Application 2: Real-Time Data Processing System for Weather Monitoring with Rollups and Aggregates

Objective: Develop a real-time data processing system to monitor weather conditions and provide summarized insights using rollups and aggregates. The system will utilize data from the OpenWeatherMap API (https://openweathermap.org/).

Data Source:

The system will continuously retrieve weather data from the OpenWeatherMap API. You will need to sign up for a free API key to access the data. The API provides various weather parameters, and for this assignment, we will focus on:

- **main**: Main weather condition (e.g., Rain, Snow, Clear)
- **temp:** Current temperature in Centigrade
- **feels_like**: Perceived temperature in Centigrade
- **dt**: Time of the data update (Unix timestamp)

Processing and Analysis:

- The system should continuously call the OpenWeatherMap API at a configurable interval (e.g., every 5 minutes) to retrieve real-time weather data for the metros in India. (Delhi, Mumbai, Chennai, Bangalore, Kolkata, Hyderabad)
- For each received weather update:
    - Convert temperature values from Kelvin to Celsius (tip : you can also use user preference).

Rollups and Aggregates:

1. **Daily Weather Summary:**
    - Roll up the weather data for each day.
    - Calculate daily aggregates for:
        - Average temperature
        - Maximum temperature
        - Minimum temperature
        - Dominant weather condition (give reason on this)
    - Store the daily summaries in a database or persistent storage for further analysis.

2. **Alerting Thresholds:**

    - Define user-configurable thresholds for temperature or specific weather conditions (e.g., alert if temperature exceeds 35 degrees Celsius for two consecutive updates).
    - Continuously track the latest weather data and compare it with the thresholds.
    - If a threshold is breached, trigger an alert for the current weather conditions. Alerts could be displayed on the console or sent through an email notification system (implementation details left open-ended).

3. **Implement visualizations:**
    - To display daily weather summaries, historical trends, and triggered alerts.

Test Cases:

1. **System Setup:**
    - Verify system starts successfully and connects to the OpenWeatherMap API using a valid API key.

2. **Data Retrieval:**
    - Simulate API calls at configurable intervals.
    - Ensure the system retrieves weather data for the specified location and parses the response correctly.

3. **Temperature Conversion:**
    - Test conversion of temperature values from Kelvin to Celsius (or Fahrenheit) based on user preference.

4. **Daily Weather Summary:**
    - Simulate a sequence of weather updates for several days.
    - Verify that daily summaries are calculated correctly, including average, maximum, minimum temperatures,and dominant weather condition.

5. **Alerting Thresholds:**
    - Define and configure user thresholds for temperature or weather conditions.
    - Simulate weather data exceeding or breaching the thresholds.
    - Verify that alerts are triggered only when a threshold is violated.

Bonus:

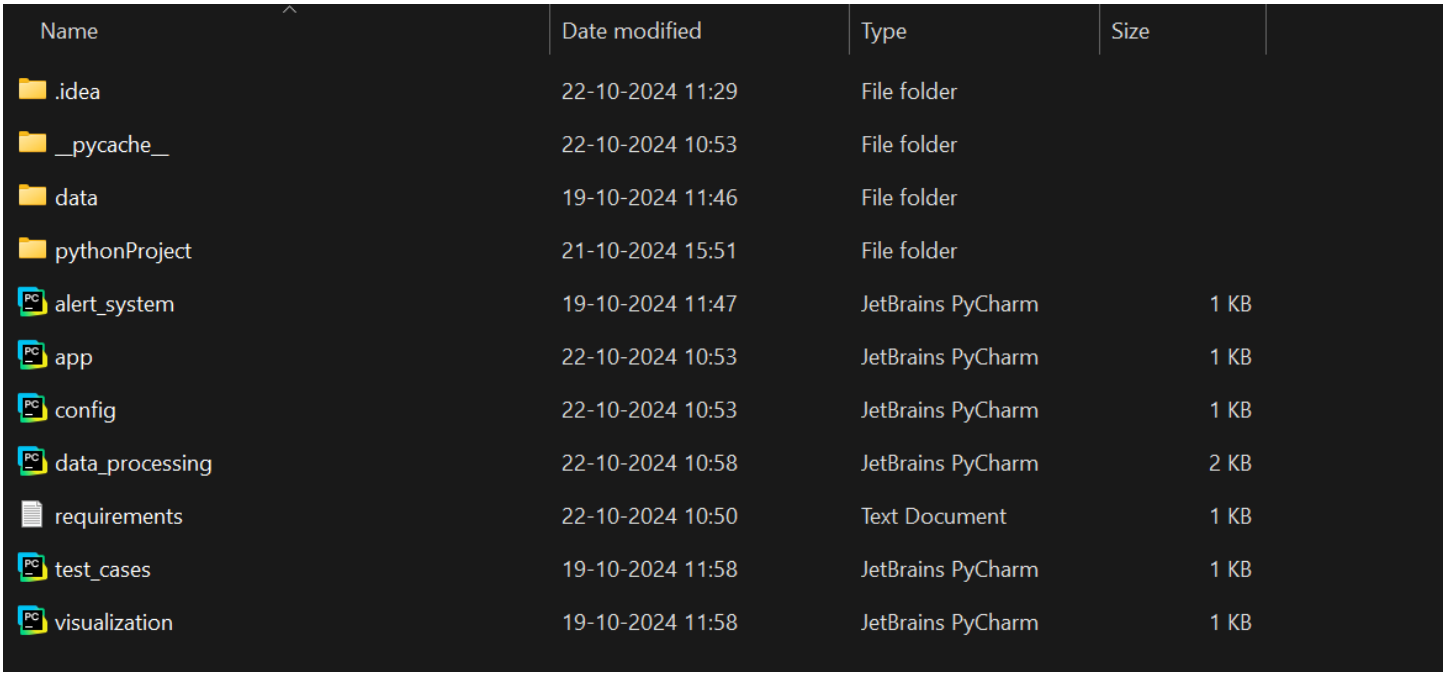
- Extend the system to support additional weather parameters from the OpenWeatherMap API (e.g., humidity, wind speed) and incorporate them into rollups/aggregates.
- Explore functionalities like weather forecasts retrieval and generating summaries based on predicted conditions.

Solution:

File Structure

weather_monitoring_system/

```
│
├── app.py                  # Main application file
├── config.py               # Configuration file (API key, cities, etc.)
├── data_processing.py      # Data processing, rollups, and aggregate functions
├── alert_system.py         # Alerting and threshold monitoring
├── visualization.py        # Code for creating visualizations
├── test_cases.py           # Test cases for the system
├── requirements.txt        # Python dependencies
└── data/
    └── daily_weather_summary.csv # CSV file to store daily weather summaries
```



Name	Date modified	Type	Size
.idea	22-10-2024 11:29	File folder	
__pycache__	22-10-2024 10:53	File folder	
data	19-10-2024 11:46	File folder	
pythonProject	21-10-2024 15:51	File folder	
alert_system	19-10-2024 11:47	JetBrains PyCharm	1 KB
app	22-10-2024 10:53	JetBrains PyCharm	1 KB
config	22-10-2024 10:53	JetBrains PyCharm	1 KB
data_processing	22-10-2024 10:58	JetBrains PyCharm	2 KB
requirements	22-10-2024 10:50	Text Document	1 KB
test_cases	19-10-2024 11:58	JetBrains PyCharm	1 KB
visualization	19-10-2024 11:58	JetBrains PyCharm	1 KB

Tools & Technologies

- **Language**: Python
- **Database**: SQLite or CSV files (for lightweight persistence)
- **API Integration**: OpenWeatherMap API
- **Visualization**: Matplotlib or Plotly

- **Alerting**: Email (optional) or console notifications
- **Test Framework**: Python's built-in unittest

Step-by-Step Instructions

Step 1: Create requirements.txt for Dependencies

First, create a **requirements.txt** file to manage dependencies. This will make it easy for others to install necessary packages.

txt

```
requests

pandas

matplotlib

plotly
```

You can install these dependencies using:

pip install -r requirements.txt

Step 2: Create *config.py* for Configurations

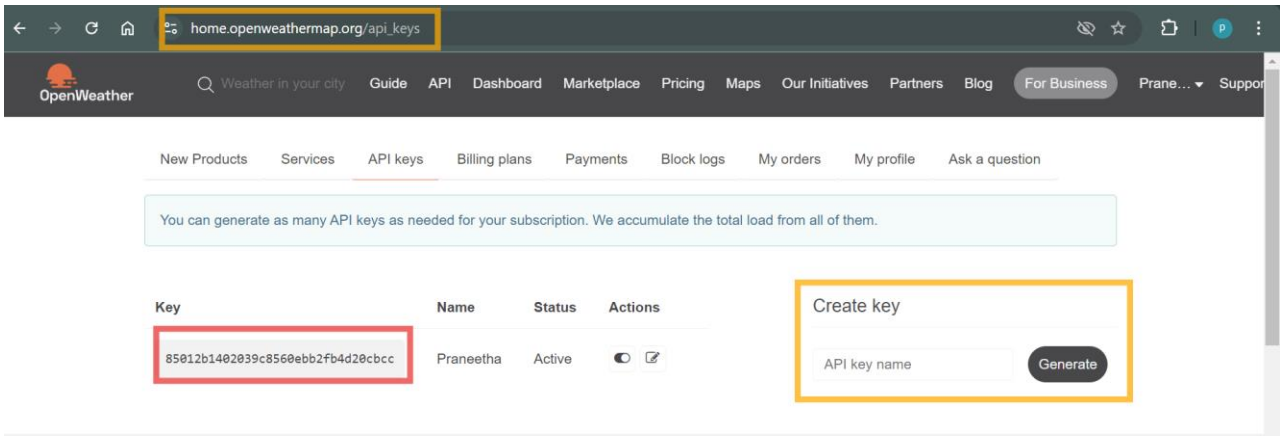
This file will store API keys, city names, and other configurations.

python

```
# config.py

API_KEY = "your_openweathermap_api_key"

BASE_URL = "http://api.openweathermap.org/data/2.5/weather"

CITIES = ['Delhi', 'Mumbai', 'Chennai', 'Bangalore', 'Kolkata', 'Hyderabad']

POLL_INTERVAL = 300  # Time interval for API calls in seconds (e.g., 5 minutes)
```

Get API_KEY from https://home.openweathermap.org/api_keys

- Create Account
- Default Api Key or Generate Key



Step 3: Create *data_processing.py* for Data Handling

This file will handle retrieving weather data, converting temperatures, and calculating aggregates.

```
# data_processing.py

import requests

import pandas as pd

from config import API_KEY, BASE_URL, CITIES


# Function to retrieve weather data

def get_weather_data(city):

    url = f"{BASE_URL}?q={city}&appid={API_KEY}"

    response = requests.get(url)

    data = response.json()

    temp_kelvin = data['main']['temp']

    weather_condition = data['weather'][0]['main']

    timestamp = data['dt']

    return {

        'city': city,

        'temp_celsius': kelvin_to_celsius(temp_kelvin),

        'weather_condition': weather_condition,

        'timestamp': timestamp

    }
```

```
# Temperature conversion
def kelvin_to_celsius(temp_kelvin):
    return temp_kelvin - 273.15

# Function to roll up daily data into aggregates
def calculate_daily_aggregates(weather_data):
    df = pd.DataFrame(weather_data)
    daily_summary = {
        'average_temp': df['temp_celsius'].mean(),
        'max_temp': df['temp_celsius'].max(),
        'min_temp': df['temp_celsius'].min(),
        'dominant_condition': df['weather_condition'].mode()[0]
    }
    return daily_summary

# Function to store the summary in a CSV file
def save_daily_summary(summary, file_path="data/daily_weather_summary.csv"):
    df = pd.DataFrame([summary])
    df.to_csv(file_path, mode='a', header=not pd.io.common.file_exists(file_path), index=False)
```

Step 4: Create *alert_system.py* for Alerts

This module monitors the data and triggers alerts based on user-defined thresholds.

```
# alert_system.py

def check_alerts(weather_data, temp_threshold=35):
    consecutive_exceeds = 0
    for record in weather_data:
        if record['temp_celsius'] > temp_threshold:
```

```
            consecutive_exceeds += 1
            if consecutive_exceeds >= 2:
                trigger_alert(record['city'], record['temp_celsius'])
        else:
            consecutive_exceeds = 0

def trigger_alert(city, temperature):
    print(f"ALERT: {city} has exceeded {temperature}°C for two consecutive updates!")
    # Optional: Send email using smtplib if required
```

Step 5: Create *visualization.py* for Data Visualization

This file creates visualizations like daily temperature trends and alert history.

```
# visualization.py
import pandas as pd
import matplotlib.pyplot as plt

def plot_weather_trends(file_path="data/daily_weather_summary.csv"):
    df = pd.read_csv(file_path)
    df.plot(x='date', y=['average_temp', 'max_temp', 'min_temp'], kind='line')
    plt.title("Daily Temperature Trends")
    plt.xlabel("Date")
    plt.ylabel("Temperature (°C)")
    plt.show()
```

Step 6: Create *app.py* to Run the Main Application

This is the main file that orchestrates the API calls, data processing, and alerting.

```
# app.py

import time
from data_processing import get_weather_data, calculate_daily_aggregates, save_daily_summary
from alert_system import check_alerts
```

```
from config import CITIES, POLL_INTERVAL

def main():
    weather_data = []
    while True:
        for city in CITIES:
            data = get_weather_data(city)
            weather_data.append(data)
            print(f"Retrieved weather data for {city}: {data['temp_celsius']}°C")

        # After collecting all cities' data, process daily aggregates
        daily_summary = calculate_daily_aggregates(weather_data)
        save_daily_summary(daily_summary)

        # Check for any alerts
        check_alerts(weather_data)

        # Wait before next API call
        time.sleep(POLL_INTERVAL)

if __name__ == "__main__":
    main()
```

Step 7: Create *test_cases.py* for Unit Tests

This file will test different functionalities of the system.

```
# test_cases.py

import unittest

from data_processing import kelvin_to_celsius, calculate_daily_aggregates



class TestWeatherSystem(unittest.TestCase):

    def test_kelvin_to_celsius(self):

        self.assertEqual(kelvin_to_celsius(273.15), 0)  # Test freezing point

        self.assertEqual(kelvin_to_celsius(298.15), 25) # Test room temperature



    def test_daily_aggregates(self):

        test_data = [

            {'temp_celsius': 25, 'weather_condition': 'Clear'},

            {'temp_celsius': 30, 'weather_condition': 'Clouds'},
```

```
            {'temp_celsius': 27, 'weather_condition': 'Clear'}
        ]
        result = calculate_daily_aggregates(test_data)
        self.assertEqual(result['average_temp'], 27.33)
        self.assertEqual(result['max_temp'], 30)
        self.assertEqual(result['min_temp'], 25)
        self.assertEqual(result['dominant_condition'], 'Clear')


if __name__ == '__main__':
    unittest.main()
```

Run tests using:

python test_cases.py

How to Run the System

1. **Install dependencies**:

   pip install -r requirements.txt

2. **Set up the API key** in "*config.py.*"
3. **Run the main application**:

   python app.py

4. **Run tests**:

   python test_cases.py

5. **Visualize the data** by running:

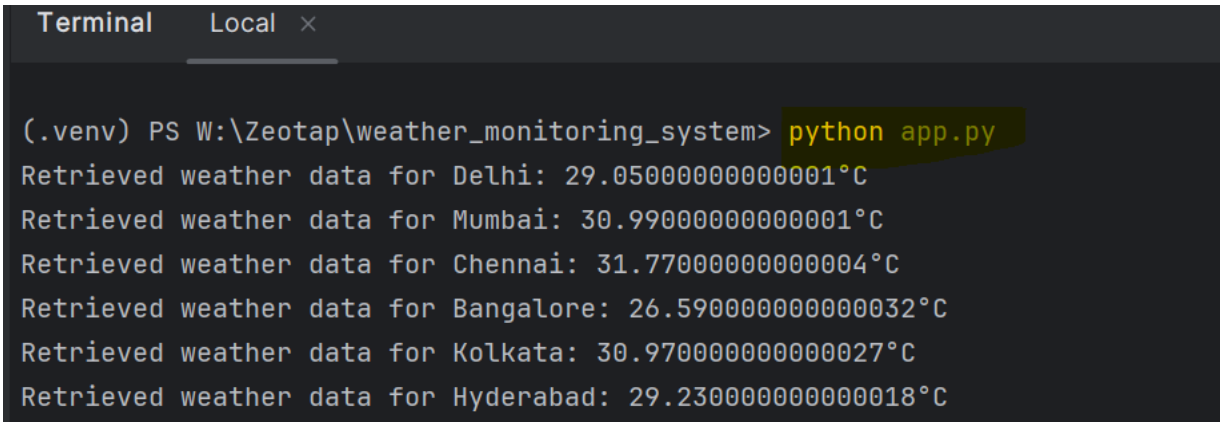
   python visualization.py

By following these steps, you'll have a fully functioning real-time weather monitoring system complete with API integration, data rollups, alerts, and visualizations.

Output:

Date: 22-10-2024

Time: AT 10:55 AM



Time: At 11:45 AM