Text → green means I wrote all by myself or that I referred to the documentation available (Coursera or Juce tutorial)
Text → orange means I referred to a 3rd party source for possible solution(eg stackoverflow/juce forum)
Text → red means I asked someone for help with this section
Text → unhighlighted means this is from the learning materials provided on Coursera

CrossFader.cpp

```
#include "CrossFader.h"
#include "../JuceLibraryCode/JuceHeader.h"


Crossfader::Crossfader(DJAudioPlayer* _player1, DJAudioPlayer* _player2, DeckGUI*
_deckGUI1, DeckGUI* _deckGUI2
) : player1(_player1),player2(_player2), deckGUI1(_deckGUI1), deckGUI2(_deckGUI2)
{
        //setting up the crossfader slider listener
        addAndMakeVisible(CFSlider);
        CFSlider.addListener(this);

        //modifying the UI of the crossfader slider
        CFSlider.setRange(0.0, 1.0);
        CFSlider.setSliderStyle(Slider::SliderStyle::LinearVertical);
        CFSlider.setValue(0.5);
        CFSlider.setTextBoxStyle(juce::Slider::NoTextBox, false, 0, 0);
        CFSlider.setColour(Slider::backgroundColourId, juce::Colours::grey);
        CFSlider.setColour(Slider::trackColourId, juce::Colours::white);

}

Crossfader::~Crossfader()
{
}

void  Crossfader::paint(Graphics& g)
{
        //background of this class
        g.fillAll(juce::Colour(100 * CFSlider.getValue(), 0, 100 * (1 - CFSlider.getValue())));
        g.setColour(Colours::black);
        g.drawRect(getLocalBounds(), 1);

        //the text above the crossfader slider
        g.setColour(Colours::white);
        g.setFont(14.0f);
        g.drawText("Cross", 5, 5, getWidth(),20, Justification::centredLeft, true);
        g.drawText("Fader", 5, 25, getWidth(), 20, Justification::centredLeft, true);
}

void Crossfader::resized()
{
	CFSlider.setBounds(0, 40, getWidth(), getHeight()-80);
}


void Crossfader::sliderValueChanged(Slider* slider)
{
	if (slider == &CFSlider)
	{
	//retrieve the value of the volume slider from both decks when the crossfader slider is
moved
	float currentV1 = deckGUI1->volSlider.getValue();
	float currentV2 = deckGUI2->volSlider.getValue();

	//Applying the new volume of each players
	//This retrieving the CFSlider's value,
	//then apply it accordingly in setGain().
	auto mix = slider->getValue();
	player1->setGain(mix * currentV1);
	player2->setGain((1 - mix) * currentV2);

	//Allow the process of each deck changing their player's volume more naturally.
	deckGUI1->CFValue = mix;
	deckGUI2->CFValue = 1 - mix;
	}
}
```

CrossFader.h

```
#pragma once
#include "../JuceLibraryCode/JuceHeader.h"
#include "DJAudioPlayer.h"
#include "DeckGUI.h"

class Crossfader : public Component,
	public Slider::Listener
{
public:
	Crossfader(DJAudioPlayer* player1, DJAudioPlayer* player2, DeckGUI* deckGUI1,
DeckGUI* deckGUI2);
	~Crossfader();

	void paint(Graphics&) override;
	void resized() override;
	//define the interactables' listeners
	void sliderValueChanged(Slider* slider) override;

private:
	//define the interactables
	Slider CFSlider;

	//define variables for the class's inputs
	DJAudioPlayer* player1;
	DeckGUI* deckGUI1;
	DJAudioPlayer* player2;
	DeckGUI* deckGUI2;

	JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(Crossfader)
};
```

DeckGUI.cpp

```
#include "../JuceLibraryCode/JuceHeader.h"
#include "DeckGUI.h"
#include <iostream>
#include <string>
#include <sstream>

DeckGUI::DeckGUI(DJAudioPlayer* _player,
		AudioFormatManager &	formatManagerToUse,
		AudioThumbnailCache &	cacheToUse,
		int _rC, int _gC, int _bC
	) : player(_player),
	waveformDisplay(formatManagerToUse, cacheToUse),
	rC(_rC), gC(_gC), bC(_bC)
{
	//setting up the listeners
	addAndMakeVisible(playNstopButton);
	addAndMakeVisible(repeatButton);

	addAndMakeVisible(volSlider);
	addAndMakeVisible(speedSlider);
	addAndMakeVisible(waveformDisplay);
	addAndMakeVisible(posSlider);
	addAndMakeVisible(repeatSlider);

	playNstopButton.addListener(this);
	repeatButton.addListener(this);

	volSlider.addListener(this);
	speedSlider.addListener(this);
	posSlider.addListener(this);
	repeatSlider.addListener(this);

	//changing the UI of the listeners
	volSlider.setRange(0.0, 2.0);
	volSlider.setSliderStyle(juce::Slider::Rotary);
	volSlider.setTextBoxStyle(Slider::TextBoxLeft,true,50,15);
	volSlider.setValue(0.2);
	volSlider.setNumDecimalPlacesToDisplay(2);
	volSlider.setColour(Slider::textBoxOutlineColourId,Colours::limegreen);
	volSlider.setColour(Slider::textBoxBackgroundColourId,Colours::grey);
	volSlider.setColour(Slider::rotarySliderOutlineColourId, juce::Colours::grey);
	volSlider.setColour(Slider::rotarySliderFillColourId, juce::Colours::white);

	speedSlider.setRange(0.0, 3.0);
	speedSlider.setSliderStyle(Slider::SliderStyle::LinearVertical);
	speedSlider.setTextBoxStyle(Slider::TextBoxLeft, true, 50, 15);
	speedSlider.setValue(1.0);
	speedSlider.setNumDecimalPlacesToDisplay(2);
	speedSlider.setColour(Slider::textBoxOutlineColourId, Colours::limegreen);
	speedSlider.setColour(Slider::textBoxBackgroundColourId, Colours::grey);
	speedSlider.setColour(Slider::backgroundColourId, juce::Colours::grey);
	speedSlider.setColour(Slider::trackColourId, juce::Colours::white);

	posSlider.setRange(0.0, 1.0);
	posSlider.setSliderSnapsToMousePosition(true);
	posSlider.setSliderStyle(juce::Slider::LinearBar);
	posSlider.setTextBoxStyle(juce::Slider::NoTextBox, true, 0, 0);
	posSlider.setColour(juce::Slider::trackColourId, juce::Colours::transparentWhite);

	repeatSlider.setRange(0.0, 1.0);
	repeatSlider.setTextBoxStyle(juce::Slider::NoTextBox, false, 0, 0);
	repeatSlider.setColour(Slider::thumbColourId, juce::Colours::red);
	repeatSlider.setValue(0.0);

	//To store the CFSlider's value
	CFValue = 0.5;

	//To allow vinyl disk and cue arm to rotate
	startTimer(20);
	handAngle = 0.0;
	audioStateP = true;
}

DeckGUI::~DeckGUI()
{
	stopTimer();
}
```

```
void DeckGUI::paint (Graphics& g)
{
	//set the deck's background color
	g.fillAll(juce::Colour(rC,gC,bC));

	g.setColour (Colours::white);
	g.drawRect(getLocalBounds(), 1);
	g.setFont (20.0f);

	//to display the loaded song's name and duration
	g.drawText("Title : "+title, 10, 2, getWidth() / 4 - 10, getHeight() / 8,
Justification::centredLeft, true);
	g.drawText("Duration : "+songduration, 10, 2 + getHeight() / 8, getWidth() / 4 - 10,
getHeight() / 8, Justification::centredLeft, true);

	//to display the checkpoint's time stamp
	g.setFont(14.0f);
	std::string checkpointLine = checkpointPos + "/" + songduration;
	g.drawText("Checkpoint : " + checkpointLine, getWidth() * 3/8 + 10, getHeight() / 3,
getWidth() / 3, getHeight() / 6, Justification::centredLeft, true);

	//vinyl disk
	g.saveState();
	auto vinylDisk = ImageCache::getFromMemory(BinaryData::vinyldisk_png,
BinaryData::vinyldisk_pngSize);
	g.addTransform(juce::AffineTransform::translation(getWidth() / 20, getHeight() /2));
	g.addTransform(juce::AffineTransform::rotation(angle, vinylDisk.getWidth() / 2,
vinylDisk.getHeight() / 2));
	g.drawImageAt(vinylDisk, 0, 0, false);
	g.restoreState();

	//cue arm
	g.saveState();
	auto vinylHand = ImageCache::getFromMemory(BinaryData::vinylHand_png,
BinaryData::vinylHand_pngSize);
	g.addTransform(juce::AffineTransform::translation(getWidth() / 20  -
vinylDisk.getWidth()/1.25, getHeight() / 2));
	if (player->isPlaying()) {
	if (handAngle < 0.5) {
	handAngle = handAngle + 0.1;;
	}
	}
	else {
	if (handAngle > 0.1) {
	handAngle = handAngle - 0.1;
	}
	}
```

```
        g.addTransform(juce::AffineTransform::rotation(handAngle, vinylHand.getWidth() / 2,
vinylHand.getHeight() / 2));
        g.drawImageAt(vinylHand, 0, 0, false);
        g.restoreState();
}

void DeckGUI::resized()
{
        playNstopButton.setBounds(getWidth() / 4, 0, getWidth() / 8, getHeight() / 3);
        repeatButton.setBounds(getWidth() / 4, getHeight()/3, getWidth() / 8, getHeight() / 6);

        volSlider.setBounds(getWidth()*5/8, 0, getWidth()/4, getHeight()*2/3);
        speedSlider.setBounds(getWidth()*7/8, 0, getWidth()/8, getHeight()*2/3);

        waveformDisplay.setBounds(getWidth()/4, getHeight()*2/3, getWidth()*3/4,
getHeight()/3);
        posSlider.setBounds(getWidth() / 4, getHeight() * 2 / 3, getWidth()*3/4, getHeight() /
3);

        repeatSlider.setBounds(getWidth() / 4, getHeight() * 2 / 3 - 10, getWidth() * 3 / 4, 10);
}


void DeckGUI::buttonClicked(Button* button)
{
        if (button == &playNstopButton)
        {
        if (audioStateP) {
        player->start();
        audioStateP = false;
        button->setButtonText("STOP");
        }
        else {
        player->stop();
        audioStateP = true;
        button->setButtonText("PLAY");
        //setButtonText() allows the button to display different text after each interaction
        }
        }
        if (button == &repeatButton)
        {
        double RPS = repeatSlider.getValue();
        if (RPS >= 0 && RPS <= 1) { //ensuring the slider value is inside an acceptable range
        double songTotalLength = player->getLengthInSeconds();
        double location = songTotalLength * RPS;
        player->setPositionRelative(location);
        }
        }
```

```
}

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

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

        if (slider == &posSlider)
        {
        if (slider->getValue() >= 0 && slider->getValue() <= 1) {
        double songTotalLength = player->getLengthInSeconds();
        double location = songTotalLength * slider->getValue();
        player->setPositionRelative(location);
        }
        }
        if (slider == &repeatSlider)
        {
        //to allow the red vertical line of the checkpoint system to display itself at the right
position
        waveformDisplay.checkPointPosition = slider->getValue();
        if (songduration != " ") {
        //songduration is a string var that contain string in the format of "hh:mm:ss"
        //it is converted into seconds in the format of int s.
        int songdurationinSec = convertToSeconds(songduration);
        double relativeTimeinSec = songdurationinSec * slider->getValue();
        double relativeTimeinMin = relativeTimeinSec / 60;
        double relativeTimeinHr = relativeTimeinSec / 3600;
        //to get the remainder seconds
        int seconds = (relativeTimeinMin - floor(relativeTimeinMin)) * 60;
        //to get the remainder seconds in term of minutes
        int minutes = (relativeTimeinHr - floor(relativeTimeinHr)) * 60;
        //to get the hour
        int hours = floor(relativeTimeinHr);

        //to convert seconds,minutes, and hours into string
        std::string hourString = std::to_string(hours);
        std::string minString = std::to_string(minutes);
        std::string secString = std::to_string(seconds);

        //add 0 to the front of seconds,minutes, or hours if they are lesser than 10
        if (hours < 10) {
```

```
                        hourString = "0" + hourString;
            }
            if (minutes < 10) {
                        minString = "0" + minString;
            }
            if (seconds < 10) {
                        secString = "0" + secString;
            }

            //compiling the time into proper string format.
            checkpointPos = hourString + ":" + minString + ":" + secString;
            }
            }

}

bool DeckGUI::isInterestedInFileDrag (const StringArray &files)
{
  std::cout << "DeckGUI::isInterestedInFileDrag" << std::endl;
  return true;
}

void DeckGUI::filesDropped (const StringArray &files, int x, int y)
{
  //only load one song when dragged onto the deck
  if (files.size() == 1){
            auto songURL = URL{ File{files[0]} };
            player->loadURL(songURL);
            waveformDisplay.loadURL(songURL);
            //remove the directory of the input, leaving on the filename and its format
            title = juce::URL::removeEscapeChars(songURL.getFileName());
            //get the audio's time in the format of "hh:mm:ss"
            songduration = getSongsTime(File{ files[0] });
  }
}

void DeckGUI::timerCallback()
{
            //if an audio is playing, rotate the disk
            if (player->isPlaying())
            {
            //this allow the vinyl disk to increase its rotate speed according to the audio's speed
            angle += 0.05 * speedSlider.getValue();
            }

            //when the angle exceed 2PI, reset it back to 0
            if (angle >= 2 * juce::MathConstants<float>::pi) {
            angle -= 2 * juce::MathConstants<float>::pi;
```

```
	}

	//allow the changes to the vinyl disk and the cue arm to be displayed
	repaint();
	waveformDisplay.setPositionRelative(player->getPositionRelative());
}


int DeckGUI::convertToSeconds(const std::string& timeStr) {
	int hours, minutes, seconds;
	char colon;
	//to convert string into time format
	std::istringstream timeInStream(timeStr);

	//allow the variables to know what to takes based on the format of "hh:mm:ss"
	timeInStream >> hours >> colon >> minutes >> colon >> seconds;

	//convert it into seconds, and then return the output
	return hours * 3600 + minutes * 60 + seconds;
}

//get an audio's duration in string format
std::string DeckGUI::getSongsTime(const juce::File chosenFile) {
	formatManager.registerBasicFormats();
	auto reader = formatManager.createReaderFor(chosenFile);
	//essentially, finding how many seconds it takes to play the audio fully
	auto duration = reader->lengthInSamples / reader->sampleRate;

	//to convert it into human calender time format
	std::time_t epochTime(duration);
	tm* calenderTime = gmtime(&epochTime);

	std::string hours = std::to_string(calenderTime->tm_hour);
	std::string minutes = std::to_string(calenderTime->tm_min);
	std::string seconds = std::to_string(calenderTime->tm_sec);

	if (calenderTime->tm_hour < 10) {
	hours = "0" + hours;
	}

	if (calenderTime->tm_min < 10) {
	minutes = "0" + minutes;
	}

	if (calenderTime->tm_sec < 10) {
	seconds = "0" + seconds;
	}
```

```
        //to compile into string format of "hh:mm:ss"
        std::string songTime = hours + ":" + minutes + ":" + seconds;

        return songTime;
}
```

DeckGUI.h

```
#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#include "DJAudioPlayer.h"
#include "WaveformDisplay.h"

class DeckGUI        : public Component,
                public Button::Listener,
                public Slider::Listener,
                public FileDragAndDropTarget,
                public Timer
{
public:
        DeckGUI(DJAudioPlayer* player,
        AudioFormatManager &    formatManagerToUse,
        AudioThumbnailCache &    cacheToUse,
        int rC, int gC, int bC);
        ~DeckGUI();

        void paint (Graphics&) override;
        void resized() override;

        void buttonClicked (Button *) override;

        void sliderValueChanged (Slider *slider) override;

        bool isInterestedInFileDrag (const StringArray &files) override;
        void filesDropped (const StringArray &files, int x, int y) override;

        void timerCallback() override;

        int convertToSeconds(const std::string& timeStr);
        std::string getSongsTime(const juce::File chosenFile);

        //variables to be used by other functions
        Slider volSlider;
        Slider repeatSlider;

        WaveformDisplay waveformDisplay;
        float angle = 0.0;
```

```
	float handAngle;

	juce::String title = " ";
	std::string songduration = " ";
	std::string checkpointPos = " ";

	float CFValue;
private:
	//to retrieve the colour values for the deck's background
	int rC;
	int gC;
	int bC;

	juce::AudioFormatManager formatManager;

	AudioPlayHead* playHead;
	AudioPlayHead::CurrentPositionInfo currenPositionInfo;

	//to define the deck's listeners
	TextButton playNstopButton{ "PLAY" };
	TextButton repeatButton{ "JUMP TO CHECKPOINT" };

	bool audioStateP;

	Slider speedSlider;
	Slider posSlider;

	FileChooser fChooser{"Select a file..."};

	DJAudioPlayer* player;

	JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (DeckGUI)
};
```

DJAudioPlayer.cpp

```
#include "DJAudioPlayer.h"

DJAudioPlayer::DJAudioPlayer(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 AudioSourceChannelInfo& bufferToFill)
{
        resampleSource.getNextAudioBlock(bufferToFill);

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

void DJAudioPlayer::loadURL(URL audioURL)
{
        auto* reader =
formatManager.createReaderFor(audioURL.createInputStream(false));
        if (reader != nullptr)
        {
        std::unique_ptr<AudioFormatReaderSource> newSource (new
AudioFormatReaderSource (reader, true));
        transportSource.setSource (newSource.get(), 0, nullptr, reader->sampleRate);

        readerSource.reset (newSource.release());
        }
}

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);
        }

}

float DJAudioPlayer::getGain() {
        return transportSource.getGain();
}

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);
	}
}
void DJAudioPlayer::setPosition(double posInSecs)
{
	//with the changes I made in deckgui, I clean up this function to contain only what I
need from it.
	transportSource.setPosition(posInSecs);
}

void DJAudioPlayer::setPositionRelative(double pos)
{
	setPosition(pos);
}


void DJAudioPlayer::start()
{
	transportSource.start();
}
void DJAudioPlayer::stop()
{
  transportSource.stop();
}

double DJAudioPlayer::getPositionRelative()
{
	return transportSource.getCurrentPosition() / transportSource.getLengthInSeconds();
}

//to check of the audio is playing
bool DJAudioPlayer::isPlaying()
{
	return transportSource.isPlaying();
}

double DJAudioPlayer::getLengthInSeconds()
{
	return transportSource.getLengthInSeconds();
}
```

DJAudioPlayer.h

```
#pragma once

#include "../JuceLibraryCode/JuceHeader.h"

class DJAudioPlayer : public AudioSource {
  public:

        DJAudioPlayer(AudioFormatManager& _formatManager);
        ~DJAudioPlayer();

        void prepareToPlay (int samplesPerBlockExpected, double sampleRate) override;
        void getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill) override;
        void releaseResources() override;

        void loadURL(URL audioURL);

        void setGain(double gain);
        float getGain();
        void setSpeed(double ratio);
        void setPosition(double posInSecs);
        void setPositionRelative(double pos);

        bool isPlaying();

        void start();
        void stop();

        double getPositionRelative();
        double getLengthInSeconds();

private:
        AudioFormatManager& formatManager;
        std::unique_ptr<AudioFormatReaderSource> readerSource;
        AudioTransportSource transportSource;
        ResamplingAudioSource resampleSource{&transportSource, false, 2};

};
```

Main.cpp → I did not touch it at all

MainComponent.cpp

```
#include "MainComponent.h"

MainComponent::MainComponent()
{
        //to define the size of the DJ application
        setSize (800, 600);

        //some require permissions to open input channels so request that here
        if (RuntimePermissions::isRequired (RuntimePermissions::recordAudio)
        && ! RuntimePermissions::isGranted (RuntimePermissions::recordAudio))
        {
        RuntimePermissions::request (RuntimePermissions::recordAudio,
                        [&] (bool granted) { if (granted)  setAudioChannels (2, 2); });
        }
        else
        {
        //to specify the number of input and output channels
        setAudioChannels (0, 2);
        }

        //allow the components to be visible
        addAndMakeVisible(deckGUI1);
        addAndMakeVisible(deckGUI2);

        addAndMakeVisible(crossFader);

        addAndMakeVisible(playlistComponent);

        formatManager.registerBasicFormats();
}

MainComponent::~MainComponent()
{
        //to shut down the audio device and clears the audio source.
        shutdownAudio();
}

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 MainComponent::getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill)
{
        mixerSource.getNextAudioBlock(bufferToFill);
}
```

```
void MainComponent::releaseResources()
{
	//to be called when the audio device stops
	player1.releaseResources();
	player2.releaseResources();
	mixerSource.releaseResources();
}

void MainComponent::paint (Graphics& g)
{
	g.fillAll (getLookAndFeel().findColour (ResizableWindow::backgroundColourId));
	g.setColour(Colours::white);
	g.setFont(14.0f);
}

void MainComponent::resized()
{
	//to define the components' position
	crossFader.setBounds(0, 0, getWidth() / 20, getHeight() *  9/ 12);
	deckGUI1.setBounds(getWidth()/20, 0, getWidth() * 19/20, getHeight() * 4.5/12);
	deckGUI2.setBounds(getWidth()/20, getHeight()*4.5/12, getWidth()*19/20 ,
getHeight() *  4.5/12);
	playlistComponent.setBounds(0, getHeight()*9/12, getWidth(), getHeight()*3/12);
}
```

MainComponent.h

```
#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#include "DJAudioPlayer.h"
#include "DeckGUI.h"
#include "PlaylistComponent.h"
#include "CrossFader.h"

class MainComponent   : public AudioAppComponent
{
public:
	MainComponent();
	~MainComponent();

	void prepareToPlay (int samplesPerBlockExpected, double sampleRate) override;
	void getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill) override;
	void releaseResources() override;

	void paint (Graphics& g) override;
	void resized() override;
private:
        AudioFormatManager formatManager;
        AudioThumbnailCache thumbCache{100};

        DJAudioPlayer player1{formatManager};
        //added the colour format at the end to define the deck's background colour
        DeckGUI deckGUI1{&player1, formatManager, thumbCache, 100,0,0};

        DJAudioPlayer player2{formatManager};
        //added the colour format at the end to define the deck's background colour
        DeckGUI deckGUI2{&player2, formatManager, thumbCache,0,0,100};

        //define the crossfader class
        Crossfader crossFader{&player1,&player2,&deckGUI1,&deckGUI2};

        MixerAudioSource mixerSource;


        PlaylistComponent playlistComponent{ &player1,&player2, &deckGUI1, &deckGUI2
};

        JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MainComponent)
};
```

PlaylistComponent.cpp

```
#include <JuceHeader.h>
#include "PlaylistComponent.h"
using namespace std;

//by taking the players and the decks as inputs, the playlist can interact with them to load
songs
PlaylistComponent::PlaylistComponent(DJAudioPlayer* _player1, DJAudioPlayer* _player2,
DeckGUI* _deckGUI1, DeckGUI* _deckGUI2
) : player1(_player1), player2(_player2), deckGUI1(_deckGUI1), deckGUI2(_deckGUI2)
{
        //define the listener
        addAndMakeVisible(loadButton);
        loadButton.addListener(this);

        //define the columns available of the table
        tableComponent.getHeader().addColumn("title", 1, 300);
        tableComponent.getHeader().addColumn("length", 2, 100);
        tableComponent.getHeader().addColumn(" ", 3, 133);
        tableComponent.getHeader().addColumn(" ", 4, 133);
        tableComponent.getHeader().addColumn(" ", 5, 133);
tableComponent.getHeader().setColour(TableHeaderComponent::backgroundColourId,
juce::Colours::white);

        tableComponent.setModel(this);

        addAndMakeVisible(tableComponent);
}

PlaylistComponent::~PlaylistComponent()
{
}

void PlaylistComponent::paint (juce::Graphics& g)
{
        g.fillAll(juce::Colour(20, 20, 20));

        g.setColour (juce::Colours::black);
        g.drawRect (getLocalBounds(), 1);
        g.setFont (14.0f);
}

void PlaylistComponent::resized()
{
        //setting the bounds of the load button and the playlist table
        loadButton.setBounds(0, 0, getWidth(), 20);
        tableComponent.setBounds(0, 20, getWidth(), getHeight()-20);
}

int PlaylistComponent::getNumRows(){
        return trackTitles.size();
}

void PlaylistComponent::paintRowBackground(Graphics & g, int rowNumber, int width, int
height, bool rowIsSelected){
        if (rowIsSelected) {
        //if the user select a row, it will highlight it with purple, else, it will be dark gray.
        g.fillAll(Colours::mediumpurple);
        }else{
        g.fillAll(juce::Colour(40, 40, 40));
        }
}
void PlaylistComponent::paintCell(Graphics & g, int rowNumber, int columnId, int width, int
height, bool rowIsSelected){
        if (columnId == 1)
        {
        //display the audio file's name
        g.setColour(Colours::yellow);
        g.drawText(trackTitles[rowNumber], 2, 0, width, height, Justification::centredLeft,
true);
        }
        if (columnId == 2)
        {
        //display the audio file's duration
        g.setColour(Colours::yellow);
        g.drawText(songDuration[rowNumber], 2, 0, width, height, Justification::centredLeft,
true);
        }
}

Component* PlaylistComponent::refreshComponentForCell(int rowNumber, int columnId,
bool isRowSelected, Component *existingComponentToUpdate) {
        //create listeners to load audio into deck1, to load audio into deck2, and to remove
the file from the playlist
        if(columnId == 3){
        if(existingComponentToUpdate==nullptr){
        TextButton* btn = new TextButton{"Remove"};
        String id{std::to_string(rowNumber)};
        btn->setComponentID(id);

        btn->addListener(this);
        existingComponentToUpdate=btn;
        btn->onClick = [this] {removeFromPlaylist(); };
        }
        }
        if (columnId == 4) {
        if (existingComponentToUpdate == nullptr) {
        TextButton* btn = new TextButton{ "Deck1" };
        String id{ std::to_string(rowNumber) };
        btn->setComponentID(id);

        btn->addListener(this);
        existingComponentToUpdate = btn;
        btn->onClick = [this] {loadAudioIntoDeck1();};
        }
        }
        if (columnId == 5) {
        if (existingComponentToUpdate == nullptr) {
        String id{ std::to_string(rowNumber) };
        TextButton* btn = new TextButton{ "Deck2"};
        btn->setComponentID(id);

        btn->addListener(this);
        existingComponentToUpdate = btn;
        btn->onClick = [this] {loadAudioIntoDeck2(); };
        }
        }
        return existingComponentToUpdate;
}


void PlaylistComponent::buttonClicked(Button * button) {
        if (button == &loadButton)
        {
        //to load one audio file into the playlist
        auto fileChooserFlags = FileBrowserComponent::canSelectFiles;
        fChooser.launchAsync(fileChooserFlags, [this](const FileChooser& chooser)
        {
        auto songURL = URL{ chooser.getResult() };
        song.add(songURL);
        //add the necessary detail into arrays, to be displayed in the table
        trackTitles.add(juce::URL::removeEscapeChars(songURL.getFileName()));
        songDuration.add(deckGUI1->getSongsTime(chooser.getResult()));
        tableComponent.updateContent(); //to display changes to the table from the new
element of the arrays
        });
        }

        //to save the id of the row the button is clicked.
        selected = button->getComponentID().getDoubleValue();
}

//these functions rely on int id saved by the variable 'selected' to load or remove the correct
audio
void PlaylistComponent::removeFromPlaylist() {
        song.remove(selected);
        trackTitles.remove(selected);
        tableComponent.updateContent();
}

void PlaylistComponent::loadAudioIntoDeck1() {
        player1->loadURL(song[selected]);
        //setting up the required prerequisite when loading a song into the deck
        deckGUI1->waveformDisplay.loadURL(song[selected]);
        deckGUI1->angle = 0.0;
        deckGUI1->title = trackTitles[selected];
        deckGUI1->songduration = songDuration[selected];
}

void PlaylistComponent::loadAudioIntoDeck2() {
        player2->loadURL(song[selected]);
        //setting up the required prerequisite when loading a song into the deck
        deckGUI2->waveformDisplay.loadURL(song[selected]);
        deckGUI2->angle = 0.0;
        deckGUI2->title = trackTitles[selected];
        deckGUI2->songduration = songDuration[selected];
}

//by interating the dragged files, this allow multiple files to be added to the playlist at a time
void PlaylistComponent::filesDropped(const StringArray& files, int x, int y) {
        for (int i = 0; i < files.size(); i = i + 1) {
        song.add(URL{ File{files[i]} });
        trackTitles.add(juce::URL::removeEscapeChars(juce::URL{ File{files[i]}
}.getFileName()));

        songDuration.add(deckGUI1->getSongsTime(File{ files[i] }));

        tableComponent.updateContent();
        }
}

bool PlaylistComponent::isInterestedInFileDrag(const StringArray& files) {
        return true;
}
```

PlaylistComponent.h

```
#pragma once

#include <JuceHeader.h>
#include <vector>
#include <string>

#include "DJAudioPlayer.h"
#include "DeckGUI.h"

class PlaylistComponent  : public juce::Component, public TableListBoxModel, public
Button::Listener, public juce::TextEditor::Listener,
                    public FileDragAndDropTarget
{
public:
        PlaylistComponent(DJAudioPlayer* player1, DJAudioPlayer* player2, DeckGUI*
deckGUI1, DeckGUI* deckGUI2);
        ~PlaylistComponent() override;

        void paint (juce::Graphics&) override;
        void resized() override;

        int getNumRows() override;
        void paintRowBackground(Graphics & g, int rowNumber, int width, int height, bool
rowIsSelected) override;
```

```
	void paintCell(Graphics & g, int rowNumber, int columnId, int width, int height, bool
rowIsSelected) override;

	Component* refreshComponentForCell(int rowNumber, int columnId, bool
isRowSelected, Component *existingComponentToUpdate) override;

	void buttonClicked(Button * button) override;
	//defining the function to remove songs from the playlist
	void removeFromPlaylist();

	//defining the functions to add an audio to their respective players.
	void loadAudioIntoDeck1();
	void loadAudioIntoDeck2();

	bool isInterestedInFileDrag(const StringArray& files) override;
	void filesDropped(const StringArray& files, int x, int y) override;

private:
	juce::AudioFormatManager formatManager;

	int selected;

	DJAudioPlayer* player1;
	DJAudioPlayer* player2;

	DeckGUI* deckGUI1;
	DeckGUI* deckGUI2;

	TableListBox tableComponent;

	//define arrays to store necessary informations of audios in the playlist
	juce::Array<juce::String> trackTitles;
	juce::Array<juce::URL> song;
	juce::Array<std::string> songDuration;

	//define the listener
	TextButton loadButton{ "LOAD A SONG INTO THE PLAYLIST" };

	FileChooser fChooser{ "Select a file...", juce::File(), "*.mp3;*.wav;*.aiff"};

	JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR
(PlaylistComponent)
};
```

WaveformDisplay.cpp

```
#include "../JuceLibraryCode/JuceHeader.h"
#include "WaveformDisplay.h"
```

```
WaveformDisplay::WaveformDisplay(AudioFormatManager &     formatManagerToUse,
                                AudioThumbnailCache &     cacheToUse) :
                                audioThumb(1000, formatManagerToUse, cacheToUse),
                                fileLoaded(false),
                                position(0)

{
  audioThumb.addChangeListener(this);
}

WaveformDisplay::~WaveformDisplay()
{
}

void WaveformDisplay::paint (Graphics& g)
{
        //change the waveform's background
        g.fillAll (juce::Colour(40, 40, 40));

        g.setColour (Colours::grey);
        g.drawRect (getLocalBounds(), 1);

        //to set the waveform's colour as yellow
        g.setColour (Colours::yellow);
        if(fileLoaded)
        {
        //to display the waveform if an audio file is loaded
        audioThumb.drawChannel(g,
        getLocalBounds(),
        0,
        audioThumb.getTotalLength(),
        0,
        1.0f
        );
        //to draw the rectangle to indicate which section the audio is being played at
        g.setColour(Colours::green);
        g.drawRect(position * getWidth(), 0, getWidth() / 20, getHeight());
        }
        else
        {
        //to display text when no audio is loaded
        g.setFont (20.0f);
        g.drawText ("LOAD A FILE", getLocalBounds(),
                Justification::centred, true);
        }

        //to draw the red vertical line of the checkpoint system
        g.setColour(Colours::red);
        g.fillRect(checkPointPosition * getWidth(), 0, 2, getHeight());
}

void WaveformDisplay::resized()
{
}

void WaveformDisplay::loadURL(URL audioURL)
{
  //to draw up the waveform of the audio file loaded
  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;
  }

}

void WaveformDisplay::changeListenerCallback (ChangeBroadcaster *source)
{
        std::cout << "wfd: change received! " << std::endl;
        //to display the waveform after loading an audio
        repaint();

}

void WaveformDisplay::setPositionRelative(double pos)
{
  //to move the green rectangle accordingly when the audio is playing
  if (pos != position)
  {
        position = pos;
        repaint();
  }
}
```

WaveformDisplay.h

```
#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
```

```
class WaveformDisplay       : public Component,
                    public ChangeListener
{
public:
        WaveformDisplay( AudioFormatManager &    formatManagerToUse,
                    AudioThumbnailCache &    cacheToUse );
        ~WaveformDisplay();

        void paint (Graphics&) override;
        void resized() override;

        void changeListenerCallback (ChangeBroadcaster *source) override;

        void loadURL(URL audioURL);

        void setPositionRelative(double pos);
        //defined in public for the checkpoint system's slider to move the red vertical line
        double checkPointPosition = 0;
private:
        AudioThumbnail audioThumb;
        bool fileLoaded;
        //to store the coordinate, to move the green box accordingly
        double position;

        JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (WaveformDisplay)
};
```