R1

R1A

To load audio files into the audio players, I created an implementation of the `ImageButton` component, and named it 'loadButton' within the `DeckGUI.h` file. After setting up 'loadButton', and using the `setImages()` function, to set its states, I called `addAndMakeVisible()` on it, in order to make it visible- all this was within the `DeckGUI.cpp` file, within the `DeckGUI` constructor. The purpose of the loadButton is to give the user the ability to load files into the app. The user is supposed to click on the button, and from there choose any audio file- everything else that goes into loading files should be hidden from the user, for it to be a well designed user interface, the process should be abstracted. In order to achieve this, I added a listener to the loadButton (`loadButton.addListener(this)`), this meant that whenever a user clicked on the loadButton an event would be triggered- more specifically the `juce::Button::Listener` class is used to receive callbacks whenever a button is clicked. Now that the loadButton had a listener, the next step was adding functionality to the button click. In other words, when the user clicked the loadButton, instead of a message being outputted to the console, the user needed to be able to browse through a file manager application, in order to pick their audio file.

Within the `DeckGUI::buttonClicked(Button* button)` function, I utilised an if-statement to get the desired functionality. The condition for the if-statement was the following: `button == &loadButton`. This checks whether the address of the button clicked, is that of the loadButton, as the loadButton is not the only button within the `DeckGUI` class, without this check the function of the button clicked, would just be the execution of all the code in the `DeckGUI::buttonClicked(Button* button)` function. If the condition is true for the user has clicked the loadButton, the following code is ran:

```
FileChooser chooser{"Select a file..."};
      if (chooser.browseForFileToOpen())
      {
          player->loadURL(URL{chooser.getResult()});
          waveformDisplay.loadURL(URL{chooser.getResult()});
          ...
      }
```

In the above code, you see that `FileChooser` named chooser has been created. In order to display the chooser, a `browseFor...()` method must be used. In this case I used the `browseForFileToOpen()` method. This method shows a dialog box, which will allow the user to choose a file to save. The file returned is then obtained by calling `getResult()`. Within the if-statement, the result of calling `getResult()` method is passed to the `loadURL()` method. Which is called on the player and waveformDisplay. The player and waveformDisplay call their respective `loadURL()` methods.

For the player, when the `loadURL()` method is called an input stream is created, and a reader is created for it, from there the audio source can be played, stopped, etc.

In the case of the waveformDisplay, when the `loadURL()` method is called the `audioThumb` is cleared, and the audioURL from the chosen file is used for the `setSource()` method. When `repaint()` is called the audioThumb is displayed for the user to see.

Point to mention- `chooser.browseForFileToOpen()` will return false if the user presses cancel, and so no file will be passed.

R1B

In order to play two or more tracks I created an instance of `MixerAudioSource` and named it mixerSource. By definition, the `MixerAudioSource` is an `AudioSource` that mixes together the output of a group of other AudioSources. Therefore it gives the program the ability to play two or more tracks at a time.

Within the `MainComponent::prepareToPlay()` function, mixerSource calls `prepareToPlay()`, which tells the source to prepare for playing. Within the same `prepareToPlay()` function mixerAudio calls the `addInputSource()` method, twice, once for each player, as demonstrated below:

```
void MainComponent::prepareToPlay (int samplesPerBlockExpected, double sampleRate)
{
    ...
    mixerSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
    mixerSource.addInputSource(&player1, false);
    mixerSource.addInputSource(&player2, false);
}
```

The `addInputSource()` method does as its name suggests, it adds an input source to the mixer. The first parameter of the method is the source that is to be added to the mixer. The second parameter is a boolean, and if it evaluates to true, the source will be deleted when it no longer serves a purpose to the mixer. On the other hand, if it evaluates to false, the source will not be deleted. As you can see in the code above, it is set to false, ergo in this case, the source will not be deleted.

Within the `MainComponent::getNextAudioBlock()` function, mixerSource calls `getNextAudioBlock()`, which is a method that is called to retrieve successive blocks of audio data. Once `prepareToPlay()` is called, `getNextAudioBlock()` continues to be called while an audio block of data is needed.

Within the `MainComponent::releaseResources()` function, mixerSource calls `releaseResources()`, this method allows the source to release anything that it no longer needs, once playback is complete. All this collectively, allows the program to play two or more tracks.

R1C

The volume of the tracks can be adjusted using the volume slider. In order to achieve this, the `DeckGUI` class inherits from `juce::Slider::Listener`, which works in similar fashion to `juce::Button::Listener` - it receives callbacks from a Slider. When the user moves the slider, it changes the value of the slider, an `addListener()` method is called on the volume slider, so that the program knows when the slider's value has changed- without it, the user could move the slider as much as they want, but their actions would have no effect on the volume. Within the `DeckGUI::sliderValueChanged()` function, an if-statement is used as a check, so the program knows which slider the user has used. If the address of the slider is that of the volume slider, then the player calls `setGain()`, and passes the slider's `getValue()` method, as a parameter. Within the `DJAudioPlayer::setGain()` function, `transportSource` calls `setGain()`, and then passes the value it received from the `DJAudioPlayer::setGain()` function as a parameter. Within the `AudioTransportSource::setGain()` function, `newGain` is assigned to `gain`, as show below:

```
void AudioTransportSource::setGain (const float newGain) noexcept
{
    gain = newGain;
}
```

Due to this, when two tracks are playing simultaneously, the user is able to adjust their volumes. This added functionality allows the user to fade tracks in and out, blend the sounds, and perhaps allow one track to be dominant over another, and vice versa- all giving a wonderful mixing effect.

R1D

By the same token, the user can adjust the speed using a slider. This is accomplished by first going through the typical motions: creating a `Slider` and naming it speedSlider; calling `addAndMakeVisible()` on it; and then adding a listener to it. Once those three tasks are complete, within the `DeckGUI::sliderValueChanged()` function, I added the following if statement:

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

The above code snippet displays how the if statement, first checks to see whether the slider in use, is indeed the speedSlider (by comparing the address), and if so the player calls the `setSpeed()` method, and passes the slider's new value to it. Within the `DJAudioPlayer::setSpeed()` function, `resampleSource` calls `setResamplingRatio()`, which changes the resampling ratio, which in turn changes the speed.

R2

R2A

In the screenshots below you can see the controls that the user has.

Not playing and not looping

Playing and looping

The first button is the load button, which allows the user to load in a track. The pink button is the rewind button, which allows the user to jump back in a track. The third/blue button is a play-pause button that toggles between play and pause, depending on the state that the player is in. The orange button is the forward button, it allows the user to jump ahead in a track. The last/red button allows the user to make the track loop or not.

R2B

When the user runs the application, they are able to play and pause a track, rewind and reverse a track, loop or not loop a track. In order to give the user the ability to rewind and forward a track, I created a forward button and a reverse button. I added listeners to both buttons, so that an event would be registered when the user clicked on one. Within the `buttonClicked()` function I created an if-statement, with a condition that compared a button to the address of the rewind button. Within the if-statement, I called `setPositionRelative()` on player. I passed `player->getPositionRelative()- 0.03` as a parameter. I created another if-statement, and this time the condition compared a button to the address of the forward button. Likewise, within the if-statement, I called `setPositionRelative()` on player. I passed `player->getPositionRelative()+ 0.03` as a parameter. By doing this, whenever the user clicks the rewind or forward button, they are able to jump ahead or behind in a track.

In order to add looping functionality, I created a loop button. I created an `enum class` called `LoopState`, the enumerator list contained two enumerator definitions: Loop, NoLoop. I set the initial `LoopState` to NoLoop. On the loop button I called `setToggleState()`- which allows a button to become toggle-able and switch between an on and off state.

Within the `buttonClicked()` function I created an if-statement that checked whether the button clicked was the loop button. If so, I then created another if-statement. This if-statement checked `LoopState`. If the `LoopState` is equal to NoLoop, then the `noLoop()` function is called, whereas if the LoopState is equal to Loop, then the `loop()` function is called. This is displayed below:

```
if (button == &loopButton) {
    std::cout << "LoopButton was clicked...\n";
    if (loopState == LoopState::NoLoop)
    {
      // If the btn is clicked and its in a no looped state- switches to looped state
      loopButton.onClick = [this]() { loop(); };
    }
    else if (loopState == LoopState::Loop)
      // If the btn is clicked and its in a looped state- switches to nolooped state
      loopButton.onClick = [this]() { noLoop(); };
  }
```

Within the `loop()` function `LoopState` is set to Loop. The `setToggleState()` function is called. The player then calls the `readerSource`, which in turn calls `setLooping(true)`, as displayed below:

```
/* as when it is not looped its 'true', and when it is looping it's false */
loopState = LoopState::Loop;

loopButton.setToggleState(false, NotificationType::dontSendNotification);
std::cout << "Loop button was clicked " << std::endl;
player->readerSource->setLooping(true);
```

The opposite is done for the `noLoop()` function:

```
loopState = LoopState::NoLoop;

loopButton.setToggleState(true, NotificationType::dontSendNotification);
std::cout << "NoLoop button was clicked " << std::endl;
player->readerSource->setLooping(false);
```

This allows the user to loop or not loop tracks. The playPause button works in a similar way. However it uses an `enum class` called `PlayState` which has the following enumerator list: Play, Stop. When the `PlayState` is equal to Stop, the `play()` function will be called, and vice versa. In this way, only one button is needed to play and pause. My aim was to take the functionality of the play button, and the stop button, and combine it into one button.

R3

R3A

Within the **`PlaylistComponent`** class, I created an instance of a **`TableListBox`** named tableComponent. **`TableListBox`** creates a table of cells, which uses **`TableHeaderComponent`** as a header. This table is where the users files go when they add them to the library- it is their music library. Also within the **`PlaylistComponent`** class, I created three **`ImageButton`**s: an addTrackButton, an addMultipleButton, and a removeTrackButton. The user can use these buttons, to add a track to the library, add multiple tracks to the library, and to remove tracks from the library. The buttons were set up in the same way as the loadButton, an instance of an **`ImageButton`** was created, **`addAndMakeVisible()`** was called, and a listener added.

When the user clicks the addTrackButton and an event is triggered, a dialog box is shown, from there the user can pick any track they want, and it will be added to their library. The same goes for the addMultipleButton. The way this works is that when the user clicks either the addTrackButton or the addMultipleButton, and the dialog box is shown, their selections are saved to a vector named files, three methods are then called, into which the vector is passed. This is illustrated in the code snippet below.

```
if (chooser.browseForMultipleFilesToOpen())
      {
        // saves tracks to vector
        for (int i = 0; i < chooser.getResults().size(); i++){
          files.push_back(chooser.getResults()[i]); // saves track files to vector
        }
        setTracks(files);   // to set the track titles
        setDuration(files); // to set the track length
        setFileType(files); // to set the track type
      }
```

The above code is taken from the addMultipleButton, as indicated by the browseForMultipleFilesToOpen() and getResults() methods.

The methods **`setTracks()`**, **`setDuration()`**, and **`setFileType()`**, then run their individual tasks. Each of the mentioned methods produce the corresponding meta data that is to be displayed within the table. At the end of each function **`tableComponent.updateContent()`** is called.

R3B

As previously mentioned, once the user clicks on the addTrackButton or the addMultipleButton, and the dialog box is shown, and their selections are added to a vector, the methods **`setTracks()`**, **`setDuration()`**, and **`setFileType()`**, are then called.

The `setTracks()` method is the method responsible for creating and displaying the filename within the music library. It takes a vector as a parameter. It then loops through the vector, and for each file it calls the `getFileNameWithoutExtension()` method, and then calls `toStdString()` on the resulting `juce` String, which is then saved to a string variable named trackString. Also for each file it calls the `getFullPathName()` method, and then calls `toStdString()` on the resulting `juce` String, which is then saved to a string variable named trackPath. This is displayed below:

```
for (int i = 0; i < trackFiles.size(); i++)
    {
      trackString = trackFiles[i].getFileNameWithoutExtension().toStdString();
      trackPath = trackFiles[i].getFullPathName().toStdString();

      /* Stops the user from selecting a track that they have already added*/
      if (std::find(std::begin(trackTitles), std::end(trackTitles), trackString) != std::end(trackTitles))
        continue;
      else {
        trackTitles.push_back(trackString);
        trackPaths.push_back(trackPath);
        ...
      }
    }
```

In the code snippet above, you can see that after the strings are created a check is run. A check is run in the form of an if statement, to stop the user from attaining duplicates of a track in the music library. For every iteration, a string is assigned to trackString, within the check this string (trackString), is compared to the strings that are already in the trackTitles vector. If the string doesn't match any string within the trackTitles vector, then the else block is run, and trackString is added to the vector. However, if it does match a string within the vector, it breaks from that iteration and continues to the next one, without adding the string to the vector.

Once the iterations are complete `tableComponent.updateContent()` is called, and the data is displayed in the table.

The `setDuration()` method is responsible for getting the length of the track, converting into typical time formatting, and displaying it within the table. `setDuration()` method takes as a parameter a vector of files, and loops through the vector. For each iteration, it converts and saves the file at that index into a URL, called fileURL. An `AudioFormatReader*` is created and fileURL is passed into it. This is displayed below:

```
for (int i = 0; i < trackDurations.size(); i++)
  {
    URL fileURL = URL(trackDurations[i]);
    double lengthSeconds = 0;
    AudioFormatReader* rd = formatManager.createReaderFor(fileURL.createInputStream(false));
    ...
}
```

Next a check is run to ensure that it is a good file. If it is a good file, lengthSeconds is then calculated by dividing the returned **lengthInSamples** value by the returned **sampleRate** value. Now that the length is calculated in seconds, the next thing to do is to convert it into hours, and display it in the hours:minutes:seconds format (00:00:00). This is the code so far:

```
for (...) {
  ...
  if (rd != nullptr) // good file!
      {
          lengthSeconds = rd->lengthInSamples / rd->sampleRate;
          delete rd;

        /* To hold the corresponding int values and later
         format for time*/
        std::string minDuration;
        std::string secDuration;
        std::string hourDuration;
  ...
  }
}
```

As you can see in the code above, three strings are created, this is where the parts of the hours:minutes:seconds format will be saved, to make up the duration. Next using some arithmetic calculations, the seconds, minutes, and hours are calculated and stored into their corresponding int variables.

```
/* Rounding adds a slight degree in accuracy (noticed 1 difference in 20 songs) */
int seconds = (int)(round(lengthSeconds * 100) / 100);
int secRemain = seconds % 60;
int minutes = (seconds - secRemain) / 60;
int minRemain = minutes % 60;
int hours = (minutes - minRemain) / 60;
```

In the above code, you can see that there are also the variables secRemain and minRemain, they are used to store the seconds and minutes that are after 60. This is so that the program doesn't end up giving a weird time value like 00:07:90, in this case it should be 00:08:30. The secRemain and minRemain are calculated using the modulus operator, and they are the values that are going to be used to make up the time format.

Once the above calculations have been made, I use a series of if-else statements to format the time. In order for the time to display as 00:08:30 rather than 0:8:30, I use an if-statement that checks whether the hours, minRemain, or secRemain are below 10. If they are a "0" will be put in front, if they aren't, no "0" will be added. This is displayed below:

```
/* To format the time */
if (hours < 10)
    hourDuration = "0" + std::to_string(hours);
else
    hourDuration = std::to_string(hours);

if (minRemain < 10)
    minDuration = "0" + std::to_string(minRemain);
else
    minDuration = std::to_string(minRemain);

if (secRemain < 10)
    secDuration = "0" + std::to_string(secRemain);
else
    secDuration = std::to_string(secRemain);
```

Once each section of the time format has been created, I then create a **`std::stringstream`** called calculateTime. I then input the hourDuration, minDuration, and secDuration, along with some colons into calculateTime. I then create a string variable called calculateTime and assign **`calculateTime.str()`** to it. The string is then added to the duration vector. This is show below:

```
/* Create typical time styling */
std::stringstream calculateTime;
calculateTime << hourDuration << ":" << minDuration << ":" << secDuration;
std::string trackDuration = calculateTime.str();
...
/* Add to vector */
duration.push_back(trackDuration);
```

Once this occurs for every iteration of the for loop, **`tableComponent.updateContent()`** is called, and the data is displayed in the table.

The **`setFileType()`** method is responsible for getting the type of the track, and then displaying it in the table.

```
void PlaylistComponent::setFileType(std::vector<File> trackTypes)
{
  for (int i = 0; i < trackTypes.size(); i++)
  {
    fileType.push_back(trackTypes[i].getFileExtension().toStdString());
  }
  tableComponent.updateContent();
}
```

The above code shows how a for loop iterates through the trackTypes vector, and for each file calls the method `getFileExtension()`, and then converts it into a `std::string`, then it pushes the string into the fileType vector. Once this is complete, `tableComponent.updateContent()` is called and the data is displayed in the table.

R3C

In order to allow the user to search for tracks within the playlist, I created an instance of the `juce::Label`, which I called search. I then called the `setText()` method and the `setEditable()` method on it. I also called `addAndMakeVisible()` and added a listener to it. The `setText()` method allowed me to set the text the label would originally display. The `setEditable()` method allowed the label to turn into a TextEditor when clicked. After positioning the search bar just above the playlist, I then implemented the function `labelTextChanged()`, which is called whenever the label's text has changed. Within that function, there is an if statement that checks whether the search box is empty or not.

If the search box is empty, the playlist will stay the same. However if the search box is not empty, the playlist will filter based on what the user types in. In order to populate the table I used vectors: trackPaths, trackTitles, duration. These vectors each have a complementary filter vector: filteredTrackPaths, filteredTracks, filteredDuration. When the non-filter vectors are being populated, their contents are also copied to their filter counterparts, so that `trackPaths = filteredTracks` etc.

```
...
else {
  ...
  trackPaths.push_back(trackPath);
  filteredTrackPaths = trackPaths;
}
```

This is shown above (a similar thing is done for the duration and tracks as well). The filter vectors are the actual vectors that populate the tables (though at the start of the program and when the search bar is empty, their content is equal to that of the non filter vectors).

Once the user types something in, the filter vectors are all cleared of their contents. A for loop is then run that iterates through the trackTitles vector. This is displayed in the code below:

```
filteredTracks.clear();
filteredTrackPaths.clear();
filteredDuration.clear();

for (int i = 0; i < trackTitles.size(); i++)
{
  filterString = trackTitles[i];
  filterPath = trackPaths[i];
  durationString = duration[i];

  if(filterString.find(userEntry) != std::string::npos) {
    filteredTracks.push_back(filterString);
    filteredTrackPaths.push_back(filterPath);
    filteredDuration.push_back(durationString);
  }
  else
    continue;
}
```

As you can see from the above code, within each iteration there is a check in the form of an if statement. userEntry is what the user types into the search bar. The track title is compared to the userEntry, and if the userEntry contains any characters that are in the track title, then the corresponding track title is pushed into the filteredTrack vector. If the userEntry doesn't contain any characters that are in the track title, then the track title is not added to the vector. Once the for loop is complete **`tableComponent.updateContent()`** is called, and the table is updated, only displaying the filtered tracks. Once the user clears the search bar, all the tracks are shown again.

R3D

I wanted the users to simply drag files from the library and drop them into whichever deck they wanted. In order to achieve this I made DeckGUI a **`juce::DragAndDropTarget`**. With that there were two virtual functions that needed to be implemented: **`isInterestedInDragSource(const SourceDetails &dragSourceDetails)`** and **`itemDropped(const SourceDetails &dragSourceDetails)`**. The **`isInterestedInDragSource()`** function checks whether the target (in my case DeckGUI) is interested in the type of object being dragged (the music file from the library). The **`itemDropped()`** function indicates whether the user has dropped something into this component. Whenever the user drops an item, this gets called, and the description can be used to decide whether the object wants to deal with it or not. Both functions take in **`const SourceDetails &dragSourceDetails`** as a parameter. **`dragSourceDetails`** contains information about the source of the drag operation.

The `isInterestedInDragSource()` function simply returns true, which means that the component (DeckGUI) wants to receive the other callbacks- in order words DeckGUI wants to receive the track. The `itemDropped()` function does a little more, as displayed below.

```
void DeckGUI::itemDropped (const juce::DragAndDropTarget::SourceDetails &dragSourceDetails) {
  std::cout << "DeckGUI item has dropped from the playlist." << std::endl;
  DBG("The name is " << dragSourceDetails.description.toString());

  player->loadURL(URL{juce::File{dragSourceDetails.description.toString()}});
  waveformDisplay.loadURL(URL{juce::File{dragSourceDetails.description.toString()}});
  ...
}
```

As you can see from the code snippet above, the description that is taken from `dragSourceDetails` is used to create a file, which is then turned into a URL, to be used as a parameter for the `loadURL()` function. The `loadURL()` function is called on the player and waveformDisplay. This is so that the track can be played, and so that the audio thumbnail can be drawn.

The information for the `dragSourceDetails` is taken from the `getDragSourceDescription()` function. This function allows rows from the table to drag and drop. If a non-null variant is returned by this function, the table will look for a `dragAndDropContainer` in its parent hierarchy- then triggering a drag and drop operation.

The `getDragSourceDescription()` function takes in the parameter `const SparseSet< int > &selectedRows`, which is just the selected row. Within the function I created a `juce::StringArray` which I named rows. I then created a for loop that iterates through the number of `selectedRows`. Within the for loop, I created an integer variable named rowNum, to which I assigned `selectedRows[i]`. I then created a `juce::String` named selected, and assigned `String(filteredTrackPaths[rowNum])` to it. I then added the `juce::String` selected to the `juce::StringArray` rows. Once the for loop iterations are completed, I return `rows.joinIntoString(", ")`. Doing this provided the description that is used in the `itemDropped()` function. These functions are what gives the users the ability to load files from the library into any deck.

R3E

In order to get the music library to persist, I had to save all the data in the library in a file, so that when the user opens the application the data is read from that file and the music library is then recreated with that same data. To achieve that, I created two `std::fstreams`, a `std::ifstream` called currentPlaylistO, and a `std::ofstream` called currentPlaylistI.

Within the destructor of the PlaylistComponent class, I created a `juce::File`, which I named f and assigned it `juce::File::getCurrentWorkingDirectory()`-this is because `juce` likes absolute paths. I then used f to create a string (filename) with the name of the .txt file. I then called the `open()` method on the `std::ofstream` currentPlaylistI. I passed in filename, and `std::ios::out` as parameters. I then created a check in the form of the if-statement. The parameter for the if-statement is the following: `currentPlaylistI.is_open()`, this is to check whether the file was able to be opened. If true, a for loop is run, that iterates the same number of times, as the size of the trackTitles vector. For each iteration, the following is ran:

```
currentPlaylistI << trackPaths[i] << "*" << trackTitles[i] << "*"
<< duration[i] << "*" << fileType[i] << std::endl;
```

The code above shows that for each iteration, the corresponding trackPaths string is written to the file, then the corresponding trackTitles string, then the duration string, and finally the fileType string. An asterisk (*) is written in between each string, to serve as a separator. The reason I used the asterisk is because some song titles contain a comma, and some even an ampersand, and I want to reduce the possibilities of errors as much as possible. After the strings are written the end line is called. Once the for loop is complete, then `currentPlaylistI.closed()` is called. In the case that the if-statement is false, a message is sent to the console, to indicate that an error has occurred.

Within the constructor of the `PlaylistComponent` class, I created a `std::string` that I named line. I reproduced the same method as before to create the filename string, as displayed below:

```
juce::File f = juce::File::getCurrentWorkingDirectory();
std::string fname = f.getFullPathName().toStdString();
std::stringstream appendString;
appendString << fname << "playlist.txt";
std::string filename = appendString.str();
```

I then called the `open()` method on the `std::ifstream` currentPlaylistO, and passed in filename as a parameter. I then created a vector of strings that I named metaData. Next I ran an if-statement, which has `currentPlaylistO.is_open()` as a parameter. If false, a message is printed to the console.

If true, a while loop is run with `getline(currentPlaylistO, line)` as its parameter- this while loop runs through all the lines of data in the .txt file. Within the while loop is the following line: `savedFiles.push_back(line)`. This saves each line within the .txt file as a string within the savedFiles vector. Once the while loop is no longer true, a for loop is run. The for loop iterates through the savedFiles vector. For each iteration, the following is ran:

```
std::stringstream extractData(savedFiles[i]);
std::string metaDatum;

while(getline(extractData, metaDatum, '*'))
{
  metaData.push_back(metaDatum);
}
```

The above code shows that each string in the saveFiles vector, is split up into smaller strings, using the asterisk as a separator, and saved as the string metaDatum, which is then added to the metaData vector. After this a check is run to check whether the metaData vector is equal to 4, if it isn't 4 an error message is printed, then the program moves onto the next iteration. If it is equal to 4, then the following is ran:

```
trackPaths.push_back(metaData[0]);
trackTitles.push_back(metaData[1]);
duration.push_back(metaData[2]);
fileType.push_back(metaData[3]);

filteredTrackPaths = trackPaths;
filteredTracks = trackTitles;
filteredDuration = duration;
```

After this, the metaData vector is cleared in preparation for the next iteration. Once the for loop is complete, `tableComponent.updateContent()` and `currentPlaylistO.close()` are called. In doing this, what was in the library when the application was closed, is there when it is open, as it persists.

R4

R4A

In order to customise the layout of the GUI, I used `juce::Grid`- which is a container that handles the anatomy for grid layouts. Within `MainComponent::resized()` function, I first created an instance of `juce::Grid`, which I named grid. I then set `grid.templateRows`, and `grid.templateColumns`, as displayed below:

```
grid.templateRows    = {
                           Track (Fr (1)), //1
                           Track (Fr (1))  //2
                       };

grid.templateColumns = {
                           Track (Fr (1)), //1
                           Track (Fr (1))  //2
                       };
```

From the above code, you can see I created a grid with the following proportions: 2 rows, 2 columns. I then set `grid.items`, and called `performLayout()` on grid. This is highlighted in the code below:

```
grid.items = {
                GridItem(&deckGUI1).withArea(1, 1, 1, 2),
                ...
                GridItem(&playlistComponent).withArea(2, 1, 2, 3)
            };

 grid.performLayout(getLocalBounds());
```

`items` is the group of items that are to be styled. The `performLayout()` method styles the items within a particular area. In the code snippet above, you can see that I passed `getLocalBounds()` as a parameter, which returns the bounds relative to the parent- in this case the whole of the app. I repeated this method to style the `DeckGUI` and the `PlaylistComponent` Classes.

I also created another class named `DJLookAndFeel`, that inherits from `juce::LookAndFeel_V4` to customise the look of the GUI. Within this class I customised the look of the sliders (**Figure 1**).

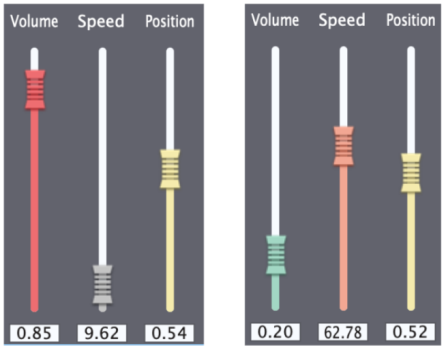

Figure 1 The sliders.

I drew and created the slider knobs within Figma (**Figure 2**), as depending on the value of the slider I wanted the knob to change colour. In order to achieve this I used `juce::Image`, and created `Image` instances for each of the knobs displayed in **Figure 2**. Within the `DJLookAndFeel` class, I overrode the `juce::LookAndFeel_V4 drawLinearSlider()` function.

Figure 2 The knobs I created in Figma.

Within the `juce::LookAndFeel_V4 drawLinearSlider()` function, there was the following if statement:

```
if (! isTwoVal)
        {
            g.setColour (slider.findColour (Slider::thumbColourId));
            g.fillEllipse (Rectangle<float> (static_cast<float> (thumbWidth),
            static_cast<float> (thumbWidth)).withCentre (isThreeVal ? thumbPoint : maxPoint));
        }
```

Within my new implementation of the function, I modified the if statement, as shown in the snippet below (spaces added and comments moved for the snippet):

```
if (! isTwoVal)
     {
         g.setColour (slider.findColour (Slider::thumbColourId));
         if (slider.isVertical()){ /* Checks whether the slider is vertical */

           if (slider.getValue() < 1.0) { /* Checks to see whether it is the vol + pos slider or the speed slider */

               if (slider.getValue() >= 0.8) { /* Checks the value of the slider */
                 g.drawImage(sliderKnobRed2, (int)maxPoint.x - (sliderKnobRed2.getWidth()/2),
                 (int)maxPoint.y - (sliderKnobRed2.getHeight()/2), sliderKnobRed2.getWidth(),
                 sliderKnobRed2.getHeight(), 0, 0, sliderKnobRed2.getWidth(), sliderKnobRed2.getHeight()); //pastel red
                 slider.setColour(juce::Slider::trackColourId , juce::Colour(0xffF26B6B)); //pastel red
               }

               else if (slider.getValue() >= 0.6) {
                 ...
               }
               ...
           }
           else if ...
         }
       }
```

Instead of the program drawing an ellipse, it instead draws an image. Once the program confirms that `(!isTwoVal)` has evaluated to true. It runs another check, in the form of an if-statement to verify that the slider is a `LinearVertical` slider, if that evaluates to true, another check is run. This new check examines the value of the slider, if the slider value is above 1.0 it continues to the next section of code. The goal of this check is to discern between the speed

slider, and the volume + position sliders. This is because the range of the speed slider is different from that of the volume and position sliders, and therefore the values needed will be different, when it comes to changing the colour of the knob.

If the slider value is less than 1.0, a chain of if-statements are run, with the conditions evaluating to true or false, based on the value of the slider. The code above shows that when the slider value is greater than or equal to 0.8, the red knob image will be drawn. This happens as when the condition is rendered true, `drawImage()` is called on g, which draws part of an image, and rescales it based on the parameters passed to it.

Next, within the `DeckGUI`'s header file, I created an instance of the `DJLookAndFeel` class, which I named appLookAndFeel. In the `DeckGUI`'s cpp file, within the constructor I passed appLookAndFeel as a parameter into `LookAndFeel::setDefaultLookAndFeel()` method. Thus changing the default-look-and-feel.

R4B + R4C

In order to display the music library component within the GUI layout, I added an instance of the `PlaylistComponent` class, to the private section of the `MainComponent` header file. Then as shown earlier I added it to the grid I created, that displays the components in the layout specified before. Below is a screenshot of the application.