Abstract

The Cocktail Finder is a user-friendly desktop application designed to facilitate the exploration of cocktail names. Harnessing the capabilities of TheCocktailDB API, this Python application allows users to quickly search for cocktails by the first letter of their names. The application is particularly useful for cocktail enthusiasts and bartenders who need a quick reference to various cocktail names, offering a simple yet effective tool for their craft.

Developed using Python and the Tkinter library, the application features an intuitive graphical user interface (GUI) that prioritizes ease of use and efficiency. Users can input a letter into the application, which then displays a list of cocktail names starting with that letter. This streamlined approach makes the Cocktail Finder an accessible and practical resource for both professionals in the hospitality industry and casual users interested in exploring different cocktails.

While the current version of the application focuses on listing cocktail names, it serves as a solid foundation for future enhancements, such as including detailed recipes, ingredients, and preparation methods. The project's source code is available on GitHub, inviting collaboration and further development from the community.

Link: https://github.com/gbylegaspi/Assessment-2---Advanced-prog.git

Project Plan (5 Days)

Day 1: Planning and Initial Design

Morning (Planning):

Define the core purpose and functionality of the application.

Research TheCocktailDB API for integration possibilities.

Afternoon (Design):

Develop initial wireframes for the GUI layout.

Draft a basic architecture for the application, including class and function outlines.

Day 2: API Integration and GUI Mockup

Morning (API Integration):

Set up Python environment and necessary libraries.

Start developing functions for API calls and handling responses.

Afternoon (GUI Development):

Create a basic GUI mockup using Tkinter, focusing on layout elements like the search bar and results display.

Day 3: GUI Development and Backend Integration

Full Day:

Complete the GUI development, ensuring all elements are in place.

Integrate the backend API logic with the GUI, allowing for real-time data fetching and display.

Day 4: Testing and Refinement

Morning (Testing):

Develop and execute test cases covering all functionalities.

Conduct manual testing to identify bugs or UX issues.

Afternoon (Refinement):

Address any bugs found during testing.

Refine the user interface and experience based on test feedback.

Day 5: Finalization and Documentation

Morning (Finalization):

Perform final checks and ensure the application is running smoothly.

Package the application for distribution if necessary.

Afternoon (Documentation):

Write comprehensive documentation, including a user guide and technical details.

Prepare the project for submission or deployment.

Note:

This condensed schedule assumes efficient time management and may require overlapping different tasks, such as designing while the environment is being set up, or starting documentation before final testing is complete. It’s an intensive timeline, ideal for an experienced developer or a team working collaboratively.

Evidence of Design: Wireframe

The design process for the Cocktail Finder application was significantly guided by a comprehensive wireframe. This wireframe served as a blueprint for the GUI layout, ensuring a user-friendly and intuitive interface. Below is a detailed description of the wireframe and its components:

Overall Layout

The wireframe presents a clean and straightforward layout, prioritizing ease of use and clarity. The GUI is designed to be intuitive, allowing users to interact with the application without needing instructions.

Components of the Wireframe

Search Area:

Located at the top of the window.

Consists of a simple text input field where users can enter a letter.

Accompanied by a 'Search' button to initiate the cocktail search.

Results Display:

Occupies the majority of the GUI, positioned below the search area.

Implemented as a scrollable list box, displaying the names of cocktails that start with the entered letter.

Allows users to scroll through the list to view all search results.

Styling and Aesthetics:

The wireframe suggests a minimalistic approach with a focus on readability and accessibility.

It includes recommendations for a color scheme that is appealing yet not overwhelming, using a light background and contrasting text colors for clarity.

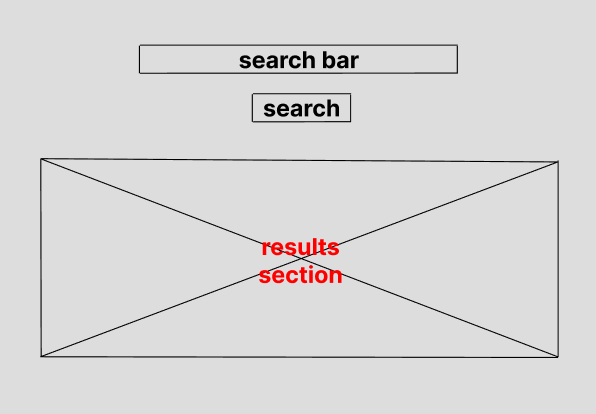
Functionality as Depicted by the Wireframe

The wireframe illustrates a straightforward flow: the user enters a letter, clicks 'Search', and views the list of cocktail names. This simplicity in design ensures that the application is accessible to a wide range of users, from cocktail enthusiasts to professional bartenders.

Importance of the Wireframe in the Development Process

The wireframe was crucial in aligning the development team's vision for the application. It served as a constant reference point throughout the development process, ensuring that the end product remained true to the planned design.

It also facilitated efficient communication about the layout and functionality between team members, and potentially, with stakeholders.



Technical Description of the Cocktail Finder Application

Overview

The Cocktail Finder is a Python-based desktop application that enables users to search for cocktail names using TheCocktailDB API. Developed with Tkinter, the application provides a graphical user interface (GUI) for easy interaction. Its main functionality allows users to enter a letter and retrieve a list of cocktail names starting with that letter.

API Integration

TheCocktailDB API: The application utilizes TheCocktailDB API to fetch cocktail names. This API was chosen for its comprehensive database of cocktails and straightforward endpoints.

Fetching Data: The APIClient class handles the API interaction. It uses Python's requests library to make GET requests to the API endpoint https://www.thecocktaildb.com/api/json/v1/1/search.php?f=. This endpoint is designed to return cocktails starting with a specific letter.

Response Handling: The application parses the JSON response from the API. The relevant data, primarily the names of the cocktails, is extracted for display.

GUI Design with Tkinter

Main Window Setup: The Tkinter library is used to create the main window of the application. The window is set to a user-friendly size and includes a title.

Layout Configuration: The application features a simple and intuitive layout, comprising a search area and a results display section.

Search Area: Includes an entry widget for users to input a letter and a button to trigger the search.

Results Display: A listbox widget displays the list of cocktail names fetched from the API. It is designed to be scrollable to accommodate a large number of results.

User Interaction Flow

Search Functionality: When a user enters a letter and clicks the 'Search' button, the application triggers a function connected to the button. This function retrieves the input, validates it (ensuring it’s a single letter), and calls the API client to fetch the corresponding data.

Displaying Results: Once the data is fetched, the application populates the listbox with the cocktail names. If no results are found or an error occurs, appropriate feedback is displayed in the listbox.

Error Handling and Validation

Input Validation: The application checks if the user's input is a single alphabetic character. If not, an error message is displayed.

API Error Handling: In case of a failed API request, the application handles this gracefully by displaying a message indicating that the data could not be fetched.

Code Structure

The code is structured into classes and functions for modularity and ease of maintenance. The APIClient class encapsulates API interactions, while the Application class, inheriting from Tkinter's Frame, handles the GUI components and user interactions.

Testing

Test Plan

The test plan for the Cocktail Finder application includes several test cases designed to validate its core functionalities and user interface. Here’s an overview:

Test Case: Valid Input

Objective: To verify the application responds correctly to valid input.

Procedure: Enter a single alphabet letter in the search field and click the search button.

Expected Result: The application displays a list of cocktail names starting with the entered letter.

Actual Result: [To be filled after testing]

Test Case: Invalid Input

Objective: To ensure the application handles invalid inputs gracefully.

Procedure: Enter invalid data (e.g., numbers, multiple letters) in the search field and click the search button.

Expected Result: The application shows an error message or prompt to enter a valid letter.

Actual Result: [To be filled after testing]

Test Case: API Failure

Objective: To test the application's behavior when the API call fails or is unavailable.

Procedure: Simulate an API failure (e.g., by disconnecting from the internet) and attempt a search.

Expected Result: The application displays a message indicating the data could not be fetched.

Actual Result: [To be filled after testing]

Evidence of Testing

Screenshots or videos of each test case should be included here, demonstrating the application's response to each scenario.

Critical Reflection

The Cocktail Finder application successfully integrates a third-party API to fetch data and presents it in a user-friendly GUI. Key strengths of the application include its straightforward interface, ease of use, and the efficient retrieval of data. Users can quickly find cocktail names based on the initial letter, which is a valuable feature for enthusiasts and professionals.

However, there are areas for improvement. For instance, expanding the application's functionality to include detailed recipes and ingredients for each cocktail would greatly enhance its value. Additionally, implementing more robust error handling and user input validation can further improve the user experience.

To improve this application, further exploration of TheCocktailDB API to fetch detailed cocktail information is necessary. Enhancing my skills in advanced GUI development with Tkinter will also be beneficial for future iterations. This project has been an excellent learning experience in API integration and Python GUI development, highlighting the importance of user-centric design and robust testing.

Appendix: Source Code

import tkinter as tk

import requests

from tkinter import font as tkFont

class APIClient:

    # APIClient class handles API requests to TheCocktailDB.

    def \_\_init\_\_(self):

        # Base URL for API requests.

        self.base\_url = "https://www.thecocktaildb.com/api/json/v1/1/search.php?f="

    def fetch\_cocktails(self, letter):

        # Fetches cocktails starting with the given letter.

        response = requests.get(self.base\_url + letter)

        if response.status\_code == 200:

            return response.json()

        else:

            return None

class Application(tk.Frame):

    # Main Application class for the GUI.

    def \_\_init\_\_(self, master=None):

        super().\_\_init\_\_(master, bg='light gray')

        self.master = master

        self.api\_client = APIClient()

        self.create\_widgets()

        self.grid(sticky="nsew")  # Allow the widget to expand in all directions (North, South, East, West).

        self.create\_grid\_config()  # Configure grid for responsive layout.

    def create\_widgets(self):

        # Setting up custom font for widgets.

        self.customFont = tkFont.Font(family="Helvetica", size=12)

        # Frame for Search Section.

        self.search\_frame = tk.Frame(self, bg='light blue', bd=2, relief='groove')

        self.search\_label = tk.Label(self.search\_frame, text="Enter a letter:", bg='light blue', font=self.customFont)

        self.search\_entry = tk.Entry(self.search\_frame, font=self.customFont)

        self.search\_button = tk.Button(self.search\_frame, text="Search", command=self.perform\_search, bg='green', fg='white', font=self.customFont)

        self.search\_label.grid(row=0, column=0, padx=5, pady=5)

        self.search\_entry.grid(row=0, column=1, padx=5, pady=5)

        self.search\_button.grid(row=0, column=2, padx=5, pady=5)

        self.search\_frame.grid(row=0, column=0, columnspan=2, sticky="ew", padx=10, pady=10)  # Ensure the frame expands horizontally.

        # Frame for Results Section.

        self.result\_frame = tk.Frame(self, bg='light gray')

        self.result\_listbox = tk.Listbox(self.result\_frame, width=40, height=15, font=self.customFont)

        self.result\_listbox.bind('<<ListboxSelect>>', self.on\_select)  # Bind selection event.

        self.result\_scrollbar = tk.Scrollbar(self.result\_frame, orient="vertical", command=self.result\_listbox.yview)

        self.result\_listbox.configure(yscrollcommand=self.result\_scrollbar.set)

        self.result\_listbox.grid(row=0, column=0, sticky="nsew")

        self.result\_scrollbar.grid(row=0, column=1, sticky="ns")

        self.result\_frame.grid(row=1, column=0, sticky="nsew", padx=10)

        # Text widget for Detail Section.

        self.detail\_text = tk.Text(self, height=10, width=30, font=self.customFont, bg='white', fg='black')

        self.detail\_text.grid(row=1, column=1, sticky="nsew", padx=10)

    def create\_grid\_config(self):

        # Configure the grid for responsive design.

        self.master.rowconfigure(0, weight=1)

        self.master.columnconfigure(0, weight=1)

        self.rowconfigure(1, weight=1)

        self.columnconfigure(1, weight=1)

        self.result\_frame.rowconfigure(0, weight=1)

        self.result\_frame.columnconfigure(0, weight=1)

    def perform\_search(self):

        # Function to handle search button click.

        letter = self.search\_entry.get().strip().lower()

        if len(letter) == 1 and letter.isalpha():

            data = self.api\_client.fetch\_cocktails(letter)

            self.populate\_results(data)

        else:

            self.result\_listbox.delete(0, tk.END)

            self.result\_listbox.insert(tk.END, "Please enter a single letter.")

    def populate\_results(self, data):

        # Populate the result listbox with data.

        self.result\_listbox.delete(0, tk.END)

        self.detail\_text.delete(1.0, tk.END)

        if data and "drinks" in data:

            for cocktail in data["drinks"]:

                self.result\_listbox.insert(tk.END, cocktail["strDrink"])

        else:

            self.result\_listbox.insert(tk.END, "No results found.")

    def on\_select(self, event):

        # Function to handle listbox selection event.

        widget = event.widget

        selection = widget.curselection()

        if selection:

            index = selection[0]

            data = widget.get(index)

            self.detail\_text.delete(1.0, tk.END)

            self.detail\_text.insert(tk.END, data)  # Placeholder for more detailed info.

root = tk.Tk()

root.title("Cocktail Finder")

root.geometry("800x500")

app = Application(master=root)

root.mainloop()