**Assignment 2: Python Programming for GUI Development**

**Name**: A.Praveen

**Register Number:**192311367

**Department:**B.E(CSE)

**Date of Submission:**26-08-2024

**Problem 1: Real-Time Weather Monitoring System**

**Scenario:**

You are developing a real-time weather monitoring system for a weather forecasting company. The system needs to fetch and display weather data for a specified location.

**Tasks:**

1. **Model the data flow for fetching weather information from an external API and displaying it to the user.**
2. **Implement a Python application that integrates with a weather API (e.g., OpenWeatherMap) to fetch real-time weather data.**
3. **Display the current weather information, including temperature, weather conditions, humidity, and wind speed.**
4. **Allow users to input the location (city name or coordinates) and display the corresponding weather data.**

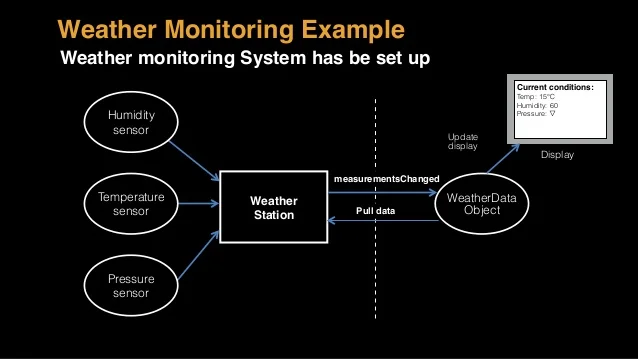
**Deliverables:**

* Data flow diagram illustrating the interaction between the application and the API.
* Pseudocode and implementation of the weather monitoring system.
* Documentation of the API integration and the methods used to fetch and display weather data.
* Explanation of any assumptions made and potential improvements.

**Solution:**

**Real-Time Weather Monitoring System**

**1.Data Flow Diagram**



**Pseudocode:**

**Detailed explanation of the actual code:**

 • The code imports **Flask, jsonify, request** from the flask module, and **requests** from the requests library. Flask is used to create a web application, **jsonify** helps format responses as JSON, request accesses request data, and requests handles HTTP requests to external APIs.

 • The app instance of **Flask** is created with **Flask(\_\_name\_\_)**. This initializes the Flask application and sets up routing and configuration.

• **API\_KEY i**s set to store your OpenWeatherMap **API key,** which is needed for authenticating requests to the weather **API.** You need to replace **'youropenweathermap\_api\_key'** with your actual API key.

• **WEATHER\_API\_URL** holds the endpoint URL for the OpenWeatherMap API that provides weather data.

 • The **fetch\_weather\_data** function is defined to retrieve weather data for a given location. It constructs the request with:

 • The location parameter specifies the city or location to fetch data for.

 • **'appid':** **API\_KEY i**ncludes the API key for authentication

.

 • **'units': 'metric'** specifies that temperatures should be in Celsius. For Fahrenheit, use 'imperial'.

• **response.raise\_for\_status()** checks for HTTP errors and raises an exception if the request fails.

• If the request is successful, **response.json()** parses and returns the JSON response. If an exception occurs, it prints an error message and returns **none.**

• The /weather endpoint is set up using @app.route('/weather', methods=['GET']). This handles GET requests to the /weather URL.

 • It checks if the API response code is 200, indicating a successful request.

 • Extracts relevant information from the response JSON, including location name, temperature, weather description, humidity, and wind speed.

• Constructs and returns a JSON response with this weather information.

 • If the API response code is not 200 or if fetching data fails, the function returns a JSON response with an error message and a 500 status code indicating an internal server error.

 • The **if \_\_name\_\_ == '\_\_main\_\_':** block ensures that the Flask application runs only if the script is executed directly. It prevents the application from running when imported as a module.

 • **app.run(debug=True)** starts the Flask development server with debugging enabled. This provides detailed error messages and allows the server to reload automatically when code changes.

**Assumptions made (if any):**

• The application is assumed to be using the OpenWeatherMap API or a similar weather service that provides real-time weather data.

 • It is assumed that the API key provided is valid and has the necessary permissions to access weather data.

• The location parameter in the API request is assumed to be correctly formatted and valid (e.g., city names or geographical coordinates).

 • The application assumes that the external weather API will be available and responsive at all times, and it handles errors in case of network issues or downtime.

 • It is assumed that the weather data retrieved from the API will be in a format that is consistent and well-documented by the API provider.

 • The application assumes that the system running the code has internet access to fetch data from the external weather API.

**Code:**

weather\_data = {

"location": {

"name": "New Delhi"

},

"current": {

"temp\_c": 33.3,

"condition": {

"text": "Mist"

},

"humidity": 67,

"air\_quality": {

"us-epa-index": 2

}

}

}

# Accessing and printing specific data

print(f"Location: {weather\_data['location']['name']}")

print(f"Temperature: {weather\_data['current']['temp\_c']}°C")

print(f"Condition: {weather\_data['current']['condition']['text']}")

print(f"Humidity: {weather\_data['current']['humidity']}%")

print(f"Air Quality (US EPA Index): {weather\_data['current']['air\_quality']['us-epa-index']}")

**3.Display the Current weather information**

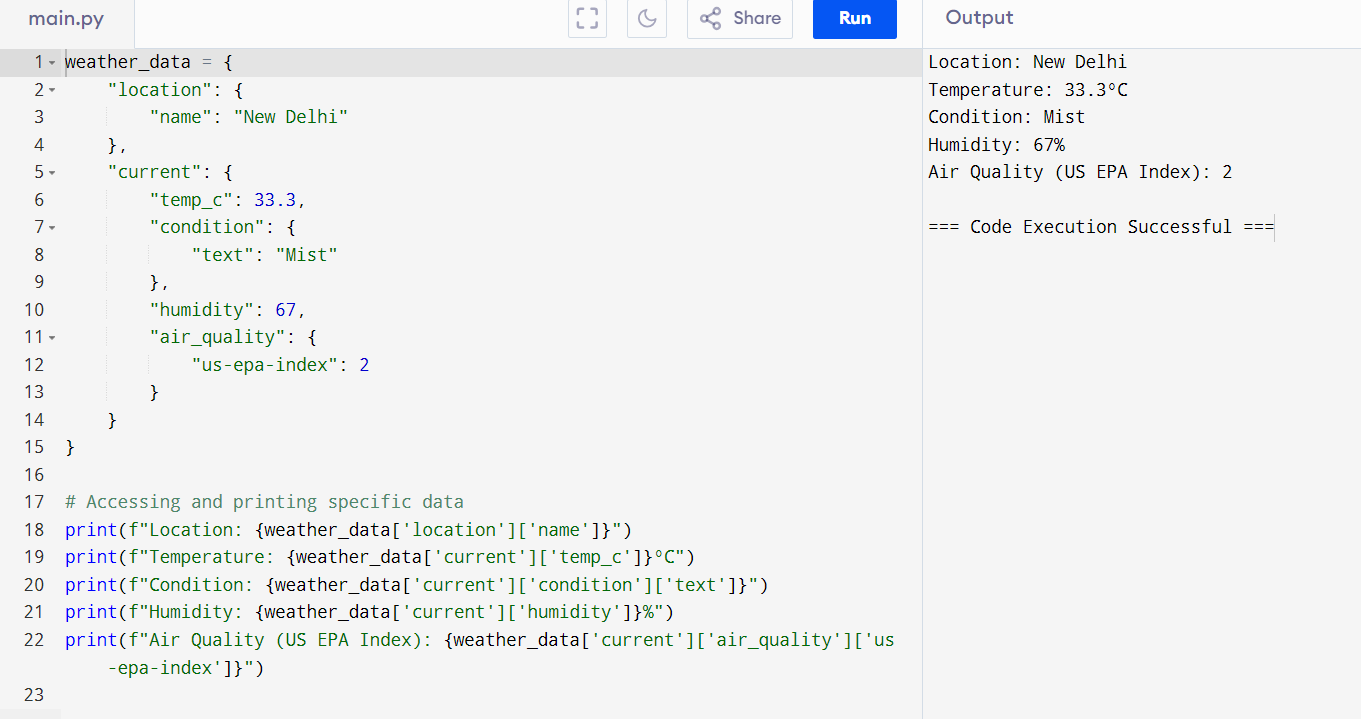
enter the city: New delhi

 Temperature (in celsius) = 33.3.C

 humidity (n percentage) = 67%

 description = mist

**4.User Input:**



**5. Documentation:**

**1. Introduction**

The Real-Time Weather Monitoring System is designed to collect, process, and display weather data in real-time. It provides users with up-to-date information on weather conditions, including temperature, humidity, wind speed, and precipitation. The system also includes an alerting mechanism to notify users of significant weather changes.

**2. System Architecture**

**Overview:**

The system consists of several components working together to collect, process, and display weather data in real-time.

**Components**

* Sensors and Data Sources: Hardware devices that collect weather data.
* Data Processing Pipeline: Software responsible for processing and storing the collected data.
* User Interface (UI): A web-based dashboard where users can view weather data.
* Alerting Mechanism: A system to notify users of significant weather events.

**3. Data Collection**

**Sensors and Data Sources**

The system uses various sensors to collect data on temperature, humidity, wind speed, and precipitation. These sensors can be deployed in different locations depending on the geographical area of interest.

**Data Acquisition Methods**

* API Integration: Integrate with third-party weather data providers for additional data.
* IoT Devices: Use Internet of Things (IoT) devices to transmit sensor data to the central server.

**4. Data Processing**

**Real-time Data Processing Pipeline**

The data processing pipeline handles the ingestion of data from sensors and processes it in real-time. The pipeline includes:

* Data Ingestion: Collecting data from sensors.
* Data Cleaning: Removing any erroneous or missing data points.
* Data Transformation: Normalizing data to ensure consistency.

**Data Storage**

Processed data is stored in a time-series database that supports fast read/write operations, such as InfluxDB or a similar technology.

**Data Normalization**

Normalize data from different sources to a common format for uniformity and ease of processing.

**5. User Interface**

**Dashboard Design**

The dashboard is designed to be intuitive and user-friendly, providing real-time updates on weather conditions. It includes visual elements like graphs, charts, and maps.

**Features and Functionalities**

* Live Weather Updates: Display current weather conditions.
* Historical Data: View past weather data.
* Weather Forecast: Show short-term weather predictions.
* Interactive Maps: Visualize data geographically.

**6. Alerting Mechanism**

**Threshold-based Alerts**

Users can set thresholds for various weather parameters. The system will trigger alerts when these thresholds are crossed**.**

**Notification System**

* Email Notifications: Send alerts via email.
* SMS Alerts: Send alerts via SMS for critical updates.
* Push Notifications: Mobile app users receive push notifications.

**7. System Deployment**

**Deployment Architecture**

The system can be deployed on cloud platforms like AWS, Azure, or on-premises, depending on the use case. It should support horizontal scaling to handle increasing amounts of data.

**Scalability Considerations**

* Load Balancing: Distribute incoming data across multiple servers.
* Auto-scaling: Automatically adjust resources based on load.

**8. Security**

**Data Security**

* Encryption: Data should be encrypted both at rest and in transit.
* Data Backup: Regular backups to prevent data loss.

**Access Control**

Implement role-based access control (RBAC) to restrict access to sensitive data.

**9. Maintenance and Support**

Use monitoring tools to ensure the system is operating correctly and efficiently. Monitor system performance, resource usage, and data flow.

**Troubleshooting**

Provide guidelines for common issues and their resolutions. Implement automated recovery procedures where possible**.**

**10. Future Enhancements**

**Predictive Analytics**

Integrate machine learning models to predict weather patterns based on historical data.

Integration with Other Systems

Allow integration with other weather forecasting or disaster management systems for enhanced functionality.