# TITLE

# MUSIC PLAYER USING TKINTER AND SQLITE3

## GROUP-13


## TEAM MEMBERS

1.PRASANNA.S.N

2.LAYASHREE.S.R

3.SWETHA. S

4.VISHALINEE.G

# CONTENTS OF OUR PROJECT

# ABSTRACT

This project presents the development of a music player application utilizing Python's Tkinter library for the graphical user interface (GUI) and SQLite for database management. The music player allows users to manage and play their music library efficiently.

**Tkinter** is employed to create an intuitive and responsive GUI, providing essential functionalities such as play, pause, stop, and skip. Users can navigate through their music collection, view track details, and interact with various controls through a user-friendly interface.

**SQLite**, an embedded database, is used to store and manage metadata associated with the music files, including track names, artists, albums, and playback history. This database integration ensures that the music player can handle large collections and maintain user preferences and playlists effectively.

The application supports various audio formats and includes features such as playlist creation, track searching, and shuffle mode. By combining Tkinter's ease of use for GUI development with SQLite's efficient data management capabilities, this project demonstrates a practical approach to building a fully functional and robust music player in Python.

Future enhancements may include support for additional audio formats, advanced audio effects, and integration with online music services to further expand the application's capabilities.

Future development possibilities include expanding support for additional audio formats, incorporating advanced audio effects, and integrating with online music streaming services to further enhance functionality. Overall, this project demonstrates a practical implementation of Python’s Tkinter and SQLite in creating a fully functional and user-centric music player application.

This detailed abstract captures the essence of the project, outlining the key features, technologies, and functionalities of the music player application. It also hints at potential future improvements, providing a comprehensive overview of the project's scope and impact.

## Here are some things to consider when creating a music player using Python's Tkinter and Pygame packages:

- **User interface:**

The music player should have a user interface that shows information about the playing file, such as the file name, length, and how much has been played.

- **Playback options:**

The music player should allow users to add songs to a playlist, play, pause, resume, rewind, and stop songs, and play the previous or next song.

- **Libraries:**

Python has several libraries that can play audio files, including Pygame, Pymedia, and Simple Audio. Pygame has music functions for playing, pausing, stopping, rewinding, and adjusting the volume and mute.

- **Tkinter:**

Tkinter is a graphical user interface (GUI) library that comes pre-installed with Python. To use Tkinter, you can import it with from Tkinter import* and initialize it with root= Tk().

- **Creating an infinite loop:**

To keep the app running until it's canceled or quit, you can use root.mainloop() to create an infinity loop.

# INTRODUCTION

In the realm of personal entertainment and multimedia management, the need for effective and intuitive music player applications is ever-present. This project addresses this need by developing a desktop music player application using Python, specifically leveraging the Tkinter library for its graphical user interface (GUI) and SQLite for robust database management. The objective is to create a comprehensive and user-friendly application that simplifies music playback and library management.

## Purpose of *the* Project:

### Objective:

- Develop a desktop music player that provides an effective and user-friendly

solution for managing and playing music files.

### Scope:

- The application allows users to handle their music library, control playback, and

organize playlists efficiently.

## Key Features and Functionalities:

## Graphical User Interface (GUI):

### Tkinter

### Main Window:

- The primary interface where users interact with the application

### Playback Controls:

- **Play**: Starts playing the selected track.
- **Pause**: Pauses the current track.
- **Stop**: Stops playback and resets the track position to the beginning.
- **Next/Previous**: Allows users to move to the next or previous track in the playlist.

## Volume Control:

- **Slider**: Allows users to adjust the audio volume.

## Playback Progress:

- **Progress Bar**: Shows the current position within the track and allows users to seek to different parts of the track.

## Playlist Management:

- **Display**: Shows the list of tracks in the current playlist.
- **Controls**: Buttons or options to add, remove, or reorder tracks in the playlist.

## Track Information:

- **Labels/Displays**: Show details such as track name, artist, and album.

## Search and Filter:

- **Search Box**: Enables users to find specific tracks by entering keywords.
- **Filter Options**: Allows sorting or filtering tracks based on criteria like artist or album.

## Database Management:

## SQLite

## Schema:

### Tracks Table:

- **Columns**: Track ID, Name, Artist, Album, Duration, File Path.

## Playlists Table:

- **Columns**: Playlist ID, Name.

## UserSettings Table:

- **Columns**: User ID, Volume, Last Track ID

## Database Operations:

## CRUD Operations:

- Create, Read, Update, and Delete operations for managing tracks, playlists, and user settings.

## Queries:

- Efficient retrieval of music and playlist data, supporting search and filter functionalities.

## Functionality:

### Music Playback:

- **Audio Format Support**: Handles various audio file formats.
- **Playback Controls**: Includes features for controlling playback and managing the audio experience.

### Playlist Management:

- **Creation/Deletion**: Users can create new playlists or delete existing ones.
- **Track Management**: Adding and removing tracks from playlists, as well as sorting and organizing them.

### User Preferences:

- **Settings Storage**: Saves user preferences such as volume level and the last played track to enhance user experience.

# OBJECTIVE

The primary objective of this project is to develop a robust and user-friendly desktop music player application using Python, integrating the Tkinter library for the graphical user interface (GUI) and SQLite for efficient database management. The application aims to provide an intuitive and comprehensive solution for managing and enjoying personal music collections.

## Key objectives include:

## Design an Intuitive GUI:

- **User Experience**: Create a visually appealing and easy-to-navigate interface using Tkinter. Ensure that essential playback controls (play, pause, stop, next, previous) and features (volume control, progress tracking) are readily accessible and user-friendly.
- **Playlist Management**: Implement functionalities for users to create, modify, and manage playlists effectively within the GUI. Include options for adding, removing, and reordering tracks in playlists.

## Efficient Data Management:

- **Metadata Storage**: Utilize SQLite to handle the storage and management of music metadata, including track names, artists, albums, file paths, and playback history.
- **User Preferences**: Store and retrieve user settings such as volume levels and last played tracks to personalize the user experience.
- **Database Operations**: Implement robust CRUD (Create, Read, Update, Delete) operations to manage tracks and playlists efficiently.

## Support for Multiple Audio Formats:

- **Playback Compatibility**: Ensure that the application supports various audio file formats, providing flexibility and versatility in music playback.

### Implementation:

- Integrate audio processing libraries (e.g., `pygame`, `pydub`) that support common formats such as MP3, WAV, and FLAC. Ensure seamless playback for all supported formats

## Enhanced User Interaction:

- **Search and Filter**: Incorporate search and filter functionalities to enable users to easily find and organize their music tracks and playlists.
- **Real-Time Feedback**: Provide real-time feedback on playback status, including progress indicators and volume adjustments.

### ➢ Implementation:

- **Search Functionality**: Implement a search box that allows users to find specific tracks by entering keywords related to track names, artists, or albums.
- **Filter Options**: Provide options to filter tracks by criteria such as genre or album to help users organize their playlists.

## Integration of GUI and Database:

**Seamless Interaction**: Ensure smooth interaction between the Tkinter GUI and the SQLite database, allowing for real-time updates and consistent data management.

### ➢ Implementation:

**Data Synchronization**: Implement mechanisms to ensure that changes in the GUI (e.g., playlist modifications) are accurately reflected in the database and vice versa.

**Real-Time Updates**: Enable real-time updates to the GUI based on database changes, such as updating the playlist view when tracks are added or removed.

## Scalability and Future Enhancements:

**Extensible Design**: Develop the application with scalability in mind, allowing for future enhancements such as support for additional audio formats, advanced audio effects, and integration with online music services.

### ➢ Implementation:

- **Modular Architecture**: Design the application with a modular approach to facilitate the addition of new features.
- **Future Enhancements**: Plan for future improvements such as support for additional audio formats, integration of advanced audio effects, and potential connectivity with online music streaming services.

➢ By achieving these objectives, the project aims to deliver a functional, efficient, and enjoyable music player application that enhances the overall user experience in managing and playing their music collection.

# TECHNOLOGY USED AND REASON WHY

The music player application utilizes a combination of technologies to deliver a fully functional and efficient desktop music management solution. The key technologies employed in this project are:

## 1. **Python:**

- **Description**: Python is a high-level, interpreted programming language known for its simplicity and readability. It is widely used for developing various types of applications, including desktop software.
- **Role in Project**: Python serves as the primary language for implementing the logic and functionality of the music player application. Its extensive standard library and third-party modules facilitate rapid development and integration.

## 2. **Tkinter:**

- **Description**: Tkinter is Python's standard GUI (Graphical User Interface) library. It provides a set of tools to create windows, dialogs, buttons, and other interactive elements in desktop applications.
- **Role in Project**: Tkinter is used to design and build the graphical user interface of the music player. It allows for the creation of various GUI components such as:
    - **Playback Controls**: Buttons for play, pause, stop, and navigation.
    - **Volume Control**: Slider for adjusting audio volume.
    - **Progress Bar**: Displays the current position in the track and allows seeking.
    - **Playlist Management**: Tools for creating and managing playlists, and viewing track details.
    - **Search and Filter**: Interface elements for searching tracks and filtering playlists.

## **3. SQLite:**

- **Description**: SQLite is a lightweight, serverless, self-contained SQL database engine. It is designed for simplicity and efficiency, making it ideal for applications requiring a local database.
- **Role in Project**: SQLite is used to manage and store data related to music tracks, playlists, and user preferences. Key aspects include:
    - **Metadata Storage**: Storing information about tracks (name, artist, album, file path) and playlists.
    - **User Settings**: Saving user preferences such as volume levels and playback history.

- **Database Operations**: Performing Create, Read, Update, and Delete (CRUD) operations to manage music data and user settings efficiently.

## 4. Audio Processing Libraries:

- **Description**: Various Python libraries can handle audio file playback and processing, such as `pygame`, `pydub`, or `playsound`.
- **Role in Project**: These libraries are used to:
  - **Playback Audio**: Handle the playback of different audio formats supported by the application.
  - **Audio Processing**: Manage tasks such as adjusting volume, seeking within tracks, and playing audio files.

## 5. File Handling and System Integration:

- **Description**: Python's built-in modules for file handling and system integration, such as `os` and `shutil`, manage file operations and interactions with the operating system.
- **Role in Project**: These modules are used to:
  - **Manage Music Files**: Handle file operations such as opening, reading, and writing music files.
  - **File Paths**: Manage paths to music files and integrate with the SQLite database to store file locations.

## 6. Additional Libraries (Optional):

- **Description**: Depending on specific requirements, additional Python libraries may be used to enhance functionality.
  - **Example Libraries**: `mutagen` for reading and writing metadata in audio files, `tkcalendar` for integrating calendar widgets, or `requests` for future integration with online services.
- **Role in Project**: These libraries can be used to extend the application's capabilities, such as integrating more advanced metadata handling or online features.

➢ By leveraging these technologies, the music player application aims to provide a cohesive and efficient solution for music management and playback, ensuring a smooth and engaging user experience.

# REASON

When creating a music player using Tkinter and SQLite, you're combining a graphical user interface (GUI) toolkit with a lightweight database to manage your music library. Here's a breakdown of the technology and reasons for their use:

## Tkinter:

- **Tkinter** is the standard GUI toolkit for Python. It provides a way to create windows, dialogs, and various user interface elements such as buttons, labels, and menus.

### 2. Uses of Tkinter:

- **Ease of Use**: Tkinter is straightforward and integrates well with Python, making it ideal for building simple GUIs without a steep learning curve.
- **Built-in**: Tkinter is included with Python, so you don't need to install any additional libraries to get started.
- **Cross-Platform**: Tkinter applications work on multiple operating systems (Windows, macOS, Linux) without modification.

### 3. Typical Components in a Music Player:

- **Buttons**: For play, pause, stop, next, and previous functions.
- **Labels and Text Fields**: To display song information (title, artist, etc.).
- **Sliders**: For controlling volume and playback position.
- **Listbox**: To display the playlist or queue of songs.

## SQLite:

### 1.SQLite:

- **SQLite** is a lightweight, serverless SQL database engine that stores data in a single file. It’s known for its simplicity and efficiency.

### 2. Uses of SQLite:

- **Simplicity**: SQLite does not require a separate server process. It is self-contained and zero-configuration, making it easy to set up and use.
- **File-Based**: Since SQLite databases are stored in a single file, managing and distributing your music database is straightforward.
- **Suitable for Small to Medium Projects**: For a personal music player with a manageable number of songs and playlists, SQLite provides ample performance without the overhead of more complex database systems.
- **Easy Integration with Python**: SQLite has native support in Python through the sqlite3 module, which makes it easy to interact with the database

### 3. Typical Use Cases in a Music Player:

- **Storing Metadata**: Information such as song titles, artists, album names, and file paths can be stored in the SQLite database.
- **Managing Playlists**: Create, modify, and manage playlists by storing playlist data in the database.
- **Search and Filter**: Efficiently query the database to find songs or manage collections.

## Combining Tkinter and SQLite

### 1. User Interface and Data Management:

- **Tkinter** handles the GUI, allowing users to interact with the music player, while **SQLite** manages the backend data, storing and retrieving music-related information.

### 2. Workflow Example:.

- **Loading Data**: When the music player starts, Tkinter can use SQLite to load song data and populate the playlist interface.
- **User Actions**: When a user selects a song or creates a playlist, Tkinter can update the database via SQLite to reflect these changes.
- **Playback Control**: Tkinter manages playback controls and displays information about the currently playing song, which is pulled from the SQLite database.

### 1. Setup and Planning:

- **Define Requirements**: Identify core features (playback controls, playlist management) and non-functional aspects (performance, cross-platform compatibility).
- **Set Up Environment**: Install Python and necessary libraries (Tkinter, audio libraries like pygame).

### 2. Design:

- **UI Design**: Sketch the layout and choose Tkinter widgets (buttons, sliders, listboxes).
- **Database Schema**: Design tables for storing song data (e.g., songs, playlists) and define fields (e.g., title, artist, file_path).

### 3. Development:

- **Database Implementation**:
  - **Create and Set Up**: Use SQLite to create tables and define CRUD operations.
- **GUI Implementation**:
  - **Create Main Window**: Use Tkinter to build the main window and add widgets.
  - **Integrate Database**: Fetch data from SQLite and populate Tkinter widgets.
- **Audio Playback**:
  - **Library Integration**: Use libraries like pygame for handling audio playback.
  - **Control Playback**: Implement play, pause, and stop functionalities.

- **Playlist Management**:
    - **Manage Playlists**: Add/remove songs and update the database.

## 4. Testing:

- **Unit Testing**: Test individual components (database functions, GUI elements).
- **Integration Testing**: Test the complete workflow from UI to database.
- **User Testing**: Gather feedback from real users to identify usability issues.

## 5. Deployment:

- **Package Application**: Use tools like PyInstaller to create an executable.
- **Create Installer**: If needed, prepare an installer for user-friendly setup.
- **Documentation**: Write a user guide and developer documentation.

## 6. Maintenance and Updates:

- **Fix Bugs**: Address any issues reported by users.
- **Add Features**: Enhance the application based on feedback.
- **Update Dependencies**: Keep libraries and tools up-to-date.

➢ These notes outline the essential steps and considerations for developing a music player with Tkinter and SQLite, from initial planning to ongoing maintenance.

➢ In summary, Tkinter provides an accessible way to build the user interface for your music player, while SQLite offers a simple and effective way to manage and store your music library data. This combination allows you to build a functional and user-friendly music player with relatively low complexity.

## 1. Seamless Interaction:

- **UI and Data Sync**: Tkinter handles the graphical interface while SQLite manages the data. For example, when a user adds a new song through the Tkinter interface, the relevant data is updated in the SQLite database, and vice versa.
- **Real-Time Updates**: Changes made through the Tkinter GUI (like adding a song to a playlist) can be immediately reflected in the SQLite database. This real-time interaction ensures that the data displayed in the interface is always current.
- **Data Synchronization**: Any changes made in the Tkinter interface (such as creating or modifying playlists) can be directly applied to the SQLite database, ensuring that the data is always synchronized with the user's actions.

2. **Efficient Development Workflow:**

- **Rapid Prototyping**: Tkinter's simplicity allows for quick design and iteration of the user interface, while SQLite's straightforward data management simplifies backend development. Together, they enable rapid prototyping and development of the music player..

**3. Data Persistence:**

- **Long-Term Storage**: SQLite ensures that all song data, playlists, and user preferences are persistently stored and can be retrieved even after the application is closed and reopened. This persistence is crucial for a music player where users expect their data to be saved between sessions.

**4.Search and Filtering:**

- Users can search for songs or filter playlists through Tkinter's interface. The queries are executed against the SQLite database, and results are presented back in the Tkinter UI.

- By combining Tkinter and SQLite, you get a robust solution for building a music player that is both user-friendly and capable of handling data efficiently.

- In summary, Tkinter and SQLite provide a powerful combination for building a music player. Tkinter offers an easy-to-use, cross-platform interface for interacting with users, while SQLite provides a lightweight, efficient way to manage and store data. Together, they enable the creation of a functional and user-friendly music player with minimal complexity

# PROJECT EXECUTION

Executing a music player project using Tkinter for the user interface and SQLite for data management involves several steps, from initial setup to final testing. Here's a step-by-step guide to help you through the process:

## 1. Project Planning:

### 1.1 Define Project Scope:

- **Features:**
  - Playback controls (play, pause, stop).
  - Metadata display (song title, artist, album, cover art).
  - Playlist management (create, save, load).
  - User settings (volume control, last played track).

### 1.2 Design the User Interface:

- **Sketch Layout:** Draw the layout of the application, including where buttons, lists, and metadata will be placed.
- **Choose Design Elements:** Select colors, fonts, and overall styling to ensure the application is both functional and visually appealing.

## 2. Setup and Environment:

### 2.1 Prepare Development Environment:

- **Install Python:** Ensure Python is installed on your system.
- **Install Necessary Libraries:** While Tkinter comes with Python, you may need additional libraries like Pillow for image handling and pygame or pydub for audio playback.

### 2.2 Set Up Virtual Environment (optional but recommended):

- **Create and Activate Virtual Environment:** This isolates your project dependencies and makes management easier.

## 3. Database Design and Setup:

### 3.1 Design Database Schema:

- **Define Tables:** Determine the tables needed for the project:

- **Songs Table:** Stores song details (title, artist, album, file path).
- **Playlists Table:** Stores playlist information (playlist name).
- **Playlist_Songs Table:** Manages the relationship between playlists and songs.
- **User_Preferences Table:** Saves user settings and preferences.

### 3.2 Initialize Database:

- **Create Database and Tables:** Use SQLite3 to set up the initial database schema based on the design.
- **Set Up Database File:** Initialize the database file that will be used to store data.

## 4. Developing the Application:

### 4.1 Build the Tkinter User Interface:

- **Create Main Window:** Set up the main application window with Tkinter.
- **Add Widgets:** Implement necessary widgets such as buttons for playback controls, labels for displaying song information, and lists for playlists.
- **Arrange Layout:** Use Tkinter's layout management tools to arrange widgets according to your design.

### 4.2 Implement Audio Playback:

- **Integrate Audio Library:** Choose and integrate a library (e.g., pygame, pydub) for handling audio playback.
- **Connect Controls:** Ensure that playback controls in the Tkinter UI are linked to the audio playback functions.

### 4.3 Connect UI with SQLite Database:

- **Manage Song Data:** Implement functionality to add, update, and retrieve song data from the SQLite database.
- **Handle Playlists:** Implement features to create, modify, and load playlists.
- **Save User Preferences:** Develop methods to save and retrieve user preferences like volume settings.

## 5. Testing:

### 5.1 Test the Application:

- **Functionality Testing:** Verify that all features work as expected (playback controls, UI elements, data handling).
- **Usability Testing:** Ensure the application is user-friendly and intuitive.
- **Database Testing:** Check that data is correctly saved, retrieved, and managed.

### 5.2 Debugging:

- **Identify Issues:** Use debugging tools and logs to find and fix bugs or issues.
- **Refine Code:** Make necessary adjustments to improve performance and reliability.

## 6. Finalization:

### 6.1 Polish the UI:

- **Refine Design:** Make final adjustments to the user interface based on testing feedback.
- **Improve Aesthetics:** Ensure that the design is cohesive and visually appealing.

### 6.2 Documentation:

- **Create User Manual:** Write instructions on how to install and use the music player.
- **Document Code:** Provide comments and documentation for the codebase to assist with future maintenance.

### 6.3 Packaging the Application:

- **Build Executable:** Use tools like PyInstaller to package the application into an executable or installer for distribution.
- **Prepare Distribution:** Ensure all necessary files are included and the application is ready for end-users.

## 7. Deployment:

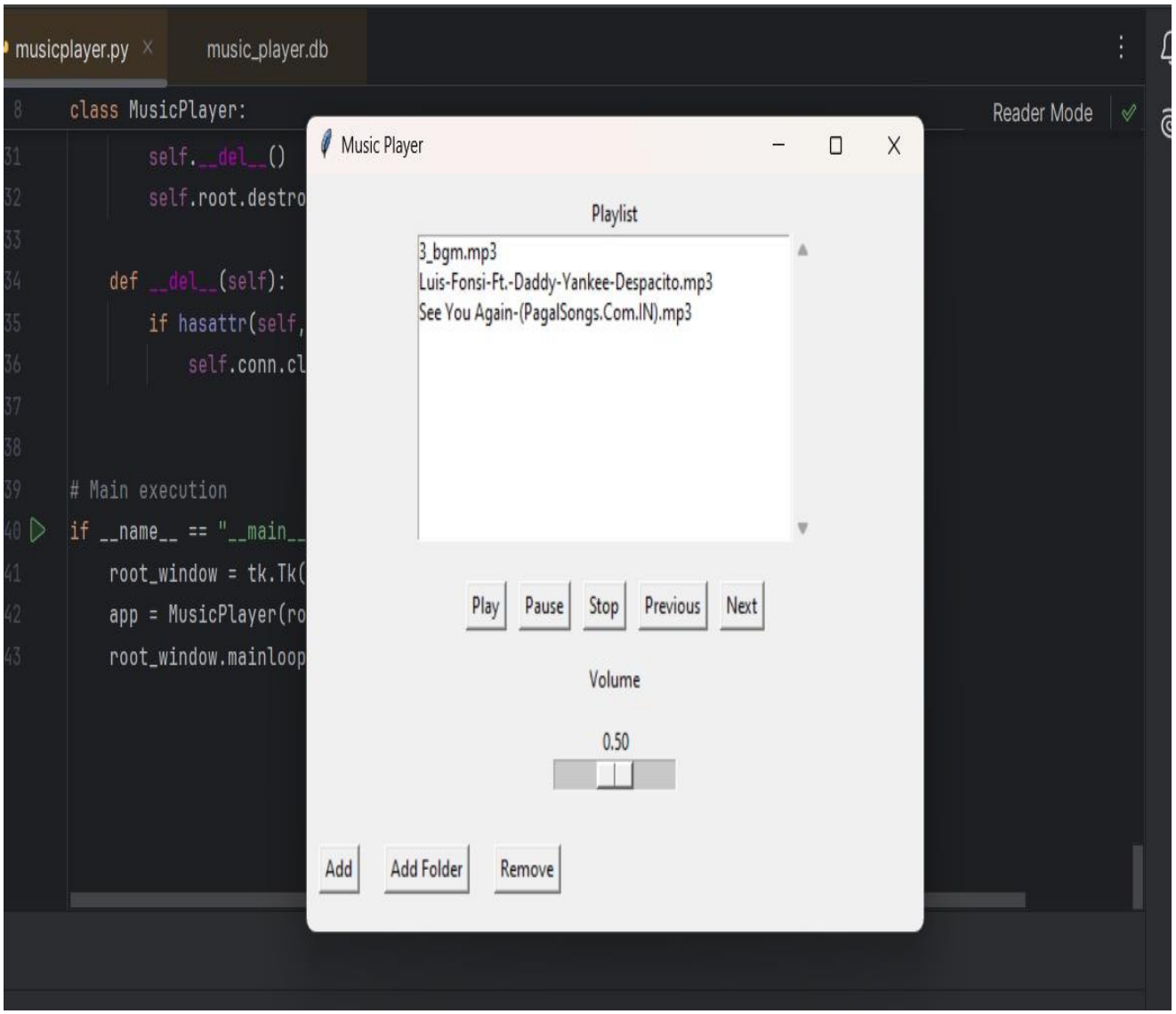
### 7.1 Distribute the Application:

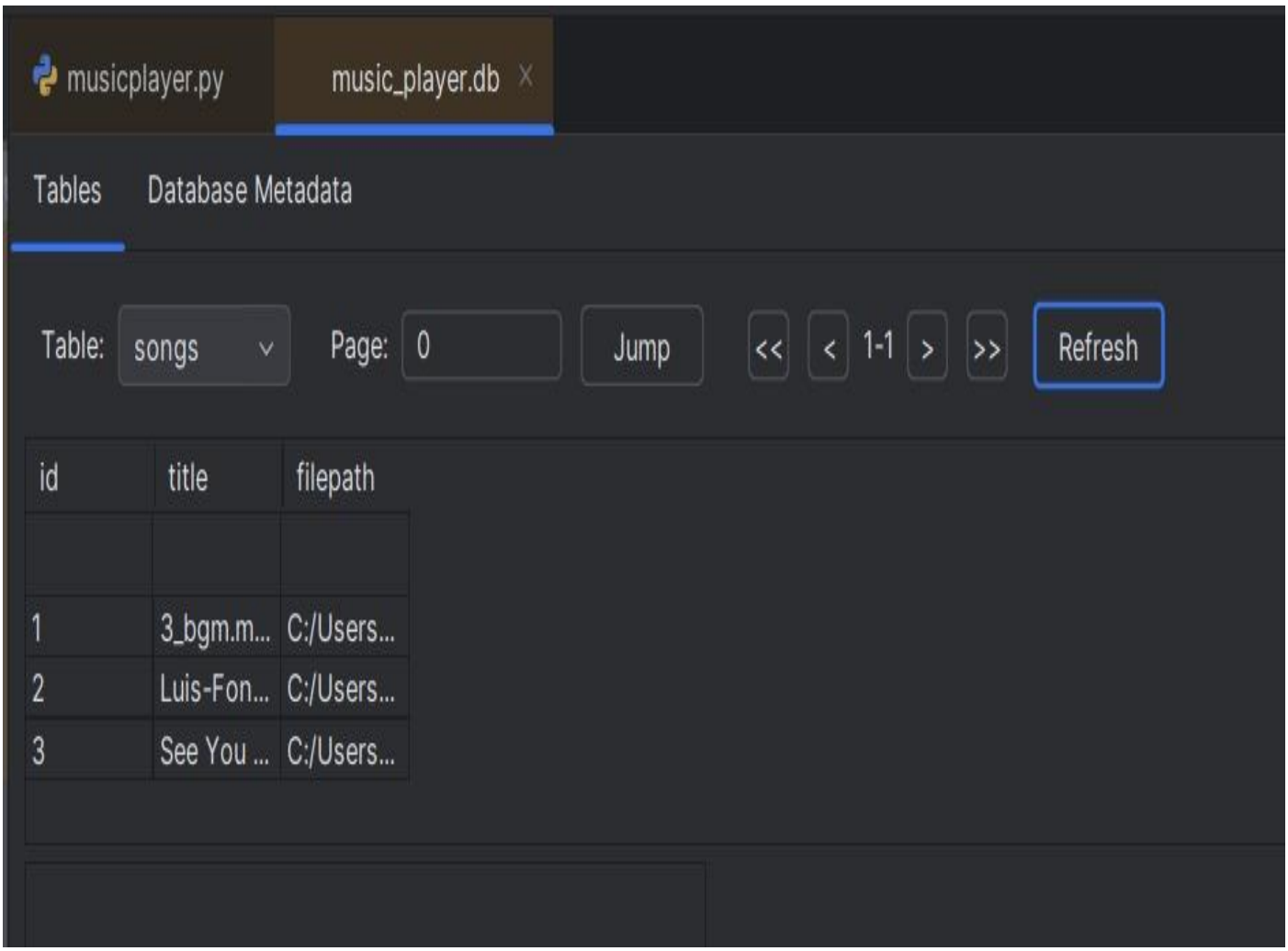
- **Release:** Share the executable or installer with users through appropriate channels (website, email, etc.).
- **Provide Support:** Be available to address user issues and feedback.

### 7.2 Maintenance and Updates:

- **Monitor Performance:** Keep track of the application's performance and any reported issues.
- **Update Regularly:** Provide updates and improvements based on user feedback and technological advancements.

# PROJECT SCREENSHOTS




musicplayer.py
music_player.db
Tables
Database Metadata
Table: songs
Page: 0
Jump
<<
<
1-1
>
>>
Refresh
id
title
filepath
1
3_bgm.m...
C:/Users...
2
Luis-Fon...
C:/Users...
3
See You ...
C:/Users...

# SOURCE CODE

```
import tkinter as tk
from tkinter import filedialog, messagebox
import sqlite3
import pygame
import os


class MusicPlayer:
    def __init__(self, root_window):
        self.root = root_window
        self.root.title("Music Player")
        self.root.geometry("500x400")

        # Initialize pygame mixer
        try:
            pygame.mixer.init()
        except pygame.error:
            messagebox.showerror("Initialization Error", "Pygame mixer could not be
initialized.")
            self.root.quit()

        self.current_song = None
        self.playlist = []

        # Initialize UI components
        self.init_ui()

        # Initialize database and load playlist
        self.init_db()
        self.load_playlist()

    def init_ui(self):
        # Playlist Frame
        self.playlist_frame = tk.Frame(self.root)
        self.playlist_frame.pack(pady=10)

        self.playlist_label = tk.Label(self.playlist_frame, text="Playlist")
        self.playlist_label.pack()

        self.playlist_box = tk.Listbox(self.playlist_frame, width=50, height=10)
        self.playlist_box.pack(side=tk.LEFT)
        self.playlist_box.bind("<Double-1>", self.on_song_select)
```

```
        self.scrollbar = tk.Scrollbar(self.playlist_frame, orient=tk.VERTICAL)
        self.scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
        self.playlist_box.config(yscrollcommand=self.scrollbar.set)
        self.scrollbar.config(command=self.playlist_box.yview)

        # Control Buttons
        self.controls_frame = tk.Frame(self.root)
        self.controls_frame.pack(pady=10)

        self.play_button = tk.Button(self.controls_frame, text="Play",
command=self.play_song)
        self.play_button.grid(row=0, column=0, padx=5)

        self.pause_button = tk.Button(self.controls_frame, text="Pause",
command=self.pause_song)
        self.pause_button.grid(row=0, column=1, padx=5)

        self.stop_button = tk.Button(self.controls_frame, text="Stop",
command=self.stop_song)
        self.stop_button.grid(row=0, column=2, padx=5)

        self.previous_button = tk.Button(self.controls_frame, text="Previous",
command=self.play_previous)
        self.previous_button.grid(row=0, column=3, padx=5)

        self.next_button = tk.Button(self.controls_frame, text="Next",
command=self.play_next)
        self.next_button.grid(row=0, column=4, padx=5)

        # Volume Control
        self.volume_label = tk.Label(self.root, text="Volume")
        self.volume_label.pack(pady=5)

        self.volume_scale = tk.Scale(self.root, from_=0, to_=1, orient=tk.HORIZONTAL,
resolution=0.01,
                                     command=self.adjust_volume)
        self.volume_scale.set(0.5)  # Default volume level
        self.volume_scale.pack(pady=5)

        # Add and Remove Buttons
        self.add_button = tk.Button(self.root, text="Add", command=self.add_song)
        self.add_button.pack(side=tk.LEFT, padx=10)

        self.add_folder_button = tk.Button(self.root, text="Add Folder",
command=self.add_folder)
        self.add_folder_button.pack(side=tk.LEFT, padx=10)
```

```
        self.remove_button = tk.Button(self.root, text="Remove", command=self.remove_song)
        self.remove_button.pack(side=tk.LEFT, padx=10)

        # Handle window close event
        self.root.protocol("WM_DELETE_WINDOW", self.on_closing)

    def init_db(self):
        try:
            self.conn = sqlite3.connect('music_player.db')
            self.cursor = self.conn.cursor()
            self.cursor.execute('''
                CREATE TABLE IF NOT EXISTS songs (
                    id INTEGER PRIMARY KEY,
                    title TEXT,
                    filepath TEXT
                )
            ''')
            self.conn.commit()
        except sqlite3.Error as e:
            messagebox.showerror("Database Error", f"Failed to connect to database: {e}")
            self.root.quit()

    def load_playlist(self):
        """Load playlist from the database into the listbox."""
        self.playlist_box.delete(0, tk.END)
        self.cursor.execute('SELECT id, title FROM songs')
        songs = self.cursor.fetchall()
        self.playlist = songs
        for song in songs:
            self.playlist_box.insert(tk.END, song[1])

    def add_song(self):
        """Add a single song to the playlist and database."""
        filepath = filedialog.askopenfilename(filetypes=[("Audio Files", "*.mp3 *.wav")])
        if filepath:
            if not filepath.lower().endswith(('.mp3', '.wav')):
                messagebox.showerror("File Error", "Selected file format is not supported.")
                return

            title = os.path.basename(filepath)
            try:
                self.cursor.execute('''
                    INSERT INTO songs (title, filepath)
                    VALUES (?, ?)
                ''', (title, filepath))
```

```
            self.conn.commit()
            self.load_playlist()
        except sqlite3.Error as e:
            messagebox.showerror("Database Error", str(e))

    def add_folder(self):
        """Add all audio files from a selected folder to the playlist and database."""
        folder_path = filedialog.askdirectory()
        if folder_path:
            supported_formats = ('.mp3', '.wav')
            files_added = 0

            for file_name in os.listdir(folder_path):
                if file_name.lower().endswith(supported_formats):
                    filepath = os.path.join(folder_path, file_name)
                    title = file_name
                    try:
                        self.cursor.execute('''
                            INSERT INTO songs (title, filepath)
                            VALUES (?, ?)
                        ''', (title, filepath))
                        files_added += 1
                    except sqlite3.IntegrityError:
                        # If the file already exists, just skip it
                        continue
                    except sqlite3.Error as e:
                        messagebox.showerror("Database Error", str(e))

            self.conn.commit()
            if files_added > 0:
                self.load_playlist()
                messagebox.showinfo("Success", f"Added {files_added} files to playlist.")
            else:
                messagebox.showinfo("Info", "No supported audio files found in the selected
folder.")

    def remove_song(self):
        """Remove the selected song from the playlist and database."""
        selected_index = self.playlist_box.curselection()
        if selected_index:
            song_id = self.playlist[selected_index[0]][0]
            try:
                self.cursor.execute('DELETE FROM songs WHERE id = ?', (song_id,))
                self.conn.commit()
                self.load_playlist()
                self.stop_song()
```

```
        except sqlite3.Error as e:
            messagebox.showerror("Database Error", str(e))

    def on_song_select(self, event):
        """Handle song selection from the listbox."""
        selected_index = self.playlist_box.curselection()
        if selected_index:
            song_id = self.playlist[selected_index[0]][0]
            try:
                self.cursor.execute('SELECT filepath FROM songs WHERE id = ?', (song_id,))
                filepath = self.cursor.fetchone()[0]
                self.current_song = filepath
            except sqlite3.Error as e:
                messagebox.showerror("Database Error", str(e))

    def play_song(self):
        if self.current_song:
            try:
                pygame.mixer.music.load(self.current_song)
                pygame.mixer.music.play()
            except pygame.error as e:
                messagebox.showerror("Playback Error", str(e))

    def pause_song(self):
        pygame.mixer.music.pause()

    def stop_song(self):
        pygame.mixer.music.stop()

    def play_song_at_index(self, index):
        if not self.playlist:
            return
        self.playlist_box.selection_clear(0, tk.END)
        self.playlist_box.selection_set(index)
        self.on_song_select(None)
        self.play_song()

    def play_previous(self):
        if not self.playlist:
            return
        current_index = self.playlist_box.curselection()
        if current_index:
            new_index = (current_index[0] - 1) % len(self.playlist)
            self.play_song_at_index(new_index)

    def play_next(self):
```

```
        if not self.playlist:
            return
        current_index = self.playlist_box.curselection()
        if current_index:
            new_index = (current_index[0] + 1) % len(self.playlist)
            self.play_song_at_index(new_index)

    def adjust_volume(self, volume):
        """Adjust the volume of the playback."""
        pygame.mixer.music.set_volume(float(volume))

    def on_closing(self):
        self.__del__()
        self.root.destroy()

    def __del__(self):
        if hasattr(self, 'conn'):
            self.conn.close()


# Main execution
if __name__ == "__main__":
    root_window = tk.Tk()
    app = MusicPlayer(root_window)
    root_window.mainloop()
```