

## Contents

Project specification summary.....	2
Graphical User Interface design.....	3
Introduction.....	3
Initial Screen UI .....	5
Sign Up UI .....	6
Log In UI.....	7
MainScreen UI .....	8
Add a Song UI .....	10
Create a Playlist UI .....	12
List Available Songs UI.....	13
Song Information UI .....	15
List Available Playlists UI .....	17
Playlist Information UI.....	18
Statistics UI .....	20
Settings UI .....	22
Player UI .....	23
Class diagram .....	25
Development methodology .....	28
Sprints distribution .....	28
Tools and resources .....	29
Time Costs .....	30
Conclusions .....	32
Bibliography .....	33

## Project specification summary

For the second semester project, we are asked by the OOPD's Project Development and Research Department to implement a new program called Espotifai. As the name may hint, its purpose is to allow the user to store music locally either by itself or organized into different playlists.

Following the concepts learned in class, we structured our project using Layered Architecture. Our components were organized into horizontal layers based on the three main layers (Presentation, Business and Persistence).

The experience starts once the user decides to log in or sign up. To be able to ensure that the username and email were unique, we stored them in the database as primary keys. Moreover, we stored the password which had to compile with a series of aspects in order to guarantee the user's safety.

Once the user is all set up, he will come across our main menu where he will be able to choose whether he wants to add a song, view the available songs, create a playlist, or see all our playlists.

If the user decides to add a song, he will have to provide us with the following data title, genre, album, author, and file with the corresponding song either (.wav or .mp3). Moreover, when listing the available songs, the user will have the possibility to access detailed information regarding the desired song. This information is composed of stored data with the incorporation of the lyrics to allow the user to not only listen to the song but to deeply feel it when singing it. As from the view we will be capable of playing a song and deleting it.

Furthermore, if the user decides to create a playlist, apart from the title he will be asked to enter a short description to give other users a glimpse of what can be expected from that particular playlist. These playlists will be accessed from the list available playlists from where the user will be able to decide which playlist to acquire in order to simply view it or to play the contained songs.

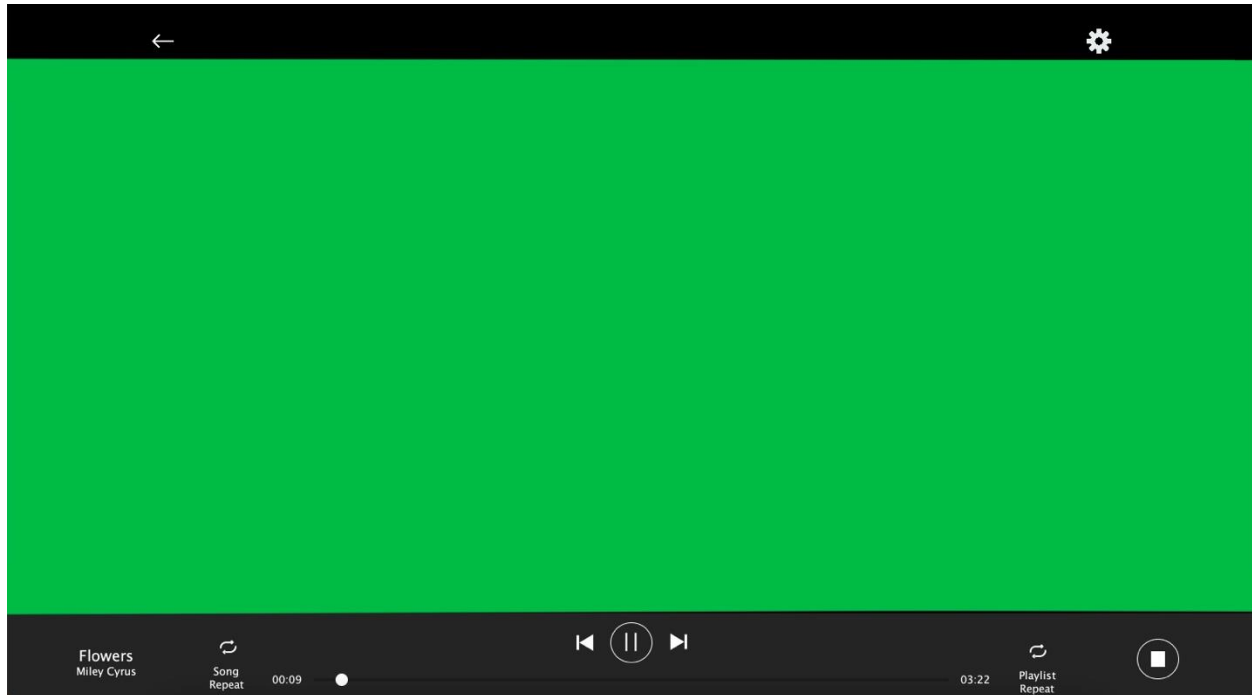
Lastly, the music statistics option will show a chart with the music genres that exist in the application, and the number of songs that are part of each of the genres.

It is worth mentioning that from all the views, once logged in, the user will have the opportunity of controlling the song being played (pausing it, going to the next one, going back or putting in repeat either the song or the playlist) and accessing settings to delete their account or simply log out.

## Graphical User Interface design

### Introduction

To develop the graphical interface of our application we used *Swing*, one of the Java Foundation Classes *JFC* that extends from the Abstract Window Toolkit *AWT*. This one provided us with several components and tools to arrange these, which we used to achieve the desired functionalities.

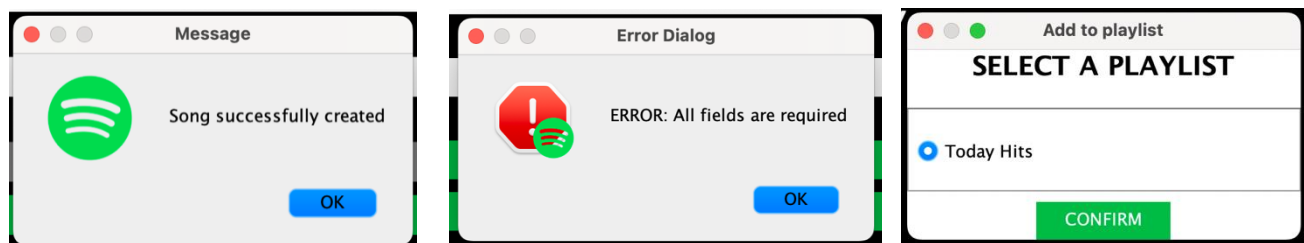


We wanted most of our views to have their top and bottom parts fixed so that the user could access the player and the settings/back icons from anywhere in our application. Hence, we created the class *MainFrame* (which extends from a *JFrame*) and added a *JPanel* with a *Border Layout* to it so that we could place these two sections on the *NORTH* and *SOUTH* regions of the screen. Having these, we added a *Card Layout* to the *CENTER* region so that we could switch between the different screens while maintaining the top and bottom parts (green rectangle representing the screen to switch). We took this approach as we believed it was more convenient and optimal than having to replicate this *Border Layout* with the top and bottom panels in each of our app's views.

Although for most of our views, we need both top and bottom panels, some didn't require them. For instance, the first three UIs explained below don't show the player, as we didn't want this one to be available until entering your credentials. Therefore, we hid it until the user logged into the application.

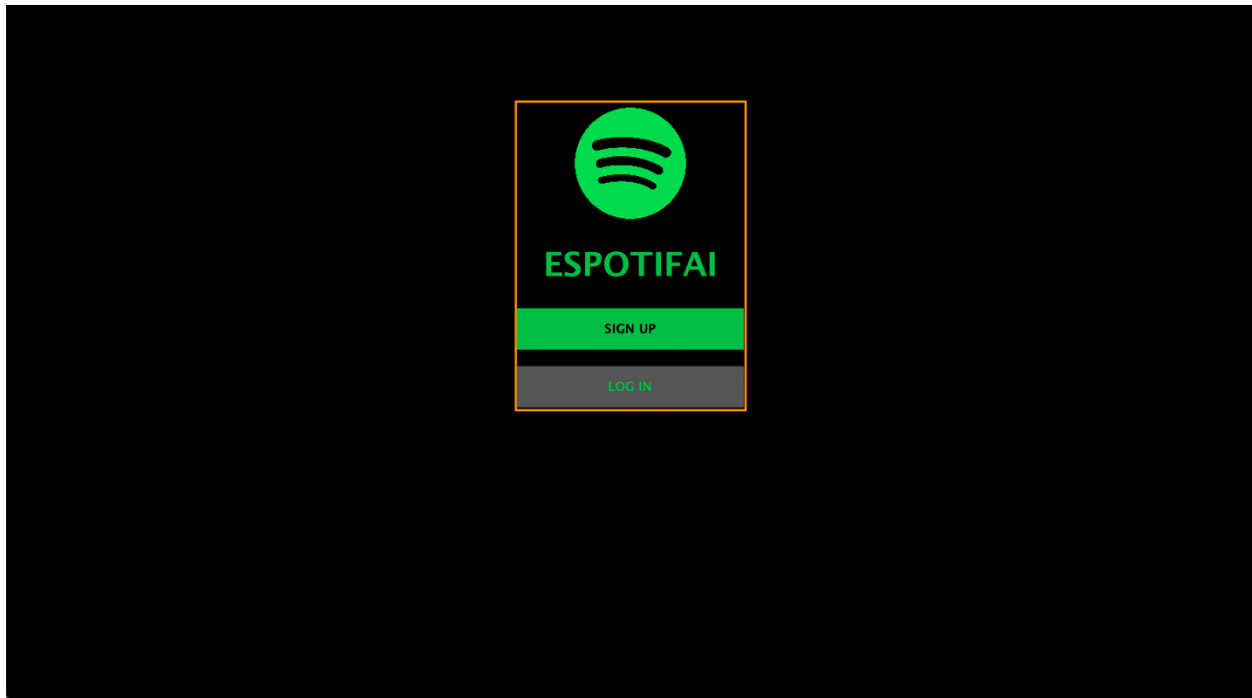
Regarding the top bar, this one would also need to be hid in one of our views (*Initial Screen UI*). For the rest, this one could change its contents by showing either an arrow, the settings icon, or both. We want to display the arrow to allow the user to go back on screens such as the *Log In* and *Sign Up*, when the user hasn't yet begun their session and shouldn't have access to settings. On the other hand, when the user is in the *Main Screen*, we don't want to allow them to go to the previous screen, so we will only show the settings icon. Lastly, if its none of these cases, we will show both.

Furthermore, we also use the above-mentioned class to display all the pop-ups of our application, which either show errors to the user, print success messages, or display different options we need the user to choose from. Some examples are displayed below:



Noticed how we included the icon of our app in these messages.

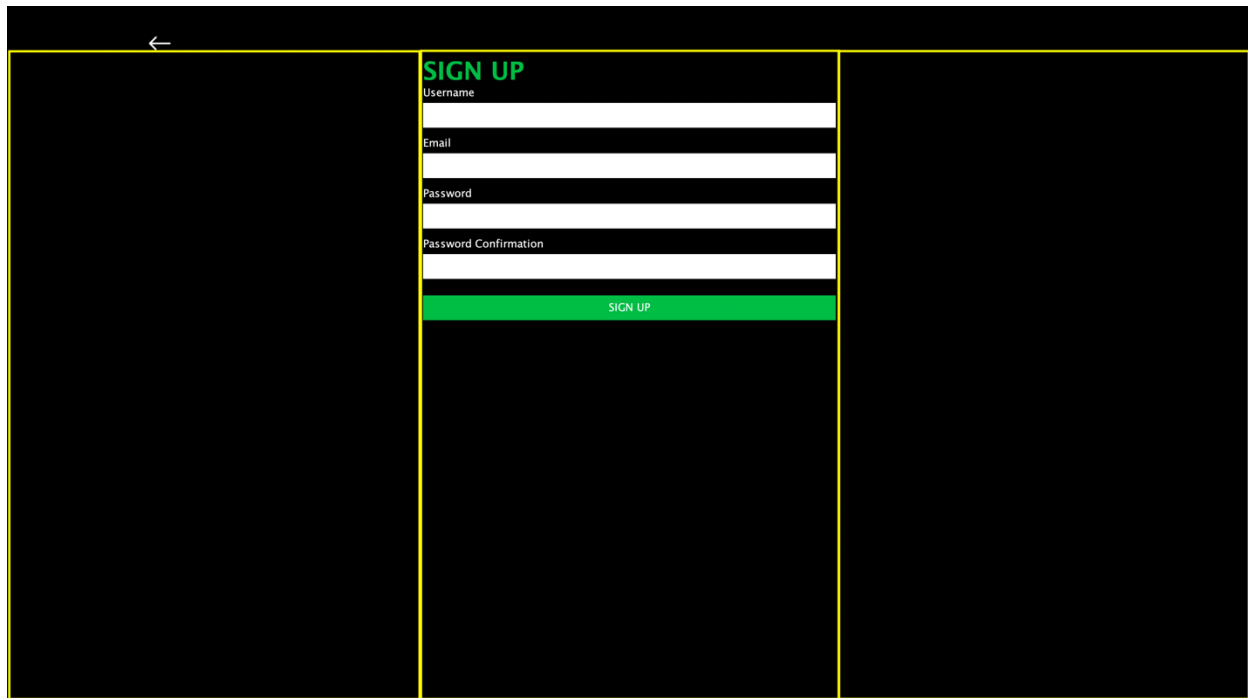
## Initial Screen UI



When the user first comes in contact with our application, an access menu appears where he can choose between signing up or logging in. The placement and color of the components was not arbitrary as we applied our Design and Usability knowledge. We placed all our elements in the middle as we opt for a clean and concise look and chose a green button for signing up instead of grey to unconsciously guide the user towards creating an account instead of logging in, the more the merrier!

In terms of Swing, we used a Box Layout as we wanted to arrange our components in a single vertical line. The reason why we chose this layout instead of a Grid Layout, for example, it is due to the fact that with a Box Layout we can have components with different sizes whereas with a Grid Layout all the elements have the same width and height. Additionally, in order to be able to correctly resize the layout, a Box Layout adjusts the size of the components based on the available space, making the design more responsive when resizing the window. Lastly, to place both JLabels (logo and title) and both JButtons we added rigid areas which were also handy to create separations between the mentioned components.

## Sign Up UI

A UI mockup for a sign-up screen. The screen has a black background. At the top left, there is a white back arrow. The main content is centered in a vertical column. It starts with the text "SIGN UP" in green. Below it are four white input fields with black labels: "Username", "Email", "Password", and "Password Confirmation". At the bottom of this column is a green button with the text "SIGN UP" in white. The entire form is enclosed in a thin yellow border.

If the user decides to create an account, he will be redirected from the previous screen to this one. In order to create an account, the user must enter a valid username, email and password. These consist of a unique username, a unique email that follows the standard email format and a password with 8 characters (minimum 1 capital letter and 1 number) that matches the confirmation password. If any of these considerations is not meant, a series of pop-ups, with the particular error, will appear. In response, the contents of the Password and Password Confirmation fields will disappear to protect our user's data.

The layout used to achieve this design is a Grid Layout as a way of placing everything in the center in a single vertical line. To create this illusion, this layout has three columns and one row, being the second row the actual content and the first and third one rigid areas. Moreover, in the center we added a Box Layout with a page axis format to place the vertical content. For the title and subtitles we used a JLabel and JTextField for the user's input. With the use of rigid areas, we were able to create separations so that the view would not appear as compact in between the different fields. Lastly, mouse listeners were added to the JButton to sign up to with the objective of making the view more interactive and joyful.

## Log In UI

If the user decides to log in into his existing account, he will be redirected from the access screen to the one found above. In order to log in into an account, the user must enter a valid username or email and the corresponding password linked to that account. If the username does not exist, the password is incorrect or a blank field is found, an error will appear. In order to improve the user's experience, we added a direct link from this view to the Sign-Up view so the user is able to reach directly if, for example, the incorrect button was pressed.

The layout used to achieve this design is a Grid Layout (marked in orange on the screenshot above) as a way of placing everything in the center in a single vertical line. To create this illusion, this layout has three columns and one row, being the second row the actual content and the first and third one rigid areas. Moreover, in the center we added a Box Layout with a page axis format to place the vertical content (marked in red). For the title and subtitles we used a JLabel and JTextField for the user's input. With the use of rigid areas, we were able to create separations so that the view would not appear as compact in between the different fields. Lastly, mouse listeners were added to the JButton to log in with the objective of making the view more interactive and joyful.

## MainScreen UI



This is the first screen the user sees whenever they log into our app, from which they have the option to access the rest of the views by clicking one of the option buttons. As we previously mentioned, the top bar (including the settings icon) and the player are fixed in the screen from this point onwards, so the user can access them from any view. Also, as these are managed by the *FrameController* the MainScreen UI only comprises the middle part in between the top and bottom yellow lines.

The main layout we used for this UI is a Border layout within a JPanel that allowed us to define the margins you can see on both sides of the screen. We achieved this by adding a rigid area to the *WEST* and *EAST* regions. We took this approach, as opposed to the Grid Layout we defined in many other views (with one row and three columns), as we needed the center component to be larger than the sides, which we couldn't have done with the grid (as all cells are forced to have the same size).

Regarding the center part of our view, we filled it with another JPanel with a second Border Layout. We followed this procedure as we wanted to have a dedicated (smaller) space for the title on top and use all the remaining screen to fit our option buttons. Therefore, we set the title on the *NORTH* region of the new layout and put the buttons on the *CENTER*.

To split the title in two lines we needed to use two different JLabels (as each have a different font). We also wanted to leave a little space between the top bar and the *Welcome* message. Hence, we decided to place these components in a Grid Layout of three rows and a single column



(painted in orange on the screenshot above), filling the first cell with a rigid box to fill the blank space). We believe this was the right decision as having equal spaces between the components makes our view cleaner and more organized. We could have achieved a similar result by using a Box Layout and try to center it to the left. However, we usually get problems when centering components to a specific side, so we took this approach instead.

We filled the *CENTER* region of the inner Border Layout with a new JPanel using a Flow Layout. We chose this one as we needed the components to be placed in a horizontal line. We first thought of doing it with a Grid Layout, however, this one would have forced us to make the spaces between buttons as big as them. We believed that would make our application less appealing to the user, so we stuck to the Flow Layout.

To achieve the final look, we added the buttons (with their predefined size) as JButtons components to the layout by also inserting a rigid box after each of them. This way, we assured we leaved a little room in between buttons achieving a more professional look.

## Add a Song UI

The screenshot shows a mobile application interface for adding a new song. The screen is divided into three vertical sections. The left section is a large empty black area. The middle section contains a form with the following elements: a green title 'ADD A SONG', a subtitle 'Share your music with the world!', and input fields for 'Title', 'Genre' (a dropdown menu with 'Please select' and a blue arrow), 'Album', 'Artist', and 'File'. Below these fields are two buttons: a grey 'SELECT FILE' button and a green 'ADD SONG' button. The right section is a large empty black area. At the bottom of the screen is a dark grey playback bar with icons for previous, play/pause, and next, along with a progress bar showing '00:00'. On the far left of the bar are 'Song Repeat' and '00:00' labels, and on the far right are '00:00', 'Playlist Repeat', and a square stop icon.

This screen allows the user to add a new song to the application by first filling in some information about as well as uploading its .mp3 or .wav file.

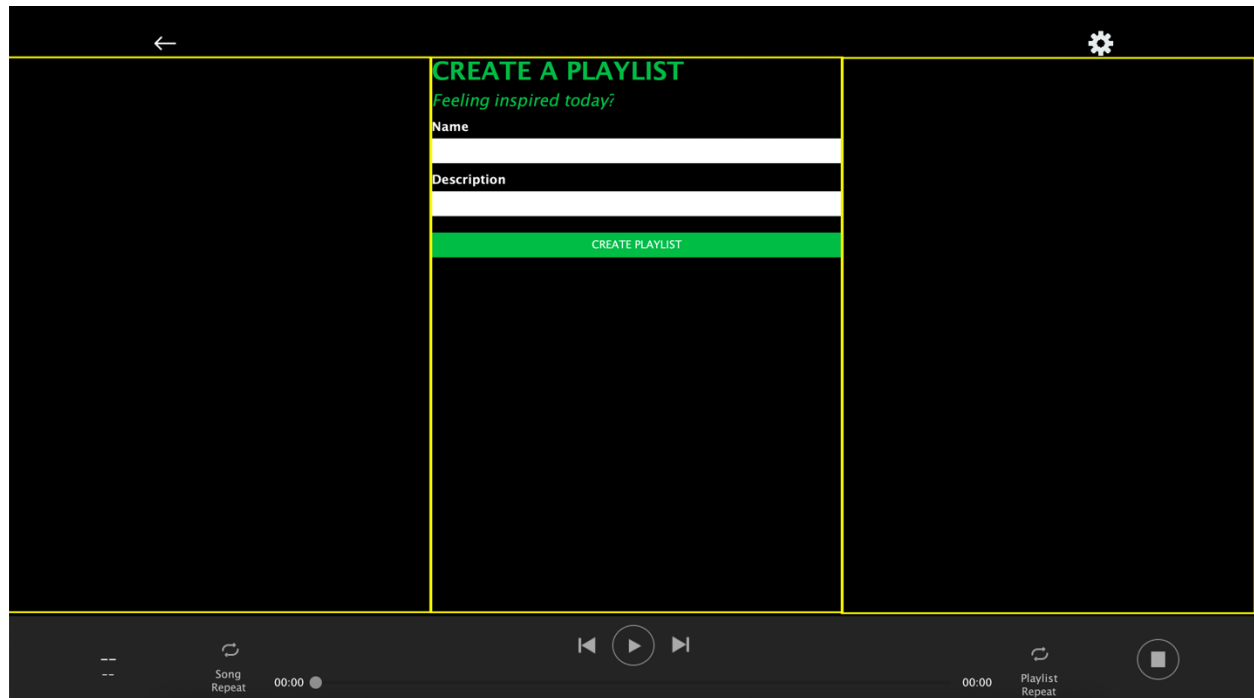
We used a JPanel with a Grid Layout to achieve the final look of this view. We chose this approach as we wanted all the fields to be centered in the screen while having margins on the sides. On the previous view, we couldn't use this layout as we needed the center to be larger than the margins. However, in this case, we believed it look well compensated to have the screen split in three. Moreover, as we are not using a fixed rigid box for these, the view is completely resizable and adapts the size of the grid to the screen.

We filled the side cells with small rigid boxes (just because we needed to place the components in order), but these don't affect the size of the grid. For the center cell, we created a new JPanel with a Box Layout to place all the components in a vertical order and let them expand to the maximum size of the cell.

We used multiple JLabels for the titles and JTextFields for the information we needed the user to enter. Also, as we just want our user to pick a genre from a given set (so that we can correctly sort all the songs in statistics) we added a JComboBox that lets them choose from one among a drop-down list. Lastly, we needed the user to upload the song file. To do so, we linked a FileChooser to the *SELECT FILE* JButton so that the user gets a pop-up screen that allows them to select the file from their laptop. We added all these components sequentially (in the desired order) into our layout to achieve the desired format. Once the user presses the *Add Song* button,

which we will know by the listener we attached to that JButton component, we check if the user has entered all the fields. We added all these components sequentially (in the desired order) into our layout to achieve the desired format. Once the user presses the *Add Song* JButton, which we will know by the listener we attached to that JButton component, we check if the user has entered all the fields. If they haven't, we prompt an error message. However, once all the inputs are inserted and are valid, we add the song to our system and display a success message.

## Create a Playlist UI

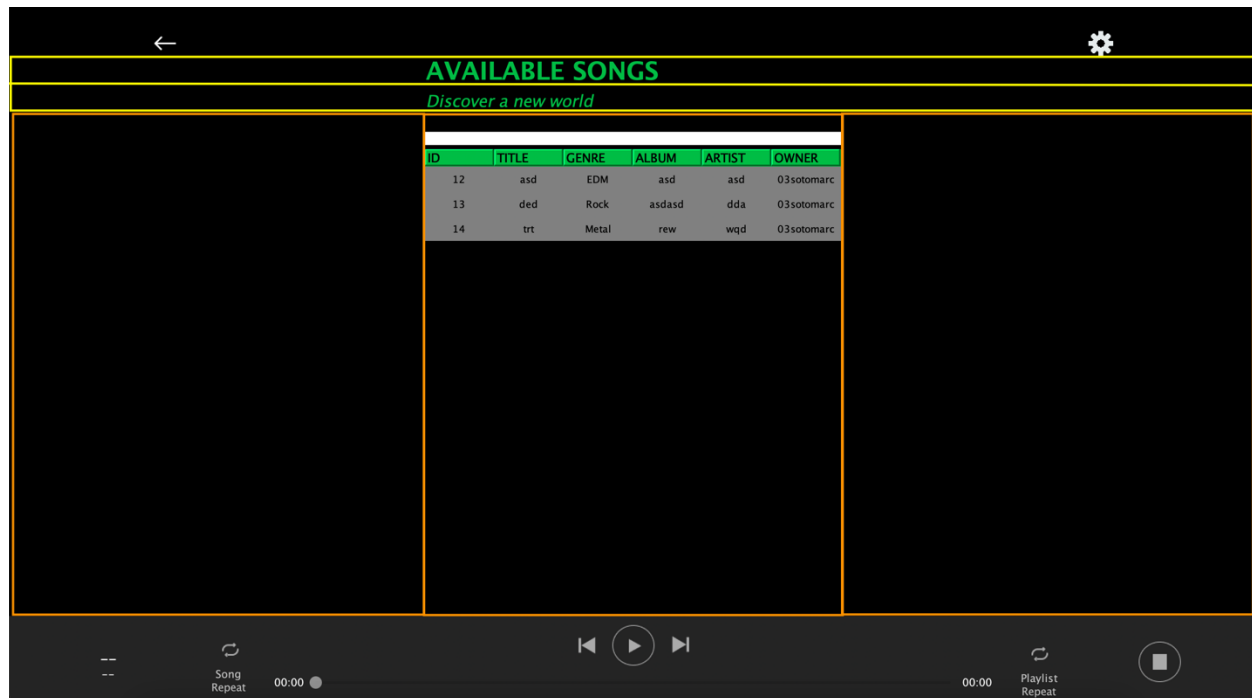


This view allows the user to create a new playlist with a given *name* and *description* so that they can organize their music by genres, mood, similar artists...

In terms of UI, this view is quite similar to the one we just discussed. We separated the content of the screen in three different columns by using a JPanel with a Grid Layout. By placing all our components on the middle column and leaving the other two empty, we achieve this clean and minimalistic look we want our application to have.

To distribute the different components of the view within the central column we decided to use a JPanel with a Box Layout, so that we could place the components vertically and make them expand to the maximum size of the cell. The only components that were required for this UI were JLabels for displaying the titles, JTextFields for allowing the user to enter new information, and JButtons to add functionalities to the screen. In this case, the JButton checks if both fields have been entered and creates a new playlist with those attributes, redirecting the user back to the main screen once the process is completed successfully.

## List Available Songs UI



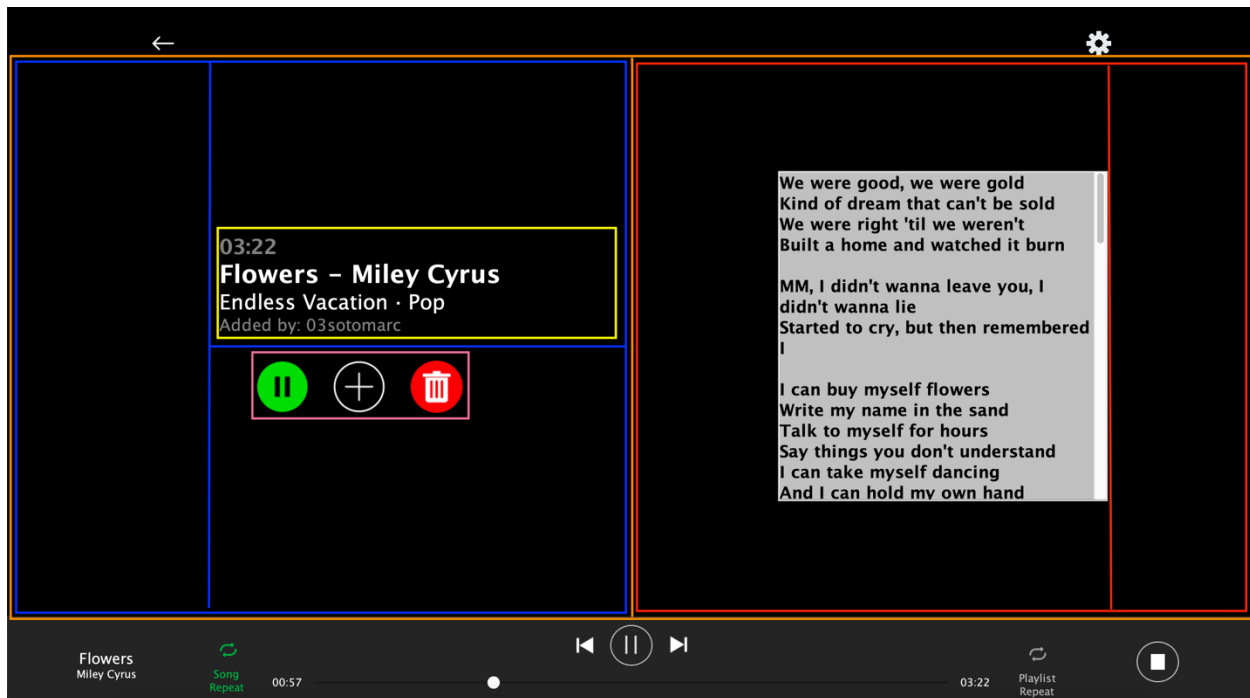
If the user wants to view all the stored songs, he will come to this view from the main screen or when adding a song to a playlist. With the objective of making the user's life effortless, we implemented a search bar to go through the contents found in the table. These contents are the following, song id, title, genre, album, artist, and owner. Once the user clicks on a cell in the table, he will get redirected to the detailed view displaying the key information regarding that song.

In terms of Swing, a Box Layout was created for the whole view. However, to place the top labels, as the text was long, we implemented a JPanel with a Grid Layout (marked in yellow in the screenshot above) of two rows and a single column, where we added both the title and subtitle JLabels of the view. Then, we added a second JPanel with a Grid Layout of a single row and three columns (marked in orange) so that we could place the table at the center and leave the borders of the screen empty. To add the corresponding table with its search bar, we added a last JPanel with a Box Layout to the center cell of the orange grid.

To actually implement the table, we created a class called JTableModel which, as the name suggests, extends from DefaultTableModel. This class was useful in order to create the actual JTable for each UI that needed one. Hence, we reused the design for all of the tables and updated them with the use of each controller. This design was based on the colors chosen for this project, the header is green and black while the table is grey. Simulating the colors used for the JButtons placed in the main screen. Lastly, to help the user find the desired song easier, we added at the top of the table a search bar to go through all of the contents stored.

This search bar consists of a JTextField with a TableRowSorter and a DocumentListener. The user enters the desired filter through the JTextField and the DocumentListener will search through all the table to find the songs that match that criteria while the TableRowSorter will make sure only the correct songs get displayed. Once the user has found the fitting song, we will get redirected to the UI found below.

## Song Information UI



If the user wants to play a song, or simply see the song's details he will come to this view. Inside, he will be able to see from the name of the song, duration, artist, album, owner and genre to the song's lyrics. Also, the user can play the song, add it to a playlist or delete it. If either one of these actions the corresponding tables will be updated with the new data, as mentioned previously. The user will be aware of it as a pop will appear.

To distribute the different components a grid layout of 1 row and 2 columns has been used (marked in orange in the screenshot above), to split the screen in half and for both sides to be resizable. Leaving the right half to the lyrics area and the left to display the information of the song and the different buttons to make any of the actions previously described.

In the left grid, a panel with a grid bag layout of 2 rows and 2 columns has been used (marked in blue). Firstly, a rigid area has been added so the information and the buttons have some separation with the left border of the screen. After, a panel with a box layout vertically aligned (marked in yellow) has been added in the first row second column, to place all the information of the song: duration on the top, followed by the title and the artist in the same row separated by a dash, then the album and the genre in the same row separated by a dash, and finally the name of the user who add this song to the system.

Lastly, a panel with a flow layout horizontally aligned (marked in pink) has been used to place all the buttons and has been added in the second row second column. In that panel the play/pause button is added at the start, followed by the add to a playlist button and, at the end, the bin

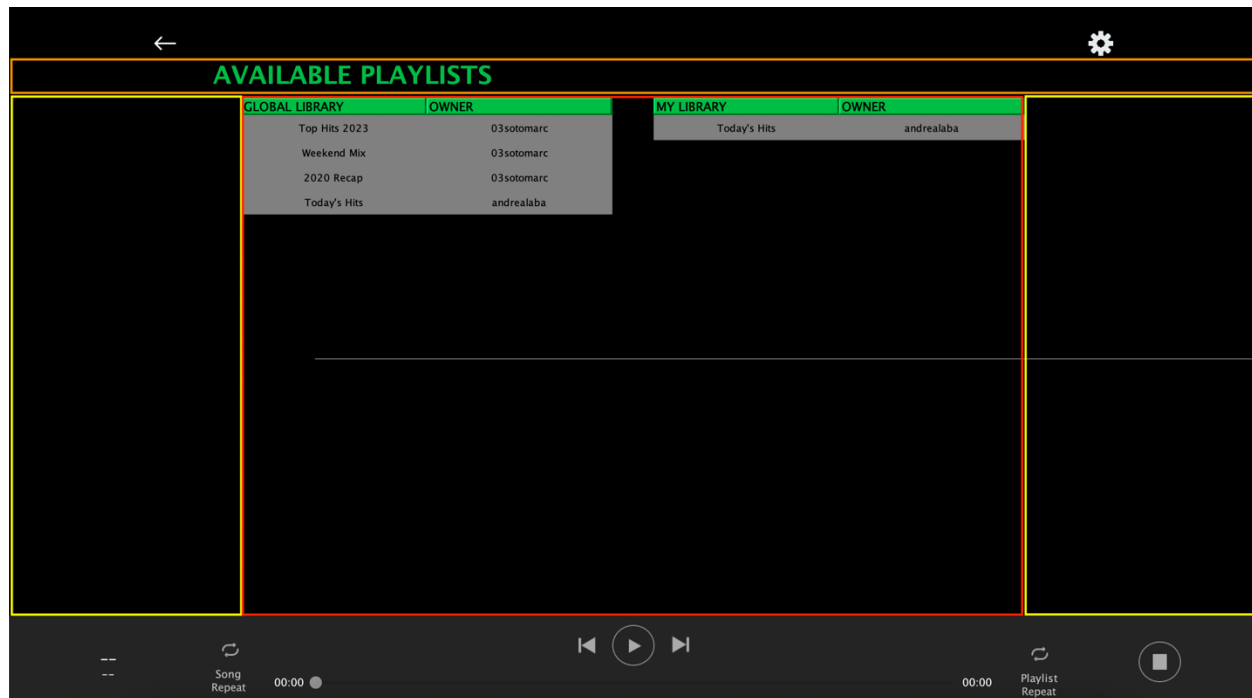
button to delete a song is added if the user who entered this screen is the user who uploaded the song to the system.

In the right cell of the outer grid, another panel with a grid bag layout has been used but being this of 1 row and 2 columns (marked in red). Adding in the first column a scroll pane with a vertical scroll bar only when needed and a text area attached to it, so the lyrics fetched from the API are placed in the text area and a vertical scroll bar appears if the lyrics are bigger than the space available, so the user can scroll down to continue reading the lyrics. After, a rigid area is added to the second column, to leave a separation between the lyrics scroll pane and the right border of the screen.

When trying to add a song to a playlist, a pop up will appear using a JDialog. This popup was designed using a JPanel with BoxLayout (page axis format) centered in the middle of the JPanel. Inside this BoxLayout we placed a JLabel to indicate the user he should select a playlist. To display all of the available playlists we used a Grid Layout with one column and a series of JRadioButtons with the name of playlist to let the user select the desired one. Moreover, a JScrollPane was used for when there's more playlists than the ones that can be seen in the pop up hence, the user can scroll. Lastly, the user will confirm the selection by using the JButton.



## List Available Playlists UI

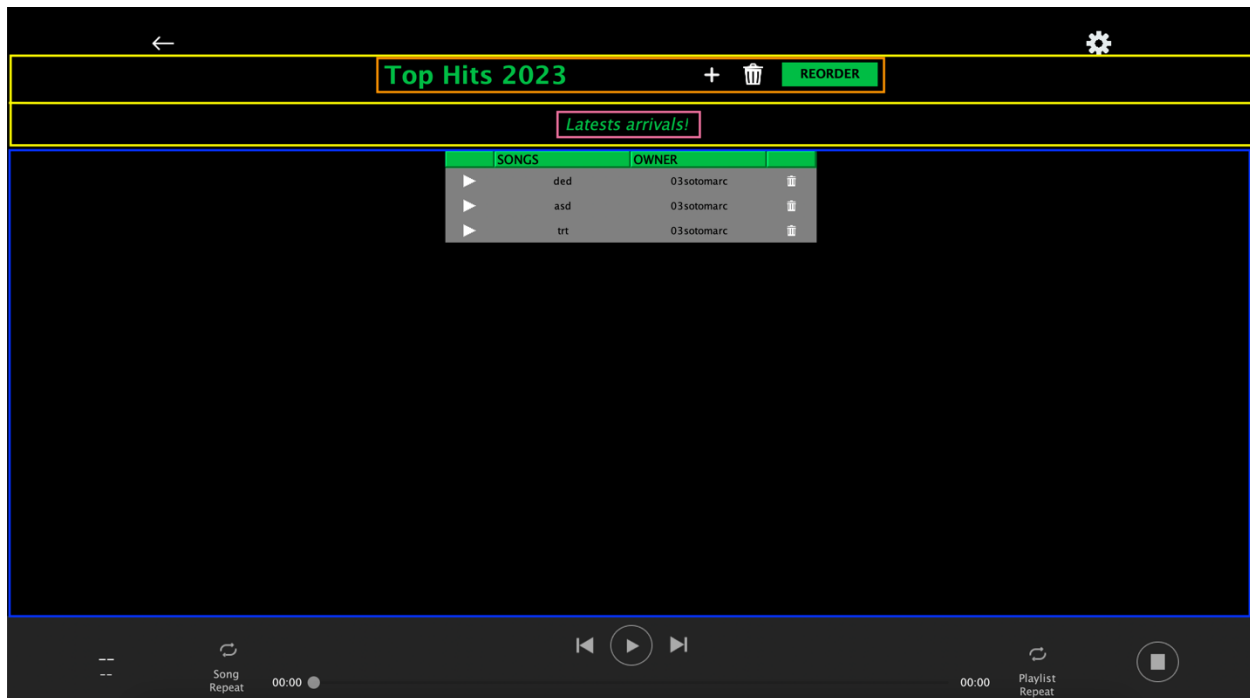


If the user wants to view all the available songs either his or globally, he will come to the list available playlist view from the main screen. With the objective of making the design clear and concise we placed two tables, one for the user's playlists and one global one. When the user clicks on a cell in the table, he will get redirected to the playlist information view in order to get all of the details and play a song.

In order to implement the table, we created a class called `JTableModel` which, as the name suggests, extends from `DefaultTableModel`. This class was useful in order to create the actual `JTable` for each UI that needed one. Hence, we reused the design for all of the tables and updated them with the use of each controller. This design was based on the colors chosen for this project, the header is green and black while the table is grey. Simulating the colors used for the `JButtons` placed in the main screen.

Moreover, a `JPanel` with a `Border Layout` was implemented to be able to add some rigid boxed to the *SOUTH* and *WEST* regions (marked in yellow) and leave some space at the borders of the screen. Also, we added a `Flow Layout` to place the tables side by side with a rigid area to create separation. This `Flow Layout` was then added on to the `Border Layout` in the *CENTER* region (marked in red). Lastly for the `JLabels`, we implemented another `Flow Layout` to place both titles and a rigid area (marked in orange), so it did not look as compact. Then this layout was added to the north of the `Border Layout`. When a cell is clicked, the user will be redirected to the desired playlists.

## Playlist Information UI



This screen shows the user all the information of a specific playlists. This being: its name, description, and songs it contains. It also gives the user the option to add songs to the playlists or delete the playlists. Moreover, the user can remove any song from the playlists, play it, or reorder them. As you can see, we selected to play the song Flowers (pause icon on second row) and this one started playing on the bottom bar.

As we wanted to split the view in two different parts: the title and the table with the list of the songs, we created a JPanel with a Border Layout. We used this one instead of a Grid Layout as we needed both parts to have different sizes. Then, we added the desired components to its *NORTH* and *CENTER* regions.

Again, we wanted to divide the *NORTH* region in two different lines: one for the name of the playlists and the option buttons and a different one for the playlist's description. Hence, we created a JPanel with a Grid Layout of two rows and a single column (marked as yellow in the screenshot above) and added it to this region. We couldn't use a Box Layout in this case as we wanted both components to be placed differently.

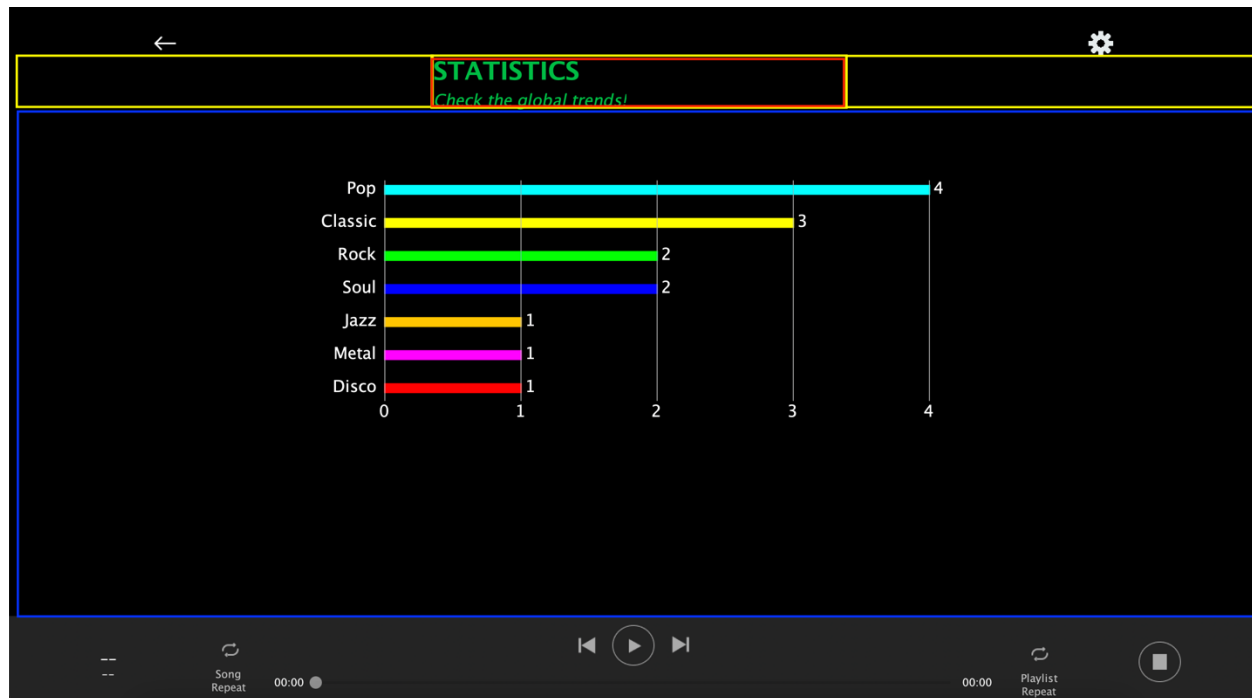
Wanting to place more than one component on the first row of the top grid forced us to create a new JPanel with a Flow Layout so that we could add this one next to the other one (marked with orange), adding a rigid box in between the name and buttons to leave the desired space. Then, as we wanted to place the description right below the name, we created a JPanel with a Box Layout that allowed us to place the playlists description in the middle. We could have only achieved the same look with a Flow Layout by forcing the separation between the description

and the screen edge with a rigid box. However, that case wouldn't let our application resize as expected. Hence, we opted for this approach that allows the user to resize and still places the description where we wanted it to be.

Lastly, we wanted the *CENTER* region of our Border Layout (blue) to show the lists of songs in a table. As we couldn't directly place this component into the Border Layout, we created a JPanel with a Flow Layout to which we added the table. Then, we placed this one in the desired region.

To display the songs in the table and add the options to play/pause and delete them, we used the above-mentioned table component we created and added the play/pause and remove icons to the side columns. To create all the buttons of this view we used JButton components and customized them accordingly, by giving them a title and a color (such as the reorder button) or loading an icon (such as the delete or play).

## Statistics UI



This view lets the user see graphically how many songs there are for each genre. In concrete, it displays a bar chart that shows how many songs are stored in the system for each genre, until a maximum of ten genres (the ones with most songs).

As in this view only the title and a bar chart were needed a JPanel with a border layout was used to split the top part (NORTH, marked with yellow) where the title and a phrase must be and the rest of the screen where the chart is (CENTER, marked with blue). Although some other layout could have been used to split the screen in two, the border layout was used as we want the top part to have the minimum space needed, to leave the rest of the screen for the bar chart to be the biggest possible.

In the NORTH of the layout, a grid layout with 1 row and 3 columns is used to place the title and the phrase in the middle column and 2 rigid areas on the side columns. We made it like that so when the screen is resized, the rigid areas on the sides and the title get resized as well, so the title is always on the middle no matter the screen size. Finally, in the middle column as we wanted to put a title and a phrase, we used a box layout aligned vertically (marked in red), as we want the title to be on top and the phrase to be slightly below. Also, between the title and the phrase a small rigid area was placed to have more separation between both labels.

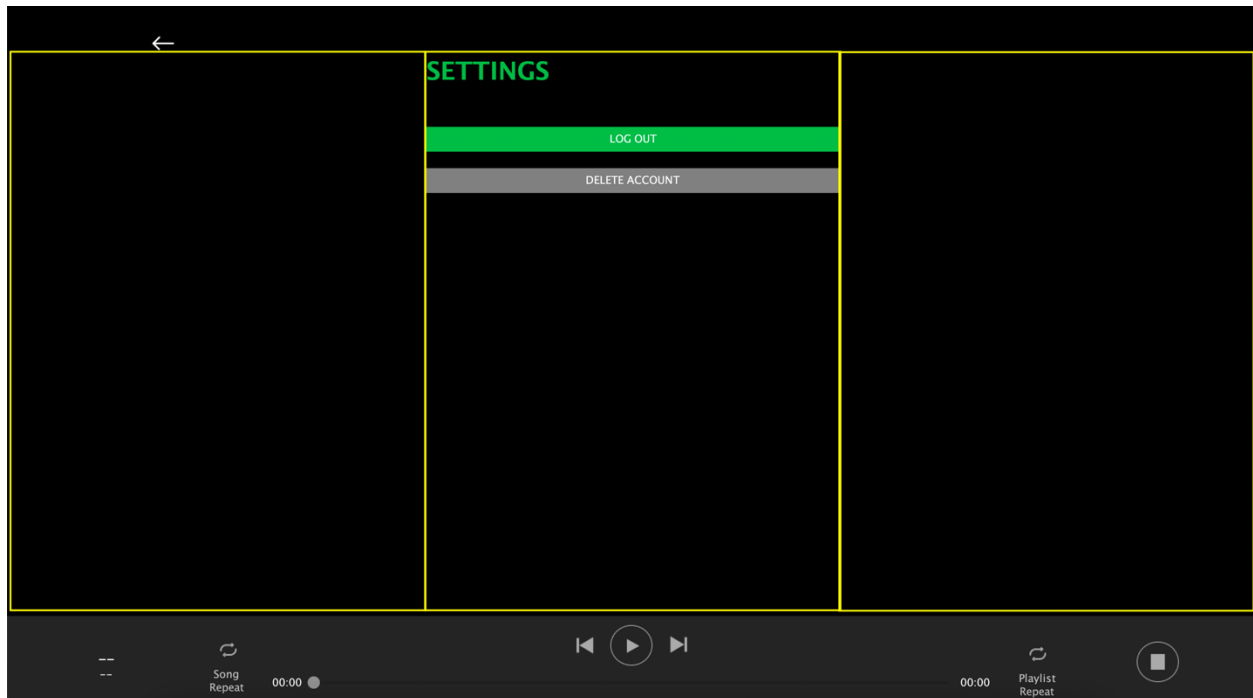
In the CENTER of the layout, a box layout is used again (marked in blue). Here, a lot of layouts could have been used as the bar chart is going to be drawn into another panel using the graphics2d class. And once drawn, this panel is going to be added to the box layout.

As said before, to draw the bar chart, the `graphics2d` class is going to be used. With this class, once the genres and the number of songs is known, then the names of the genres will be drawn on the y axis, from the top left part of the chart to the bottom left part, starting to draw them at different distances, so that they all end just before they y axis, and none crosses it.

Once the genres are drawn, one rectangle of distinct color and at the end of them the number of songs for each genre is drawn next to the genres (where the y axis starts and at the height of the genre they represent). Each rectangle is drawn proportional to the number of songs of the genre they represent, so the one with the most songs has a length of 500 pixels (for example), and a genre with half the songs has a rectangle with half the length.

Finally, once the genres, the rectangles and the number of songs of each are drawn, different vertical lines will be drawn to divide the chart for a better understanding. To do this, before drawing them the maximum number of songs is checked, if it is divisible for one of the numbers the program supports, then it will draw the same number of lines as that divisor. If the maximum number of songs is a prime number non divisible for any of the divisors stored in the system, then it will add one to the maximum number of songs and will check another time for the divisors, but now being an even number. Moreover, the number of songs that would represent a bar going from the start until any of the lines is drawn below the lines.

## Settings UI



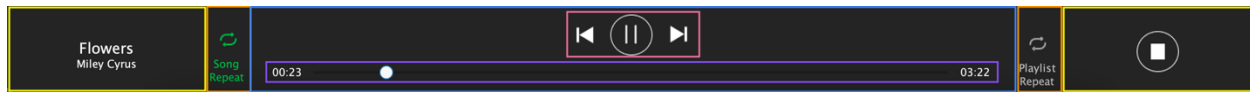
This is one of the simplest views of our app, as it is groups only three components: a JLabel displaying the desired title and two JButtons that give the user the option to log out of our application or delete their account, which would remove all their songs and playlists from the system.

Following the same approach as we have seen in multiple of our other views, this UI uses a JPanel with a Grid Layout to structure the screen in three columns and a single row. We did this to better organize our application and give it a minimalistic and clean appearance.

To place our components in a vertical way we used a JPanel with a Box Layout placed at the center cell of the grid. Like this, we were able to add the different components sequentially (with some rigid boxes between them to leave the correct spacings) and achieve the desired look.

Also, we decided to make the log out button green to give it priority over the gray one, as we don't want our users to feel visually attracted to the second button and delete their account accidentally.

## Player UI

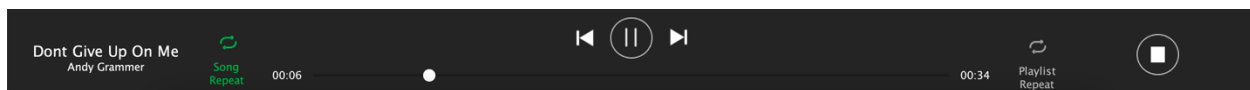


This is the UI that allows the user to interact with the music that is playing from any given screen. From here, the user can view the name and artists of the current song, select to repeat the song/playlist, rewind, pause/resume and skip the song; or completely stop the playing (right hand button).

To distribute the components as desired, we created a JPanel with a Border Layout. On its *EAST* and *WEST* regions, we placed the yellow components. The rest of them are located in its *CENTER*. We decided to use this type of layout as we needed to structure the view in three blocks, always having the player at the center and have this one bigger than the sides. Therefore, it wasn't possible to use any other layout (apart from Grid bag) to achieve this, as a Flow or Box Layout wouldn't allow us to combine between components placed horizontally and vertically or fix the player at the center. Also, a Grid Layout would only work in this case if the sides and the center had the same size.

Both sides use the same structure, a JPanel with a Box Layout. We took this approach as we needed to arrange all the components vertically (the rigid box used for leaving space between the top of the player and other components, the two JLabels used for displaying the song title and artist and the JButton used for stopping the music playing). Also, as we wanted to leave margins between these components and the edge of the screen, we aligned them to the middle of the layout. To ensure everything is fixed in the screen, we used the `setPreferredSize` method on both of these layouts to define the space we wanted them to occupy.

We initially achieved a similar result by using a Grid Layout with three columns and one row. This approach worked perfectly for the right corner. However, we realized that those songs with a longer title weren't fitting right in the screen and couldn't expand to the entire size of the left component (as they were placed on the center cell of the grid and this one was fixed). Therefore, we decided to remove this and arrange the UI using Box Layouts, which allows us to have long song titles as you can see below.



Moving to the *CENTER* region (blue and orange sections), as we still had to place many components and we wanted the loop buttons to be at the sides, we added another JPanel with a Border Layout so that we could place them at the *WEST* and *EAST* regions (marked with orange in the screenshot above). As both required a JButton and a JLabel to correctly name each functionality, directly placing them onto the desired regions wasn't an option. Hence, we made

the orange sections be a JPanel with a Box Layout to be able to correctly add these two components vertically, `3align` them and let them expand to the maximum size of their cell.

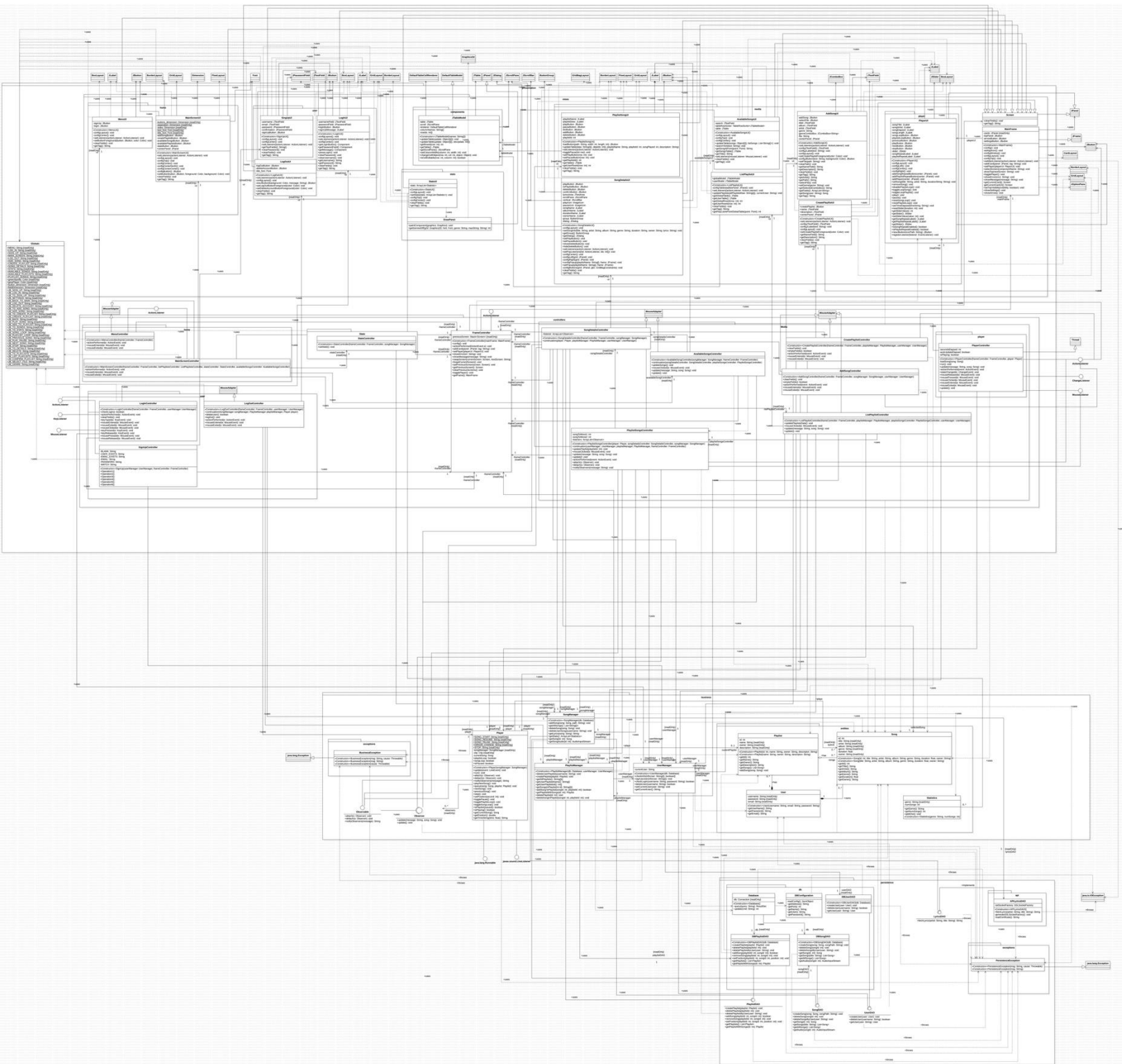
Also, it is worth mentioning these get painted green whenever they are active and gray when they aren't (see picture above where song loop button is active and the playlist one isn't). These buttons allow the user to keep playing their favorite track on repeat or loop their dearest playlist.

Lastly, the missing components were added to the *CENTER* region (marked in blue) in a Box Layout so that we could place them vertically (first the control buttons and then the player slider). Both are a set of components: the control buttons consist of three JButtons and the slider has the slider itself and the tags displaying the song length and the elapsed time. Therefore, we created a horizontal Box Layout for each of these so that we could place the components accordingly (purple and pink groups) before adding them to the external Box Layout (blue).

These buttons allow the user to pause/resume the current song. Also, they can restart the song by pressing the top left button if they have missed their favorite part or skip it if they come across a song they don't like (or have already heard too many times). Furthermore, they can check the duration of the song at the right side of the slider. To see how much of the song they have already heard, they can check the elapsed time (left side of the slider) or check the status of the slider, as this moves accordingly.



## Class diagram



Note: as the diagram is not readable, we have included it in the ZIP folder.

The diagram features three main packages that describe the layered architecture pattern we have strictly followed in the development of the project, with the integration of the Model-View-Controller design pattern. These are Presentation, Business and Persistence. The Presentation layer focuses on allowing the program to communicate with the user, in our case, the Presentation layer controls all aspects related to the UI and communicates with the Business layer to perform on-demand logic that will enable the different functionalities the program offers. The Business layer typically holds the core logic of the program, however, as our program is mainly based on user interaction, the Business layer serves more as a layer that allows proper communication between the Persistence layer and the Presentation layer. The logic in the Business layer mainly processes data obtained directly from the Persistence layer, with few exceptions, such as the Player, which features a more independent logic from the data. Finally, the Persistence layer is in charge of allowing CRUD operations on the set of data our program requires to operate, the CRUD operations occur inside DAOs which are linked to entities representing a piece of data.

Looking into the Persistence layer, our program required persisting to an SQL database, we created a DAO implementation for almost each table in the database: DBUserDAO, DBSongDAO and DBPlaylistDAO. Each of these is able to perform CRUD operations related to their entities. The only attribute they have is of type Database which allows the physical communication with the database. Database is a class that establishes communication with the database and allows different operations, such as queries and updates, through its methods. The initial connection is done in its constructor with the help of an instance of DBConfiguration, this class reads the *config.json* file that holds the database connection data.

Apart from accessing a database, our program required communication with an API. To accomplish this, another DAO called LyricsDAO was created. Communication between the Business layer and the Persistence layer is done through interfaces extracted from the DAOs, this isolates our current DAO implementation from Business allowing the Persistence layer to be easily maintainable and scalable. Finally, Persistence has a PersistenceException class that forwards a user-friendly cause of any exception thrown in the Persistence layer, should any rise.

Next up, the Business layer, which powers the main logic of the program as the Model component of the program, has a manager class per entity. As explained before, the entities only serve as means of holding information, meaning that all their attributes only store data. Which is why none of their relations are direct associations as these classes are not dependent on any others. Due to the fact of, once again the entities only being means of storing information in the RAM, the managers do not need to hold these as attributes as they only need to be used in methods, meaning that all the manager to entity relations are associations. However, this is not the case with the Player to Playlist and Player to Song relation, because the player needs to hold which song is playing at all times together with the information of if that song is being played from a

playlist and from which playlist, which is why these two relations are aggregations. Where the relations differ is in the entity-to-entity relations, where for example playlist has songs and can also not exist without songs which is why they have an aggregation, for reasons such as these other entity-to-entity relations are aggregations and compositions.

In the presentation all the views have direct associations to component classes as and a few associations, this is because the UI classes have components that change and need to be returned for listener motives so they are direct associations, on the other side other components such as JLabels which are generally only used in methods, are only associated to the UIs, all UIs also extend from a custom class: Screen, the reason for this is so that the main frame can move between views in the cards panel and detach itself from having to have all views as attributes, and not have to do cases for each view by using abstract methods. Each UI class has its controller class which is directly associated to it as the controller needs to have the object of the UI to call changes on it, all the controllers are also connected to the main frame controller since an error message is not unique to a view, these are handled by the frame controller, so when an exception happens the controllers need to the object of the frame controller so that it can call on the main frame to display the error, directly associating the controllers to the frame controller. Also, some other controllers are connected between each other as before going into some views, the controllers need to be notified that their view is about to be entered so some controllers need to be directly associated to each other, obviously without bidirectionalities. Lastly each controller extends or implements many listeners classes in order to listen to their UIs components and in some cases components from other UIs, which is why the controllers have many generalization and interface realizations connections with listener classes or interfaces.

## Development methodology

### Sprints distribution

To ensure we efficiently managed our time and kept a satisfactory working rate through the development of the project we broke it down into different tasks and distributed them through four 2-week sprints. This methodology allowed us to organize ourselves as a group and distribute tasks to clearly see what stage of the project we were at.

During the first sprint of the project, we designed the UIs of our application using Figma (which we then took as a reference when coding them in Java), planned and implemented the database we use to store all the information regarding users and their music and created the first version of our diagram. This one has been updated multiple times during the course of the project. Thinking about the initial classes we would need, their responsibilities, relationship with others, attributes, and methods was extremely useful for start coding the project.

Regarding the division of tasks among the group members, we decided it would be best to brainstorm the main ideas together so that we agreed on a start point. Once we had already talked about the main tables our database would have, the classes we would need to add to our diagram, and the look we wanted our app to have; we took a divide-and-conquer approach to perform the tasks more efficiently.

We followed a similar method for the next sprints, always agreeing on the main ideas together and then splitting the tasks so that each of us could work on some functionality independently and then catch up during and at the end of each sprint to evaluate our progress. Moreover, we helped each other whenever a problem was encountered so that we could fix this one quicker and continue working.

On the second sprint, we already started coding the project and creating the different UIs. We began with the main functionalities to build a solid base to which we could keep adding new functionalities progressively. By the end of this sprint, we had already implemented the *Log In*, *Sign Up*, *Log Out*, *Main Screen*, and *Add Song* screens with all the logic behind them.

The third sprint of the project was a week longer than the rest. We took advantage of this to implement as many tasks as we managed to. At this point, we had already implemented most of the views and their functionalities, being able to play music, add songs to playlists, list the available songs, among others.

Lastly, we used the last sprint to thoroughly test our code to try and find any bugs we might have overseen. During this inspection, we found a couple mistakes and fixed them. Moreover, we started documenting all our steps on this report, updated our class diagram to its final version, as well as properly commented our code following a Javadoc structure.

## Tools and resources

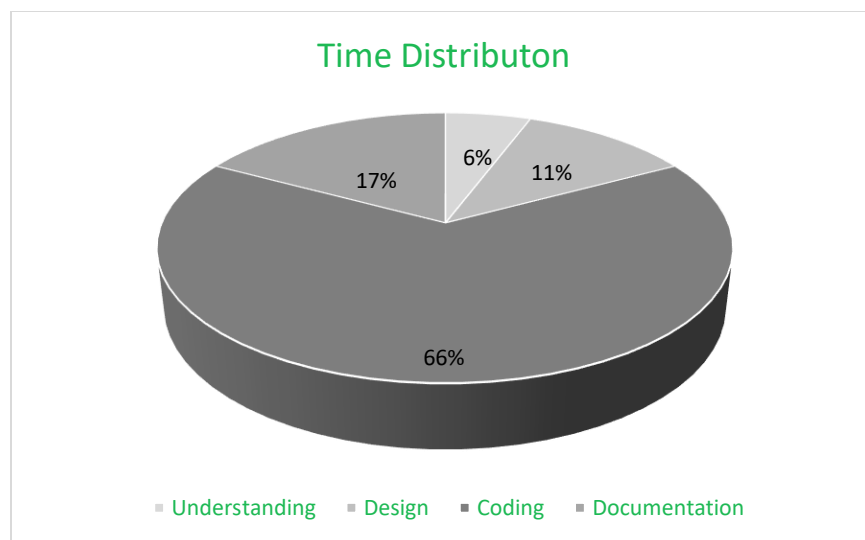
With objective of implementing our project the most efficient way possible, we used Jira and Bitbucket, apart from IntelliJ in order to develop the actual project. Jira was key in order to properly distribute the work and see where everybody was at, without having to constantly ask. Thus, it helped us stay organized and collaborate effectively. Moreover, thanks to its integration with Bitbucket there was a constant flow of code between the two tools. Bitbucket provided us a platform to manage our code in a way where we can create branches, commit and merge everything seamlessly. This creation of branches allowed us to work in parallel on separate features, such as bug fixes, and once it was completed the branches could be merged back together.

## Time Costs

Below you can find the specific amount of time we dedicated to each of the following sections during the different sprints and the total duration of each task as well as the amount of time we worked in each sprint. Note that this is an estimation of the actual time we spent as we didn't measure exactly how long each tasks took.

	Sprint 1	Sprint 2	Sprint 3	Sprint 4	TOTAL
Understanding	3H	1H	0H	0H	4H
Design	6H	1H	1H	0H	8H
Coding	1H	15H	25H	5H	46H
Documentation	0H	0H	0H	12H	12H
<b>TOTAL</b>	<b>10H</b>	<b>22H</b>	<b>26H</b>	<b>15H</b>	<b>70H</b>

To better visualize the data plotted in the table above we created the following pie chart that illustrates the percentage of time dedicated to each section



As you can see, the part that took us more time was the coding of the app, as this was the hardest and longest to implement. We worked on this mostly on the second and third sprints, leaving only some minor bugs to fix or aspects to improve on the last sprint, and dedicating around an hour on the first sprint to create and implement our database.

The second part that took us the longest was the documentation of this project, which happened entirely on the last sprint. This was then followed by the design of our application, which we did on the first sprint to have the mockups ready before starting coding.

Lastly, the task that took us the least time was understanding the requirements and functionalities we had to implement. We spent around three hours on the first sprint trying to comprehend the statement identify the functionalities that needed to be implemented. On the second sprint, we also took a bit of time re-reading the statement to ensure we did not forget anything.

Overall, we dedicated around 70 hours to the development of the project and worked especially hard on the second and third sprints. We believe we properly managed our time and worked constantly during the whole project, which allowed us to successfully implement it in the required time.

## Conclusions

The development of this project has helped us reinforce all the concepts learned in OOPD classes and widened our knowledge regarding Swing by applying the theory given in class on a real-life problem.

Designing this application with the tools provided by Swing in Java has taught us how to properly use the different components and allow the user to interact with them. Also, by trying different arrangements for our UIs, we have been able to clearly see the advantages and disadvantages of each of the layouts and, most importantly, understand their limitations and identify when it is better to use each of them.

Also, the functionalities of the project required us to perform several tasks simultaneously for what we decided to use threads. For instance, to make the slider increase every second or make the player wait for a new song to start. This helped us not only to improve our implementation of this process, but also to recognize when these are necessary or useful to apply.

Working on this project has given us the opportunity to put into practice our knowledge of layer architecture design patterns once again; writing a code that respects its principles by separating the project in three horizontal layers (Presentation, Business, and Persistence) and defining the appropriate relationships between their classes. Concepts like polymorphism and decoupling were also used to provide the best quality to our code.

Moreover, the division of tasks in sprints and the use of tools such as Jira and Bitbucket allowed us to organize ourselves as a group and manage our time efficiently by always keeping track of what we had done and what had yet to be implemented. These tools also helped us divide the work and structure our sprints to ensure we maintained a constant workflow that allowed us to finish the project on time.

Overall, we were able to successfully complete the project and implement our Espotifai application that allows users to listen to their favorite music and organize it in playlists. We are satisfied with the achieved outcome and believe it is the fruit of a well-coordinated and hardworking group.



## Bibliography

These are the sources from where we obtained the information to develop this project:

*Oracle Help Center: Java Platform, Standard Edition (Java SE) 8.*

- <https://docs.oracle.com/javase>

*StackOverflow*

- <https://stackoverflow.com/questions/>

*TutorialsPoint*

- <https://www.tutorialspoint.com/index.htm>