# MUSICIFY

**FINAL SEMESTER EXAM PROJECT PROGRESS REPORT**

(Prepared for the Final Exam Project of the First Semester)



**By :**

**Bethania Putri J. N (25031554115)**

**Rajendra Alif A. C (25031554137)**

**Veni Mayline Nababan (25031554197)**

**Lecturer: Hasanuddin Al-Habib, S.Si., M.Si**

**STUDY PROGRAM DATA SCIENCE**

**FACULTY OF MATHEMATICS AND NATURAL SCIENCES STATE UNIVERSITY OF SURABAYA**

**2025**

# TABLE OF CONTENTS

CHAPTER 1 2

INTRODUCTION 2

* 1. [BACKGROUND 2](#_TOC_250009)
  2. [PROBLEM STATEMENT 2](#_TOC_250008)
  3. [OBJECTIVES 3](#_TOC_250007)

CHAPTER 2 4

ANALYSIS AND DESIGN 4

* 1. APPLICATION NECESSITY ANALYSIS [4](#_TOC_250006)
  2. FLOWCHART5
  3. USER INTERFACE DESIGN SKETCH6

CHAPTER 3 7

IMPLEMENTATION 7

* 1. [CODE EXPLANATION](#_TOC_250003) 7
  2. [APPLICATION SCREENSHOTS](#_TOC_250002) 11
  3. [LINK PROJECT](#_TOC_250002) 13

[REFERENCES](#_TOC_250002)

## BACKGROUND

# CHAPTER 1 INTRODUCTION

Music applications have become an essential part of modern digital media consumption. With the rapid growth of digital audio libraries and the increasing demand for personalized listening experiences, users rely heavily on music players to organize tracks, manage playlists, explore audio files, and enjoy seamless playback. While many platforms offer these capabilities, a significant number of lightweight local music players fail to provide advanced customization options, visually engaging interfaces, and modern interactive features. In particular, many lack customizable GUI elements, smooth animations, and real-time synchronization between audio output and visual components features that greatly enhance user experience.

*Musicify* is a Python-based desktop music player designed to address these limitations by leveraging the Tkinter library for GUI development. The purpose of this project is to create a fully functional, visually appealing, and smooth music-playing application that incorporates modern features often found in larger, more resource-heavy media players. These features include interactive sliders for volume and track navigation, responsive buttons and controls, dynamic animations, and real-time updates that reflect current playback status. Additionally, *Musicify* integrates the ability to render album cover images, creating a more immersive and aesthetically pleasing interface.

The project aims to deliver a lightweight yet powerful alternative to traditional media players by combining customizable Tkinter components with an efficient audio engine capable of handling playback smoothly. By focusing on performance, usability, and visual design, *Musicify* provides users with a streamlined experience without sacrificing functionality. Ultimately, this project demonstrates how Python and Tkinter can be used to produce a modern, polished multimedia application that balances simplicity with advanced features.

## PROBLEM STATEMENT

Many desktop music players available today provide a limited and outdated listening experience. Users often encounter rigid interfaces, inconsistent playback controls, and a lack of engaging visual elements such as album artwork, making the overall experience feel less immersive compared to modern platforms. These shortcomings reduce user satisfaction, hinder smooth interaction with their music library, and fall short of the expectations shaped by contemporary services like Spotify. Additionally, many modern players rely heavily on internet connectivity, which becomes inconvenient for users who prefer or need an offline, fully local music experience.

To address these user-centered issues, we developed *Musicify*, an offline desktop music player built with Python and Tkinter, designed to deliver a smoother, more visually appealing, and internet-independent listening environment. By incorporating features such as interactive sliders, responsive playback controls, and automatic cover-image rendering, *Musicify* provides a modern, engaging, and reliable way to enjoy local music collections. This project aims to offer users a desktop music player that feels contemporary, intuitive, and accessible anytime without the need for an internet connection.

## OBJECTIVES

The purpose of making this *Musicify* includes:

* + 1. Develop a fully functional desktop music player using Python Tkinter.
    2. Implement a real-time visual engine (Pixel Sync Engine) that synchronizes animations with audio playback.
    3. Create a clean, modern UI with custom buttons, frames, sliders, and icons.
    4. Provide a user-friendly and stable application suitable for real-world use.

# CHAPTER 2 ANALYSIS AND DESIGN

## APPLICATION NECESSITY ANALYSIS

The development of *Musicify* is based on several user needs and limitations found in many local music players. This analysis is divided into the following parts:

* + 1. Target User Demographics
       - Digital audio enthusiasts who maintain large collections of locally stored music files and need efficient organization and playback features.
       - Privacy-conscious users who prefer offline, self-hosted applications without cloud services or data tracking.
       - Offline-first users who require a reliable music player that works smoothly without an Internet connection.
    2. Functional Requirements
       - *Musicify* requires an audio playback system that supports MP3 and WAV formats, including controls such as play, pause, resume, stop, seek, and volume adjustment.
       - The application needs a library management system capable of storing metadata such as title, artist, album, track number, duration, genre, file path, and album artwork. It must also support adding, editing, deleting, and bulk importing music files.
       - A queue system is needed to allow users to create and manage playlists, including adding, removing, reordering, and displaying upcoming tracks.
       - The user interface must be simple and easy to navigate, featuring a sidebar, a main content area, an album view, a track list, and playback controls.interaction.
    3. Technology Stack Justification
       - **Python 3.x** is chosen for its cross-platform support, ease of learning, large library ecosystem, and suitability for desktop application development.
       - **Tkinter** is used for the GUI because it is lightweight, included in Python by default, and provides the necessary components for building a simple desktop interface.
       - **Pygame** is used for audio playback due to its easy-to-use API and support for basic audio formats such as MP3 and WAV.
       - **Pillow (PIL)** handles image processing tasks such as loading and displaying album artwork.
       - **Structured text files** are used for data storage because they are easy to read, simple to implement, and require no external database.

## FLOWCHART

## *Musicify* is designed as a digital music player that allows users to manage and enjoy their personal music collections through a simple and user-friendly interface. Unlike traditional music players with limited functionality, *Musicify* aims to provide a structured system for adding, storing, and playing songs efficiently.

## The flowchart illustrates the sequence of processes within the *Musicify* application, starting from launching the application, adding music files along with their metadata, and continuing to song playback. It also shows how the system handles user interactions, such as selecting tracks and controlling playback. This flow diagram helps ensure a clear and logical progression of operations, allowing users to navigate the application smoothly from start to finish.

## 

## USER INTERFACE DESIGN SKETCH

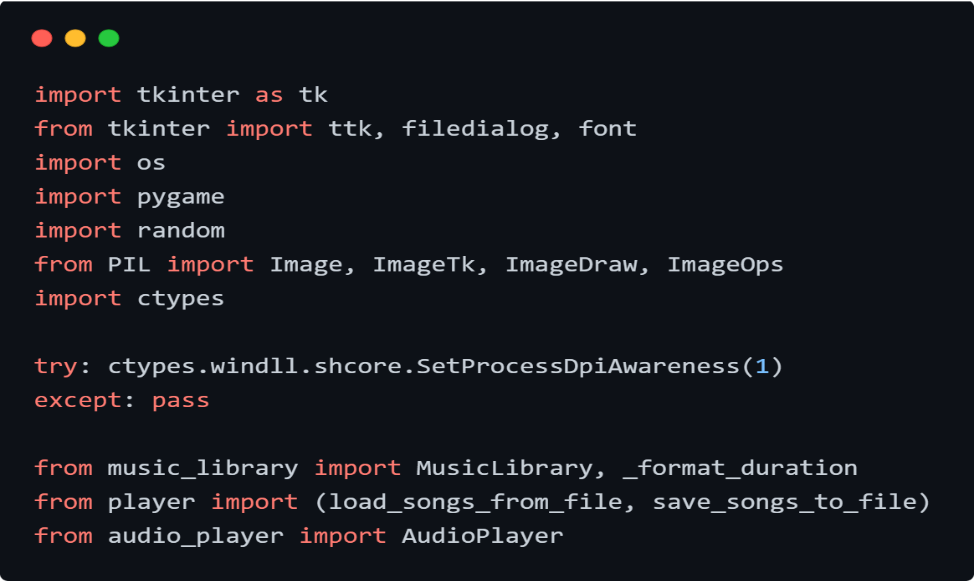
## The next steps in the *Musicify* project involve a combination of technical refinement and feature expansion. Core improvements include enhancing synchronization between the playback slider and the audio engine, increasing interface consistency, and improving album art rendering through automatic resizing and scaling.

## Beyond these system-level enhancements, the project aims to improve user experience by integrating additional informational and personalization features. These include displaying detailed song metadata, providing an in-app lyrics viewer for convenience, and introducing a basic user profile system to support a more customized interaction with the application.

# CHAPTER 3 IMPLEMENTATION

## CODE EXPLANATION

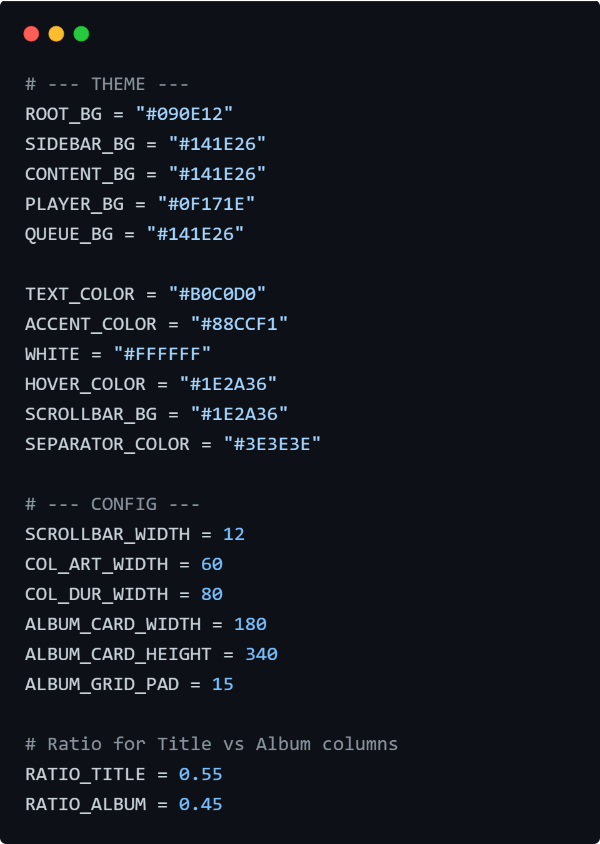
The implementation of *Musicify* V1 is carried out using the Python programming language. *Musicify* utilizes Tkinter for a native desktop GUI and Pygame for its audio processing engine. The application handles image processing for album art using the Pillow (PIL) library.



Gambar 3. 1 Libraries for *Musicify*   
Source: Personal documentation

Importing Libraries:

* Tkinter – used to build the core application interface, including the main window, frames, buttons, and the custom canvas-based slider for seeking tracks.
* pygame – used to handle the audio engine, specifically for loading music files, controlling playback (play, pause, stop), and managing the audio mixer.
* PIL (pillow) – used to process image assets, such as resizing album art and creating dynamic rounded (circular) icons for a modern visual style.
* os – used to manage file paths and navigate the operating system's directory structure to locate song files and assets dynamically.
* random – used to implement the “shuffle” functionality, allowing the application to randomize the order of the song queue.
* ctypes – used to configure high-DPI awareness on Windows systems, ensuring that text and UI elements appear rendered on high-resolution screens, and not blurry.



Gambar 3. 2 Code for Musicify Theme Configuration

Source: Personal documentation

The configuration section defines the color palette and layout constants for the application. The theme utilizes a dark mode aesthetic with specific hex codes:

* ROOT\_BG → the main background color of the application window.
* SIDEBAR\_BG → the highlight color used for the navigation sidebar.
* ACCENT\_COLOR → the highlight color used for active elements, such as the slider track and artist names.
* TEXT\_COLOR → the primary color for general text to ensure readability against the dark background.
* HOVER\_COLOR → the color applied to buttons or list items when the mouse cursor hovers over them, providing visual feedback.

Additionally, the configuration constants control the layout dimensions:

* SCROLLBAR\_WIDTH → defines the fixed width of the custom scrollbar.
* ALBUM\_CARD\_WIDTH/HEIGHT → sets the standard size for album cards in the grid view to ensure uniformity.
* RATIO\_TITLE/ALBUM → determines how much horizontal space the Title column (55%) takes compared to the Album column (45%) in the song list.

In simple terms:

This code defines the “look and feel” of the music player by storing all color codes and size measurements in one place. This makes it easy to change the entire color scheme (e.g., from blue to green) just by editing these few lines, ensuring the design remains consistent across every screen.

3.1.1 Main Interface (gui\_main.py)

This file serves as the entry point and the primary visual controller of the application. It defines the MusicifyApp class, which inherits from tk.Tk. Key implementation details include:

Custom Widgets: Standard Tkinter widgets were insufficient for the desired "modern" look. A custom ModernSlider class was created inheriting from tk.Canvas to draw a seek bar with rounded caps and interactive dragging support.

Layout Management: The interface uses a PanedWindow to create resizable Sidebar and Content areas. A custom ScrollableFrame class was implemented to handle scrolling for the song lists and queue, as Tkinter frames do not support scrolling natively.

Visual Styling: The UI implements a dark theme (using hex codes like #090E12 and #141E26) and dynamic rounded images. The make\_round\_image utility function uses alpha masking to create smooth, circular album art icons programmatically.

3.1.2 Audio Player Logic (audio\_player.py)

This module encapsulates the low-level audio processing and playback logic, serving as the interface between the application and the pygame mixer. It manages the playback state independently of the user interface. Key features include:

Queue Management: The class maintains two primary lists, self.queue and self.history, to manage playback flow. The play\_next\_from\_queue function automatically handles the transition to the next track when a song finishes, ensuring continuous playback without user intervention.

Event-Driven Updates: To ensure the GUI stays synchronized with the audio state (e.g., updating the progress bar or changing the “Play” button to “Pause”), the class uses callback functions (self.on\_song\_changed, self.on\_playback\_state\_changed). This allows the backend to trigger UI updates safely when the internal state changes.

Playback Control: It wraps standard pygame mixer functions to provide robust control over media, including precise seeking (seek), toggling pause states, and handling volume or track skipping logic.

3.1.3 **Data Structure & Library (music\_library.py)**

This file acts as the application's “brain” regarding data organization, implementing Object-Oriented Programming (OOP) principles to structure how music data is handled in memory.

Class Inheritance: A base class MediaItem is defined to hold common attributes like title and duration. The primary Song class inherits from MediaItem, extending it with specific music attributes such as artist, album, genre, and file paths.

Efficient Lookup: The MusicLibrary class uses a Python dictionary (self.all\_songs) rather than a simple list to store songs. This allows for O(1) time complexity when checking for duplicates (preventing the same song from being added twice) by using the song title as a unique key.

Sorting and Categorization: The module includes helper methods to organize data for different views, such as get\_sorted\_song\_list (which sorts by Artist then Album) and get\_songs\_by\_album (which groups songs into a dictionary keyed by album name for the “Albums” view).

3.1**.4 File Persistence (player.py)**

This module handles the Input/Output (I/O) operations required to save the user's library to the disk, ensuring that added songs are not lost when the program is closed.

Data Serialization: The save\_songs\_to\_file function converts Song objects into a specific string format using a pipe delimiter (|). This text-based serialization allows complex object data to be stored in a simple, human-readable .txt file.

Data Loading: The load\_songs\_from\_file function reads songs.txt line-by-line, splitting the string by the | delimiter to reconstruct Song objects. It includes error handling (try-except blocks) to safely skip corrupted lines or missing files without crashing the application.

## APPLICATION SCREENSHOTS

## Some visual components of the *Musicify* application need to be presented to help readers better understand the interface, flow, and overall design of the system. Therefore, the following figure provides a visual representation of the corresponding section of the application:

## All Songs Page (When First Opened)

## 

## Gambar 3.5 All Songs Page

## Source: Personal documentation

## Albums Page

## 

## Gambar 3.6 Albums Page

## Source: Personal documentation

## Playing a Song

## 

## Gambar 3.7 Playing a Song

## Source: Personal documentation

## Adding a Song

## 

## Gambar 3.8 To add a song, use this window by clicking “Add New Song”

## Source: Personal documentation

## Queue Bar

## 

## Gambar 3.9 Displayed on the right side of the GUI, with clear queue and volume adjustment feature below

## Source: Personal documentation

## 

## ATTACHEMNT

## 

## 

## 

## 

## 

## 

## 

## 

## gui\_main.py

## 

## 

## audio\_player.py

## 

## music\_library.py

## player.py

## Link Project

<https://drive.google.com/drive/u/2/folders/13-UMqPgu217jxoVpUkbdIAmPupwjeUUN>

# REFERENCES

# GeeksforGeeks. (2024). *Build a music player with Tkinter and Pygame in Python*. Retrieved December 15, 2024, from <https://www.geeksforgeeks.org/python/build-a-music-player-with-tkinter-and-pygame-in-python/>

# Python Software Foundation. (2024). *tkinter — Python interface to Tcl/Tk*. Retrieved December 15, 2024, from <https://docs.python.org/3/library/tkinter.html>

# Python Software Foundation. (2024). *tkinter — Python interface to Tcl/Tk*. Retrieved December 15, 2024, from <https://docs.python.org/3/library/tkinter.html>

# Pygame Documentation. (2024). *pygame.mixer — Pygame module for loading and playing sounds*. Retrieved December 15, 2024, from <https://www.pygame.org/docs/ref/mixer.html>