File Overview:

- **Main.cpp:** contains the basic startup code for a JUCE application
- **MainComponent.h:** Define the main user interface component
- **MainComponent.cpp:** Implementation of the main user interface component
- **DeckGUI.h:** Defines the graphical interface for the DJ deck
- **DeckGUI.cpp:** Implementation of the graphical interface of the DJ deck
- **DJAudioPlayer.h:** Define audio playback functionality
- **DJAudioPlayer.cpp:** Implementation of audio playback functionality
- **WaveformDisplay.h:** Define the waveform visualization component
- **WaveformDisplay.cpp:** Controls the waveform visualization
- **PlaylistComponent.h:** Set up the playlist management component
- **PlaylistComponent.cpp:** Implementation of the playlist management
- **RotatingDisk.h:** Define the rotating disk component
- **RotatingDisk.cpp:** Controls the rotating disk animation component
- **DrumPadComponent.h:** Set up the drum pad interface
- **DrumPadComponet.cpp:** Controls how the drum pads works

Main.cpp

```
// Followed the starter code

/*

==============================================================================

  This file contains the basic startup code for a JUCE application.

==============================================================================

*/

// Include the juce library and the header file

#include <JuceHeader.h>

#include "MainComponent.h"

// Definition of the main application class

class OtoDecks2Application  : public juce::JUCEApplication

{

public:
```

```
//==============================================================================
=========

  // Constructor for the class

  OtoDecks2Application() {}

  // application name

  const juce::String getApplicationName() override       { return
ProjectInfo::projectName; }

  // application version

  const juce::String getApplicationVersion() override    { return
ProjectInfo::versionString; }

  // Allow opening multiple copies of the application

  bool moreThanOneInstanceAllowed() override             { return true; }

//==============================================================================

  void initialise (const juce::String& commandLine) override

  {

    // Main window of the application

    mainWindow.reset (new MainWindow (getApplicationName()));

  }


  void shutdown() override

  {

    // Application's shutdown code

    mainWindow = nullptr;

  }

//==============================================================================

  void systemRequestedQuit() override

  {

    // To close the app
    quit();
  }
  void anotherInstanceStarted (const juce::String& commandLine) override
  {
    // When you launch another instance of the application while this one is still
running,
    // this method is called, and the commandLine parameter provides you with the
    // command - line arguments for the other instance.
  }
//==============================================================================
  /*
    This class implements the desktop window that contains an instance of
    our MainComponent class.
  */
  // Definition of the main window class
  class MainWindow    : public juce::DocumentWindow
  {
  public:
    MainWindow (juce::String name)
      : DocumentWindow (name,
                juce::Desktop::getInstance().getDefaultLookAndFeel()
                              .findColour (juce::ResizableWindow::backgroundColourId),
                DocumentWindow::allButtons)
    {
      setUsingNativeTitleBar (true);
      setContentOwned (new MainComponent(), true);

     #if JUCE_IOS || JUCE_ANDROID
```

```
            setFullScreen (true);
           #else
            setResizable (true, true);
            centreWithSize (getWidth(), getHeight());
           #endif
            setVisible (true);
        }
        void closeButtonPressed() override
        {
            // This is called when the user tries to close this window. Here, we'll just
            // ask the app to quit when this happens, but you can change this to do
            // whatever you need.
            JUCEApplication::getInstance()->systemRequestedQuit();
        }

        /* Note: Be careful if you override any DocumentWindow methods - the base
           class uses a lot of them, so by overriding you might break its functionality.
           It's best to do all your work in your content component instead, but if
           you really have to override any DocumentWindow methods, make sure your
           subclass also calls the superclass's method.
        */
    private:
        JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MainWindow)
    };
private:
    std::unique_ptr<MainWindow> mainWindow;
};
```

```
//==============================================================================
// This macro generates the main() routine that launches the app.
START_JUCE_APPLIC
```

// End of following starter code

MainComponent.h

// **Followed the starter code**

// The header file contains the main part of the DJ app which combines the 2 audio players, their controls, a playlist, and a mixer to handle playback and user interaction.

```
// Include all the libraries and the header files
#pragma once
#include <JuceHeader.h>
#include "DJAudioPlayer.h"
#include "DeckGUI.h"
#include "PlaylistComponent.h"
#include "WaveformDisplay.h"
//==============================================================================
// Include all JUCE classes
class MainComponent  : public juce::AudioAppComponent
{
public:
//==============================================================================
   // Called to set up everything when the program starts
   MainComponent();
   // Cleans up everything when the program closes
   ~MainComponent() override;
//==============================================================================
   // Prepares the audio system for playback
   void prepareToPlay (int samplesPerBlockExpected, double sampleRate) override;
```

```
  // Continuously provides the next piece of audio to play
  void getNextAudioBlock (const juce::AudioSourceChannelInfo& bufferToFill) override;
  // Free up memory when audio stops
  void releaseResources() override;
//==============================================================================
  // Paint function to draw on the screen
  void paint (juce::Graphics& g) override;
  // To resize the components on the screen
  void resized() override;
private:
  // To load and play different types of audio files
  juce::AudioFormatManager formatManager;
  // Stores the waveform previews of audio files
  juce::AudioThumbnailCache thumbCache{100};
  // 1st audio player that plays songs
  DJAudioPlayer player1{ formatManager };
  // End of following starter code
  // Deck 1 controls
  DeckGUI deckGUI1{ &player1, formatManager, thumbCache,
juce::Colours::deepskyblue };
  // Followed the starter code
  // 2nd audio player that plays songs
  DJAudioPlayer player2{formatManager};
  //End of following starter code
  // Deck 2 controls
  DeckGUI deckGUI2{ &player2, formatManager, thumbCache , juce::Colours::red };
```

// **Followed the starter code**

```
    // Mixes both deck audio together
    juce::MixerAudioSource mixerSource;
    // Manage and show the song list that was loaded into the playlist table
    PlaylistComponent playlistComponent;
    // Prevents copying this class to avoid memory issues
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MainComponent)
};
```

// End of following starter code

MainComponent.cpp

```
// The cpp file contains the implementation of the header file which is the main part of the DJ app which combines the 2 audio players,
// their controls, a playlist, and a mixer to handle playback and user interaction.
// Include the header file
#include "MainComponent.h"
//==============================================================================
// Contructor to initialise the main component
MainComponent::MainComponent()
    : deckGUI1(&player1, formatManager, thumbCache, juce::Colours::deepskyblue),
    deckGUI2(&player2, formatManager, thumbCache, juce::Colours::red)
{
    // Main window size
    setSize (1100, 600);
```

// **Followed the starter code**

```
    // Some platforms require permissions to open input channels so request that here
    if (juce::RuntimePermissions::isRequired (juce::RuntimePermissions::recordAudio)
        && ! juce::RuntimePermissions::isGranted
(juce::RuntimePermissions::recordAudio))
```

```
  {
    // If accepted, ask for permission and set up audio channels.
    juce::RuntimePermissions::request (juce::RuntimePermissions::recordAudio,
                          [&] (bool granted) { setAudioChannels (granted ? 2 : 0, 2); });
  }
  else
  {
    // Specify the number of input and output channels that we want to open
    setAudioChannels (0, 2);
  }
  // Show the decks on the screen
  addAndMakeVisible(deckGUI1);
  addAndMakeVisible(deckGUI2);
  // Register basic audio file formats
  formatManager.registerBasicFormats();
```
// End of following starter code
```
  // To load a track into Deck 1 when selected from the playlist
playlistComponent.onLoadToDeck1 = [this](const juce::String& track)
   {
      DBG("Loading track to Deck 1: " + track);

      // Start from the current working directory
      juce::File currentDir = juce::File::getCurrentWorkingDirectory();

      // Search for the "tracks" folder by moving up the directory tree
      while (currentDir.exists() && !currentDir.getChildFile("tracks").exists())
      {
        currentDir = currentDir.getParentDirectory();
```

```
    }

    // If "tracks" is found, construct the full path to the track file
    juce::File trackFile;
    if (currentDir.getChildFile("tracks").exists())
    {
        trackFile = currentDir.getChildFile("tracks").getChildFile(track + ".mp3");
    }

    // Check if the file exists
    if (trackFile.exists())
    {
        DBG("Track exists: " + trackFile.getFullPathName());

        // Load the track
        deckGUI1.loadTrack(trackFile.getFullPathName());

        // Show the title in the deck
        deckGUI1.setCurrentTrackTitle(track);
    }
    else
    {
        // Console statement when the track is not found
        DBG("Track file not found: " + trackFile.getFullPathName());
    }
};
```

```
// To load a track into Deck 2 when selected from the playlist
playlistComponent.onLoadToDeck2 = [this](const juce::String& track)
  {
      DBG("Loading track to Deck 2: " + track);

      // Start from the current working directory
      juce::File currentDir = juce::File::getCurrentWorkingDirectory();

      // Search for the "tracks" folder by moving up the directory tree
      while (currentDir.exists() && !currentDir.getChildFile("tracks").exists())
      {
         currentDir = currentDir.getParentDirectory();
      }

      // If "tracks" is found, construct the full path to the track file
      juce::File trackFile;
      if (currentDir.getChildFile("tracks").exists())
      {
         trackFile = currentDir.getChildFile("tracks").getChildFile(track + ".mp3");
      }

      // Check if the file exists
      if (trackFile.exists())
      {
         DBG("Track exists: " + trackFile.getFullPathName());

         // Load the track
         deckGUI2.loadTrack(trackFile.getFullPathName());
```

```
            // Show the title in the deck
            deckGUI2.setCurrentTrackTitle(track);
        }
        else
        {
            // Console statement when the track is not found
            DBG("Track file not found: " + trackFile.getFullPathName());
        }
    };
```
// **Followed the starter code**

```
    // Show the playlist component on the screen
    addAndMakeVisible(playlistComponent);
}


MainComponent::~MainComponent()
{
    // This shuts down the audio device and clears the audio source.
    shutdownAudio();
}
//==============================================================================
void MainComponent::prepareToPlay (int samplesPerBlockExpected, double
sampleRate)
{
    // Add player 1 and player 2 into the mixer
    mixerSource.addInputSource(&player1, false);
    mixerSource.addInputSource(&player2, false);
    // Prepare the mixer to play at the given sample rate

    mixerSource.prepareToPlay(samplesPerBlockExpected, sampleRate);

}


void MainComponent::getNextAudioBlock(const juce::AudioSourceChannelInfo&
bufferToFill)

{

    // Get audio from the mixer

    mixerSource.getNextAudioBlock(bufferToFill);

}


void MainComponent::releaseResources()

{

    // Remove all audio sources from the mixer

    mixerSource.removeAllInputs();


    // Release the resources

    mixerSource.releaseResources();

    player1.releaseResources();

    player2.releaseResources();

}

//==============================================================================

void MainComponent::paint (juce::Graphics& g)

{

    // Background fill

    g.fillAll (getLookAndFeel().findColour (juce::ResizableWindow::backgroundColourId));

}

void MainComponent::resized()
```

```
{
   // Position and size of deck 1
   deckGUI1.setBounds(0, 0, getWidth() / 2, getHeight() / 2);
   // Position and size of deck 2
   deckGUI2.setBounds(getWidth() / 2, 0, getWidth() / 2, getHeight() / 2);
   // Position and sizr of playlist component
   playlistComponent.setBounds(0, getHeight() / 2, getWidth(), getHeight() / 2);
}
```

// End of following starter code

DeckGUI.h

```
// The header file contains the DJ deck, handling playback, UI and file loading.
/*
==============================================================================
   DeckGUI.h
   Created: 28 Jan 2025 10:53:35pm
   Author:  Subathra
==============================================================================
*/
```

// **Followed the starter code**

```
// Include all the libraries and the header files
#pragma once
#include <JuceHeader.h>
#include "DJAudioPlayer.h"
#include "WaveformDisplay.h"
#include "RotatingDisk.h"
#include "DrumPadComponent.h"
//==============================================================================
/*
```

```
*/
// Include all JUCE classes
class DeckGUI  : public juce::Component,
            public juce::Button::Listener,
            public juce::Slider::Listener,
            public juce::FileDragAndDropTarget,
            public juce::Timer
{
public:
  // Constructor which has the deck GUI with player, format manager and cache
  DeckGUI(DJAudioPlayer* _player,
       juce::AudioFormatManager & formatManagerToUse,
       juce::AudioThumbnailCache & cacheToUse,
       juce::Colour color);

  // Cleans up everything when the program closes
  ~DeckGUI() override;

  // Paint function to draw on the screen
  void paint (juce::Graphics&) override;

  // To resize the components on the screen
  void resized() override;

  // Called when a button is clicked
  void buttonClicked(juce::Button* button) override;

  // Called when slider value changes
```

```
    void sliderValueChanged(juce::Slider* slider) override;

    // Checks if files are dragged into the component
    bool isInterestedInFileDrag(const juce::StringArray& files) override;

    // Called when files are dropped in the component
    void filesDropped(const juce::StringArray& files, int x, int y) override;

    // Update UI at regular intervals
    void timerCallback() override;
// End of following starter code

    // Loads audio track into the deck
    void loadTrack(const juce::String& filePath);

    // Display the title track below the waveform
    void setCurrentTrackTitle(const juce::String& title);

private:

    // loop variable to keep track if the looping is on
    bool looping = false;

    // Store the current title of the track
    juce::String currentTrackTitle;

// Followed the starter code
    // Button to play the track
    juce::TextButton playButton{ "PLAY" };


    // Button to stop the track
    juce::TextButton stopButton{ "STOP" };
```
// End of following starter code
```
    // Button to loop the track
    juce::TextButton loopButton{ "LOOP" };

```
// **Followed the starter code**
```
   // Button to load the track
    juce::TextButton loadButton{ "LOAD" };


    // Volumn slider to adjust the volumn of the track
    juce::Slider volSlider;


    // Speed slider to adjust the speed of the track
    juce::Slider speedSlider;


    // Position slider to adjust the position of the track
    juce::Slider posSlider;
```
// End of following starter code
```
    // Volumn label to be shown on the slider
    juce::Label volSliderLabel;


    // Speed label to be shown on the slider
    juce::Label speedSliderLabel;


    // Position label to be shown on the slider
    juce::Label posSliderLabel;


// Followed the starter code
    // Pointer to the audio player
    DJAudioPlayer* player;


    // File chooser for loading tracks
    juce::FileChooser fChooser{ "Select a file..." };
// Waveform of the loaded track
    WaveformDisplay waveformDisplay;
// End of following starter code


    // Custom UI theme
    juce::LookAndFeel_V4 lookandfeel;


    // Rotating disk effect
    RotatingDisk rotatingDisk;


    // Drum pad for real feature in dj program
    DrumPadComponent DrumPad;


// Followed the starter code
    // Prevents copying this class to avoid memory issues
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (DeckGUI)
};
// End of following stater code
```

DeckGUI.cpp

```
// The cpp file contains the implementation of the DJ deck, handling playback, UI and
file loading.

/*
==============================================================================

  DeckGUI.cpp

  Created: 28 Jan 2025 10:53:35pm

  Author:  Subathra

==============================================================================

*/


// Include the header file and juce library

#include <JuceHeader.h>

#include "DeckGUI.h"

// Constructor for the class

DeckGUI::DeckGUI(DJAudioPlayer* _player,

  juce::AudioFormatManager& formatManagerToUse,

  juce::AudioThumbnailCache& cacheToUse,

  juce::Colour color

) : player(_player),

waveformDisplay(formatManagerToUse, cacheToUse, color)

{

// Followed the starter code

  // Show the play button on the screen

  addAndMakeVisible(playButton);

  // Show the stop button on the screen

  addAndMakeVisible(stopButton);

// End of following starter code

  // Show the loop button on the screen

  addAndMakeVisible(loopButton);
```

```
// Followed the starter code
   // Show the load button on the screen
   addAndMakeVisible(loadButton);
   // Show volumn slider on the screen
   addAndMakeVisible(volSlider);
   // Show speed slider on the screen
   addAndMakeVisible(speedSlider);
   // Show position slider on the screen
   addAndMakeVisible(posSlider);
// End of following starter code
   // Show volumn slider label on the screen
   addAndMakeVisible(volSliderLabel);
   // Show speed slider label on the screen
   addAndMakeVisible(speedSliderLabel);
   // Show position slider label on the screen
   addAndMakeVisible(posSliderLabel);
// Followed the starter code
   // Show the waveform on the screen
   addAndMakeVisible(waveformDisplay);
// End of following starter code
   // Show the rotating disk on the screen
   addAndMakeVisible(rotatingDisk);
   // Show the drum pad component on the scrren
   addAndMakeVisible(DrumPad);
// Followed the starter code
   // Listener for the play button
   playButton.addListener(this);
   // Listener for the stop button
```

```
  stopButton.addListener(this);
// End of following starter code
  // Listerner for the loop button
  loopButton.addListener(this);
// Followed the starter code
  // Listener for the load Button
  loadButton.addListener(this);
  // Listener for the volumn slider
  volSlider.addListener(this);
  // Listener for the speed slider
  speedSlider.addListener(this);
  // Listener for the position slider
  posSlider.addListener(this);
  // Range of the volumn slider
  volSlider.setRange(0.0, 1.0);
  // Range of the speed slider
  speedSlider.setRange(0.1, 100.0);
  // Range of the position slider
  posSlider.setRange(0.0, 1.0);
// End of following starter code
  // Volumn slider label
  volSliderLabel.setText("Volume", juce::NotificationType::dontSendNotification);
  volSliderLabel.setFont(juce::Font(14.0f));


  // Speed slider label
  speedSliderLabel.setText("Speed", juce::NotificationType::dontSendNotification);
  speedSliderLabel.setFont(juce::Font(14.0f));
```

```
// Position slider label
posSliderLabel.setText("Position", juce::NotificationType::dontSendNotification);
posSliderLabel.setFont(juce::Font(14.0f));

// Make the values of volumn slider to be 2 decimal places
volSlider.setNumDecimalPlacesToDisplay(2);

// Default volumn to be 80%
volSlider.setValue(0.8);

// Make the volumn slider to be vertical bar with no text box
volSlider.setSliderStyle(juce::Slider::SliderStyle::LinearBarVertical);
volSlider.setTextBoxStyle(juce::Slider::NoTextBox, false, 0, 0);

// Make the values of speed slider to be 2 decimal places
speedSlider.setNumDecimalPlacesToDisplay(2);

// Make the speed slider to be vertical bar with no text box
speedSlider.setSliderStyle(juce::Slider::SliderStyle::LinearBarVertical);
speedSlider.setTextBoxStyle(juce::Slider::NoTextBox, false, 0, 0);

// Make the values of position slider to be 2 decimal places
posSlider.setNumDecimalPlacesToDisplay(2);

// Make the position slider to be vertical bar with no text box
posSlider.setSliderStyle(juce::Slider::SliderStyle::LinearBarVertical);
posSlider.setTextBoxStyle(juce::Slider::NoTextBox, false, 0, 0);
```

```
    // Slider color to be light grey
    getLookAndFeel().setColour(juce::Slider::trackColourId,
juce::Colours::lightslategrey);

    // Play button color to be green
    playButton.setColour(juce::TextButton::buttonColourId, juce::Colours::green);

    // Stop button color to be red
    stopButton.setColour(juce::TextButton::buttonColourId, juce::Colours::red);

    // Load button color to be yellow
    loadButton.setColour(juce::TextButton::buttonColourId, juce::Colours::goldenrod);

// Followed the starter code
    // Start timer
    startTimer(200);
}

DeckGUI::~DeckGUI()
{
    // Stop timer when the app closes
    stopTimer();
}
void DeckGUI::paint(juce::Graphics& g)
{
    // Background fill
    g.fillAll(getLookAndFeel().findColour(juce::ResizableWindow::backgroundColourId));
```

```
  // Font color to be white
  g.setColour(juce::Colours::white);
```
// End of following stater code
```
  // Size and bold the font
  g.setFont(juce::Font(15.0f, juce::Font::bold));

  // Show the track title in the deck
  g.drawText(currentTrackTitle, -10, getHeight() - 230, getWidth() - 20, 20,
juce::Justification::centred);
}
void DeckGUI::resized()
{
  // Size of row
  double rowH = getHeight() / 9;

  // Postion of the waveform
  waveformDisplay.setBounds(0, 0, getWidth(), rowH * 2);

  // Position of the volumn slider
  volSlider.setBounds(getWidth() / 3 - 150, rowH * 2 + 15, 30, getHeight() / 4 + 85);

  // Postion of the speed slider
  speedSlider.setBounds(getWidth() / 3 - 110, rowH * 2 + 15, 30, getHeight() / 4 + 85);

  // Postion of the position slider
  posSlider.setBounds(getWidth() / 3 - 70, rowH * 2 + 15, 30, getHeight() / 4 + 85);

  // Postion of the rotating disk
  rotatingDisk.setBounds(getWidth() / 3 - 30, rowH * 2 + 20, 150, 150);

  // Postion of the drum pads
  DrumPad.setBounds(getWidth() / 3 + 115, rowH * 2 + 15, 250, 200);

  // Postion of the play button
  playButton.setBounds(((getWidth() - (getWidth() * 0.6)) / 5) + 40, rowH * 8 - 5,
(getWidth() * 0.6) / 4, rowH);

  // Postion of the stop button
  stopButton.setBounds(((getWidth() - (getWidth() * 0.6)) / 5) + 50 + (getWidth() * 0.6) /
4, rowH * 8 - 5, (getWidth() * 0.6) / 4, rowH);

  // Postion of the load button
  loadButton.setBounds(((getWidth() - (getWidth() * 0.6)) / 5) + 60 + ((getWidth() * 0.6) /
4) * 2, rowH * 8 - 5, (getWidth() * 0.6) / 4, rowH);

  // Postion of the loop button
  loopButton.setBounds(((getWidth() - (getWidth() * 0.6)) / 5) + 70 + ((getWidth() * 0.6) /
4) * 3, rowH * 8 - 5, (getWidth() * 0.6) / 4, rowH);

  // Postion of the volumn slider label
  volSliderLabel.setBounds(volSlider.getX() - 70, volSlider.getY() + volSlider.getHeight() /
2 - 142, 100, 20);

  // Rotate the volumn slider label to be vertical
  volSliderLabel.setTransform(
    juce::AffineTransform::rotation(-juce::MathConstants<float>::halfPi)
    .translated(volSlider.getX() - 15, volSlider.getY() + volSlider.getHeight() / 2 + 20)
  );
  // Make the label center
  volSliderLabel.setJustificationType(juce::Justification::centred);

  // Position of the speed slider label
  speedSliderLabel.setBounds(speedSlider.getX() - 110, speedSlider.getY() +
speedSlider.getHeight() / 2 - 142, 100, 20);

  // Rotate the speed slider label to be vertical
  speedSliderLabel.setTransform(
    juce::AffineTransform::rotation(-juce::MathConstants<float>::halfPi)
    .translated(speedSlider.getX() - 15, speedSlider.getY() + speedSlider.getHeight() / 2 +
20)
  );

  // Make the label center
  speedSliderLabel.setJustificationType(juce::Justification::centred);

  // Postion of the position slider label
  posSliderLabel.setBounds(posSlider.getX() - 148, posSlider.getY() +
posSlider.getHeight() / 2 - 142, 100, 20);

  // Rotate the position slider label to be vertical
  posSliderLabel.setTransform(
    juce::AffineTransform::rotation(-juce::MathConstants<float>::halfPi)
    .translated(posSlider.getX() - 15, posSlider.getY() + posSlider.getHeight() / 2 + 20)
  );

  // Make the label center
  posSliderLabel.setJustificationType(juce::Justification::centred);
}
// Followed the starter code
void DeckGUI::buttonClicked(juce::Button* button)
{
  if (button == &playButton)
  {
    // Console statement when play button is clicked
    DBG("Play Button was clicked");

    // Start playback
    player->start();
// End of following starter code
    // Start the rotation of the disk when the play button is clicked
    rotatingDisk.startRotation();
  }
// Followed the starter code
  if (button == &stopButton)
  {
    // Console statement when stop button is clicked
    DBG("Stop Button was clicked");

    // Stop playback
    player->stop();
// End of following starter code
    // Stop the rotation of disk when the stop button is clicked
    rotatingDisk.stopRotation();
  }
```

```
    if (button == &loopButton)
    {
        // loop state
        looping = !looping;

        // Set looping state in the player
        player->loop(looping);

        if (looping)
        {
            // Console statement for the loop button
            DBG("Loop is on");

            // Set the button color to be blue when the loop is on
            loopButton.setColour(juce::TextButton::buttonColourId, juce::Colours::blue);
        }
        else
        {
            // Console statement for the lopp button
            DBG("Loop is off");

            // Set the button color back to normal when the loop is off
            loopButton.setColour(juce::TextButton::buttonColourId,
juce::Colours::transparentBlack);
        }

        // Update the UI
        repaint();
```

```
  }
// **Followed the starter code**
  if (button == &loadButton)
  {
    // Allow to select files
    auto fileChooserFlags = juce::FileBrowserComponent::canSelectFiles;

    // File chooser to select files
    fChooser.launchAsync(fileChooserFlags, [this](const juce::FileChooser& chooser)
      {
        // Get the file
        auto chosenFile = chooser.getResult();

        // Load the file into the player
        player->loadURL(juce::URL{ chosenFile });

        // Load the file into the waveform display
        waveformDisplay.loadURL(juce::URL{ chosenFile });
       // **End of following starter code**
        // Reset the rotation disk to the starting position
        rotatingDisk.resetRotation();

        // Get the track title
        juce::String trackTitle = chosenFile.getFileNameWithoutExtension();

        // Display the track title
        setCurrentTrackTitle(trackTitle);
```

```
                // Update the UI
                repaint();
            });
    }
}
// Followed the starter code
void DeckGUI::sliderValueChanged(juce::Slider* slider)
{
    if (slider == &volSlider)
    {
        // Adjust the volumn
        player->setGain(slider->getValue());
    }
    else if (slider == &speedSlider)
    {
        // Adjust the speed
        player->setSpeed(slider->getValue());
       // End of following starter code
        // Adjust the speed of the disk
        rotatingDisk.setRotationSpeed(slider->getValue());
    }
// Followed the starter code
    else if (slider == &posSlider)
    {
        // Adjust the postion
        player->setPostionRelative(slider->getValue());
// End of following starter code
        // Adjust the postion of the disk
```

```
      rotatingDisk.setRotationAngle(slider->getValue() *
juce::MathConstants<float>::twoPi);
  }
}
// **Followed the starter code**
// To check if a file is dropped
bool DeckGUI::isInterestedInFileDrag(const juce::StringArray& files)
{
  // Console statment when a file is dragged
  DBG("DeckGUI::isInterestedInFileDrag");
  return true;
}

void DeckGUI::filesDropped(const juce::StringArray& files, int x, int y)
{
  // Only 1 file is dropped
  if (files.size() == 1)
  {
    // Get the dropped file
    juce::File file{ files[0] };
    // If the file is available
    if (file.existsAsFile())
    {
      // Load the track into the deck
      player->loadURL(juce::URL{ file });
      // Update waveform
      waveformDisplay.loadURL(juce::URL{ file });
    }
  }
}
void DeckGUI::timerCallback()
{
  // Get current playback position
  double pos = player->getPositionRelative();

  // Update waveform display position
  if (pos >= 0.0 && pos <= 1.0)
  {
    waveformDisplay.setPositionRelative(pos);
  }
```
// End of following starter code
```
    // Check if the track has ended and stop rotation if loop mode is off
    if (pos >= 1.0)
    {
        // When loop mode is on
        if (looping)
        {
            // Restart from the beginning
            player->setPostionRelative(0.0);

            // Start playback again
            player->start();
        }
        else
        {
            // Stop playback when loop mode is off
            player->stop();

            // Stop disk rotation
            rotatingDisk.stopRotation();
        }
    }
}
void DeckGUI::loadTrack(const juce::String& filePath)
{
    // Load the track into the deck
    player->loadURL(juce::URL{ juce::File{filePath} });

    // Display the waveform
    waveformDisplay.loadURL(juce::URL{ juce::File{filePath} });

    // Reset the disk to the starting position
    rotatingDisk.resetRotation();

    // Update the UI
    repaint();
}
void DeckGUI::setCurrentTrackTitle(const juce::String& title)
{

    // Set the current track title to the title of the file
    currentTrackTitle = title;

    // Update the UI
```

```
  repaint();
}
```

DJAudioPlayer.h

```
// The header file contains the functions for loading, playing, stopping, looping, and
controlling audio tracks.
/*
==============================================================================

  DJAudioPlayer.h

  Created: 28 Jan 2025 4:46:56pm

  Author:  Subathra

==============================================================================
*/
```

// **Followed the starter code**

```
// Include the juce library
#pragma once

#include <JuceHeader.h>

// Include the JUCE class
class DJAudioPlayer : public juce::AudioSource {
  public:
    // Constructor which takes reference to audioformatmanager
    DJAudioPlayer(juce::AudioFormatManager& _formatManager);

    // Cleans up everything when the program closes
    ~DJAudioPlayer();

    // Prepare the audio player before playback
    void prepareToPlay(int samplesPerBlockExpected, double sampleRate) override;

    // Add the next audio block to the audio buffer
    void getNextAudioBlock(const juce::AudioSourceChannelInfo& bufferToFill) override;

    // Release resources
    void releaseResources() override;

    // Load track into the deck using URL
    void loadURL(juce::URL audioURL);

    // Set the gain of the player
    void setGain(double gain);

    // Set the speed of the track
    void setSpeed(double ratio);

    // Set the track position in seconds
    void setPosition(double posInSecs);

    // Set the track position relative
    void setPostionRelative(double pos);

    // Start the track
    void start();

    // Stop the track
    void stop();

    // Get the relative position of the playhead
    double getPositionRelative();
// End of following the starter code
    // Loop the track
    void loop(bool loop);
// Followed the starter code
  private:
    // Reference to audio format manager
    juce::AudioFormatManager& formatManager;

    // Pointer to audio source
    std::unique_ptr<juce::AudioFormatReaderSource> readerSource;

    // Manages audio playback
    juce::AudioTransportSource transportSource;

    // Resampling source for changing speed
    juce::ResamplingAudioSource resampleSource{&transportSource, false, 2};
};
// End of following starter code
```

DJAudioPlayer.cpp

```
// The cpp files containus the implementation for loading, playing, stopping, looping,
and controlling audio tracks.
/*
==============================================================================
  DJAudioPlayer.cpp
  Created: 28 Jan 2025 4:46:56pm
  Author:  Subathra
==============================================================================
*/
```
// **Followed the starter code**
```
// Include the header file
#include "DJAudioPlayer.h"


// Constructor for the class
DJAudioPlayer::DJAudioPlayer(juce::AudioFormatManager& _formatManager)
  : formatManager(_formatManager)
{
}
// Destructor
DJAudioPlayer::~DJAudioPlayer()
{
}


void DJAudioPlayer::prepareToPlay(int samplesPerBlockExpected, double sampleRate)
{
  // Transport source for track
  transportSource.prepareToPlay(samplesPerBlockExpected, sampleRate);


  // Resample source for track
  resampleSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
}
void DJAudioPlayer::getNextAudioBlock(const juce::AudioSourceChannelInfo&
bufferToFill)
```

```
{
   // Fill the buffer with the following audio block from the resample source
   resampleSource.getNextAudioBlock(bufferToFill);
}
void DJAudioPlayer::releaseResources()
{
   // Release transport source resources
   transportSource.releaseResources();

   // Release resample source resources
   resampleSource.releaseResources();
}

void DJAudioPlayer::loadURL(juce::URL audioURL)
{
   // Reader for the given URL
   auto* reader = formatManager.createReaderFor(audioURL.createInputStream(false));

   // Check if the file was loaded successfully
   if (reader != nullptr)
   {
      // Create new audioformatereadersource from the reader
      std::unique_ptr<juce::AudioFormatReaderSource> newSource(new
juce::AudioFormatReaderSource(reader, true));

      // Set new source for transportsource with the reader sample rate
      transportSource.setSource(newSource.get(), 0, nullptr, reader->sampleRate);
```

```
        // reset readersource to take ownership of the new source
        readerSource.reset(newSource.release());
    }
}

void DJAudioPlayer::setGain(double gain)
{
    // Check if the gain value is within the range
    if (gain < 0 || gain > 1.0)
    {
        // Console statement if the range is out
        DBG("DJAudioPlayer::setGain gain should be between 0 and 1");
    }
    else {
        // Set the gain for the transport source
        transportSource.setGain(gain);
    }
}
void DJAudioPlayer::setSpeed(double ratio)
{
    // Ensure the ratio is within a safe range
    if (ratio < 0.1 || ratio > 100.0)
    {
        DBG("DJAudioPlayer::setSpeed ratio should be between 0.1 and 100");
        return; // Exit the function to prevent crashing
    }

    // Set resample ratio for track speed adjustment
    resampleSource.setResamplingRatio(ratio);
}
void DJAudioPlayer::setPosition(double posInSecs)
{
    // Move the playhead to the specific position
    transportSource.setPosition(posInSecs);
}

void DJAudioPlayer::setPostionRelative(double pos)
{
    // Convert position value into absoulute time in seconds
    double posInSecs = transportSource.getLengthInSeconds() * pos;

    // set the playback position to the calculated absolute time
    setPosition(posInSecs);
}

void DJAudioPlayer::start()
{
    // Start track from the current position
    transportSource.start();
}
void DJAudioPlayer::stop()
{
    // stop track
    transportSource.stop();
}
```

// End of following the starter code

```
void DJAudioPlayer::loop(bool loop)
{
   // loop state in the transport source
   transportSource.setLooping(loop);


   // Restart the track if the loop is on
   if (loop && getPositionRelative() >= 1.0)
   {
      // Restart from beginning
      setPostionRelative(0.0);


      // Start the track
      start();
   }
}
```

// **Followed the starter code**

```
double DJAudioPlayer::getPositionRelative()
{
   // return the current position as a fraction of the total track length
   return transportSource.getCurrentPosition() / transportSource.getLengthInSeconds();
}
```

// End of following the starter code

WaveformDisplay.h

// The header file contains the functions which display the waveform of the audio file, manage the loading of audio files, and updating the display based on the playhead position.

```
/*
==============================================================================
  WaveformDisplay.h

  Created: 30 Jan 2025 1:31:02pm

  Author:  Subathra

==============================================================================

*/

// Followed the starter code

// Include all the libraries and the header files

#pragma once

#include <JuceHeader.h>

#include "DJAudioPlayer.h"

//==============================================================================

/*

*/

// Include all JUCE classes

class WaveformDisplay  : public juce::Component,

                  public juce::ChangeListener

{

public:

  // Constructor

  WaveformDisplay(juce::AudioFormatManager& formatManagerToUse,

            juce::AudioThumbnailCache& cacheToUse,

            juce::Colour color);


  // Cleans up everything when the program closes

  ~WaveformDisplay() override;


  // Paint function to draw on the screen

  void paint (juce::Graphics&) override;
  // To resize the components on the screen
  void resized() override;

  // Called when audio player state changes
  void changeListenerCallback(juce::ChangeBroadcaster* source) override;

  // Loads the audio file from a URL
  void loadURL(juce::URL audioURL);

  // Set the position of the playhead in the waveform display
  void setPositionRelative(double pos);

private:
  // Audiothumbnail object to render waveform
  juce::AudioThumbnail audioThumb;
// End of following the stater code
  // Color of the waveform
  juce::Colour waveformColour;
// Followed the starter code
  // Variable to check if an audio file is loaded
  bool fileLoaded;

  // Variable to store current position of the playhead
  double position;

  // Prevents copying this class to avoid memory issues
  JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (WaveformDisplay)
```

```
};
```

// End of following the starter code

WaveformDisplay.cpp

// The header file contains the implementations which display the waveform of the audio file, manage the loading of audio files, and updating the display based on the playhead position.

```
/*
==============================================================================

  WaveformDisplay.cpp

  Created: 30 Jan 2025 1:31:02pm

  Author:  Subathra

==============================================================================

*/


// Include all the libraries and the header file

#include <JuceHeader.h>

#include "WaveformDisplay.h"


//==============================================================================

// Constructor to set up the WaveformDisplay object's color, format manager, and cache

WaveformDisplay::WaveformDisplay(juce::AudioFormatManager& formatManagerToUse,

  juce::AudioThumbnailCache& cacheToUse, juce::Colour color) :

  audioThumb(1000, formatManagerToUse, cacheToUse),

  waveformColour(color),

  fileLoaded(false),

  position(0)

{
```

```
// **Followed the starter code**
   // Add as a listener to audiothumbnail changes
   audioThumb.addChangeListener(this);
}


// Destructor
WaveformDisplay::~WaveformDisplay()
{
}


void WaveformDisplay::paint(juce::Graphics& g)
{
   // Background fill
   g.fillAll(getLookAndFeel().findColour(juce::ResizableWindow::backgroundColourId));


   // Set boarder color to be grey
   g.setColour(juce::Colours::grey);
// End of following starter code
   // Draw a border
   g.drawRect(getLocalBounds(), 2);


   // Set the waveform color
   g.setColour(waveformColour);
// Followed the starter code
   // If file is loaded
   if (fileLoaded)
   {
      // Draw the waveform using the audiothumbnail object
```

```
        audioThumb.drawChannel(g,
          getLocalBounds(),
          0,
          audioThumb.getTotalLength(),
          0,
          1.0f
        );

        // Set playhead color to be light green
        g.setColour(juce::Colours::lightgreen);
// End of following the starter code
        // Draw a very thin rectangle for the playhead
        g.drawRect(position * getWidth(), 0, getWidth() / 200, getHeight());

        // fill the playhead
        g.fillRect(position * getWidth(), 0, getWidth() / 200, getHeight());
    }
// Followed the starter code
    else {
        // Font size to 20
        g.setFont(juce::FontOptions(20.0f));

        // Message to display in the center
        g.drawText("File not loaded...", getLocalBounds(), juce::Justification::centred, true);
    }
}

void WaveformDisplay::resized()
```

```
{
}

void WaveformDisplay::loadURL(juce::URL audioURL)
{
   // Clear existing waveform data
   audioThumb.clear();

   // Set the source to new URL
   fileLoaded = audioThumb.setSource(new juce::URLInputSource(audioURL));
}
void WaveformDisplay::changeListenerCallback(juce::ChangeBroadcaster* source)
{
   // Update the UI when changes happens
   repaint();
}

void WaveformDisplay::setPositionRelative(double pos)
{
   // If the position change
   if (pos != position)
   {
      // update the position of the playhead
      position = pos;

      // Update the UI
      repaint();
   }
```

}

// End of following starter code

PlaylistComponent.h

// The header file contains the functions which manages a playlist UI with functionalities such as displaying tracks, filtering, importing music, and interacting with buttons for loading tracks to decks.

```
/*
==============================================================================

  PlaylistComponent.h

  Created: 30 Jan 2025 7:33:53pm

  Author:  Subathra

==============================================================================

*/
```

// **Followed the starter code**

```
// Include all the libraries

#pragma once


#include <JuceHeader.h>

#include <vector>

#include <string>

//==============================================================================

/*

*/

// Include all JUCE classes

class PlaylistComponent  : public juce::Component,

                 public juce::TableListBoxModel,

                 public juce::Button::Listener,

                 public juce::TextEditor::Listener
```

```
{
public:
   // Constructor
   PlaylistComponent();

   // Destructor
   ~PlaylistComponent() override;

   // Paint function to draw on the screen
   void paint (juce::Graphics&) override;

   // To resize the components on the screen
   void resized() override;

   // The number of rows in the table
   int getNumRows() override;

   // Paint the background of each row
   void paintRowBackground(juce::Graphics& g, int rowNumber, int width, int height,
bool rowIsSelected) override;
   // Paint each cell in the table
   void paintCell(juce::Graphics& g, int rowNumber, int columnId, int width, int height,
bool rowIsSelected) override;
   // Refresh a custome component inside a table cell
   juce::Component* refreshComponentForCell(int        rowNumber, int columnId, bool
isRowSelected, Component* existingComponentToUpdate) override;
   // Called when a button is clicked
   void buttonClicked(juce::Button* button) override;
```
// End of following the starter code

```
    // Function for the search box

    void textEditorTextChanged(juce::TextEditor&) override;


    // Callback function for loading track to Deck 1

    std::function<void(const juce::String&)> onLoadToDeck1;


    // Callback function for loading track to Deck 2

    std::function<void(const juce::String&)> onLoadToDeck2;

private:

    // Import music into the playlist table

    void importMusic();


    // Text box for the searching of tracks

    juce::TextEditor searchBox;


    // Store the title of tracks after user filter

    std::vector<std::string> userFilteredTrackTitles;

    // Store the duration of the track

    std::vector<std::string> trackDurations;


    // Store file paths for all tracks

    std::vector<std::string> trackFilePaths;
```

// **Followed the starter code**

```
    // Table which shows the playlist of tracks

    juce::TableListBox tableComponent;


    // Store the titles of all tracks

    std::vector<std::string> trackTitles;
```

// End of following the starter code

// Import button

juce::TextButton importButton{ "IMPORT MUSIC" };

// File choose dialog to select files

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

// **Followed the starter code**

// Prevents copying this class to avoid memory issues

JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (PlaylistComponent)

};

// End of following the starter code

PlaylistComponent.cpp

// The cpp file contains the implementations which manages a playlist UI with functionalities such as displaying tracks, filtering, importing music, and interacting with buttons for loading tracks to decks.

/*

==

PlaylistComponent.cpp

Created: 30 Jan 2025 7:33:53pm

Author: Subathra

==

*/

// **Followed the starter code**

// Include all the libraries and the header file

#include <JuceHeader.h>

#include "PlaylistComponent.h"

#include <algorithm>

```
//==============================================================================
// Constructor
PlaylistComponent::PlaylistComponent()
{
   // Add title column in the table
   tableComponent.getHeader().addColumn("Track title", 1, 548);
```
// End of following the starter code
```
   // Add duration column in the table
   tableComponent.getHeader().addColumn("Duration(hhmmss)", 2, 170);

   // Add deck 1 load column in the table
   tableComponent.getHeader().addColumn("Load to", 3, 150);

   // Add deck 2 load column in the table
   tableComponent.getHeader().addColumn("Load to", 4, 150);

   // Add delete column in the table
   tableComponent.getHeader().addColumn("Delete", 5, 60);
   // Set the row height
   tableComponent.setRowHeight(40);
```
// **Followed the starter code**
```
   // Set the model for the table
   tableComponent.setModel(this);

   // Show the table on the screen
   addAndMakeVisible(tableComponent);
```
// End of following the starter code
```
   // Set up Import Music Button
```

```
importButton.setButtonText("Import Music");

// Listener for the import button
importButton.addListener(this);

// Show the import button on the screen
addAndMakeVisible(importButton);

// Show the search box on the screen
addAndMakeVisible(searchBox);

// Listener for the search box
searchBox.addListener(this);

// Placeholder text
searchBox.setTextToShowWhenEmpty(juce::String{ "Search" }, juce::Colours::black);

// Set text color to black
searchBox.setColour(juce::TextEditor::textColourId, juce::Colours::black);

// Background fill for the search box
searchBox.setColour(juce::TextEditor::backgroundColourId, juce::Colours::lightblue);

// Font style for the search box
searchBox.setFont(juce::Font(15.0f, juce::Font::bold));

// text justification in search box
searchBox.setJustification(juce::Justification::centredLeft);
```

```
}


// Destructor
PlaylistComponent::~PlaylistComponent()
{
  // Remove model from the table when the table is destroyed
  tableComponent.setModel(nullptr);
}
```
// **Followed the starter code**
```
void PlaylistComponent::paint (juce::Graphics& g)
{
  // Background fill
  g.fillAll (getLookAndFeel().findColour (juce::ResizableWindow::backgroundColourId));
// clear the background


  // Set color to grey for the outer boarder line
  g.setColour (juce::Colours::grey);


  // draw an outline around the component
  g.drawRect (getLocalBounds(), 1);
}
```
// End of following starter code
```
int PlaylistComponent::getNumRows()
{
  // Returns the number of rows in the table
  if (userFilteredTrackTitles.empty())
  {
    return trackTitles.size();
  }
  else
  {
    return userFilteredTrackTitles.size();
  }
}
void PlaylistComponent::paintRowBackground(juce::Graphics& g, int  rowNumber, int
        width, int        height, bool rowIsSelected)
{
  // If row is selected
  if (rowIsSelected)
  {
    // Fill the row with lightblue
    g.fillAll(juce::Colours::lightblue);
  }
  else {
    // Else fill the row with white
    g.fillAll(juce::Colours::white);
  }
}

void PlaylistComponent::paintCell(juce::Graphics& g, int rowNumber, int columnId, int
width, int height, bool rowIsSelected)
{
  // To store the list of track titles that should be used
  const std::vector<std::string>* titlesToUse;

  // Decide whether to use the full track list or the filtered list
  if (userFilteredTrackTitles.empty())
```

```
    {
        // Use the full track list if no filtering is applied
        titlesToUse = &trackTitles;
    }
    else
    {
        // Use filtered list if search is applied
        titlesToUse = &userFilteredTrackTitles;
    }

    // Track duration column
    if (columnId == 2)
    {
        // Draw track duration in the corresponding cell
        g.drawText(trackDurations[rowNumber], 2, 0, width - 4, height,
juce::Justification::centredLeft, true);
    }
    // Track title column
    else
    {
        // Draw the track title in the corresponding cell
        g.drawText((*titlesToUse)[rowNumber], 2, 0, width - 4, height,
juce::Justification::centredLeft, true);
    }
}

// Create and update components for the buttons in the table
juce::Component* PlaylistComponent::refreshComponentForCell(int rowNumber, int
columnId, bool isRowSelected, Component* existingComponentToUpdate)
```

```
{
   if (columnId == 3 || columnId == 4 || columnId == 5)
   {
      if (existingComponentToUpdate == nullptr)
      {
         // If it is the delete column
         if (columnId == 5)
         {
            // Drawable Button to put the image in the button
            auto* imgButton = new juce::DrawableButton("DeleteButton",
juce::DrawableButton::ImageOnButtonBackground);

            // Start from the current working directory
            juce::File currentDir = juce::File::getCurrentWorkingDirectory();

            // Search for the "samples" folder by moving up the directory tree
            while (currentDir.exists() && !currentDir.getChildFile("samples").exists())
            {
               currentDir = currentDir.getParentDirectory();
            }

            // If "samples" is found, construct the full path to "delete_icon.png"
            juce::File deleteIconFile;
            if (currentDir.getChildFile("samples").exists())
            {
               deleteIconFile =
currentDir.getChildFile("samples").getChildFile("delete_icon.png");
            }
```

```
        // Log the directory for debugging
        juce::Logger::writeToLog("Looking for delete icon at: " +
deleteIconFile.getFullPathName());

        // Ensure the file exists
        if (!deleteIconFile.existsAsFile())
        {

          // Console statement when the icon is not found
          juce::Logger::writeToLog("Error: Delete icon not found at " +
deleteIconFile.getFullPathName());

          // Return null is the file does not exists
          return nullptr;
        }

        // Load the image
        juce::Image deleteImage = juce::ImageFileFormat::loadFrom(deleteIconFile);

        // If the image is valid
        if (deleteImage.isValid())
        {
          auto drawableImage = std::make_unique<juce::DrawableImage>();
          drawableImage->setImage(deleteImage);

          // Set the image in the button
          imgButton->setImages(drawableImage.get());
        }
        // Set component id to row column format
        imgButton->setComponentID(juce::String(rowNumber) + "-" +
juce::String(columnId));

        // Listerner for the button
        imgButton->addListener(this);

        // Update the existing component
        existingComponentToUpdate = imgButton;
      }
      else
      {
        // Pointer to a TextButton
        juce::TextButton* btn;

        if (columnId == 3)
        {
          // Button for deck 1
          btn = new juce::TextButton{ "Deck 1" };
        }
        else
        {
          // Button for deck 2
          btn = new juce::TextButton{ "Deck 2" };
        }

        // unique ID to the button using row and column info
        btn->setComponentID(juce::String(rowNumber) + "-" + juce::String(columnId));
```

```
            //  Listener for the button
            btn->addListener(this);

            // Update the existing component
            existingComponentToUpdate = btn;
          }

        }
        // Update the existing component
        return existingComponentToUpdate;
    }
    // Return null if no component is needed
    return nullptr;
}
void PlaylistComponent::resized()
{
    // Size of the search box
    searchBox.setBounds(10, 10, getWidth() - 2 * 10, 30);

    // Size of the tablecomponent
    tableComponent.setBounds(10, 45, getWidth() - 2 * 10, getHeight() - 30 - 65);

    // Size of the import button
    importButton.setBounds(10, getHeight() - 45, getWidth() - 10 * 2, 40);
}

void PlaylistComponent::buttonClicked(juce::Button* button)
```

```
{
    // If import button is clicked
    if (button == &importButton)
    {
        // Call the function
        importMusic();
        return;
    }

    // Get the ID of the button
    juce::String buttonID = button->getComponentID();
    juce::StringArray tokens;

    // Split the component ID into row and column
    tokens.addTokens(buttonID, "-", "");

    // Ensure there are 2 parts
    if (tokens.size() != 2) return;

    // Extract the row number
    int row = tokens[0].getIntValue();

    // Extract the column ID
    int columnId = tokens[1].getIntValue();

    // A non-const reference to store the track list
    const std::vector<std::string>* titlesToUsePtr = &userFilteredTrackTitles;
```

```
// If the filtered list is empty, use the full track list instead
if (titlesToUsePtr->empty())
{
   titlesToUsePtr = &trackTitles;
}

// Use *titlesToUsePtr to access the selected track list
const auto& titlesToUse = *titlesToUsePtr;

// Ensure valid row
if (row < 0 || row >= titlesToUse.size()) return;

// Find the actual index in trackTitles
auto it = std::find(trackTitles.begin(), trackTitles.end(), titlesToUse[row]);

// Ensure the track exists
if (it == trackTitles.end()) return;

// Get original index
int actualIndex = std::distance(trackTitles.begin(), it);

// If deck 1 button is clicked
if (columnId == 3)
{
   // Call deck 1 function
   if (onLoadToDeck1) onLoadToDeck1(trackTitles[actualIndex]);
}
```

```
    // If deck 2 button is clicked
    else if (columnId == 4)
    {
        // Call deck 2 function
        if (onLoadToDeck2) onLoadToDeck2(trackTitles[actualIndex]);
    }

    // If delete button is clicked
    else if (columnId == 5)
    {
        // Remove track from original list
        trackTitles.erase(trackTitles.begin() + actualIndex);

        // Remove duration from the original list
        trackDurations.erase(trackDurations.begin() + actualIndex);

        // If filtering is active, also remove from filtered list
        if (!userFilteredTrackTitles.empty())
        {
            userFilteredTrackTitles.erase(userFilteredTrackTitles.begin() + row);
        }

        // Refresh UI
        tableComponent.updateContent();
    }
}

void PlaylistComponent::importMusic()
```

```
{
  // Allow file selection
  auto fileChooserFlags = juce::FileBrowserComponent::canSelectFiles |
juce::FileBrowserComponent::canSelectMultipleItems;

  // Launch file chooser
  fChooser.launchAsync(fileChooserFlags, [this](const juce::FileChooser& chooser)
    {
      // Get the selected files
      auto chosenFiles = chooser.getResults();

      // Loop through each file
      for (const auto& file : chosenFiles)
      {
        // Get track name without the extension
        std::string trackName = file.getFileNameWithoutExtension().toStdString();

        // Check if the track is already in the playlist
        if (std::find(trackTitles.begin(), trackTitles.end(), trackName) != trackTitles.end())
        {
          // Alert message is there are duplicate tracks in the playlist
          juce::AlertWindow::showMessageBoxAsync(
            juce::AlertWindow::WarningIcon,
            "                    Duplicate Track",
            "      The track '" + trackName + "' is already in the playlist.",
            "OK"
          );
          continue;
        }

        // Track folder path
        // Track folder path
        juce::File tracksFolder =
juce::File::getSpecialLocation(juce::File::currentExecutableFile)
                    .getParentDirectory()
                    .getParentDirectory()
                    .getParentDirectory()
                    .getParentDirectory()
                    .getParentDirectory()
                    .getParentDirectory()
                    .getChildFile("tracks");

        // Ensure the tracks folder exists
        if (!tracksFolder.exists())
          tracksFolder.createDirectory();

        // Create the destination file path inside the tracks folder
        juce::File destinationFile = tracksFolder.getChildFile(file.getFileName());

        //  Check is the file exists
        if (!destinationFile.existsAsFile())
        {
          // Copy the file to the tracks folder
          bool success = file.copyFileTo(destinationFile);

          if (!success)
```

```
            {
                // Console statement if the copying does not work
                juce::Logger::writeToLog("Failed to copy track to tracks folder: " +
destinationFile.getFullPathName());
                continue;
            }
        }

        // Console statement for the track being copied successfully
        juce::Logger::writeToLog("Track copied to: " +
destinationFile.getFullPathName());

        // Set up formatmanager to prepare for audio track
        juce::AudioFormatManager formatManager;

        // Register audio formats
        formatManager.registerBasicFormats();

        // Create audio reader for the copied file
        juce::AudioFormatReader* reader =
formatManager.createReaderFor(destinationFile);
        if (reader != nullptr)
        {
            // Check duration of the tracks in seconds
            double InSec = static_cast<double>(reader->lengthInSamples) / reader-
>sampleRate;

            // Convert seconds into min and sec format
            int min = static_cast<int>(InSec) / 60;
```

```
                int sec = static_cast<int>(InSec) % 60;

                // Duration as a string
                std::string formattedDuration = juce::String::formatted("%02d:%02d", min,
sec).toStdString();

                // Store track details
                trackTitles.push_back(trackName);
                trackDurations.push_back(formattedDuration);
                trackFilePaths.push_back(destinationFile.getFullPathName().toStdString()); //
Store new path

                // Free up memory
                delete reader;
            }
        }
        // Update content
        tableComponent.updateContent();
    });
}

void PlaylistComponent::textEditorTextChanged(juce::TextEditor&)
{
    // Search text entered by the user
    std::string searchTerm = searchBox.getText().toStdString();

    // Clear the previous search
    userFilteredTrackTitles.clear();
    // If search box is empty
    if (searchTerm.empty())
    {
        // Update content
        tableComponent.updateContent();
        return;
    }

    // Perform case-insensitive search
    for (const auto& song : trackTitles)
    {
        //  Convert search term and song names to lowercase
        std::string lowerSong = song;
        std::string lowerSearchTerm = searchTerm;

        std::transform(lowerSong.begin(), lowerSong.end(), lowerSong.begin(), ::tolower);
        std::transform(lowerSearchTerm.begin(), lowerSearchTerm.end(),
lowerSearchTerm.begin(), ::tolower);

        // Check if the song name contains the search term
        if (lowerSong.find(lowerSearchTerm) != std::string::npos)
        {
            // If there is a match, add it to the filtered list
            userFilteredTrackTitles.push_back(song);
        }
    }

    // Update content
```

```
  tableComponent.updateContent();
}
```

RotatingDisk.h

// The header file contains the functions which handles the rotating disk, to start, stop, reset rotation, adjust speed, and customize the rotation angle.

```
/*
==============================================================================

  RotatingDisk.h
  Created: 22 Feb 2025 4:09:23pm
  Author:  Subathra

==============================================================================
*/
// Include the juce library
#pragma once

#include <JuceHeader.h>

// Include all JUCE classes
class RotatingDisk : public juce::Component, public juce::Timer
{
public:
  // Constructor
  RotatingDisk();

  // Paint function to draw on the screen
  void paint(juce::Graphics& g) override;

  // To resize the components on the screen
```

```
    void resized() override;

    // Start the rotation of the disk
    void startRotation();

    // Stop the rotation of the disk
    void stopRotation();

    // Reset the rotatation of the disk
    void resetRotation();

    // Timer callback to update the rotation
    void timerCallback() override;

    // Adjust speed dynamically
    void setRotationSpeed(double speed);

    // Adjust the rotation angle
    void setRotationAngle(float newAngle);


private:
    // Variable to check if the disk is rotating
    bool isRotating;

    // Current rotation angle
    double rotationAngle = 0.0;
    // Current rotation speed
    double rotationSpeed = 0.0;

    // Variable to store the glow effect for the disk
    float glowPhase;

    // Prevents copying this class to avoid memory issues
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(RotatingDisk)
};
```

RotatingDisk.cpp

// The cpp file contains the implementation which handles the rotating disk, to start, stop, reset rotation, adjust speed, and customize the rotation angle.

```
/*
==============================================================================

    RotatingDisk.cpp
    Created: 22 Feb 2025 4:09:23pm
    Author:  Subathra

==============================================================================
*/
// Include all the libraries and the header file
#include <JuceHeader.h>
#include "RotatingDisk.h"

// Constructor to set up the rotating disk
RotatingDisk::RotatingDisk() : rotationAngle(0.0f), isRotating(false), rotationSpeed(0.1f),
glowPhase(0.0f)
{
    // set Disk size
    setSize(100, 100);
}

void RotatingDisk::paint(juce::Graphics& g)
{
    // Calculate the radius of the rotating circle
    float radius = (getWidth() - 20) / 2;

    // In the red ellipse, set the steps for drawing fine circles.
    float redRadiusStep = 20.0f / 5;

    // Dynamic RGB colors based on a sine wave for smooth cycling
    float r = (std::sin(glowPhase) + 1.0f) * 127.5f;
    float gVal = (std::sin(glowPhase + juce::MathConstants<float>::pi / 3) + 1.0f) * 127.5f;
    float b = (std::sin(glowPhase + juce::MathConstants<float>::pi * 2 / 3) + 1.0f) * 127.5f;

    // X position of the tip
    float tipX = getWidth() / 2 + radius * std::cos(rotationAngle);

    // Y position of the tip
    float tipY = getHeight() / 2 + radius * std::sin(rotationAngle);

    // Combine the RGB into a color for the dynamic glow
    juce::Colour dynamicGlow = juce::Colour((uint8_t)r, (uint8_t)gVal, (uint8_t)b);

    // Create a neon radial gradient
    juce::ColourGradient glowGradient(dynamicGlow, getWidth() / 2, getHeight() / 2,
        juce::Colours::transparentBlack, getWidth(), getHeight(), true);
    // Set the gradient as the fill color
    g.setGradientFill(glowGradient);

    // Fill the disk area with the gradient
    g.fillEllipse(0, 0, getWidth(), getHeight());

    // Draw a black ellipse on top of the gradient ellipse
    g.setColour(juce::Colours::black);
    g.fillEllipse(10, 10, getWidth() - 20, getHeight() - 20);
    g.drawEllipse(10, 10, getWidth() - 20, getHeight() - 20, 2.0f);

    // Grey color for the fine circles
    g.setColour(juce::Colour(150, 150, 150));

    // Draw fine circles for the black ellipse
    for (float r = (((getWidth() - 20) / 2) / 10); r < (getWidth() - 20) / 2; r += (((getWidth() - 20) /
  2) / 10))
    {
        g.drawEllipse(getWidth() / 2 - r, getHeight() / 2 - r, r * 2, r * 2, 0.2f);
    }

    // Draw a small red ellipse in the center
    g.setColour(juce::Colours::red);
    g.fillEllipse(getWidth() / 2 - 20.0f, getHeight() / 2 - 20.0f, 20.0f * 2, 20.0f * 2);

    // Draw fine circles for the red ellipse
    for (float r = redRadiusStep; r < 20.0f; r += redRadiusStep)
    {
        g.drawEllipse(getWidth() / 2 - r, getHeight() / 2 - r, r * 2, r * 2, 0.5f); // Thinner strokes
    }

    // Set the color of the indicator to be grey
    g.setColour(juce::Colour(150, 150, 150));

    // Make a path for the indicator line that rotates.
    juce::Path indicator;

    // Start at the center
    indicator.startNewSubPath(getWidth() / 2, getHeight() / 2);

    // Draw line to the rotating tip
    indicator.lineTo(getWidth() / 2 + radius * std::cos(rotationAngle), getHeight() / 2 +
radius * std::sin(rotationAngle));

    // Draw the stroke path
    g.strokePath(indicator, juce::PathStrokeType(2.0f));

    // Color of the tip ellipse
    g.setColour(juce::Colours::white);

    // Draw the tip ellipse
    g.fillEllipse(tipX - 4.0f, tipY - 4.0f, 4.0f * 2, 4.0f * 2);
}
void RotatingDisk::resized()
{
    // Update the UI

    repaint();

}


void RotatingDisk::startRotation()

{

    // Set isRotating to true to start rotation

    isRotating = true;


    // Start timer

    startTimer(50);


    if (rotationSpeed == 0.0f) {

        // Set a default speed

        rotationSpeed = 0.1f;

    }

}


void RotatingDisk::stopRotation()

{

    // Set isRotating to false to stop rotation

    isRotating = false;


    // Stop timer

    stopTimer();

}


void RotatingDisk::resetRotation()
```

```
{
   // Stop rotation
   stopRotation();

   // Reset the angle to be 0
   rotationAngle = 0.0;

   // Update the UI
   repaint();
}

void RotatingDisk::timerCallback()
{
   // If the disk is rotating
   if (isRotating)
   {
      // Increase the rotation angly by rotation speed
      rotationAngle += rotationSpeed;

      // if the angle exceeds 360 degree
      if (rotationAngle >= juce::MathConstants<float>::twoPi)

         // Set the angle back to 0
         rotationAngle -= juce::MathConstants<float>::twoPi;

      // Update the UI
      repaint();
   }
```

```
    // Increase the glow phase for color cycle
    glowPhase += 0.1f;

    // If glow phase exceeds twopi
    if (glowPhase > juce::MathConstants<float>::twoPi)

        // Set the glow phase back to 0
        glowPhase -= juce::MathConstants<float>::twoPi;

    // Updates the UI
    repaint();
}

void RotatingDisk::setRotationSpeed(double speed)
{
    // Scale the speed factor
    rotationSpeed = speed * 0.1f;
}

void RotatingDisk::setRotationAngle(float newAngle)
{
    // New rotation angle
    rotationAngle = newAngle;

    // Updates the UI
    repaint();
}
```

DrumPadComponent.h

// The header file contains the functions of the management of the drum pads with audio playback, volume control, and looping functionality.

```
/*
==============================================================================

  DrumPadComponent.h

  Created: 24 Feb 2025 11:07:32pm

  Author:  Subathra

==============================================================================

*/

// Include the juce library

#pragma once


#include <JuceHeader.h>


// Include all the JUCE classes

class DrumPadComponent : public juce::Component,

  public juce::Button::Listener,

  public juce::Slider::Listener,

  public juce::Timer,

  public juce::ChangeListener

{

public:

  // Constructor

  DrumPadComponent();


  // Destructor
```

```
    ~DrumPadComponent() override;

    // Paint function to draw on the screen
    void paint(juce::Graphics&) override;

    // To resize the components on the screen
    void resized() override;

    // Called when a button is clicked
    void buttonClicked(juce::Button* button) override;

    // Called when slider value changes
    void sliderValueChanged(juce::Slider* slider) override;

    // Update UI at regular intervals
    void timerCallback() override;

    // Handles audio changes
    void changeListenerCallback(juce::ChangeBroadcaster* source) override;

private:

    // Load audio file
    void loadSample(int padIndex, const juce::String& fileName);

    // Stop the blinking effect
    void stopBlinking(int padIndex);
```

```
// Keeps track of the blinking state

bool isBlinking[6] = { false };


// Controls blinking effect

bool blinkState = false;


// Keeps track of loop state

bool isLooping[6] = { false };


// Butons for drun pads

juce::TextButton drumPads[6];


// Volumn sliders for each drum pad

juce::Slider volumeSliders[6];


// Toggle buttons for looping

juce::ToggleButton loopButtons[6];


// Manages audio file formats

juce::AudioFormatManager formatManager;


// Handles audio track

juce::AudioTransportSource transportSources[6];


// Store audio sources

std::unique_ptr<juce::AudioFormatReaderSource> readerSources[6];


// Manages audio devices
```

```
    juce::AudioDeviceManager deviceManager;


    // Handles audio track for eaach drum pad
    juce::AudioSourcePlayer audioSourcePlayer[6];


    // Default color of the drum pad
    juce::Colour defaultPadColor = juce::Colours::black;


    // Color for each drum pad
    juce::Colour padColors[6] = { juce::Colours::red, juce::Colours::blue,
juce::Colours::green,
                                  juce::Colours::orange, juce::Colours::purple, juce::Colours::yellow };


    // Prevents copying this class to avoid memory issues
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(DrumPadComponent)
};
```

DrumPadComponent.cpp

```
// The cpp file contains the implrmentation of the management of the drum pads with
audio playback, volume control, and looping functionality.
/*

======================================================================
========


    DrumPadComponent.cpp
    Created: 24 Feb 2025 11:07:32pm
    Author:  Subathra
```

```
==============================================================================
*/

// Include the header file
#include "DrumPadComponent.h"

// Constructor to initializes drum pads, volume sliders, loop buttons, and audio
DrumPadComponent::DrumPadComponent()
{
  // Register audio formats
  formatManager.registerBasicFormats();

  // Initialize audio device with 2 output channels
  deviceManager.initialiseWithDefaultDevices(0, 2);

  // Loop through each drum pad
  for (int i = 0; i < 6; ++i)
  {
    // Listener for the drum pads
    drumPads[i].addListener(this);

    // Default color for the drum pad
    drumPads[i].setColour(juce::TextButton::buttonColourId, defaultPadColor);

    // Show the drum pad on the screen
    addAndMakeVisible(drumPads[i]);
```

```
// Volumn slider range

volumeSliders[i].setRange(0.0, 1.0, 0.05);


// Set the starting value of the volume slider to be 80%

volumeSliders[i].setValue(0.8);


// Listener for the slider

volumeSliders[i].addListener(this);


// Style for the volumne to be increase decrease buttons

volumeSliders[i].setSliderStyle(juce::Slider::SliderStyle::IncDecButtons);


// Show the volumn slider on the screen

addAndMakeVisible(volumeSliders[i]);


// Text for the button

loopButtons[i].setButtonText("Loop");


// Set clicking toggle state

loopButtons[i].setClickingTogglesState(true);


// Listener for the loop button

loopButtons[i].addListener(this);


// Show the loop buttons on the screen

addAndMakeVisible(loopButtons[i]);


// Load the tracks for the drum pads
```

```
        loadSample(i, "sample" + juce::String(i + 1) + ".mp3");

        // Add audio call back
        deviceManager.addAudioCallback(&audioSourcePlayer[i]);

        // Listener for audio state changes
        transportSources[i].addChangeListener(this);
    }

    // Start timer
    startTimer(500);
}

DrumPadComponent::~DrumPadComponent()
{
    for (int i = 0; i < 6; ++i)
    {
        // Remove audio source
        transportSources[i].setSource(nullptr);

        // Remove change listener
        transportSources[i].removeChangeListener(this);

        // Remove audio callbacks
        deviceManager.removeAudioCallback(&audioSourcePlayer[i]);
    }
    // Stop the blinking effect
    stopTimer();
```

```
}

void DrumPadComponent::paint(juce::Graphics& g)
{

}

void DrumPadComponent::resized()
{
  for (int i = 0; i < 6; ++i)
  {
    // x-axis
    int x = 10 + (i % 3) * 80;

    // y-axis
    int y = 10 + (i / 3) * 80;

    // position of the drum pads
    drumPads[i].setBounds(x, y, 50, 50);

    // position of the volumn sliders to be beside the drum pads
    volumeSliders[i].setBounds(x + 50, y + 5, 30, 40);

    // position of the loop button below the drum pad
    loopButtons[i].setBounds(x, y + 55, 50, 15);
  }
}
```

```
void DrumPadComponent::buttonClicked(juce::Button* button)
{
  // Loop through all the drum pads
  for (int i = 0; i < 6; ++i)
  {
    // Check is the button is clicked
    if (button == &drumPads[i])
    {
      // If the drum pad is playing, stop it
      if (transportSources[i].isPlaying())
      {
        // Stop track and reset position to the beginning
        transportSources[i].stop();
        transportSources[i].setPosition(0);

        // Stop the blinking
        stopBlinking(i);

        // Disable loop
        isLooping[i] = false;

        // Update loop state to off
        loopButtons[i].setToggleState(false, juce::dontSendNotification);
      }
      else
      {
        // If not playing, start track
        transportSources[i].setPosition(0);
```

```
            transportSources[i].start();

            // Start blinking effect
            isBlinking[i] = true;
          }
        }
        else if (button == &loopButtons[i])
        {
          // update the loop state based on the toggle
          isLooping[i] = loopButtons[i].getToggleState();

          // Console statement for looping of the different drum pads
          DBG("Looping for Pad " + juce::String(i + 1) + (isLooping[i] ? " enabled" : "
disabled"));
        }
    }
}

void DrumPadComponent::sliderValueChanged(juce::Slider* slider)
{
    // Loop through all volumn sliders
    for (int i = 0; i < 6; ++i)
    {
        if (slider == &volumeSliders[i])
        {
          // Set the volumn to the corresponding transport source
          transportSources[i].setGain(static_cast<float>(volumeSliders[i].getValue()));
```

```
            // Console statement for the change of volumn
            DBG("Pad " + juce::String(i + 1) + " volume set to " +
juce::String(volumeSliders[i].getValue()));
        }
    }
}

void DrumPadComponent::timerCallback()
{
    // Toggle blink state for drum pads
    blinkState = !blinkState;

    // Loop thorough all drum pads to update the blinking effect
    for (int i = 0; i < 6; ++i)
    {
        // If the pad is blinking, update its color
        if (isBlinking[i])
        {
            // Based on blinkstate update the color of the drum pad
            if (blinkState)
                drumPads[i].setColour(juce::TextButton::buttonColourId, padColors[i]);
            else
                drumPads[i].setColour(juce::TextButton::buttonColourId, defaultPadColor);

        }
    }
    // Update the UI
    repaint();
```

```
}

void DrumPadComponent::changeListenerCallback(juce::ChangeBroadcaster* source)
{
    // Loop through all transport sources to check for changes
    for (int i = 0; i < 6; ++i)
    {
        // If the transport source has finished playing, update accordingly
        if (source == &transportSources[i] && !transportSources[i].isPlaying())
        {
            // If loop is on, restart track
            if (isLooping[i])
            {
                // Restart the track
                transportSources[i].setPosition(0);
                transportSources[i].start();

                // Console statement for loop of drum pad
                DBG("Looping Pad " + juce::String(i + 1));
            }
            else
            {
                // Stop blinking effect, if loop state is off
                stopBlinking(i);

                // Console statement when the drum pad track is finished
                DBG("Pad " + juce::String(i + 1) + " track finished playing");
            }
        }
    }
}

void DrumPadComponent::stopBlinking(int padIndex)
{
    // Stop blinking effect for the drum pad
    isBlinking[padIndex] = false;

    // Set the drum pad to default color
    drumPads[padIndex].setColour(juce::TextButton::buttonColourId, defaultPadColor);
}

void DrumPadComponent::loadSample(int padIndex, const juce::String& fileName)
{
    // current working directory
    juce::File current_directory = juce::File::getCurrentWorkingDirectory();

    // Search for the project root by looking for a unique file or folder
    while (current_directory.exists() &&
!current_directory.getChildFile("samples").exists())
    {
        current_directory = current_directory.getParentDirectory();
    }

    // If "samples" folder is found, construct the full path
    juce::File file;
    if (current_directory.getChildFile("samples").exists())
    {
        file = current_directory.getChildFile("samples").getChildFile(fileName);
    }



    // Check if the file exists
    if (!file.existsAsFile())
    {
        DBG("File not found: " + file.getFullPathName());
        return;
    }

    // Create an audio reader for the file
    std::unique_ptr<juce::AudioFormatReader>
reader(formatManager.createReaderFor(file));

    // If creation of reader was successful, load the sample
    if (reader != nullptr)
    {
        // Create and audio format reader source
        readerSources[padIndex] =
std::make_unique<juce::AudioFormatReaderSource>(reader.release(), true);

        // set it for the transport source
        transportSources[padIndex].setSource(readerSources[padIndex].get(), 0, nullptr,
readerSources[padIndex]->getAudioFormatReader()->sampleRate);

        // Set the transport source to the audio source player
        audioSourcePlayer[padIndex].setSource(&transportSources[padIndex]);
        // Console statement if the sample track is loaded
        DBG("Loaded sample: " + file.getFullPathName());
    }
    else
    {
        // Console statement if the sample track is not loaded
        DBG("Failed to load sample: " + file.getFullPathName());
    }
}
```