Introduction

The **Weather App** is a Python-based graphical user interface (GUI) application designed to fetch and display current weather information for a specified city. Developed using the Tkinter library for the GUI, it allows users to input the name of a city and retrieve real-time weather data, including temperature, humidity, visibility, weather conditions, sunrise and sunset times, and more.

The app utilizes the **OpenWeatherMap API** to obtain weather data and the **TimezoneFinder** and **pytz** libraries to determine the local time of the searched location. It features intuitive design elements such as weather icons that change based on current conditions (e.g., clear, cloudy, rainy), and labels that dynamically update with relevant weather information.

This project demonstrates the integration of Python's standard and third-party libraries to create a real-time, interactive application. The threading mechanism ensures that the GUI remains responsive while the app fetches data from the API in the background, providing a smooth user experience.

By implementing this project, users can gain practical experience with:

- GUI development using Tkinter.
- API handling and JSON data processing.
- Multi-threading for non-blocking UI.
- Basic error handling and user input validation.

Functionalities

The **Weather App** offers several key functionalities that allow users to interact with the application and obtain real-time weather information. Below is a breakdown of the main features:

1. **City Search Functionality**:
    - Users can enter the name of a city in the search textbox and either click the search button or press the "Enter" key to retrieve weather information.
    - The app fetches real-time weather data using the OpenWeatherMap API and displays it on the interface.
    - Error handling ensures that appropriate messages are shown if the city is not found or if the user fails to enter a city name.
2. **Real-time Weather Information Display**:
    - The application provides detailed weather information, including:
        - **City Name**: Displays the name of the searched city.

        - **Local Time**: Shows the local time of the city based on its geographical coordinates.
        - **Temperature**: Displays the current temperature in degrees Celsius.
        - **Weather Description**: Provides a brief description of the current weather (e.g., clear, cloudy, rainy).
        - **Feels Like**: Shows the "feels like" temperature, indicating how the temperature actually feels based on humidity and other factors.
        - **Humidity**: Displays the current humidity percentage.
        - **Pressure**: Provides the atmospheric pressure in millibars (mBar).
        - **Visibility**: Displays visibility in kilometers.
3. **Sunrise and Sunset Times**:
    - The app calculates and displays the local sunrise and sunset times for the specified city using the city's geographical location and the time zone derived through the TimezoneFinder library.
4. **Dynamic Weather Icons**:
    - Based on the current weather condition (e.g., Clear, Clouds, Rain, Haze), the app updates a weather icon that visually represents the weather.
    - Different icons are loaded to correspond to different weather types for an intuitive and user-friendly interface.
5. **Reset Functionality**:
    - The app includes a "Reset" button that clears all displayed data, including the city name, weather information, and weather icon, allowing users to perform a new search.
6. **Exit Functionality**:
    - An "Exit" button is provided for easy termination of the application. When clicked, it prompts the user with a confirmation dialog box to confirm whether they want to exit the app.
7. **Threading for Non-blocking UI**:
    - To ensure that the application remains responsive while fetching weather data, a separate thread is used for the API requests. This prevents the interface from freezing during data retrieval, creating a smoother user experience.
8. **Error Handling**:
    - The app includes error handling for various scenarios:
        - If the user is not connected to the internet, a warning is displayed.
        - If an invalid city name is entered or the city is not found, an error message is shown.
        - Generic error handling ensures that the app can gracefully handle unforeseen issues, preventing crashes.

Technology Used

The **Weather App** leverages several technologies and libraries to provide its functionality and user interface. Below is a list of the primary technologies used in the project:

1. **Python**:
    - The core programming language used for the app's logic and functionality.
2. **Tkinter**:
    - The built-in Python library used to create the graphical user interface (GUI). Tkinter provides tools for designing windows, buttons, labels, and textboxes, enabling user interactions with the application.
3. **OpenWeatherMap API**:
    - An external API used to fetch real-time weather data, such as temperature, humidity, pressure, visibility, and weather conditions for the specified city.
4. **Requests Library**:
    - A popular Python library for making HTTP requests to the OpenWeatherMap API. It is used to retrieve weather data in JSON format, which is then processed by the app.
5. **Pillow (PIL)**:
    - A Python Imaging Library (PIL) used for handling and manipulating images. In this project, Pillow is used to load, resize, and display various icons (e.g., weather conditions, location icon) in the Tkinter GUI.
6. **TimezoneFinder**:
    - A Python library used to determine the timezone of a given city based on its geographical coordinates (longitude and latitude). This allows the app to display the local time for the searched city.
7. **pytz**:
    - A library used to handle time zones in Python. It works in conjunction with TimezoneFinder to display the correct local time for the searched city.
8. **Threading**:

- The Python threading module is used to ensure the app's GUI remains responsive while the app fetches weather data in the background. This prevents the UI from freezing when handling network requests.

9. **ConfigParser**:

   - A Python library used to manage configuration files. It allows the app to store and retrieve the OpenWeatherMap API key from an external config.ini file, keeping the API key secure and separate from the source code.

Requirements

Hardware	Software
Computer [High – performance computer {PC or Laptop]	Python version 3.x or higher.
Internet Connectivity	Operating System
RAM Minimum: 4 GB	Code Editor [VS code, PyCharm etc.]
	OpenWeatherMap API Key
	Python Libraries [Tkinter, Pillow, Requests, TimezoneFinder, pytz, ConfigParser]

Source Code and Output

```
import configparser
from tkinter import *
from tkinter import messagebox
from PIL import Image,ImageTk
import threading
import requests
import datetime
import pytz
from timezonefinder import TimezoneFinder

class Weather(Tk):
    def __init__(self):
        super().__init__()
        self.title("Weather App")
        self.geometry("800x480")
        self.iconbitmap(r"Images/weather_icon.ico")
        self.resizable(False,False)
        # self.__gui()
        threading.Thread(target=self.__gui).start()

    def __gui(self):
        # placing the black border for search
        self.img=Image.open(r"Images/black_border.png")
        self.resizeimg=self.img.resize((275,35))
        self.finalimg=ImageTk.PhotoImage(self.resizeimg)
        Label(image=self.finalimg).place(x=20,y=20)

        # creating the search button
        self.img1=Image.open(r"Images/search_btn.png")
        self.resizeim1=self.img1.resize((29,29))
        self.finalimg1=ImageTk.PhotoImage(self.resizeim1)
        self.b1=Button(image=self.finalimg1,bg="black",command=self.threading)
        self.b1.place(x=297,y=22)
        self.bind("<Return>",self.threading)

        # creating the search textbox
        self.search=StringVar()
        self.search_textbox=Entry(textvariable=self.search,font=("Segoe UI",14,'bold'),width=24,justify="center",relief="flat")
        self.search_textbox.place(x=25,y=25)

        # creating the current weather label to display the city name and city time
        Label(text="Current Weather :",font='Arial 14 bold',fg="red").place(x=590,y=7)

        # location image logo
        self.img2=Image.open(r'Images/location.png')
        self.resizeimg2=self.img2.resize((20,20))
        self.finalimg2=ImageTk.PhotoImage(self.resizeimg2)
        Label(image=self.finalimg2).place(x=595,y=36)

        # location label
        self.location=Label(text='',font='Calibri 15')
        self.location.place(x=620,y=34)

        # time label for the searched city
        self.timelbl=Label(text="",font=("Cambria",16))
        self.timelbl.place(x=590,y=60)

        # creating the label for the logo according to main
        self.img3=Image.open(r"Icons/main.png")
        self.resizeimg3=self.img3.resize((200,190))
        self.finalimg3=ImageTk.PhotoImage(self.resizeimg3)
        self.icons=Label(image=self.finalimg3)
        self.icons.place(x=70,y=110)

        # creating the label to display the temperature
        self.temperature=Label(text="",font=("Cambria",75,'bold'))
        self.temperature.place(x=270,y=140)
        self.degree=Label(text="",font="Cambria 40 bold")
        self.degree.place(x=390,y=135)

        # feels like label and sunny or fog like labels
        self.feel=Label(text="",font=("Nirmala UI",16,"bold"))
        self.feel.place(x=280,y=245)
```

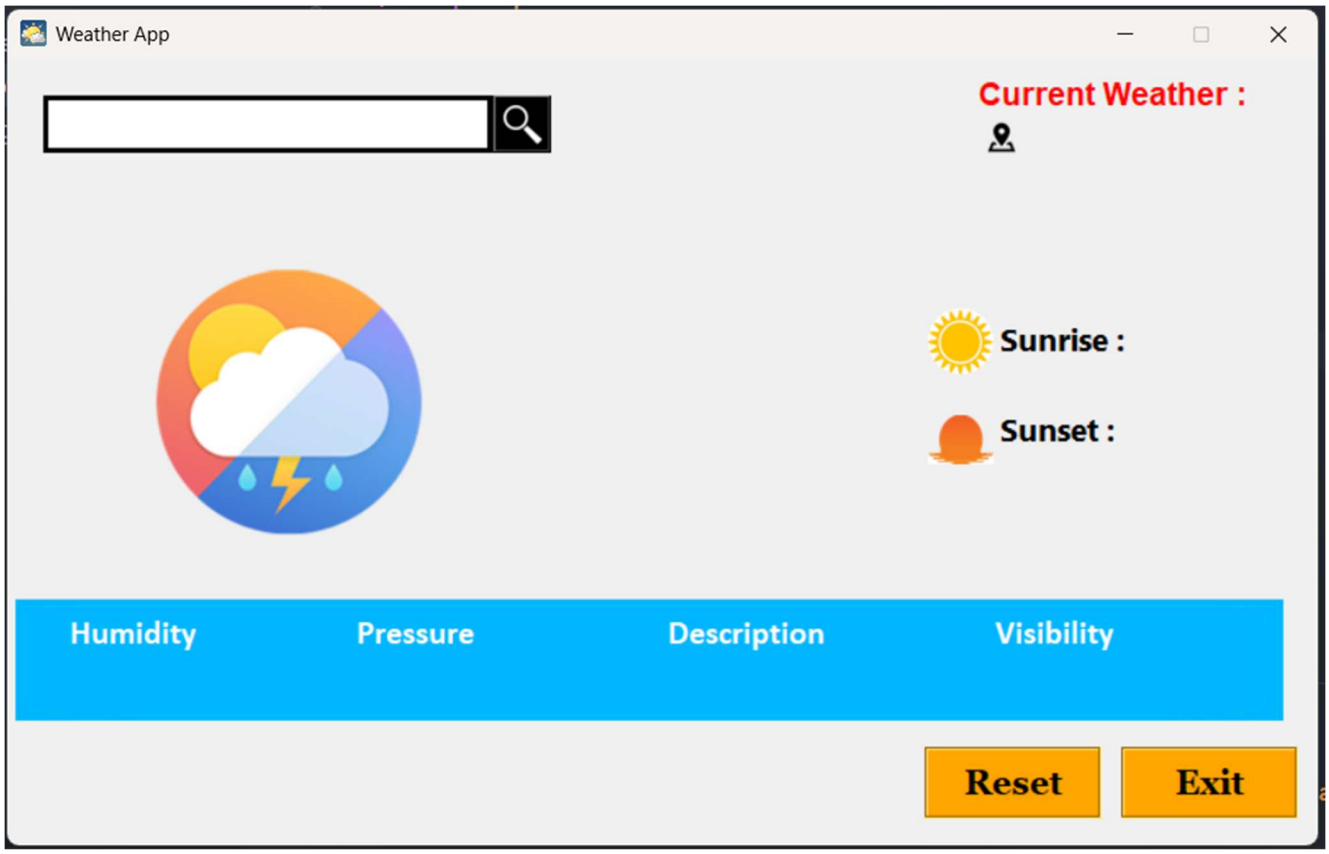
```
        # sunrise logo
        self.finalimg4=ImageTk.PhotoImage(image=Image.open(r"Images/sunrise.png").resize((40,40)))
        Label(image=self.finalimg4).place(x=560,y=150)
        self.sunrise=Label(text="Sunrise : ",font=("Segoe UI",14,'bold'))
        self.sunrise.place(x=603,y=155)

        #sunset logo
        self.finalimg5=ImageTk.PhotoImage(image=Image.open(r"Images/sunset.png").resize((40,30)))
        Label(image=self.finalimg5).place(x=560,y=215)
        self.sunset=Label(text="Sunset : ",font=("Segoe UI",14,'bold'))
        self.sunset.place(x=603,y=210)

        # bottom bar
        self.finalimg6=ImageTk.PhotoImage(image=Image.open(r'Images/bottom_bar.png').resize((770,70)))
        Label(image=self.finalimg6,bg='#00b7ff').place(x=5,y=330)

        # placing the labels
        Label(text="Humidity",font="Calibri 15 bold",bg='#00b7ff',fg='white').place(x=35,y=335)
        Label(text="Pressure",font="Calibri 15 bold",bg='#00b7ff',fg='white').place(x=210,y=335)
        Label(text="Description",font="Calibri 15 bold",bg='#00b7ff',fg='white').place(x=400,y=335)
        Label(text="Visibility",font="Calibri 15 bold",bg='#00b7ff',fg='white').place(x=600,y=335)

        # humidity label
        self.humidity=Label(text="",font=("Calibri",15,'bold'),bg='#00b7ff',fg='black')
        self.humidity.place(x=50,y=361)

        # pressure label
        self.pressure=Label(text="",font=("Calibri",15,'bold'),bg='#00b7ff',fg='black')
        self.pressure.place(x=203,y=361)

        # description label
        self.des=Label(text="",font=("Calibri",15,'bold'),bg='#00b7ff',fg='black')
        self.des.place(x=405,y=361)

        # visibility label
        self.vis=Label(text="",font=("Calibri",15,'bold'),bg='#00b7ff',fg='black')
        self.vis.place(x=610,y=361)

        # exit and reset button
        Button(text='Exit',font=("Georgia",16,"bold"),bg='orange',fg='black',width=7,relief='groove',command=self.exit).place(x=680,y=420)
        Button(text='Reset',font=("Georgia",16,"bold"),bg='orange',fg='black',width=7,relief='groove',activebackground="blue",activeforeground='white',com
mand=self.clear).place(x=560,y=420)

    def __get_weather(self):
        try:
            # getting the weather information
            city=self.search.get()
            config_file=configparser.ConfigParser()
            config_file.read("config.ini")
            api=config_file['Openweather']['api']
            data=f'https://api.openweathermap.org/data/2.5/weather?q={city}&appid={api}'
            weather=requests.get(data).json()
            self.__set_information(weather=weather)

        except requests.exceptions.ConnectionError:
            messagebox.showwarning('Connect',"Connect to The internet")
        except:
            messagebox.showerror('Error',"Some Errored Occured\nTry again Later!")

    def __set_information(self,weather):
        # print(weather)
        if weather['cod']=='404' and weather['message']=='city not found':
            messagebox.showerror("Error","Entered City Not Found")
            self.search.set("")
        elif weather['cod']=='400' and weather['message']=='Nothing to geocode':
            messagebox.showinfo("Warning",'Enter The city name')
            self.search.set('')
        else:
            # getting time according to timezone
            lon=weather['coord']['lon']  # longitutde
            lat=weather['coord']['lat']  # latitude
            tf=TimezoneFinder()
            result=tf.timezone_at(lng=lon,lat=lat)
            home=pytz.timezone(result)
            local=datetime.datetime.now(home).strftime("%d/%m/%y  %I:%M %p")
```

```
        self.timelbl['text']=local
        self.des['text']=weather['weather'][0]['description']
        self.feel['text']=f"Feels Like {int(weather['main']['feels_like']-273)}° | {weather['weather'][0]['main']}"
        type=weather['weather'][0]['main']
        self.place_image(type)

        # sets the temperature and degree label
        temp=int(weather['main']['temp']-273)
        self.degree['text']="°C"
        if temp>=100:
            self.degree.place(x=450,y=135)
        elif temp<=9 and temp>=0:
            self.degree.place(x=340,y=135)
        elif temp<=99 and temp>=10:
            self.degree.place(x=390,y=135)
        elif temp<0 and temp>=-9:
            self.degree.place(x=358,y=135)
        elif temp<=-10 and temp>=-99:
            self.degree.place(x=419,y=135)
        self.temperature['text']=int(weather['main']['temp']-273)
        self.humidity['text']=weather['main']['humidity'],'%'
        self.pressure['text']=weather['main']['pressure'],'mBar'
        self.location.config(text=weather['name'])
        self.vis['text']=int(weather['visibility']/1000),'km'
        self.sunrise['text']=f"Sunrise : \n{datetime.datetime.fromtimestamp(int(weather['sys']['sunrise'])).strftime('%d/%m/%y  %I:%M %p')}"
        self.sunset['text']=f"Sunset : \n{datetime.datetime.fromtimestamp(int(weather['sys']['sunset'])).strftime('%d/%m/%y   %I:%M %p')}"

    def place_image(self,type):
        if type=="Clear":
            img="clear.png"
            self.set_image(img)
        elif type=="Clouds":
            img='clouds.png'
            self.set_image(img)
        elif type=="Rain":
            img='rain.png'
            self.set_image(img)
        elif type=='Haze':
            img='haze.png'
            self.set_image(img)
        else:
            img='main.png'
            self.set_image(img)

    def set_image(self,img):
      self.img3=Image.open(f"Icons/{img}")
      self.resizeimg3=self.img3.resize((190,190))
      self.finalimg3=ImageTk.PhotoImage(self.resizeimg3)
      self.icons=Label(image=self.finalimg3)
      self.icons.place(x=70,y=110)

    def clear(self):
        self.des.config(text="")
        self.vis.config(text="")
        self.pressure.config(text="")
        self.humidity.config(text="")
        self.sunset.config(text="Sunset :")
        self.sunrise.config(text="Sunrise :")
        self.feel.config(text="")
        self.degree.config(text="")
        self.temperature.config(text="")
        self.timelbl.config(text="")
        self.location.config(text="")
        self.search.set("")
        img='main.png'
        self.set_image(img)

    def exit(self):
        a=messagebox.askyesno('Confirmation',"Are You sure You Want To Exit !")
        if a==True:
            self.destroy()

    def threading(self,event=0):
        t1=threading.Thread(target=self.__get_weather)
        t1.start()
```

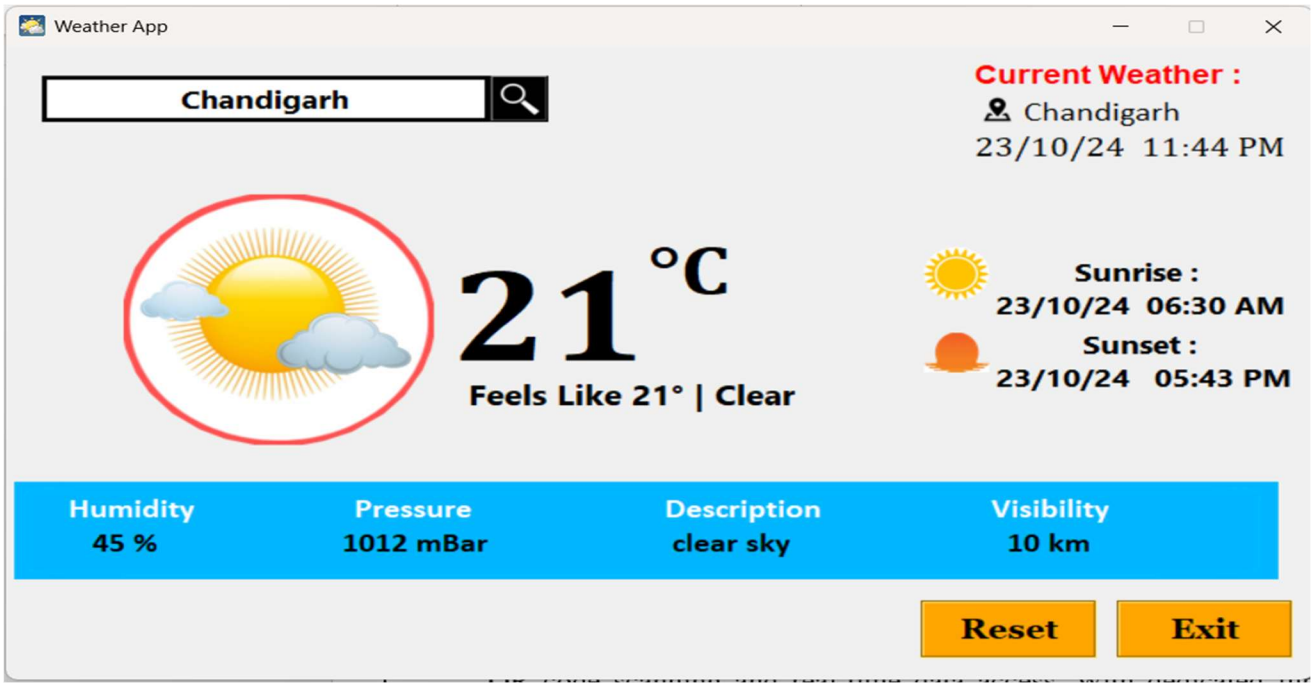
```
if __name__=="__main__":
    c=Weather()
    c.mainloop()
```

Output

Landing UI:



Searching City:



Learning Outcome

1. **Understanding of Python GUI Development**:
    - Gain hands-on experience with Tkinter, Python's built-in library for creating graphical user interfaces (GUIs), learning how to structure and design an intuitive user interface.
2. **Working with External APIs**:
    - Learn how to interact with external APIs, such as OpenWeatherMap, by sending HTTP requests and processing JSON data to retrieve and display weather information.
3. **Error Handling and User Input Validation**:
    - Implement error handling mechanisms to manage various issues such as invalid city names, network connection errors, and API errors, ensuring a seamless user experience.
4. **Multithreading in Python**:
    - Gain a practical understanding of Python's multithreading capabilities, using threads to perform background tasks (like fetching data) while keeping the GUI responsive.
5. **Manipulating and Displaying Images in Python**:
    - Learn how to use the Pillow library to manipulate and display images in a GUI, such as weather icons and location logos.
6. **Time Zone Management**:
    - Develop skills in managing time zones using the pytz and TimezoneFinder libraries to display accurate local time for various cities based on geographical coordinates.
7. **File Handling with ConfigParser**:
    - Understand how to manage configuration files securely using the ConfigParser library, enabling separation of sensitive data (like API keys) from the main application code.
8. **Real-world Problem Solving**:

- Apply programming skills to solve a real-world problem—building an app that provides real-time weather information—enhancing both coding proficiency and problem-solving capabilities.

Conclusion

The **Weather App** project serves as an excellent introduction to real-world application development using Python. Through this project, students and developers learn how to build a fully functional, user-friendly application that integrates multiple technologies, such as GUI design, API interaction, and multithreading.

The project also demonstrates the importance of error handling, responsiveness, and clear design in application development, ensuring a smooth user experience. By completing this project, learners gain valuable insights into core programming concepts while exploring practical solutions for data fetching, image processing, and time management.

Overall, this project is a stepping stone toward mastering Python and developing more complex, data-driven applications in the future.