameWithoutExtension + " already loaded",
                 "OK",
                 nullptr
                );
            }
        }
    }
}
```

```
juce::String PlaylistComponent::getLength(juce::URL audioURL)
{
    playerForPlaylist->loadURL(audioURL);
    double seconds{playerForPlaylist->getLengthInSeconds() };
    juce::String minutes{secondsToMinutes(seconds)} ;
    return minutes;
}

juce::String PlaylistComponent::secondsToMinutes(double
seconds)
{
    int secondsRounded{ int (std::round(seconds)) } ;
    juce::String min{ std::to_string(secondsRounded / 60 )};
    juce::String sec{std::to_string(secondsRounded %60)};

    if(sec.length() < 2)
    {
        sec = sec.paddedLeft('0', 2);
    }
    return juce::String{ min + ":" + sec } ;
}

bool PlaylistComponent::isInTracks(juce::String
fileNameWithoutExtension)
{
    return(std::find(tracks.begin(),tracks.end(),
                    fileNameWithoutExtension) !=
           tracks.end());
}
```

- **R3B:**

The file name of the loaded file will be displayed due to the file.getFilenameWithoutExtension function, and the song length will be more complicated because it takes long time for me to figure out how does it work.

Moreover, a function named getLength is called, and in the function, we use the pointer of palyerforplaylist to access the functions in DJAudio player, and we can use the load URL and getLengthInSeconds functions. Hence, we can the length of the music by seconds, and we can transfer the result to minutes by the seconds to minutes function.

Lastly, the Is in track function is comparatively simple that it iterates from the begin of the vector to the end of the vector to see if their same name in the vector.

- **R3C:**

```
searchField.setTextToShowWhenEmpty("search music name here
and press ENTER !",juce::Colours::blue);

searchField.onReturnKey = [this]{searchLibrary
(searchField.getText());};
```

Search field is a function inherited from juce::textEditor, and we add the visible box to the table list box as usual with the add and make visible function, and the search field will connect with the searchLibrary to implement functions. Furthermore, we need a function to locate the music name in the

tracks. We will pass the search text that you typed in the field to the function and find the tokens in the tracks and locate it.

```
void PlaylistComponent::searchLibrary(juce::String searchText){
    DBG("search library for the track :<searchText);
    if ( searchText != "")
    {
        int rowNumber = whereInTracks(searchText);
        library.selectRow(rowNumber);
    }
    else
    {
        library.deselectAllRows();
    }
}

int PlaylistComponent::whereInTracks(juce::String searchText)
{
    auto it= find_if(tracks.begin(), tracks.end(),
        [&searchText](const Track& obj) {return obj.title.contains(searchText);});
    int i=-1;
    if( it != tracks.end())
    {
        i = std::distance(tracks.begin(), it);
    }
    return i;
}
```

### • R3D:

```
void PlaylistComponent::loadInPlayer(DeckGUI* deckGUI)
{
    int selectedRow( library.getSelectedRow() );
    if ( selectedRow != -1)
    {
        DBG("Adding: " <<tracks[selectedRow].title<<" to player ");
        deckGUI->loadFile(tracks[selectedRow].URL);
    }
    else
    {
        juce::AlertWindow::showMessageBox(juce::AlertWindow::AlertIconType::InfoIcon,
            "Add to deck information",
            "Please selecte a track to add to deck ",
            "ok",
            nullptr
        );
    }
}

void DeckGUI::loadFile(juce::URL audioURL)
{
    DBG("DeckGUI::loadFile called");
    player->loadURL(audioURL);
    waveformDisplay.loadURL(audioURL);
}
```

Users can use the selectedRow function of the library to get rows from the music library, and once the row is selected, and we will click the button to load the file into the DeckGUI. DeckGUI has the ability to access the pointer of player in the DJAudio, and we can access the real component that loads the music, and it is load URL in the DJaudio, and we can load one to each other with two buttons, since we have two DeckGUIs.

### • R3E:

• In order to save the playlist in the library and I used the save library and load library functions to deliver.

```

void PlaylistComponent::saveLibrary()
{
    std::ofstream myLibrary("My-library.csv");

    for(Track& t: tracks)
    {
        myLibrary <<t.file.getFullPathName()<< "," <<t.length<< "\n";
    }
}

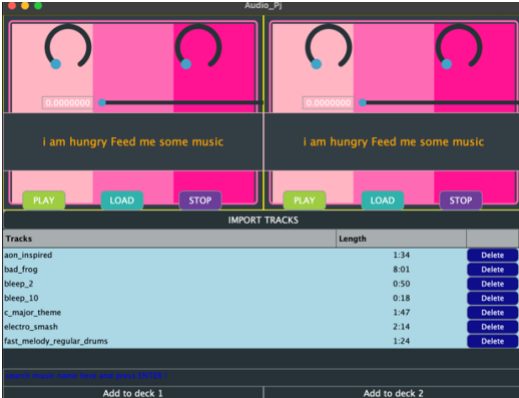
void PlaylistComponent::loadLibrary()
{
    std::ifstream myLibrary("My-library.csv");
    std::string filePath;
    std::string length;

    if( myLibrary.is_open())
    {
        while(getline(myLibrary, filePath, ',')) {
            juce::File file{ filePath };
            Track newTrack{ file };

            getline(myLibrary, length);
            newTrack.length = length;
            tracks.push_back(newTrack);
        }
    }
    myLibrary.close();
}

```

- R4:**  
 The layer out of the DJ audio player designed to show the conditions of the players, and it is a simplified DJ audio player, and the core concept is that the colour of the player will inform the different conditions of the player. For example, the music stopped, and the background of the player will turn to whole red, and DeckGUI is pink while playing music, which is indicative.



```

void DeckGUI::resized()
{
    // This method is where you should set the bounds of any child
    // components that your component contains..
    double rowH = getHeight()/10 ;

    playButton.setBounds(rowH, rowH*9, getWidth()/6, rowH);
    loadButton.setBounds(rowH*5, rowH*9, getWidth()/6, rowH);
    stopButton.setBounds(rowH*9, rowH*9, getWidth()/6, rowH);
    playButton.setColour(juce::TextButton::buttonColourId, juce::Colours::yellowgreen);
    stopButton.setColour(juce::TextButton::buttonColourId, juce::Colours::rebeccapurple);
    loadButton.setColour(juce::TextButton::buttonColourId, juce::Colours::lightseagreen);
    waveformDisplay.setBounds(0, rowH*5, getWidth(), rowH*3);
    posSlider.setBounds(rowH*2, rowH*4, getWidth(), rowH);

    auto border = 4;

    auto area = getLocalBounds();

    auto dialArea = area.removeFromTop (area.getHeight() / 3);

    volSlider.setBounds (dialArea.removeFromLeft (dialArea.getWidth() / 2).reduced (border));
    speedSlider.setBounds (dialArea.reduced (border));
}

```

- R4A:**  
 Moreover, we can use buttons and sliders to control the playing music. For example, the volume is the left side, and the speed is the right-hand side slider. The delete button could delete the music already in the playlist, and the waveform display is in the middle of those buttons, and it is very close to the position of the music because they are closely related, and users can easily identify where the is the position of the music and manipulate it.



#### R4B:

R2 is the control deck playback section, and we can see this section blends with waveform display, and three buttons, and the position control. We can see three buttons corresponding to the three-line colours at the back of the DeckGui, and buttons will trigger the change of the background painting. For instance, colour flag is used here to set the colour of the background, and it works with the button click. Three buttons stays at the button of the DeckGui.

```
void DeckGUI::buttonClicked(juce::Button* button)
{
    //just look at the memory address clicked on if you get the memo
    if(button == &playButton)
    {
        std::cout<<" play button was clicked "<<std::endl;
        player->start();
        colorFlag=1;
        repaint();
    }
    if(button == &stopButton)
    {
        std::cout<<"stop button was clicked "<<std::endl;
        player->stop();
        colorFlag=2;
        repaint();
    }
    if(button == &loadButton)
    {
        juce::FileChooser chooser("Select a file...");
        if(chooser.browseForFileToOpen())
        {
            loadFile(juce::URL{ chooser.getResult() });
        }
        colorFlag=3;
        repaint();
    }
}

if(colorFlag ==1 )
{
    g.fillAll(juce::Colours::palevioletred);
}

if(colorFlag ==2 )
{
    g.fillAll(juce::Colours::red);
}

if(colorFlag ==3)
{
    g.fillAll(juce::Colours::hotpink);
}
```

#### R4C: GUI layout includes the music library component from R3

The music library component starts with the import tracks button, and this is the core function of the library with three headers, tracks name, length of the music, and the delete button to control the library. Furthermore, users will see an empty library and once the library is filled with tracks, and it will be full. The search field section is just below the list of the files, and also the add to deck1 and the add to deck 2 buttons stay at the bottom of the table list box.

```
void PlaylistComponent::resized()
{
    // This method is where you should set the bounds of any child
    // components that your component contains..
    tableComponent.setBounds(0, 0, getWidth(), getHeight());
    importButton.setBounds(0, 0, getWidth(), getHeight() / 10);
    library.setBounds(0, 1 * getHeight() / 10, getWidth(), 13 * getHeight() / 10);
    searchField.setBounds(0, 13 * getHeight() / 16, getWidth(), 1.5*getHeight() / 16);
    addToPlayer1Button.setBounds(0, 14 * getHeight() / 16, getWidth() / 2 , 2*getHeight() / 16);
    addToPlayer2Button.setBounds(getWidth() / 2, 14 * getHeight() / 16, getWidth() / 2, 2*getHeight() / 16);
    //set columns
    library.getHeader().setColumnWidth(1, 12.8 * getWidth() / 20);
    library.getHeader().setColumnWidth(2, 5 * getWidth() / 20);
    library.getHeader().setColumnWidth(3, 2 * getWidth() / 20);
}
```