Github repository: <https://github.com/mrodriguez229/SDEV265GroupProject_FNM/tree/master>

**Task.py:**

import sqlite3

class Task:

    """

    Represents a task in the task management app with title, description, due date, priority, status, and category.

    """

    def \_\_init\_\_(self, title, description, due\_date, priority, status, category):

        """

        Initializes a task with the provided details.

        Parameters:

        title (str): Task title.

        description (str): Task description.

        due\_date (str): Task due date.

        priority (str): Task priority (e.g., High, Medium, Low).

        status (str): Task status (e.g., Pending, Completed).

        category (str): Task category.

        """

        self.title = title  # Title of the task

        self.description = description  # Task description

        self.due\_date = due\_date  # Task due date

        self.priority = priority  # Task priority

        self.status = status  # Task status

        self.category = category  # Task category

    def save(self, task\_id):

        """

        Saves or updates the task in the database.

        Parameters:

        task\_id (int): ID of the task to update.

        """

        with sqlite3.connect('tasks.db') as conn:

            cursor = conn.cursor()  # Create a cursor to execute SQL commands

            # Update task in the database

            cursor.execute('''

                UPDATE tasks

                SET title = ?, description = ?, due\_date = ?, priority = ?, status = ?

                WHERE id = ?

            ''', (self.title, self.description, self.due\_date, self.priority, self.status, task\_id))

            conn.commit()  # Save changes to the database

    def delete(self, task\_id):

        """

        Deletes a task from the database.

        Parameters:

        task\_id (int): ID of the task to delete.

        """

        with sqlite3.connect('tasks.db') as conn:

            cursor = conn.cursor()  # Create a cursor to execute SQL commands

            # Delete task from the database

            cursor.execute('DELETE FROM tasks WHERE id = ?', (task\_id,))

            conn.commit()  # Save changes to the database

**Tasks.py**

from database import create\_connection

def add\_task(task):

    """

    Adds a new task to the database.

    Parameters:

    task (Task): An instance of the Task class with task details.

    """

    conn = create\_connection()  # Connect to the database

    cursor = conn.cursor()  # Create a cursor to execute SQL commands

    # Insert the new task into the tasks table

    cursor.execute('''

        INSERT INTO tasks (title, description, due\_date, priority, status, category)

        VALUES (?, ?, ?, ?, ?, ?)

    ''', (task.title, task.description, task.due\_date, task.priority, task.status, task.category))

    conn.commit()  # Save the new task in the database

    conn.close()  # Close the connection

def get\_tasks():

    """

    Retrieves all tasks from the database, ordered by due date.

    Returns:

    list: A list of all tasks.

    """

    conn = create\_connection()  # Connect to the database

    cursor = conn.cursor()  # Create a cursor to execute SQL commands

    # Select all tasks ordered by due date

    cursor.execute('SELECT \* FROM tasks ORDER BY due\_date')

    tasks = cursor.fetchall()  # Fetch all tasks

    conn.close()  # Close the connection

    return tasks  # Return the task list

def delete\_task(task\_id):

    """

    Deletes a task from the database by task ID.

    Parameters:

    task\_id (int): The unique identifier of the task.

    """

    conn = create\_connection()  # Connect to the database

    cursor = conn.cursor()  # Create a cursor to execute SQL commands

    # Delete the task with the given task ID

    cursor.execute('DELETE FROM tasks WHERE id = ?', (task\_id,))

    conn.commit()  # Save the deletion

    conn.close()  # Close the connection

def update\_task(task, task\_id):

    """

    Updates a task in the database.

    Parameters:

    task (Task): An instance of the Task class with updated details.

    task\_id (int): The unique identifier of the task to update.

    """

    conn = create\_connection()  # Connect to the database

    cursor = conn.cursor()  # Create a cursor to execute SQL commands

    # Update the task details in the database

    cursor.execute('''

        UPDATE tasks

        SET title = ?, description = ?, due\_date = ?, priority = ?, status = ?, category = ?

        WHERE id = ?

    ''', (task.title, task.description, task.due\_date, task.priority, task.status, task.category, task\_id))

    conn.commit()  # Save the updated task

    conn.close()  # Close the connection

def get\_categories():

    """

    Retrieves all distinct task categories from the database.

    Returns:

    list: A list of unique categories.

    """

    conn = create\_connection()  # Connect to the database

    cursor = conn.cursor()  # Create a cursor to execute SQL commands

    # Select distinct categories

    cursor.execute('SELECT DISTINCT category FROM tasks')

    categories = cursor.fetchall()  # Fetch all unique categories

    conn.close()  # Close the connection

    return categories  # Return the category list

def get\_tasks\_by\_status(status):

    """

    Retrieves tasks by their status.

    Parameters:

    status (str): The task status (e.g., 'Completed', 'Pending').

    Returns:

    list: A list of tasks with the given status.

    """

    conn = create\_connection()  # Connect to the database

    cursor = conn.cursor()  # Create a cursor to execute SQL commands

    # Select tasks with the given status

    cursor.execute("SELECT \* FROM tasks WHERE status = ?", (status,))

    tasks = cursor.fetchall()  # Fetch tasks with the matching status

    conn.close()  # Close the connection

    return tasks  # Return the task list

**Database.py**

import sqlite3

from tkinter import messagebox

def create\_connection():

    """

    Establishes a connection to the SQLite database.

    Returns:

    sqlite3.Connection: Connection object to interact with the database.

    """

    conn = None

    try:

        conn = sqlite3.connect('tasks.db')  # Connect to the tasks database

        conn.execute("PRAGMA foreign\_keys = ON")  # Enable foreign keys

    except sqlite3.Error as e:

        print(f"Connection Error: {e}")  # Print connection error if it occurs

    return conn

def create\_tables():

    """

    Creates the necessary tables in the database if they don't exist.

    """

    conn = create\_connection()  # Connect to the database

    cursor = conn.cursor()  # Create a cursor to execute SQL commands

    # Create the tasks table if it doesn't exist

    cursor.execute('''

        CREATE TABLE IF NOT EXISTS tasks (

            id INTEGER PRIMARY KEY AUTOINCREMENT,

            title TEXT NOT NULL,

            description TEXT,

            due\_date TEXT NOT NULL,

            priority TEXT,

            status TEXT,

            category TEXT

        )

    ''')

    conn.commit()  # Save changes

    conn.close()  # Close the connection

def add\_task(title, description, due\_date, priority, status, category):

    """

    Adds a new task to the database.

    Parameters:

    title (str): Task title.

    description (str): Task description.

    due\_date (str): Task due date.

    priority (str): Task priority.

    status (str): Task status.

    category (str): Task category.

    """

    conn = create\_connection()  # Connect to the database

    cursor = conn.cursor()  # Create a cursor to execute SQL commands

    try:

        # Insert the new task into the tasks table

        cursor.execute('''

            INSERT INTO tasks (title, description, due\_date, priority, status, category)

            VALUES (?, ?, ?, ?, ?, ?)

        ''', (title, description, due\_date, priority, status, category))

        conn.commit()  # Save the new task

    except sqlite3.Error as e:

        messagebox.showerror("Database Error", f"Failed to add the task. Error: {e}")

    finally:

        conn.close()  # Close the connection

def update\_task(task\_id, title, description, due\_date, priority, status, category):

    """

    Updates an existing task in the database.

    Parameters:

    task\_id (int): ID of the task to update.

    title (str): Updated task title.

    description (str): Updated task description.

    due\_date (str): Updated due date.

    priority (str): Updated task priority.

    status (str): Updated task status.

    category (str): Updated task category.

    """

    conn = create\_connection()  # Connect to the database

    cursor = conn.cursor()  # Create a cursor to execute SQL commands

    try:

        # Update the task in the tasks table

        cursor.execute('''

            UPDATE tasks

            SET title = ?, description = ?, due\_date = ?, priority = ?, status = ?, category = ?

            WHERE id = ?

        ''', (title, description, due\_date, priority, status, category, task\_id))

        conn.commit()  # Save the updated task

    except sqlite3.Error as e:

        messagebox.showerror("Database Error", f"Failed to update the task. Error: {e}")

    finally:

        conn.close()  # Close the connection

def delete\_task(task\_id):

    """

    Deletes a task by its ID.

    Parameters:

    task\_id (int): ID of the task to delete.

    """

    conn = create\_connection()  # Connect to the database

    cursor = conn.cursor()  # Create a cursor to execute SQL commands

    try:

        # Delete the task with the given ID

        cursor.execute('DELETE FROM tasks WHERE id = ?', (task\_id,))

        conn.commit()  # Save the deletion

    except sqlite3.Error as e:

        messagebox.showerror("Database Error", f"Failed to delete the task. Error: {e}")

    finally:

        conn.close()  # Close the connection

def get\_tasks():

    """

    Retrieves all tasks from the database.

    Returns:

    list: A list of tasks.

    """

    conn = create\_connection()  # Connect to the database

    cursor = conn.cursor()  # Create a cursor to execute SQL commands

    tasks = []

    try:

        # Fetch all tasks from the tasks table

        cursor.execute('SELECT \* FROM tasks')

        tasks = cursor.fetchall()  # Retrieve all tasks

    except sqlite3.Error as e:

        messagebox.showerror("Database Error", f"Failed to fetch tasks. Error: {e}")

    finally:

        conn.close()  # Close the connection

    return tasks  # Return the list of tasks

# Ensure the tasks table is created when the module is loaded

create\_tables()

**Setting.py**

import tkinter as tk

class TaskManagerApp(tk.Tk):

    """

    Main application class for the Task Manager app, managing windows and themes.

    """

    def \_\_init\_\_(self):

        """

        Initializes the main window, sets up the pages, and applies the default theme.

        """

        super().\_\_init\_\_()  # Initialize the parent class (Tk)

        self.title("Task Manager")  # Set window title

        self.geometry("1024x768")  # Set window size

        container = tk.Frame(self)  # Create a container frame for the pages

        container.pack(side="top", fill="both", expand=True)  # Configure layout

        self.frames = {}  # Store page frames in a dictionary

        for F in (DashboardPage, MyTasksPage, SettingsPage):  # Loop through pages

            page\_name = F.\_\_name\_\_  # Get page name

            frame = F(parent=container, controller=self)  # Create page instance

            self.frames[page\_name] = frame  # Add to frames dictionary

            frame.grid(row=0, column=0, sticky="nsew")  # Position page in grid

        self.theme = "light"  # Default theme

        self.themes = {  # Theme options for light and dark modes

            "light": {"bg": "#d3d3d3", "fg": "#000000"},  # Light theme

            "dark": {"bg": "#333333", "fg": "#FFFFFF"}  # Dark theme

        }

        self.show\_frame("DashboardPage")  # Show the dashboard page by default

    def show\_frame(self, page\_name):

        """

        Displays the specified page.

        Parameters:

        page\_name (str): The name of the page to show.

        """

        frame = self.frames[page\_name]  # Get the requested page frame

        frame.tkraise()  # Bring the page to the front

        frame.update\_theme(self.theme)  # Apply the current theme to the page

    def apply\_theme(self, theme):

        """

        Applies the selected theme to all pages.

        Parameters:

        theme (str): The name of the theme to apply (e.g., 'light', 'dark').

        """

        self.theme = theme  # Set the new theme

        for frame in self.frames.values():  # Loop through all pages

            frame.update\_theme(theme)  # Update the theme for each page

class DashboardPage(tk.Frame):

    """

    Dashboard page class displaying the main dashboard.

    """

    def \_\_init\_\_(self, parent, controller):

        """

        Initializes the dashboard page.

        Parameters:

        parent (tk.Widget): The parent container.

        controller (TaskManagerApp): The controller to manage navigation and themes.

        """

        tk.Frame.\_\_init\_\_(self, parent)  # Initialize the parent class

        self.controller = controller  # Reference to the controller

        label = tk.Label(self, text="Dashboard", font=("Helvetica", 24))  # Create a dashboard label

        label.pack(pady=10)  # Add padding to the label

        # Button to navigate to My Tasks page

        button = tk.Button(self, text="Go to My Tasks", command=lambda: controller.show\_frame("MyTasksPage"))

        button.pack()

    def update\_theme(self, theme):

        """

        Updates the dashboard theme.

        Parameters:

        theme (str): The name of the theme to apply.

        """

        self.config(bg=self.controller.themes[theme]["bg"])  # Update background color

class MyTasksPage(tk.Frame):

    """

    My Tasks page class displaying the task list.

    """

    def \_\_init\_\_(self, parent, controller):

        """

        Initializes the My Tasks page.

        Parameters:

        parent (tk.Widget): The parent container.

        controller (TaskManagerApp): The controller to manage navigation and themes.

        """

        tk.Frame.\_\_init\_\_(self, parent)  # Initialize the parent class

        self.controller = controller  # Reference to the controller

        label = tk.Label(self, text="My Tasks", font=("Helvetica", 24))  # Create a label for My Tasks

        label.pack(pady=10)  # Add padding to the label

    def update\_theme(self, theme):

        """

        Updates the theme of the My Tasks page.

        Parameters:

        theme (str): The name of the theme to apply.

        """

        self.config(bg=self.controller.themes[theme]["bg"])  # Update background color

class SettingsPage(tk.Frame):

    """

    Settings page class for adjusting application settings, such as themes.

    """

    def \_\_init\_\_(self, parent, controller):

        """

        Initializes the settings page with options for theme changes.

        Parameters:

        parent (tk.Widget): The parent container.

        controller (TaskManagerApp): The controller to manage navigation and themes.

        """

        tk.Frame.\_\_init\_\_(self, parent)  # Initialize the parent class

        self.controller = controller  # Reference to the controller

        label = tk.Label(self, text="Settings", font=("Helvetica", 24))  # Create a settings label

        label.pack(pady=10)  # Add padding to the label

        # Radio buttons for selecting themes

        self.theme\_var = tk.StringVar(value="light")  # Variable to store the selected theme

        # Light theme radio button

        light\_rb = tk.Radiobutton(self, text="Light Theme", variable=self.theme\_var, value="light", command=self.set\_theme)

        light\_rb.pack(pady=5)

        # Dark theme radio button

        dark\_rb = tk.Radiobutton(self, text="Dark Theme", variable=self.theme\_var, value="dark", command=self.set\_theme)

        dark\_rb.pack(pady=5)

    def set\_theme(self):

        """

        Applies the selected theme.

        """

        self.controller.apply\_theme(self.theme\_var.get())  # Apply the selected theme

    def update\_theme(self, theme):

        """

        Updates the theme of the settings page.

        Parameters:

        theme (str): The name of the theme to apply.

        """

        self.config(bg=self.controller.themes[theme]["bg"])  # Update background color

if \_\_name\_\_ == "\_\_main\_\_":

    app = TaskManagerApp()  # Create the main app instance

    app.mainloop()  # Start the Tkinter main loop

**Ui.py**

import tkinter as tk

from tkinter import ttk, messagebox

from tkcalendar import Calendar, DateEntry

from tasks import add\_task, get\_tasks, delete\_task, update\_task

from task import Task

import datetime

import hashlib

from PIL import Image, ImageTk

class TaskInputFrame(tk.Frame):

    """

    A frame used for inputting task details such as title, description, due date, priority, status, and category.

    """

    def \_\_init\_\_(self, parent, controller, theme, style, update\_task\_display\_callback, \*args, \*\*kwargs):

        """

        Initializes the TaskInputFrame with input fields for task details.

        Parameters:

        parent (tk.Widget): The parent widget.

        controller (tk.Widget): The main application controller.

        theme (dict): The current theme being applied.

        style (ttk.Style): The styling options for the widgets.

        update\_task\_display\_callback (callable): Callback function to update the task display.

        """

        super().\_\_init\_\_(parent, \*args, \*\*kwargs)

        self.controller = controller

        self.theme = theme

        self.style = style

        self.update\_task\_display\_callback = update\_task\_display\_callback

        self.configure(bg=theme["bg"])  # Set background color based on the theme

        # Task Title Input

        task\_title\_frame = tk.Frame(self, bg=theme["bg"])

        task\_title\_frame.pack(fill="x", padx=20, pady=5)

        self.task\_title\_label = tk.Label(task\_title\_frame, text="Task Title", font=("Arial", 12), width=15, anchor="w", bg=theme["bg"], fg=theme["fg"])

        self.task\_title\_label.pack(side="left")

        self.title\_entry = tk.Entry(task\_title\_frame, width=35, bg="white", fg="black")

        self.title\_entry.pack(side="left", padx=10)

        # Task Description Input

        task\_desc\_frame = tk.Frame(self, bg=theme["bg"])

        task\_desc\_frame.pack(fill="x", padx=20, pady=5)

        self.task\_desc\_label = tk.Label(task\_desc\_frame, text="Task Description", font=("Arial", 12), width=15, anchor="w", bg=theme["bg"], fg=theme["fg"])

        self.task\_desc\_label.pack(side="left")

        self.desc\_text = tk.Text(task\_desc\_frame, height=5, width=35, bg="white", fg="black")

        self.desc\_text.pack(side="left", padx=10)

        # Task Category Input

        task\_category\_frame = tk.Frame(self, bg=theme["bg"])

        task\_category\_frame.pack(fill="x", padx=20, pady=5)

        self.task\_category\_label = tk.Label(task\_category\_frame, text="Category", font=("Arial", 12), width=15, anchor="w", bg=theme["bg"], fg=theme["fg"])

        self.task\_category\_label.pack(side="left")

        self.category\_combobox = ttk.Combobox(task\_category\_frame, values=["Work", "School", "Personal", "Home", "Other"], state="readonly", width=33, style="TCombobox")

        self.category\_combobox.set("Select Category")

        self.category\_combobox.pack(side="left", padx=10)

        # Task Due Date Input

        task\_due\_date\_frame = tk.Frame(self, bg=theme["bg"])

        task\_due\_date\_frame.pack(fill="x", padx=20, pady=5)

        self.task\_due\_date\_label = tk.Label(task\_due\_date\_frame, text="Due Date", font=("Arial", 12), width=15, anchor="w", bg=theme["bg"], fg=theme["fg"])

        self.task\_due\_date\_label.pack(side="left")

        self.due\_date\_entry = DateEntry(task\_due\_date\_frame, width=20, background=theme["bg"], foreground=theme["fg"])

        self.due\_date\_entry.pack(side="left", padx=10)

        # Task Priority Input

        task\_priority\_frame = tk.Frame(self, bg=theme["bg"])

        task\_priority\_frame.pack(fill="x", padx=20, pady=5)

        self.task\_priority\_label = tk.Label(task\_priority\_frame, text="Priority", font=("Arial", 12), width=15, anchor="w", background=theme["bg"], foreground=theme["fg"])

        self.task\_priority\_label.pack(side="left")

        self.priority\_var = tk.StringVar(value="Medium")

        self.priority\_options = ttk.Combobox(task\_priority\_frame, textvariable=self.priority\_var, values=["Low", "Medium", "High"], state="readonly", width=33, style="TCombobox")

        self.priority\_options.pack(side="left", padx=10)

        # Task Status Input

        task\_status\_frame = tk.Frame(self, bg=theme["bg"])

        task\_status\_frame.pack(fill="x", padx=20, pady=5)

        self.task\_status\_label = tk.Label(task\_status\_frame, text="Status", font=("Arial", 12), width=15, anchor="w", background=theme["bg"], foreground=theme["fg"])

        self.task\_status\_label.pack(side="left")

        self.status\_var = tk.StringVar(value="Pending")

        self.status\_options = ttk.Combobox(task\_status\_frame, textvariable=self.status\_var, values=["Pending", "In Progress", "Completed"], state="readonly", width=33, style="TCombobox")

        self.status\_options.pack(side="left", padx=10)

        # Create Task Button

        self.create\_task\_button = tk.Button(self, text="Create Task", font=("Arial", 12), width=20, command=self.create\_task, bg=theme["bg"], fg=theme["fg"], relief="raised", bd=2)

        self.create\_task\_button.pack(pady=20)

    def update\_theme(self, theme):

        """

        Updates the current theme for the frame and its widgets.

        Parameters:

        theme (dict): The theme to apply to the widgets.

        """

        self.controller.apply\_theme\_to\_widgets(theme, self.task\_title\_label, self.task\_desc\_label, self.task\_category\_label, self.task\_due\_date\_label, self.task\_priority\_label, self.task\_status\_label, self.create\_task\_button)

    def is\_valid\_date(self, date\_str):

        """

        Validates if the date string is in the correct format (MM/DD/YY).

        Returns True if valid, False otherwise.

        """

        try:

            # Ensure the date follows the correct format

            datetime.datetime.strptime(date\_str, '%m/%d/%y')

            return True

        except ValueError:

            return False

    def create\_task(self):

        """

        Gathers task input data and attempts to create a new task in the system.

        If any validation fails, an error message is shown.

        """

        task\_title = self.title\_entry.get().strip()  # Task title input

        task\_desc = self.desc\_text.get("1.0", tk.END).strip()  # Task description input

        task\_category = self.category\_combobox.get().strip()  # Task category input

        task\_due\_date = self.due\_date\_entry.get().strip()  # Task due date input

        task\_priority = self.priority\_var.get().strip()  # Task priority input

        task\_status = self.status\_var.get().strip()  # Task status input

        # Input validation

        if not task\_title:

            messagebox.showerror("Input Error", "Task title cannot be empty!")

            return

        if not task\_due\_date:

            messagebox.showerror("Input Error", "Due date cannot be empty!")

            return

        if not self.is\_valid\_date(task\_due\_date):

            messagebox.showerror("Input Error", "Invalid date format. Please use MM/DD/YY. Example: 12/31/24.")

            return

        if not task\_category:

            messagebox.showerror("Input Error", "Please select a category.")

            return

        if not task\_priority:

            messagebox.showerror("Input Error", "Please select a priority level.")

            return

        if not task\_status:

            messagebox.showerror("Input Error", "Please select a status.")

            return

        try:

            # Create new task object and save it

            new\_task = Task(task\_title, task\_desc, task\_due\_date, task\_priority, task\_status, task\_category)

            add\_task(new\_task)

            self.clear\_inputs()  # Clear input fields after creating task

            self.update\_task\_display\_callback()  # Update task list display

        except Exception as e:

            messagebox.showerror("Error", f"Failed to create task: {e}")  # Show error message if task creation fails

    def clear\_inputs(self):

        """Clears all input fields after task creation."""

        self.title\_entry.delete(0, tk.END)

        self.desc\_text.delete("1.0", tk.END)

        self.category\_combobox.set("Select Category")

        self.due\_date\_entry.set\_date(datetime.datetime.now())  # Reset to current date

        self.priority\_var.set("Medium")

        self.status\_var.set("Pending")

class DashboardFrame(tk.Frame):

    """

    A frame used for displaying the list of tasks and handling task-related actions on the dashboard.

    """

    def \_\_init\_\_(self, parent, theme, style, \*args, \*\*kwargs):

        """

        Initializes the DashboardFrame with a list of tasks and a scrollbar for navigation.

        Parameters:

        parent (tk.Widget): The parent widget.

        theme (dict): The current theme being applied.

        style (ttk.Style): The styling options for the widgets.

        """

        super().\_\_init\_\_(parent, \*args, \*\*kwargs)

        self.theme = theme  # Current theme applied

        self.style = style  # Styling options for ttk widgets

        self.configure(bg=theme["bg"])  # Set the background color based on the theme

        # Frame for task list display

        self.my\_tasks\_frame = tk.Frame(self, bg=theme["bg"])

        self.my\_tasks\_frame.pack(fill="both", expand=True, padx=10, pady=10)

        # Scrollbar for the task list

        self.my\_tasks\_scrollbar = tk.Scrollbar(self.my\_tasks\_frame)

        self.my\_tasks\_scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

        # Listbox for displaying tasks

        self.my\_tasks\_listbox = tk.Listbox(self.my\_tasks\_frame, yscrollcommand=self.my\_tasks\_scrollbar.set, width=50, height=10)

        self.my\_tasks\_listbox.pack(fill="both", expand=True)

        self.my\_tasks\_scrollbar.config(command=self.my\_tasks\_listbox.yview)

        self.update\_my\_tasks()  # Populate the task list

    def update\_my\_tasks(self):

        """

        Updates the list of tasks displayed in the listbox by fetching tasks from the database.

        """

        self.my\_tasks\_listbox.delete(0, tk.END)  # Clear the current listbox

        tasks = get\_tasks()  # Fetch all tasks

        self.displayed\_tasks = sorted(tasks, key=lambda task: datetime.datetime.strptime(task[3], '%m/%d/%y'))  # Sort tasks by due date

        # Insert each task into the listbox

        for task in self.displayed\_tasks:

            task\_info = f"{task[1]} - Due: {task[3]} - Priority: {task[4]}"

            self.my\_tasks\_listbox.insert(tk.END, task\_info)

        self.my\_tasks\_listbox.bind("<Double-1>", self.open\_task\_window)  # Bind double-click to open task window

    def open\_task\_window(self, event):

        """

        Opens a new window to edit or delete the selected task.

        Parameters:

        event (tk.Event): The event object when a task is selected from the listbox.

        """

        selected\_index = self.my\_tasks\_listbox.curselection()  # Get selected task index

        if selected\_index:

            selected\_index = selected\_index[0]

            selected\_task = self.displayed\_tasks[selected\_index]

            self.show\_edit\_task\_window(selected\_task)  # Open the edit task window

    def show\_edit\_task\_window(self, task):

        """

        Displays a window that allows the user to edit or delete the selected task.

        """

        edit\_window = tk.Toplevel(self)  # Create a new window for editing the task

        edit\_window.title("Edit Task")

        window\_width, window\_height = 700, 700

        screen\_width = self.winfo\_screenwidth()

        screen\_height = self.winfo\_screenheight()

        x = int((screen\_width / 2) - (window\_width / 2))  # Center the window on the screen

        y = int((screen\_height / 2) - (window\_height / 2))

        edit\_window.geometry(f"{window\_width}x{window\_height}+{x}+{y}")

        # Apply theme to the window

        edit\_window.configure(bg=self.theme["bg"])

        # Task Title

        title\_label = tk.Label(edit\_window, text="Task Title", bg=self.theme["bg"], fg=self.theme["fg"])

        title\_label.pack(pady=5)

        title\_entry = tk.Entry(edit\_window, bg=self.theme["entry\_bg"], fg=self.theme["entry\_fg"])

        title\_entry.pack(pady=5)

        title\_entry.insert(0, task[1])

        # Task Description

        desc\_label = tk.Label(edit\_window, text="Task Description", bg=self.theme["bg"], fg=self.theme["fg"])

        desc\_label.pack(pady=5)

        desc\_text = tk.Text(edit\_window, height=5, bg=self.theme["entry\_bg"], fg=self.theme["entry\_fg"])

        desc\_text.pack(pady=5)

        desc\_text.insert(tk.END, task[2])

        # Due Date

        due\_date\_label = tk.Label(edit\_window, text="Due Date", bg=self.theme["bg"], fg=self.theme["fg"])

        due\_date\_label.pack(pady=5)

        due\_date\_entry = DateEntry(edit\_window, background=self.theme["bg"], foreground=self.theme["fg"])

        due\_date\_entry.pack(pady=5)

        due\_date\_entry.set\_date(task[3])

        # Priority

        priority\_label = tk.Label(edit\_window, text="Priority", bg=self.theme["bg"], fg=self.theme["fg"])

        priority\_label.pack(pady=5)

        priority\_var = tk.StringVar(value=task[4])

        priority\_options = ttk.Combobox(edit\_window, textvariable=priority\_var, values=["Low", "Medium", "High"], state="readonly")

        priority\_options.pack(pady=5)

        # Status

        status\_label = tk.Label(edit\_window, text="Status", bg=self.theme["bg"], fg=self.theme["fg"])

        status\_label.pack(pady=5)

        status\_var = tk.StringVar(value=task[5])

        status\_options = ttk.Combobox(edit\_window, textvariable=status\_var, values=["Pending", "In Progress", "Completed"], state="readonly")

        status\_options.pack(pady=5)

        # Category

        category\_label = tk.Label(edit\_window, text="Category", bg=self.theme["bg"], fg=self.theme["fg"])

        category\_label.pack(pady=5)

        category\_var = tk.StringVar(value=task[6])

        category\_combobox = ttk.Combobox(edit\_window, textvariable=category\_var, values=["Work", "School", "Personal", "Home", "Other"], state="readonly")

        category\_combobox.pack(pady=5)

        # Button to save changes to the task

        save\_button = tk.Button(edit\_window, text="Save Changes", bg=self.theme["bg"], fg=self.theme["fg"], command=lambda: self.save\_edited\_task(task[0], title\_entry.get(), desc\_text.get("1.0", tk.END), due\_date\_entry.get(), priority\_var.get(), status\_var.get(), category\_var.get(), edit\_window))

        save\_button.pack(pady=10)

        # Button to delete the task

        delete\_button = tk.Button(edit\_window, text="Delete Task", bg=self.theme["bg"], fg=self.theme["fg"], command=lambda: self.delete\_task\_and\_close(task[0], edit\_window))

        delete\_button.pack(pady=10)

        # Store references to the window and its widgets (including labels)

        self.edit\_window = edit\_window

        self.edit\_widgets = [title\_label, title\_entry, desc\_label, desc\_text, due\_date\_label, due\_date\_entry,

                         priority\_label, priority\_options, status\_label, status\_options, category\_label,

                         category\_combobox, save\_button, delete\_button]

    def save\_edited\_task(self, task\_id, title, description, due\_date, priority, status, category, edit\_window):

        """

        Saves the changes made to the task and updates the task list.

        Parameters:

        task\_id (int): The ID of the task to update.

        title (str): The updated task title.

        description (str): The updated task description.

        due\_date (str): The updated due date.

        priority (str): The updated priority level.

        status (str): The updated task status.

        category (str): The updated task category.

        edit\_window (tk.Toplevel): The window where the task is edited.

        """

        updated\_task = Task(title, description, due\_date, priority, status, category)  # Create an updated Task object

        update\_task(updated\_task, task\_id)  # Update the task in the database

        self.update\_my\_tasks()  # Refresh the task list

        edit\_window.destroy()  # Close the edit window

    def delete\_task\_and\_close(self, task\_id, window):

        """

        Deletes the selected task and closes the edit window.

        Parameters:

        task\_id (int): The ID of the task to delete.

        window (tk.Toplevel): The window where the task is being edited.

        """

        response = messagebox.askyesno("Delete Task", "Are you sure you want to delete this task?")  # Confirmation dialog

        if response:

            delete\_task(task\_id)  # Delete the task from the database

            window.destroy()  # Close the edit window

            self.update\_my\_tasks()  # Refresh the task list

class TaskManagerApp(tk.Tk):

    """

    The main application class for the Task Manager app. Manages navigation between pages, user login, and theme switching.

    """

    def \_\_init\_\_(self):

        """

        Initializes the main application window, frames (pages), sidebar, topbar, and applies the default theme.

        """

        super().\_\_init\_\_()

        self.title("Task Manager")  # Set the title of the window

        self.geometry("700x600")  # Set the window size

        self.theme = "light"  # Default theme for the application

        self.themes = {  # Theme settings for light and dark modes

            "light": {"bg": "#d3d3d3", "fg": "#000000", "entry\_bg": "#FFFFFF", "entry\_fg": "#000000"},

            "dark": {"bg": "#333333", "fg": "#FFFFFF", "entry\_bg": "#444444", "entry\_fg": "#FFFFFF"}

        }

        self.style = ttk.Style()  # Style for ttk widgets

        self.mode\_var = tk.StringVar(value="Light Mode")  # Variable to switch between light and dark modes

        self.frames = {}  # Dictionary to hold the frames (pages)

        self.logged\_in = False  # Flag to track if the user is logged in

        # Create various sections of the application

        self.create\_sidebar()  # Create the sidebar with navigation buttons

        self.create\_topbar()  # Create the top bar with login and add task buttons

        self.create\_main\_page()  # Create the main page with the calendar

        self.create\_my\_tasks\_page()  # Create the My Tasks page

        self.create\_settings\_page()  # Create the Settings page

        self.create\_add\_task\_page()  # Create the Add Task page

        self.apply\_theme()  # Apply the default theme

        self.show\_frame("MainPage")  # Show the main page by default

    def create\_sidebar(self):

        """

        Creates the sidebar with navigation buttons for different pages.

        """

        self.sidebar = tk.Frame(self, bg=self.themes[self.theme]["bg"], width=200, bd=2, relief="ridge")

        self.sidebar.pack(side=tk.LEFT, fill=tk.Y, padx=0, pady=0)

        # Add logo and buttons to the sidebar

        logo\_image = tk.PhotoImage(file="assets/logo.png").subsample(3, 3)

        self.logo\_label = tk.Label(self.sidebar, image=logo\_image, bg=self.themes[self.theme]["bg"], bd=0)

        self.logo\_label.image = logo\_image  # Store reference to the image

        self.logo\_label.pack(pady=(10, 20))

        # Define button style

        button\_style = {"width": 15, "height": 1, "bg": self.themes[self.theme]["bg"], "fg": self.themes[self.theme]["fg"], "relief": "raised", "bd": 2, "anchor": "center"}

        # Create sidebar buttons

        self.dashboard\_button = tk.Button(self.sidebar, text="Dashboard", \*\*button\_style, command=lambda: self.show\_frame("MainPage"))

        self.dashboard\_button.pack(padx=10, pady=10, fill=tk.X)

        self.tasks\_button = tk.Button(self.sidebar, text="My Tasks", \*\*button\_style, command=self.open\_tasks\_page)

        self.tasks\_button.pack(padx=10, pady=10, fill=tk.X)

        self.settings\_button = tk.Button(self.sidebar, text="Settings", \*\*button\_style, command=lambda: self.show\_frame("SettingsPage"))

        self.settings\_button.pack(padx=10, pady=10, fill=tk.X)

        # Log out button at the bottom of the sidebar

        self.spacer = tk.Frame(self.sidebar, bg=self.themes[self.theme]["bg"])

        self.spacer.pack(expand=True, fill=tk.Y, pady=0)

        self.logout\_button = tk.Button(self.sidebar, text="Log Out", \*\*button\_style, command=self.log\_out)

        self.logout\_button.pack(padx=10, pady=10, fill=tk.X)

    def create\_topbar(self):

        """

        Creates the top bar with login and add task buttons.

        """

        self.top\_frame = tk.Frame(self, bg=self.themes[self.theme]["bg"], height=50, bd=2, relief="ridge")

        self.top\_frame.pack(side=tk.TOP, fill=tk.X, padx=0, pady=0)

        # Create login button with an image

        button\_style = {"width": 15, "height": 1, "bg": self.themes[self.theme]["bg"], "fg": self.themes[self.theme]["fg"], "relief": "raised", "bd": 2, "anchor": "center"}

        login\_image\_path = "assets/login.png"

        login\_image = Image.open(login\_image\_path)

        login\_image = login\_image.resize((100, 90), Image.LANCZOS)

        self.login\_photo = ImageTk.PhotoImage(login\_image)

        self.login\_button = tk.Button(self.top\_frame, image=self.login\_photo, command=self.open\_login\_window, bg=self.themes[self.theme]["bg"], borderwidth=0, highlightthickness=0, relief="flat")

        self.login\_button.pack(side=tk.RIGHT, padx=10, pady=0)

        # Create add task button

        self.add\_task\_button = tk.Button(self.top\_frame, text="+ Add Task", \*\*button\_style, command=lambda: self.show\_frame("AddTaskPage"))

        self.add\_task\_button.pack(side=tk.RIGHT, padx=10, pady=10)

    def create\_main\_page(self):

        """

        Creates the main page which contains a calendar to display tasks.

        """

        if "MainPage" not in self.frames:

            self.main\_frame = tk.Frame(self, bg=self.themes[self.theme]["bg"], bd=2, relief="ridge")

            self.main\_frame.pack(expand=True, fill="both", padx=20, pady=20)

            calendar\_frame = tk.Frame(self.main\_frame, bg="#ffffff", bd=1, relief="solid")

            calendar\_frame.place(relx=0.05, rely=0.05, relwidth=0.9, relheight=0.8)

            today = datetime.datetime.now()  # Get today's date

            self.calendar = Calendar(calendar\_frame, selectmode='day', year=today.year, month=today.month, day=today.day)  # Create the calendar widget

            self.calendar.pack(expand=True, fill="both")

            self.frames["MainPage"] = self.main\_frame  # Store reference to the main page

        else:

            self.clear\_calendar\_tasks()  # Clear tasks if already initialized

        if self.logged\_in:

            self.update\_calendar\_tasks()  # Update the calendar with tasks if logged in

    def clear\_calendar\_tasks(self):

        """

        Clears all tasks displayed on the calendar.

        """

        for event\_id in self.calendar.get\_calevents():

            self.calendar.calevent\_remove(event\_id)  # Remove all calendar events

    def update\_calendar\_tasks(self):

        """

        Updates the calendar by fetching tasks from the database and marking their due dates.

        """

        tasks = get\_tasks()  # Fetch all tasks

        for task in tasks:

            due\_date\_str = task[3]

            due\_date = datetime.datetime.strptime(due\_date\_str, '%m/%d/%y')

            self.calendar.calevent\_create(due\_date, task[1], 'task\_due')  # Add task due date to the calendar

        self.calendar.tag\_config('task\_due', background='lightblue', foreground='black')  # Configure the calendar event tag

    def open\_login\_window(self):

        """

        Opens the login window for user authentication.

        """

        login\_window = tk.Toplevel(self)  # Create a new login window

        login\_window.title("Log In")

        window\_width, window\_height = 300, 200

        screen\_width = self.winfo\_screenwidth()

        screen\_height = self.winfo\_screenheight()

        x = int((screen\_width / 2) - (window\_width / 2))  # Center the window on the screen

        y = int((screen\_height / 2) - (window\_height / 2))

        login\_window.geometry(f"{window\_width}x{window\_height}+{x}+{y}")

        # Create input fields for username and password

        username\_label = tk.Label(login\_window, text="Username (max 15 chars):")

        username\_label.pack(pady=10)

        username\_entry = tk.Entry(login\_window)

        username\_entry.pack(pady=5)

        username\_entry.focus\_set()  # Focus on username input

        password\_label = tk.Label(login\_window, text="Password (8-12 chars):")

        password\_label.pack(pady=10)

        password\_entry = tk.Entry(login\_window, show="\*")  # Mask the password input

        password\_entry.pack(pady=5)

        # Create the login button

        login\_button = tk.Button(login\_window, text="Login", command=lambda: self.validate\_login(username\_entry.get(), password\_entry.get(), login\_window))

        login\_button.pack(pady=10)

    def validate\_login(self, username, password, login\_window):

        """

        Validates the user login by checking the credentials.

        Parameters:

        username (str): The entered username.

        password (str): The entered password.

        login\_window (tk.Toplevel): The login window to be closed after validation.

        """

        users = self.load\_users()  # Load the predefined users

        # Perform input validation

        if not username or not password:

            tk.messagebox.showerror("Input Error", "Both username and password are required!")

        elif len(username) > 15:

            tk.messagebox.showerror("Input Error", "Username cannot exceed 15 characters!")

        elif len(password) > 12 or len(password) < 8:

            tk.messagebox.showerror("Input Error", "Password must be between 8-12 characters!")

        elif not username.isalnum():

            tk.messagebox.showerror("Input Error", "Username can only contain letters and numbers!")

        else:

            hashed\_password = hashlib.sha256(password.encode()).hexdigest()  # Hash the entered password

            if username in users and users[username] == hashed\_password:  # Check if the username and password match

                tk.messagebox.showinfo("Login Successful", f"Welcome, {username}!")

                self.logged\_in = True  # Set the logged in flag to True

                self.tasks\_button.config(state="normal")  # Enable the tasks button

                self.create\_main\_page()  # Update the main page

                self.show\_frame("MainPage")  # Show the main page

            else:

                tk.messagebox.showerror("Login Failed", "Invalid username or password")

        # Clear the input fields

        for widget in login\_window.winfo\_children():

            if isinstance(widget, tk.Entry):

                widget.delete(0, tk.END)

        login\_window.destroy()  # Close the login window

    def load\_users(self):

        """

        Loads predefined users for authentication.

        Returns:

        dict: A dictionary of users with their hashed passwords.

        """

        return {

            'admin': hashlib.sha256('password123'.encode()).hexdigest(),  # Predefined user 'admin'

            }

    def log\_out(self):

        """

        Logs the user out and resets the app state.

        """

        response = messagebox.askyesno("Log Out", "Are you sure you want to log out?")  # Ask for confirmation

        if response:

            self.logged\_in = False  # Set the logged in flag to False

            self.clear\_calendar\_tasks()  # Clear tasks from the calendar

            self.show\_frame("MainPage")  # Show the main page

            self.login\_button.config(state="normal")  # Re-enable the login button

    def create\_my\_tasks\_page(self):

        """

        Creates the My Tasks page where the user can view and manage their tasks.

        """

        theme = self.themes[self.theme]  # Get the current theme

        self.my\_tasks\_frame = DashboardFrame(self, self.themes[self.theme], self.style)  # Create the DashboardFrame

        self.my\_tasks\_frame.config(bg=theme["bg"], bd=2, relief="ridge")

        self.frames["MyTasksPage"] = self.my\_tasks\_frame

        self.my\_tasks\_frame.pack(expand=True, fill="both", padx=20, pady=20)

    def open\_tasks\_page(self):

        """

        Opens the My Tasks page if the user is logged in.

        """

        if self.logged\_in:

            self.show\_frame("MyTasksPage")  # Show the My Tasks page

        else:

            tk.messagebox.showerror("Access Denied", "You must log in to access this page.")  # Show error if not logged in

    def create\_add\_task\_page(self):

        """

        Creates the Add Task page where users can input and create new tasks.

        """

        theme = self.themes[self.theme]  # Get the current theme

        self.add\_task\_frame = TaskInputFrame(self, self, self.themes[self.theme], self.style, self.update\_tasks\_in\_my\_tasks\_page)  # Create TaskInputFrame

        self.add\_task\_frame.config(bg=theme["bg"], bd=2, relief="ridge")

        self.frames["AddTaskPage"] = self.add\_task\_frame

        self.add\_task\_frame.pack(expand=True, fill="both", padx=20, pady=20)

    def create\_settings\_page(self):

        """

        Creates the Settings page where users can adjust preferences such as the theme.

        """

        self.settings\_frame = tk.Frame(self, bg=self.themes[self.theme]["bg"], bd=2, relief="ridge")  # Create settings frame

        self.mode\_frame = tk.Frame(self.settings\_frame, bg=self.themes[self.theme]["bg"])

        self.mode\_frame.pack(pady=10, padx=50, anchor="w")

        # Create label and combobox for theme selection

        self.mode\_label = tk.Label(self.mode\_frame, text="Select Mode:", font=("Arial", 12), bg=self.themes[self.theme]["bg"], fg=self.themes[self.theme]["fg"])

        self.mode\_label.pack(side="left", padx=(0, 10))

        self.mode\_button = ttk.Combobox(self.mode\_frame, textvariable=self.mode\_var, values=["Light Mode", "Dark Mode"], background="white", foreground="black")

        self.mode\_button.pack(side="left")

        self.mode\_button.bind("<<ComboboxSelected>>", lambda event: self.apply\_theme\_from\_combobox())  # Bind combobox selection event

        self.task\_prefs\_frame = tk.Frame(self.settings\_frame, bg=self.themes[self.theme]["bg"])

        self.task\_prefs\_frame.pack(pady=10, padx=50, anchor="w")

        self.frames["SettingsPage"] = self.settings\_frame  # Store reference to the settings page

    def show\_frame(self, frame\_name):

        """

        Displays the specified frame (page) in the application.

        Parameters:

        frame\_name (str): The name of the frame (page) to display.

        """

        if frame\_name in ["MyTasksPage", "AddTaskPage"] and not self.logged\_in:

            tk.messagebox.showerror("Access Denied", "You must log in to access this page.")  # Show error if trying to access without logging in

            return

        # Hide all frames and show the requested one

        for frame in self.frames.values():

            frame.pack\_forget()

        frame = self.frames.get(frame\_name)

        if frame:

            frame.pack(fill="both", expand=True)

            if frame\_name == "MainPage":

                today = datetime.datetime.now()

                self.calendar.selection\_set(today)  # Set calendar selection to today

        else:

            print(f"Frame '{frame\_name}' not found!")  # Print error if frame is not found

    def apply\_theme(self):

        self.theme = "light" if self.mode\_var.get() == "Light Mode" else "dark"

        current\_theme = self.themes[self.theme]

        # Apply theme settings to other sections

        self.sidebar.config(bg=current\_theme["bg"], bd=2, relief="ridge")

        self.logo\_label.config(bg=current\_theme["bg"])

        self.spacer.config(bg=current\_theme["bg"])

        self.main\_frame.config(bg=current\_theme["bg"], bd=2, relief="ridge")

        self.my\_tasks\_frame.config(bg=current\_theme["bg"])

        self.add\_task\_frame.config(bg=current\_theme["bg"])

        # Update task frame theme and apply to widgets

        self.add\_task\_frame.update\_theme(current\_theme)

        self.add\_task\_frame.update\_idletasks()

        self.settings\_frame.config(bg=current\_theme["bg"], bd=2, relief="ridge")

        self.top\_frame.config(bg=current\_theme["bg"], bd=2, relief="ridge")

        self.mode\_frame.config(bg=current\_theme["bg"])

        self.mode\_label.config(bg=current\_theme["bg"], fg=current\_theme["fg"])

        self.task\_prefs\_frame.config(bg=current\_theme["bg"])

        # Apply theme to the edit window, if it exists and is open

        if hasattr(self, 'edit\_window') and self.edit\_window.winfo\_exists():

            self.apply\_theme\_to\_edit\_window(self.edit\_window, self.edit\_widgets)

        # Apply theme to sidebar and topbar buttons

        for widget in self.sidebar.winfo\_children():

            if isinstance(widget, tk.Button):

                widget.config(bg=current\_theme["bg"], fg=current\_theme["fg"], relief="raised")

        for widget in self.top\_frame.winfo\_children():

            if isinstance(widget, tk.Button):

                widget.config(bg=current\_theme["bg"], fg=current\_theme["fg"], relief="raised")

        self.style.configure("TCombobox", fieldbackground="white", foreground="black")  # Set combobox styles

    def apply\_theme\_to\_edit\_window(self, edit\_window, widgets):

        """

        Applies the theme to the edit task window and its widgets.

        Parameters:

        edit\_window (tk.Toplevel): The edit task window.

        widgets (list): A list of widgets to apply the theme.

        """

        edit\_window.configure(bg=self.themes[self.theme]["bg"])

        for widget in widgets:

            widget.configure(bg=self.themes[self.theme]["bg"], fg=self.themes[self.theme]["fg"])

    def apply\_theme\_from\_combobox(self):

        """

        Applies the theme selected from the combobox on the settings page.

        """

        self.apply\_theme()  # Call the apply\_theme method to update the theme

    def apply\_theme\_to\_widgets(self, theme, \*widgets):

        """

        Applies the selected theme to the given widgets.

        Parameters:

        theme (dict): The theme to apply to the widgets.

        widgets (tk.Widget): The widgets to update with the theme.

        """

        for widget in widgets:

            widget.config(bg=theme["bg"], fg=theme["fg"])  # Set the background and foreground color for each widget

    def update\_tasks\_in\_my\_tasks\_page(self):

        """

        Updates the task list on the My Tasks page by calling the update method of the frame.

        """

        if "MyTasksPage" in self.frames:

            self.my\_tasks\_frame.update\_my\_tasks()  # Call the update method of the My Tasks page

if \_\_name\_\_ == "\_\_main\_\_":

    app = TaskManagerApp()

    app.mainloop()

**App.py:**

from ui import TaskManagerApp

# Entry point of the Task Management application.

# TaskManagerApp is the main class for the UI logic.

try:

    app = TaskManagerApp()

    app.mainloop()

except Exception as e:

    print(f"An error occurred while running the application: {e}")