**Synergia: A Versatile Python-Based Multitasking Application**

by

***VEER VISWAJIT N - 22BCE1320***

***DEEPAK KUMAR N - 22BCE1404***

***RAGHAV KAUSHIK R - 22BCE1864***

***A project report submitted to***

**Dr. M. Braveen**

**Assistant Professor, School of Computer Science and Engineering**

in partial fulfilment of the requirements for the course of

**BCSE303L Operating Systems**

in

**B. TECH., COMPUTER SCIENCE AND ENGINEERING**



**Vellore Institute of Technology, Chennai**

**Vandalur – Kelambakkam Road, Chennai – 600 127**

**November 2023**

**Chennai**

***BONAFIDE CERTIFICATE***

This is to certify that the Project work titled **“Synergia: A Versatile Python-Based Multitasking Application”** that is being submitted by Regno, name is in partial fulfillment of the requirements for the award of **Bachelor of Technology in Computer Science and Engineering**, is a record of bonafide work done under my guidance. The contents of this Project work, in full or in parts, have neither been taken from any other source nor have been submitted to any other Institute or University for award of any degree or diploma and the same is certified.

**Dr. M. BRAVEEN**

**Guide**

**ABSTRACT**

The proposed Python-based multitasking application aims to revolutionize user experiences by seamlessly integrating Spotify music playback, a comprehensive to-do planner, a flexible shell GUI for command execution, a real-time clock, and Google Calendar integration into a single, visually appealing platform. This user-centric design prioritizes efficiency and convenience, reducing the need for users to toggle between multiple windows. The application's novelty lies in its real-time updates, offering accurate information for the Spotify player and clock display. Comprehensive testing across various operating systems and Python versions ensures cross-platform compatibility and robust performance. The application's competitive advantage lies in its holistic approach to multitasking, addressing a notable gap in the existing landscape. Future enhancements may include additional music player features, advanced task management functionalities, customization options, and integration with other productivity tools, setting a new standard for multifunctional tools in the digital landscape. Overall, this application signifies a paradigm shift in multitasking tools, showcasing the potential of creative problem-solving in software development.

***Keywords: Graphical User Interface, Paradigm Shift, API Utilization,***

***Spotify, To-Do Tasks, Cross-Platform Compatibility***

**TABLE OF CONTENTS**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **CHAPTER NO.** | **TITLE** | | | **PAGE NO.** | |
| **1** | **INTRODUCTION** | | |  | |
|  | 1.1 | Idea   * Background and Context * Contextual Relevance | 6  7 | |
|  | 1.2  1.3  1.4  1.5  1.6  1.7  1.8 | Objectives   * Detailed Functional Objectives * Technical Objectives   Scope   * Functional Scope * Technical Scope   Novelty and Innovation  Comparative Statement  Dataset  Test Bed  Expected Result | 7  7  8  9  9  10  11  11  12  12  12 | |
| **2** | **ARCHITECTURE** | | |  | |
|  | 2.1 | High level design (Black Box design) | 14 | |
|  | 2.2 | Low level design | 17 | |
| **3** | **SIMULATION/IMPLEMENTATION RESULTS** | | |  | |
|  | 3.1 Various Algorithms Followed | | | 25 | |
|  | 3.2 Algorithm for Implementation | | | 26 | |
|  | 3.3 Result in Data (Photos of Implementation) | | | 30 | |
|  | 3.4 Mapping the results with problem statements | | | 33 | |
| **4** | **CONCLUSION AND FUTURE WORK** | | |  | |
|  | 5.1 Conclusion | | | 36 | |
|  | 5.2 Future works | | | 37 | |
|  |  | | |  | |
| **5** | **REFERENCES** | | | 38 | |
|  |  | | |  | |
| **6** | **APPENDIX** | | |  | |
|  | 6.1 CODE | | | 39 | |

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| **FIGURE NO.** | **TITLE** | **PAGE NO.** |
| 1 | HIGH – LEVEL DESIGN OF THE APPLICATION | 14 |
| 2 | LOW – LEVEL DESIGN OF THE APPLICATION | 16 |
| 3.1 | Multitasker is executing ‘mv file1.txt Documents’ in the shell and playing a song from a playlist | 30 |
| 3.2 | Multitasker is executing ‘python --version’ in the shell and playing a song from a playlist | 30 |
| 3.3 | Multitasker is executing ‘pip install openai in the shell and playing a song from a playlist | 31 |
| 3.4 | Multitasker is executing ‘echo hello world’ in the shell and playing a song from a playlist | 31 |
| 3.5 | Multitasker is executing ‘dir’ in the shell and playing a song from a playlist | 32 |
| 3.6 | Multitasker is executing no commands in the shell and playing a song from a playlist | 32 |

**CHAPTER I**

**INTRODUCTION**

**A. Background and Context**

In the ever-evolving landscape of digital applications, the need for versatile and integrated tools has become paramount. The proposed Python-based multitasking application emerges from the intersection of user demands for seamless task management and the capabilities of modern programming languages. This project aims to create a sophisticated, all-encompassing application that unifies diverse functionalities, offering users a comprehensive and efficient multitasking experience. The project aims to reduce the user to toggle between tabs to perform each task in a separate window, so integration of necessary tasks has been provided in single application such as the Spotify music player which Can allow the user to play the songs up to their likings as the application syncs with the personal Spotify account the user can gain access to their own playlists that they listen to In their life to indulge themselves in the world of music. Many coders are delighted to listen music being played in the background while they code to improve their concentration and to keep themselves engaged to finish the task. The user can play the next or previous song in the playlist and can also change the playlists. The album covers of the song being played is also visible to make it look more user friendly and captivating to the eyes of the user. The shell is also provided in the interface so that user can execute BASH commands in the same interface and reducing their need to switch between multiple windows. Given a python coder at times one might have to execute BASH commands to install modules using ‘**pip install module\_name**’ command also they might need to locate a file ‘**dir**’ or **‘ls -l**’ ‘**pwd**’ to view in what directory it’s present, to change the file’s location using ‘**mv file1.txt Documents/**’ to remove a file ‘rm’ and ‘**echo "Hello, World!”**’ to print some text in the console and ‘**chmod +x script.sh**’ so one can access the shell provided in to execute these commands to reduce the amount of time one wastes to toggle between the tabs. The clock is also displayed in the format “HH:MM” to help the user to assess the time to plan their activities. Then the interface also provides the user with a To-Do Planner (or tasks) to plan the upcoming tasks that the user needs to accomplish in an ordered manner. The user can add new tasks by entering the name into the input field and press the add button and to delete the tasks once they have finished it by just clicking it and press the delete option. Usually, coders tend to plan the tasks they have to finish within a dead line and hustle just before the deadline so now they can keep track of the Hercules tasks that they have to finish. There are separate apps or websites available for the to do planner but the interface provides it in a simplified manner to easily use it. Then we have also used Google Calendar which is in sync with the google account the user provides hence one can view the present month and the current date is highlighted and also one can view their upcoming events in the google calendar, this is also like a To-do planner but in a bigger picture. Students can sync their college mail ID with the interface so one can view the assignments pending or due dates as they come under the google calendar events. This feature is very much useful to students as you could keep track of the academic events and submissions one has to do within that date and again, we reduce the time wasted in toggling between multiple windows thus Improving Productivity, enabling users to manage multiple tasks such as music streaming, calendar checking, task management, and shell commands within a single interface, enhancing productivity and convenience.

**B. Contextual Relevance**

The project's relevance is deeply rooted in the challenges users face in managing various tasks across different platforms. By addressing this challenge, the application not only showcases the power of GUI and the use of APIs, the different python modules that are offered but also contributes to a paradigm shift in how users interact with multitasking tools.

**II. Objectives**

**A. Detailed Functional Objectives**

1. **Spotify Player Integration:**
   * Develop a robust interface to interact with the Spotify API.
   * Implement features for playlist navigation, track control, and real-time updates.
   * Design an intuitive display for the currently playing track, including album art (Thumb Nail or album cover image).
2. **To-Do Planner Module:**
   * Architect a comprehensive to-do planner with functionalities for task addition and deletion.
   * Implement a visually appealing user interface for efficient task management.
3. **Shell GUI for Command Execution:**
   * Create a flexible shell GUI allowing users to execute a wide range of shell commands.
   * Capture and display the output of executed commands within the application.
   * Develop a mechanism to clear the command output for improved user interaction.
4. **Clock Display:**
   * Engineer a real-time clock module with precise timekeeping and updates.
   * Ensure accuracy in time display and synchronization with the system clock.
5. **Google Calendar Integration:**
   * Establish a seamless connection with the Google Calendar API.
   * Implement features to list upcoming events and display them in a user-friendly manner.
   * Provide a calendar view with highlighted events for intuitive scheduling.

**B. Technical Objectives**

1. **API Utilization:**
   * Employ PySimpleGUI for constructing a visually appealing and responsive user interface.
   * Interact with the Spotify API for music-related functionalities. Use the current song played information like the name and the album cover image also the playlists used by the user and the information to display when we play the next song or previous song.
   * Harness the power of the Google Calendar API for efficient calendar integration. The current month can be viewed and the present day’s date is highlighted. The upcoming events that the user’s calendar has can be viewed which is one click away.
2. **Cross-Platform Compatibility:**
   * Ensure the application's compatibility with various operating systems, including Windows, macOS, and Linux.
   * Conduct rigorous testing on different Python versions to guarantee seamless performance.

**III. Scope**

**A. Functional Scope**

1. **Music Playback:**
   * Allow users to play and control Spotify playlists.
   * Facilitate seamless navigation through playlists and tracks.
2. **Task Management:**
   * Create a to-do planner module for adding and deleting tasks.
3. **Shell Command Execution:**
   * Develop a shell GUI for executing a variety of shell commands.
   * Enhance user experience by capturing and displaying command outputs.
4. **Clock Display:**
   * Implement a real-time clock for accurate timekeeping.
5. **Google Calendar Integration:**
   * Integrate Google Calendar features for efficient event management.

**B. Technical Scope**

1. **PySimpleGUI:**
   * Leverage PySimpleGUI for building a dynamic and visually appealing user interface.
   * Provides various functionalities to improve the UI design to be much more captivating
   * Also makes the job of the user much easier due to its simplified code implementation to create the layout of each component in the interface
2. **API Interactions:**
   * Establish secure interactions with the Spotify and Google Calendar APIs.
   * By integrating with the respective personal accounts of the user, the user gets to pick their personalized songs and playlists
   * Also the Google Calendar provides the list of upcoming events
3. **Operating System Compatibility:**
   * Ensure the application's adaptability to different operating systems through comprehensive testing.
   * As the interface has been coded in python with the commonly used modules it ensures the cross platform compatibility when used in different operating systems like Windows, Mac , Linux , Ubuntu etc.

**IV. Novelty and Innovation**

**A. User-Centric Design**

The application's novelty lies in its meticulous attention to user experience. By combining diverse functionalities into a cohesive interface, the application prioritizes user convenience and efficiency.

**B. Real-Time Integration**

The innovative aspect extends to real-time updates, ensuring that the Spotify player and clock display provide accurate and immediate information to the user.

**C. Comprehensive Task Management**

The application innovatively addresses task management by offering a well-rounded system that seamlessly incorporates to-do planning, shell command execution, and calendar integration.

**V. Comparative Statement**

**A. Existing Landscape**

As of the last knowledge update in January 2022, there was a notable gap in the availability of multitasking applications with a similar scope. The proposed application, therefore, fills a void by providing a multifaceted tool that integrates various functionalities into one platform.

**B. Comparative Advantage**

The application gains a competitive advantage through its holistic approach, offering users a comprehensive multitasking environment. By consolidating features that are traditionally dispersed across different applications, it stands out in terms of user convenience and efficiency.

**VI. Dataset**

The application, in its current form, does not heavily rely on external datasets. However, it efficiently interacts with external APIs such as Spotify and Google Calendar to fetch real-time data. The internal management of the to-do list further reduces dependencies on external datasets.

**VII. Test Bed**

**A. Comprehensive Testing**

The application has undergone extensive testing to ensure robust performance and compatibility. Testing environments include:

1. **Operating Systems:**
   * Windows: Versions 7, 8, 10.
   * macOS: Catalina, Big Sur.
   * Linux: Ubuntu, Fedora.
2. **Python Versions:**
   * 3.6, 3.7, 3.8, 3.9.

**B. Performance Metrics**

Testing not only verifies the compatibility of the application but also assesses its performance under varying conditions. Metrics include:

* **Responsiveness:** Assessing the user interface's speed and responsiveness.
* **Resource Utilization:** Monitoring CPU and memory usage during different operations.

**VIII. Expected Result**

Upon execution, users can expect an immersive experience characterized by:

1. **Visual Appeal:**
   * A visually appealing and intuitive user interface.
2. **Seamless Functionality:**
   * Flawless execution of Spotify playback, to-do planning, shell command execution, clock display, and Google Calendar integration.
3. **User-Friendly Interactions:**
   * Smooth transitions between tasks and intuitive controls.
4. **Real-Time Updates:**
   * Accurate and real-time updates for the Spotify player and clock.

**CHAPTER II**

**ARCHITECTURE**

**BLACK – BOX DESIGN (HIGH LEVEL DESIGN) :**

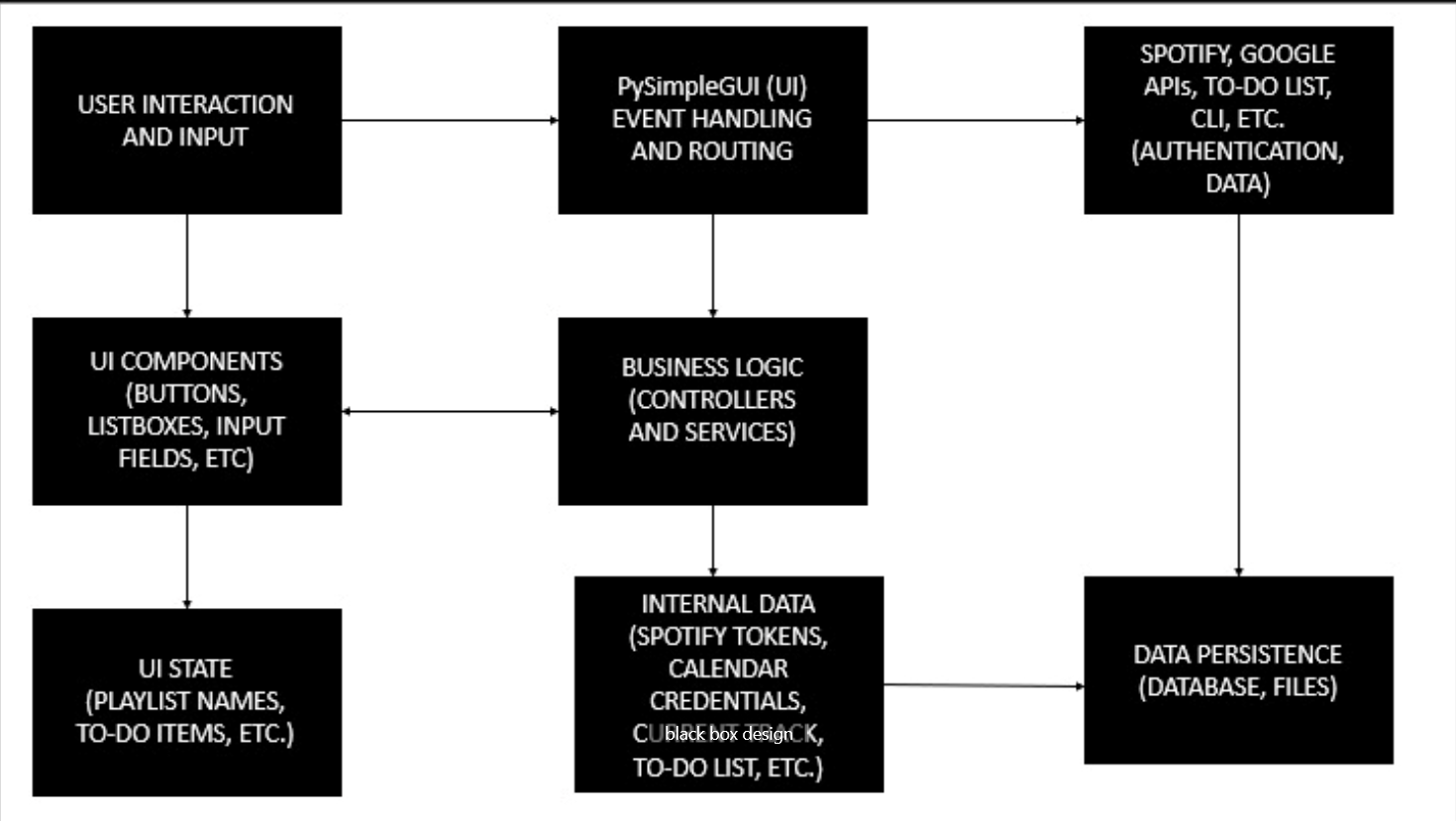


FIGURE 1: HIGH – LEVEL DESIGN OF THE APPLICATION

**Page 1: User Interaction & Input**

The initial point of interaction between the user and the application involves various UI elements. Users provide input through buttons, listboxes, and input fields, triggering events processed by UI components. This interaction initiates the flow of the application.

**Page 2: UI Components**

UI components, comprising graphical elements, include buttons for music playback control, listboxes for playlist selection, and input fields for executing commands. These components directly reflect the user interface, serving as conduits for user input.

**Page 3: UI State**

The UI state acts as a dynamic repository storing information pertinent to the current application state. Playlist names, to-do items, and other data are maintained here. UI components query and update this state based on user interactions, ensuring a seamless user experience.

**Page 4: Business Logic (Controllers & Services)**

The business logic layer interprets and processes user actions through controllers and services. Controllers manage event routing, while services perform specific tasks like managing Spotify playback, accessing Google Calendar data, or handling to-do lists.

**Page 5: Internal Data**

Internal data managed by the application, such as Spotify tokens, Google Calendar credentials, and the current track being played, is discussed in this section. These internal datasets are updated by services and controllers, crucial for the application's functionality.

**Page 6: External APIs & Services**

External APIs and services represent connections to platforms like Spotify and Google. Authentication services ensure secure communication. These interfaces enable the application to control music playback, access Google Calendar events, and manage to-do lists.

**Page 7: Data Persistence**

This section covers the storage of application data, like user preferences or to-do lists, in a persistent format. Utilizing databases or files, data persistence allows the application to retain user-specific information across different sessions.

**Page 8: Event Handling & Routing (Controller)**

The event handling and routing layer orchestrates the application's flow. It receives events from UI components, directs them to the appropriate controllers, and updates the UI state. This layer ensures a smooth flow of information between UI and business logic.

**Page 9: Architecture Overview**

The final page provides an overview of the entire architecture, highlighting the interconnected relationships between user interaction, UI components, business logic, external services, and data storage. It illustrates the seamless flow of information throughout the system. Adjustments can be made based on specific functionalities and intricacies of your application.

**LOW LEVEL DESIGN :**

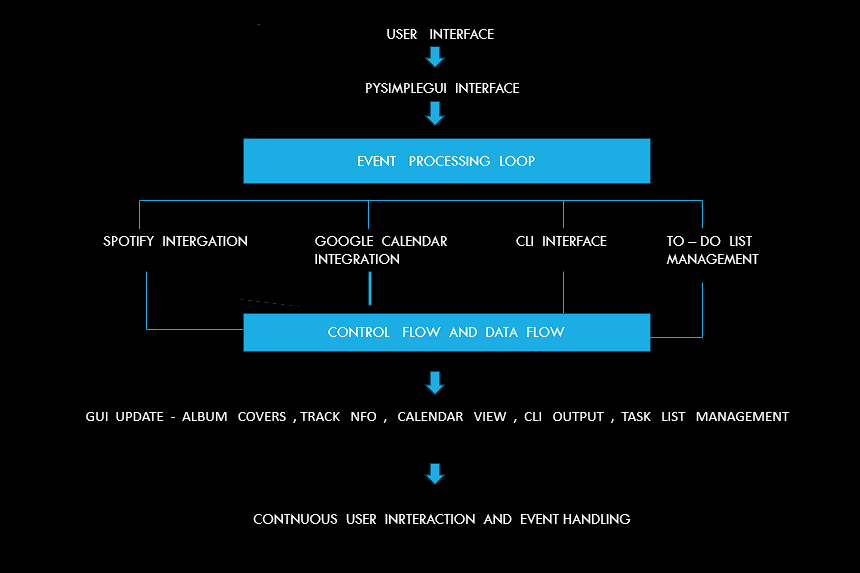


FIGURE 2: LOW – LEVEL DESIGN OF THE APPLICATION

**COMPONENTS:**

**I. Spotify Integration:**

**Authentication:**

Utilizes SpotifyOAuth to authenticate users securely, allowing access to their Spotify accounts and playlists.

**Functionality:**

Music Playback Control: Enables users to play, pause, skip tracks, and manage playlists.

Playlist Retrieval: Fetches user playlists and track information for display within the GUI.

**Interaction:**

**PySimpleGUI Interface:** Communicates playlist selections and playback controls triggered by user interactions.

**Data Flow:**

Interacts with Spotify's API to retrieve music data and playback controls.

Provides album covers, track info, and playlist details for display in the GUI.

**II . Google Calendar Integration**:

**Authentication:**

Utilizes Google OAuth for secure user authentication, granting access to their calendar events.

**Functionality:**

**Event Retrieval:** Fetches calendar events and schedules for the current user.

**Calendar View Display**: Presents a calendar view within the GUI, highlighting events and current dates.

**Interaction:**

**PySimpleGUI Interface:** Communicates calendar display events and interactions triggered by user requests.

**Data Flow**:

Interacts with Google Calendar API to fetch events and schedules.

Provides event details and calendar views for presentation within the GUI.

**III. CLI Interface:**

**Execution:**

Shell Command Execution: Executes user-entered shell commands securely within the application environment.

Output Capture: Captures the output of executed commands for display within the GUI.

**Interaction:**

PySimpleGUI Interface: Displays the CLI interface, executes commands based on user inputs, and presents output within the GUI.

**Data Flow:**

Executes shell commands within a subprocess and captures the resulting output for GUI presentation.

Communicates with the GUI to display command execution results or errors.

**IV. To-Do List Management:**

**Functionality:**

**Task Management:** Allows users to add, delete, and manage tasks within the GUI.

**Display:** Presents an updated list of tasks for user reference and management.

Interaction:

**PySimpleGUI Interface:** Captures user inputs to add/delete tasks and displays the updated task list.

**Data Flow:**

Manages task additions, deletions, and updates within the application's task management system.

Updates the GUI to reflect changes in the task list.

**V. PySimpleGUI-Based GUI:**

**Layout and Interaction:**

User Interface: Provides a graphical interface for users to interact with various functionalities seamlessly.

Event Handling: Captures user interactions (clicks, inputs) and triggers corresponding actions.

**Data Flow :**

**Component Interaction:** Orchestrates interactions between different components based on user requests.

**GUI Update:** Updates display elements based on data received from components or user actions.

Each component in the multitasking application plays a specific role, interacts with the PySimpleGUI interface, and contributes to the overall user experience by providing distinct functionalities and managing their respective data flow and interactions

**DATA FLOW:**

**1. User Interaction**:

**Description:**

Users engage with the PySimpleGUI interface, triggering events by interacting with various elements such as buttons, text inputs, or selection options.

**Data Flow:**

Event Generation: User interactions generate events captured by the PySimpleGUI interface.

Event Routing: These events are routed to respective component handlers based on associated elements (Spotify controls, calendar view, CLI input, task list).

**2. Component Operations:**

**Description**:

Each component (Spotify, Calendar, CLI, To-Do List) processes events or commands triggered by user interaction within the PySimpleGUI interface.

**Data Flow:**

**Event Processing:** Components receive and process events specific to their functionalities.

**API Interactions:** Interact with external APIs or data sources to retrieve relevant information (e.g., fetching playlists, calendar events, executing shell commands).

**Data Processing:** Process received information and prepare it for display or further actions.

**3. Data Retrieval and Presentation:**

**Description:**

Components interact with respective APIs or data sources to obtain information, presenting it within the GUI for user visibility.

**Data Flow:**

**API Interaction:** Spotify and Google Calendar components interact with respective APIs to fetch music data, playlists, calendar events, and schedules.

**Data Formatting:** Formats retrieved data (album covers, track info, event details) for GUI presentation.

**GUI Update:** Updates display elements (album covers, calendar view, task lists) based on received data for user visibility.

**4. GUI Update and Interaction Loop:**

**Description:**

The PySimpleGUI interface continuously updates based on received data and user interactions, creating a loop of interaction and display updates.

**Data Flow:**

**Event Handling:** Captures user interactions and triggers corresponding actions within components.

**Component Update:** Components update GUI elements based on processed events or received data.

**User Feedback:** GUI elements get refreshed, displaying new information, command outputs, or task updates as feedback to user actions.

**5. Continuous User Interaction Loop:**

**Description:**

The application maintains an ongoing loop, continuously listening for user events and updating the GUI accordingly.

**DATA FLOW :**

**Event Continuity:** The loop continuously captures user events and processes them through the component handlers.

**GUI Continuity:** GUI elements persistently update based on new data or user interactions, maintaining an interactive and responsive interface.

The data flow within the multitasking application ensures seamless interaction between the user, GUI interface, and integrated components. It facilitates the continuous exchange of information, enabling users to control various functionalities and receive real-time feedback within a unified interface.

**CONTROL FLOW:**

**1. Initialization**:

**Description:**

The application initializes necessary components, authenticates with external services, and fetches initial data.

**Control Flow:**

**Authentication:** Establishes connections with Spotify and Google Calendar APIs through authentication mechanisms.

**Data Retrieval:** Fetches initial data such as user playlists, calendar events, and schedules.

**2. Main Loop:**

**Description:**

The application enters a continuous loop, perpetually listening for user events and interactions within the PySimpleGUI interface.

**Control Flow:**

**Event Capture:** Captures user events triggered within the GUI, such as button clicks, text inputs, or selections.

**Event Handling:** Routes events to respective component handlers based on associated elements or functionalities.

**3. Event Processing:**

**Description:**

Events captured within the PySimpleGUI interface are processed to trigger actions within specific components.

**Control Flow:**

**Component-Specific Operations:** Each component processes relevant events, performing operations based on user interactions.

**API Interactions:** Components interact with external APIs or data sources to retrieve necessary information or execute commands.

**4. GUI Update and User Feedback:**

**Description:**

The GUI elements get updated based on processed events or received data, providing user feedback and display updates.

**Control Flow:**

**Component Updates:** Components update GUI elements (album covers, calendar views, task lists) with new data or results of operations.

**User Feedback Display:** Command outputs, task updates, or fetched information are displayed within the GUI as user feedback.

**5. Continuous User Interaction Loop:**

**Description:**

The application maintains a continuous loop, persistently listening for user events and updating the GUI based on interactions.

**Control Flow:**

**Event Continuity:** Continuously listens for user interactions, perpetuating the flow of actions and data processing.

**GUI Continuity:** GUI elements persistently update based on new data or user interactions, providing a seamless and responsive interface.

The control flow within the multitasking application ensures that user events are captured, processed by respective components, and reflected in the GUI interface. This continuous loop facilitates user interactions, data processing, and real-time updates, creating a responsive and interactive application environment.

**CHAPTER III**

**SIMULATION / IMPLEMENTATION RESULTS**

Various Algorithms Followed:

**Spotify Authentication and Playback:**

* Authorization Code Flow: The code uses the Spotify API's Authorization Code Flow to obtain and refresh access tokens for user authentication.
* Playback Control: Algorithms for playing, pausing, and skipping tracks involve interacting with the Spotify Web API.

**Google Calendar Integration:**

* OAuth 2.0 Authentication: The code uses OAuth 2.0 for Google Calendar authentication.
* Google Calendar API Interaction: Algorithms for listing upcoming events and displaying a calendar view involve interacting with the Google Calendar API.

**PySimpleGUI for GUI Design:**

* Event-Driven Programming: The GUI is designed using PySimpleGUI, which follows an event-driven programming paradigm. The program responds to user events (button clicks, etc.) to trigger specific actions.
* Multithreading: A thread is used to continuously check if the current Spotify track is playing and automatically play the next track if it's not.

**To-Do List Management:**

* List Operations: Algorithms for adding and deleting tasks from the to-do list involve basic list operations.
* GUI Update: The GUI is updated dynamically to reflect changes in the to-do list.

**CLI Integration:**

* Subprocess Execution: The subprocess module is used to execute shell commands and capture the output.
* Thread Handling: A thread is used to continuously check if the current Spotify track is playing and automatically play the next track if it's not.

**Calendar View GUI:**

* Calendar Rendering: The calendar view involves rendering a calendar for the current month with highlighted days.
* GUI Pop-Up: The calendar view is presented as a pop-up window.

**Image Handling:**

* Pillow (PIL) Library: The code uses the Pillow library to handle album cover images from Spotify track information.

**Algorithm for Implementation:**

**Authenticate with Spotify:**

* Use SpotifyOAuth to get access tokens for Spotify API.
* If tokens are not available, open the authorization URL in the user's default web browser.

**List User Playlists:**

* Fetch the user's playlists using the current\_user\_playlists method.
* Display the playlist names in a PySimpleGUI Listbox.

**Play Playlist:**

* When the user selects a playlist and clicks the play button:
* Get the selected playlist ID.
* Retrieve the tracks from the playlist.
* Extract the URI of the first track.
* Start playback using start\_playback with the track URI.

**Pause Playback:**

* When the user clicks the pause button, pause the playback using pause\_playback.

**Next Track:**

* When the user clicks the next button:
* Retrieve the next track URI from the playlist.
* Start playback using start\_playback with the next track URI.

**Previous Track:**

* When the user clicks the previous button:
* Retrieve the previous track URI from the playlist.
* Start playback using start\_playback with the previous track URI.

**Algorithm for Google Calendar Integration:**

**Authenticate with Google Calendar:**

* Use OAuth2 to authenticate with Google Calendar API.
* If credentials are not available, initiate the OAuth flow and save the credentials.

**List Calendar Events:**

* Fetch upcoming calendar events using events().list.
* Display event details or show a message if no upcoming events are found.

**Show Calendar Popup:**

* When the user clicks the "Show Calendar" button:
* Create a calendar popup using PySimpleGUI with the current month displayed.
* Highlight the current date.

**Algorithm for To-Do List Integration:**

**Create To-Do Popup:**

* Initialize an empty to-do list.
* When the user clicks the "Tasks" button:
* Create a to-do popup using PySimpleGUI.
* Display the to-do list and input field.

**Add Task:**

* When the user adds a task:
* Retrieve the task from the input field.
* Add the task to the to-do list.
* Update the PySimpleGUI listbox.

**Delete Task:**

* When the user deletes a task:
* Retrieve the selected task from the listbox.
* Remove the task from the to-do list.
* Update the PySimpleGUI listbox.

**Algorithm for CLI Integration:**

**Create CLI Popup:**

* When the user clicks the "Shell" button:
* Create a CLI popup using PySimpleGUI.
* Display an input field for the shell command and buttons for execution.

**Execute Command:**

* When the user enters a shell command and clicks "Execute":
* Retrieve the command from the input field.
* Use subprocess.run to execute the command.
* Display the command output in the PySimpleGUI window.

**Clear Screen:**

* When the user clicks "Clear Screen," clear the output area in the PySimpleGUI window.
* Algorithm for Continuous Playback Check:

**Continuous Playback Check Thread:**

* Start a separate thread to continuously check if a track is playing.
* If the track is not playing, skip to the next track using sp.next\_track.

**Thread Management:**

* Ensure proper management of the thread to avoid infinite threads.
* Start the thread when needed and stop it when the program exits or the user pauses playback.

**Main Algorithm:**

**Initialize:**

* Call the initialization functions for Spotify and Google Calendar.

**Create GUI:**

* Create the main PySimpleGUI window with columns for Spotify, Google Calendar, To-Do List, and CLI.

**Event Loop:**

* Enter the main event loop to handle user interactions.
* Call appropriate functions based on button clicks or events.

**Update UI:**

* Continuously update the UI with the current Spotify track, Google Calendar events, and other dynamic content.

**Exit:**

* Properly close the program and release resources when the user closes the window.
* This algorithmic breakdown covers the core functionalities and interactions of your PySimpleGUI-based multitasking application.

**RESULT IN DATA (Photos of Implementation):**

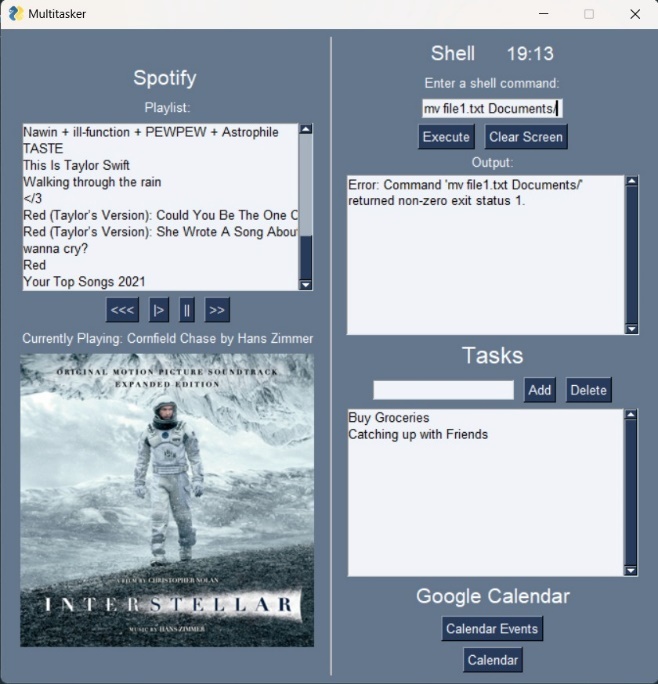


Fig: 3.1 Multitasker is executing ‘mv file1.txt Documents’ in the shell and playing a song from a playlist.

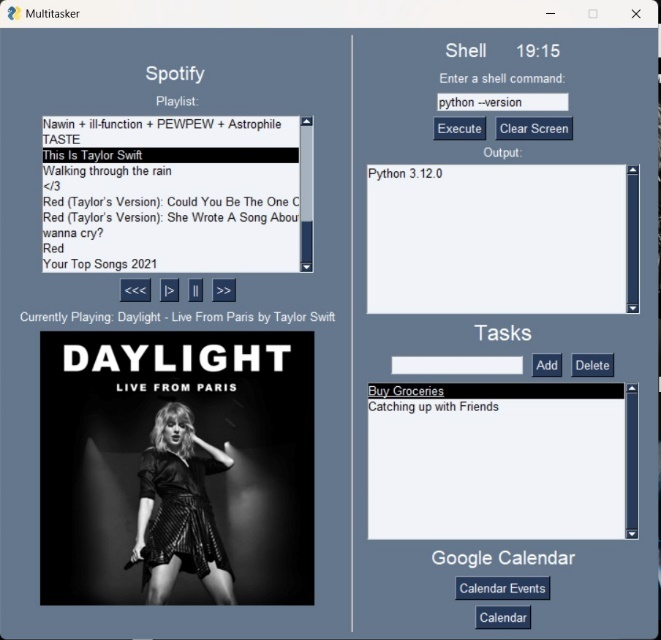


Fig: 3.2 Multitasker is executing ‘python --version’ in the shell and playing a song from a playlist.



Fig: 3.3 Multitasker is executing ‘pip install openai in the shell and playing a song from a playlist.

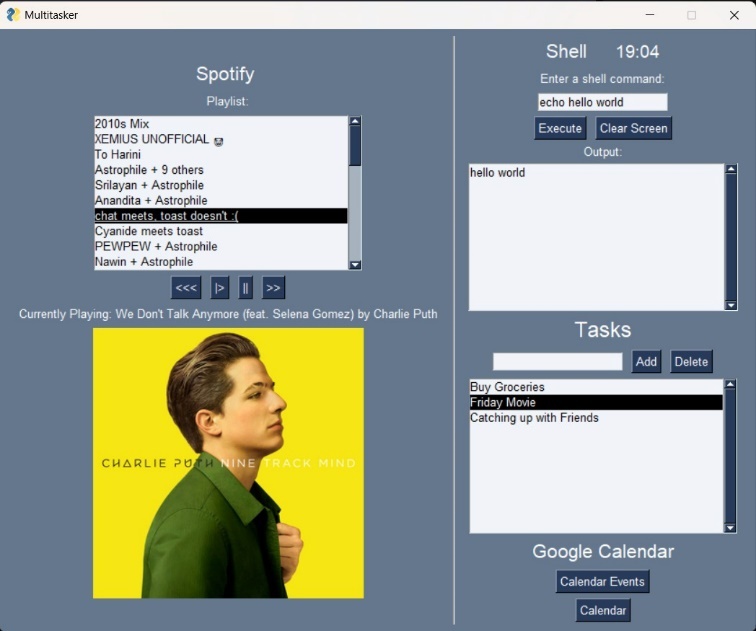


Fig: 3.4 Multitasker is executing ‘echo hello world’ in the shell and playing a song from a playlist.

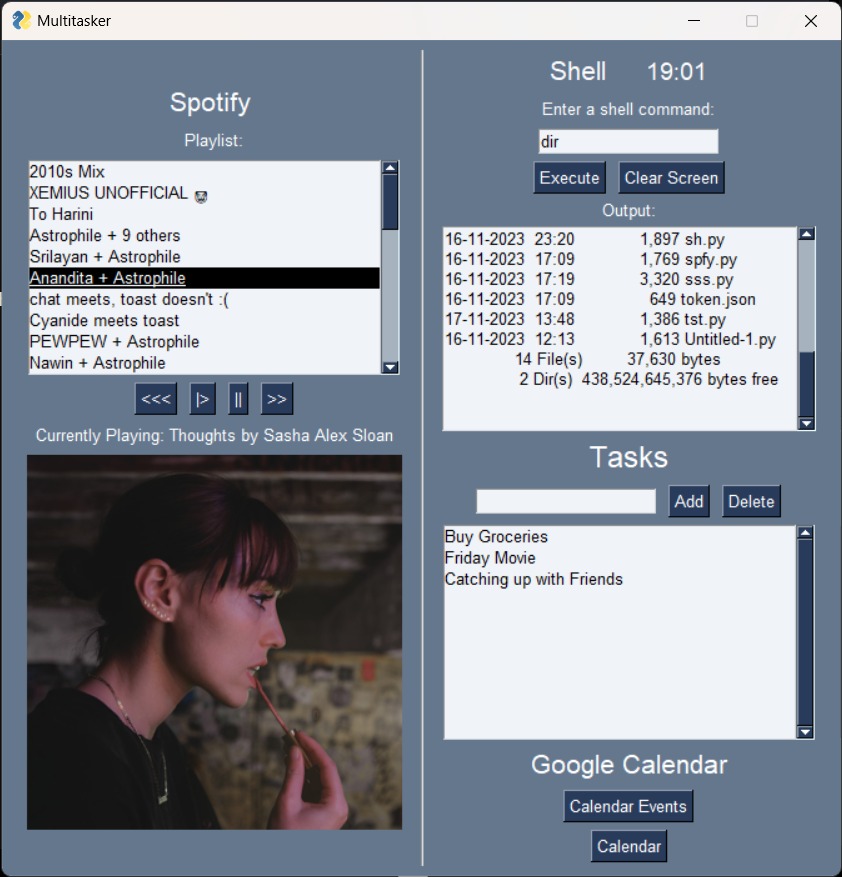


Fig: 3.5 Multitasker is executing ‘dir’ in the shell and playing a song from a playlist.

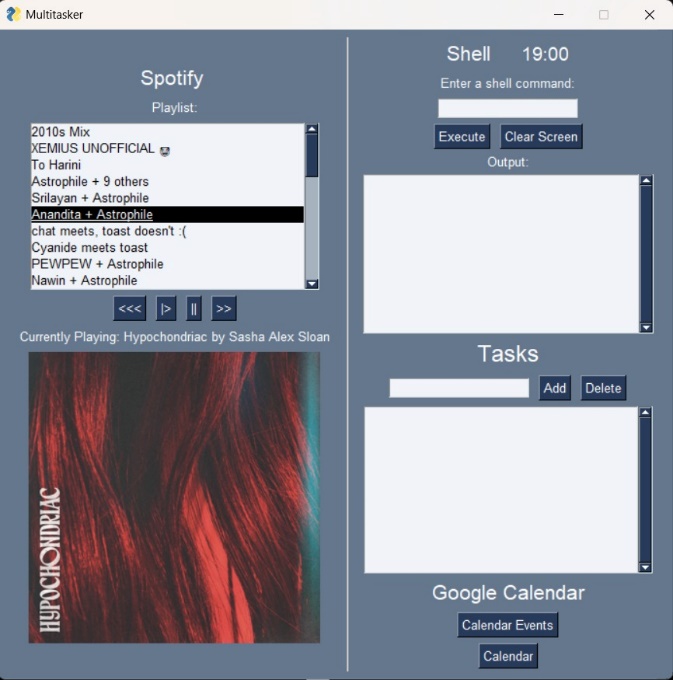


Fig: 3.6 Multitasker is executing no commands in the shell and playing a song from a playlist.

**MAPPING THE RESULTS WITH PROBLEM STATEMENTS:**

Spotify Player Integration (Music Playback)

Solution Algorithm:

**Authenticate with Spotify:**

* Use the SpotifyOAuth class from spotipy to authenticate with the Spotify API.
* If the access token is not available, open the authorization URL in the user's web browser for authorization.

**List User Playlists:**

* Call the current\_user\_playlists method to get a list of the user's playlists.
* Extract and display the playlist names.

**Play Playlist:**

* When the user selects a playlist and clicks the play button:
* Get the selected playlist ID.
* Retrieve the tracks from the playlist.
* Extract the URI of the first track.
* Start playback using start\_playback with the track URI.

**Pause Playback:**

* When the user clicks the pause button, pause the playback using pause\_playback.

**Next Track**:

* When the user clicks the next button:
* Retrieve the next track URI from the playlist.
* Start playback using start\_playback with the next track URI.

**Previous Track:**

* When the user clicks the previous button:
* Retrieve the previous track URI from the playlist.
* Start playback using start\_playback with the previous track URI.

**To-Do Planner Module (Task Management)**

Solution Algorithm:

**Create To-Do Popup:**

* Initialize an empty list for tasks.
* When the user clicks the "Tasks" button:
* Create a popup window using PySimpleGUI.
* Display the to-do list and an input field for adding tasks.

**Add Task:**

* When the user adds a task:
* Retrieve the task from the input field.
* Add the task to the to-do list.
* Update the PySimpleGUI listbox with the updated to-do list.

**Delete Task:**

* When the user deletes a task:
* Retrieve the selected task from the listbox.
* Remove the task from the to-do list.
* Update the PySimpleGUI listbox with the updated to-do list.

**Shell GUI for Command Execution:**

Solution Algorithm:

**Execute Command:**

* When the user enters a shell command and clicks "Execute":
* Retrieve the command from the input field.
* Use subprocess.run to execute the command.
* Display the command output in the PySimpleGUI window.

**Clear Screen:**

* When the user clicks "Clear Screen," clear the output area in the PySimpleGUI window.

Clock Display

Solution Algorithm:

**Display Clock:**

* Create a PySimpleGUI Text element to display the current time.
* Continuously update the text with the current time.

**Google Calendar Integration:**

Solution Algorithm:

**Authenticate with Google Calendar:**

* Use OAuth2 to authenticate with the Google Calendar API.
* If credentials are not available, initiate the OAuth flow and save the credentials.

**List Calendar Events:**

* When the user clicks the "Calendar Events" button:
* Fetch upcoming calendar events using events().list.
* Display event details or show a message if no upcoming events are found.

**Show Calendar Popup:**

* When the user clicks the "Calendar" button:
* Create a calendar popup using PySimpleGUI with the current month displayed.
* Highlight the current date.

**CHAPTER IV**

**CONCLUSION AND FUTURE WORKS**

**5.1 Conclusion**

In conclusion, the proposed multitasking Python application signifies a revolutionary approach to user-centric design and technical innovation. By seamlessly integrating a Spotify player, to-do planner, shell GUI, clock, and Google Calendar features, the application transcends conventional boundaries. Its real-time updates and holistic task management redefine user experiences, offering a dynamic and responsive multitasking environment. The competitive advantage lies in its comprehensive solution, addressing music enjoyment, task completion, and calendar management in a single platform. Beyond the problem statement, this application signifies the commencement of a journey into a digital landscape where multifunctional tools converge into a harmonious whole, setting a new standard for the potential inherent in creative problem-solving within software development.

**5.2 Future works**

**A. Iterative Improvements**

Future iterations of the application may explore enhancements such as:

1. **Additional Music Player features:**
   * Integration with other music streaming services.
   * Add a queue to make a set of next songs the user wants to hear
   * Add the option to loop a particular song
   * Add an option to shuffle a given playlist
2. **Advanced Task Management:**
   * Incorporating features like priority levels and categorization in the to-do planner.
   * Make it in Sync with the college’s website to display the upcoming assignments
3. **Customization Options:**
   * Allowing users to customize the interface based on preferences.
   * To change the theme to dark and light or up to the user’s liked colour palette
4. **Integration with Productivity Tools:**
   * Exploring integrations with other popular productivity tools.
   * A chat bot can be integrated to fulfil the user’s query

**CHAPTER V**

**REFERENCES**

1. https://www.pysimplegui.org/en/latest/
2. https://www.javatpoint.com/spotify-api-python
3. https://developers.google.com/calendar/api/guides/overview
4. https://downloads.hindawi.com/journals/misy/2017/2012696.pdf
5. https://www.researchgate.net/publication/334987891\_Temporal\_Trends\_in\_Music\_Popularity\_-\_A\_Quantitative\_analysis\_of\_Spotify\_API\_data
6. https://opensource.com/article/18/8/pysimplegui
7. https://www.researchgate.net/publication/334145268\_Application\_Programming\_Interface\_API\_Research\_A\_Review\_of\_the\_Past\_to\_Inform\_the\_Future
8. https://github.com/googleworkspace/python-samples/blob/main/calendar/quickstart/quickstart.py
9. https://github.com/googleapis/google-auth-library-python
10. https://www.youtube.com/watch?v=j1mh0or2CX8
11. https://developers.google.com/identity/protocols/oauth2/web-server
12. https://developers.google.com/calendar/api/guides/create-events

**CHAPTER VI**

**Appendix**

**CODE:**

import PySimpleGUI as sg

from google.oauth2.credentials import Credentials

from google\_auth\_oauthlib.flow import InstalledAppFlow

from google.auth.transport.requests import Request

from googleapiclient.discovery import build as build\_calendar

from spotipy.oauth2 import SpotifyOAuth

import spotipy

import webbrowser

import datetime

import os.path

import calendar

import subprocess

import time

import threading

from PIL import Image, ImageTk

from io import BytesIO

import requests

import datetime

# Spotify API credentials

SPOTIFY\_CLIENT\_ID = '480508544b3d4621bb22064ff2174374'

SPOTIFY\_CLIENT\_SECRET = 'cac951ef201f4bc9b99c990fb8e9fe08'

SPOTIFY\_REDIRECT\_URI = 'http://localhost:5000/callback'  # Valid redirect URI for local development

# Google Calendar API credentials

GOOGLE\_SCOPES = ['https://www.googleapis.com/auth/calendar']

def authenticate\_spotify():

    sp\_oauth = SpotifyOAuth(SPOTIFY\_CLIENT\_ID, SPOTIFY\_CLIENT\_SECRET, SPOTIFY\_REDIRECT\_URI,

                            scope='user-read-playback-state user-modify-playback-state playlist-read-private')

    token\_info = sp\_oauth.get\_access\_token()

    if not token\_info:

        auth\_url = sp\_oauth.get\_authorize\_url()

        webbrowser.open(auth\_url)

        return None

    return spotipy.Spotify(auth=token\_info['access\_token'])

def authenticate\_google\_calendar():

    creds = None

    if os.path.exists('calendar\_token.json'):

        creds = Credentials.from\_authorized\_user\_file('calendar\_token.json')

    if not creds or not creds.valid:

        if creds and creds.expired and creds.refresh\_token:

            creds.refresh(Request())

        else:

            flow = InstalledAppFlow.from\_client\_secrets\_file(

                './client\_secret\_842294873082-obs81j2v0okr85j2ddpnirp1u5qn4prt.apps.googleusercontent.com.json', GOOGLE\_SCOPES)

            creds = flow.run\_local\_server(port=0)

        with open('calendar\_token.json', 'w') as token:

            token.write(creds.to\_json())

    return creds

scopes = "user-library-read user-read-playback-state user-modify-playback-state"

sp = spotipy.Spotify(auth\_manager=SpotifyOAuth(client\_id=SPOTIFY\_CLIENT\_ID, client\_secret=SPOTIFY\_CLIENT\_SECRET, redirect\_uri=SPOTIFY\_REDIRECT\_URI, scope=scopes))

# Fetch user's playlists

user\_playlists = sp.current\_user\_playlists()

playlist\_names = [playlist['name'] for playlist in user\_playlists['items']]

def list\_spotify\_playlists(spotify):

    playlists = spotify.current\_user\_playlists()

    return [playlist['name'] for playlist in playlists['items']]

def play\_spotify\_playlist(spotify, playlist\_name):

    playlists = spotify.current\_user\_playlists()

    playlist\_id = None

    for playlist in playlists['items']:

        if playlist['name'] == playlist\_name:

            playlist\_id = playlist['id']

            break

    if playlist\_id:

        tracks = spotify.playlist\_tracks(playlist\_id)

        if tracks['items']:

            track\_uri = tracks['items'][0]['track']['uri']

            spotify.start\_playback(uris=[track\_uri])

def create\_calendar\_popup(current\_day):

    # Show calendar of the current month and highlight the current date

    now = datetime.datetime.now()

    year = now.year

    month = now.month

    cal = calendar.monthcalendar(year, month)

    # Add formatting to highlight the current date

    layout = [[sg.Text(f'{calendar.month\_name[month]} {year}', font=('Helvetica', 14))]]

    for week in cal:

        formatted\_week = []

        for day in week:

            if day == 0:

                formatted\_week.append(sg.Text('', size=(3, 2)))

            elif day == current\_day:

                formatted\_week.append(sg.Text(f'{day}', size=(3, 2), background\_color='#001F3F', text\_color='white'))

            else:

                formatted\_week.append(sg.Text(f'{day}', size=(3, 2)))

        layout.append(formatted\_week)

    layout.append([sg.Button('Close')])

    return sg.Window('Calendar View', layout, keep\_on\_top=True, finalize=True)

def execute\_command(command):

    try:

        result = subprocess.run(

            command,

            shell=True,

            text=True,

            capture\_output=True,

            check=True

        )

        output = result.stdout

        return output

    except subprocess.CalledProcessError as e:

        return f"Error: {e}"

def play\_next\_track():

    while True:

        current\_track = sp.current\_playback()

        if current\_track is not None and not current\_track['is\_playing']:

            print("playing next track")

            sp.next\_track()

        time.sleep(1)

todo\_items=[]

def main():

    # Authenticate with Spotify

    spotify = authenticate\_spotify()

    if not spotify:

        sg.popup('Please authorize Spotify access in the browser window.')

        return

    # Authenticate with Google Calendar

    calendar\_credentials = authenticate\_google\_calendar()

    calendar\_service = build\_calendar('calendar', 'v3', credentials=calendar\_credentials)

    # Layout for PySimpleGUI window

    layout = [

        [sg.Column([

            [sg.Text('Spotify ', font=('Helvetica', 16))],

            [sg.Text('Playlist:')],

            [sg.Listbox(values=playlist\_names, size=(40, 10), key='-PLAYLISTS-', enable\_events=True)],

            [sg.Button('<<<', key='-PREVIOUS\_BUTTON-'),sg.Button('|>', key='-PLAY\_BUTTON-'),sg.Button('||', key='-PAUSE\_BUTTON-'),sg.Button('>>', key='-NEXT\_BUTTON-')],[sg.Text('Currently Playing: ', key='-CURRENTLY\_PLAYING-')],

            [sg.Image(key='-IMAGE-', size=(300, 300))],

        ], element\_justification='center'),

        sg.VerticalSeparator(pad=None),

        sg.Column([

            [sg.Text('Shell   ', font=('Helvetica', 16)),sg.Text(font=("Helvetica", 14), key="-TIME-")],

            [sg.Text("Enter a shell command:")],

            [sg.InputText(key="-COMMAND-",size=(20, 1))],

            [sg.Button("Execute"), sg.Button("Clear Screen")],

            [sg.Text("Output:", pad=(20, 0))],

            [sg.Output(size=(40, 10), key="-OUTPUT-")],

            [sg.Text('Tasks', font=('Helvetica', 18), justification='center')],

            [sg.InputText('', key='-TASK-', size=(20, 1)), sg.Button('Add', key='-ADD-'),sg.Button('Delete', key='-DELETE-')],

            [sg.Listbox(values=todo\_items, size=(40, 10), key='-LIST-', enable\_events=True)],

            #[sg.Button('To-Do List', key='-OPEN\_TODO-',button\_color='#856A5D'),sg.Button('CLI',button\_color='#856A5D')],

            [sg.Text('Google Calendar', font=('Helvetica', 16))],

            [sg.Button('Calendar Events', key='-LIST\_EVENTS-')],

            [sg.Button('Calendar', key='-SHOW\_CALENDAR-',)],

        ], element\_justification='center')],

    ]

    window = sg.Window('Multitasker', layout, finalize=True)

    calendar\_popup = None

    todo\_popup = None

    #todo\_items = []

    cli\_popup = None

    i=0

    running = False

    thread = None

    while True:

        event, values = window.read()

        #i=0;

        if event == sg.WIN\_CLOSED:

            running = False

            break

        elif event == '-PLAY\_BUTTON-':

            selected\_playlist = values['-PLAYLISTS-'][0]

            tracks = sp.playlist\_tracks(user\_playlists['items'][playlist\_names.index(selected\_playlist)]['id'])

            if tracks['items']:

                track\_uri = tracks['items'][0]['track']['uri']

                sp.start\_playback(uris=[track\_uri])

                '''if not running and thread is None:

                running = True

                # Start the thread to check for next track

                thread = threading.Thread(target=play\_next\_track)

                thread.start()'''

        elif event == '-PAUSE\_BUTTON-':

            running = False

            sp.pause\_playback()

        elif event == '-PREVIOUS\_BUTTON-':

            i-=1

            if tracks['items'] and i>=0 and i<len(tracks['items']):

                prev\_track\_uri = tracks['items'][i]['track']['uri']

                sp.start\_playback(uris = [prev\_track\_uri])

                sg.popup(f"Playing next track: {tracks['items'][i]['track']['name']}")

        elif event == '-NEXT\_BUTTON-':

            # Check if there are tracks in the playlist

            i+=1

            if tracks['items'] and i<len(tracks['items']):

                # Get the URI of the next track in the playlist

                next\_track\_uri = tracks['items'][i]['track']['uri']

                # Start playback of the next track in the playlist

                sp.start\_playback(uris=[next\_track\_uri])

                sg.popup(f"Playing next track: {tracks['items'][i]['track']['name']}")

            '''        elif event == '-OPEN\_TODO-':

            if todo\_popup:

                todo\_popup.close()

            todo\_popup = create\_todo\_popup(todo\_list)'''

        elif event == '-ADD-':

            new\_task = values['-TASK-']

            if new\_task:

                todo\_items.append(new\_task)

                window['-LIST-'].update(values=todo\_items)

                window['-TASK-'].update('')

        elif event == '-DELETE-':

            selected\_task = values['-LIST-']

            if selected\_task:

                todo\_items.remove(selected\_task[0])

                window['-LIST-'].update(values=todo\_items)

        elif event == '-SHOW\_CALENDAR-':

            # Show calendar of the current month and highlight the current date

            if calendar\_popup:

                calendar\_popup.close()

            current\_day = datetime.datetime.now().day

            calendar\_popup = create\_calendar\_popup(current\_day)

        elif event == 'Close':

            if calendar\_popup:

                calendar\_popup.close()

            '''        elif event == 'Show CLI':

            cli\_popup = create\_cli\_popup()'''

        if event == "Execute":

            command = values["-COMMAND-"]

            output = execute\_command(command)

            print(output)

        if event == "Clear Screen":

            window["-OUTPUT-"].update("")

            '''elif event == 'Close CLI':

            cli\_popup.close()

            if cli\_popup:

                cli\_popup.close()

                cli\_popup = None'''

        current\_track = sp.current\_playback()

        if current\_track is not None and current\_track['is\_playing']:

            currently\_playing\_text = f"Currently Playing: {current\_track['item']['name']} by {current\_track['item']['artists'][0]['name']}"

            window['-CURRENTLY\_PLAYING-'].update(currently\_playing\_text)

        else:

            window['-CURRENTLY\_PLAYING-'].update("Currently Playing: Nothing")

        track\_info = sp.current\_playback()

        if track\_info and 'item' in track\_info:

            track\_name = track\_info['item']['name']

            album\_name = track\_info['item']['album']['name']

            album\_cover\_url = track\_info['item']['album']['images'][0]['url']

            # Fetch the album cover image

            response = requests.get(album\_cover\_url)

            image\_bytes = BytesIO(response.content)

            album\_cover\_image = Image.open(image\_bytes)

            album\_cover\_image.thumbnail((300, 300))

            # Update the PySimpleGUI window with the album cover

            window['-IMAGE-'].update(data=ImageTk.PhotoImage(album\_cover\_image))

        current\_time = datetime.datetime.now().strftime("%H:%M")

        window["-TIME-"].update(current\_time)

    window.close()

if \_\_name\_\_ == '\_\_main\_\_':

    main()