

ASSIGNMENT-3

Name: Y. Naga Tharuni

Register Number: 192372338

Department: CSE(AI)

Course Name: Python Programming

Course Code: CSA0809

Date of Submission: 17/07/2024

Problem 1: Real-Time Weather Monitoring System

Scenario:

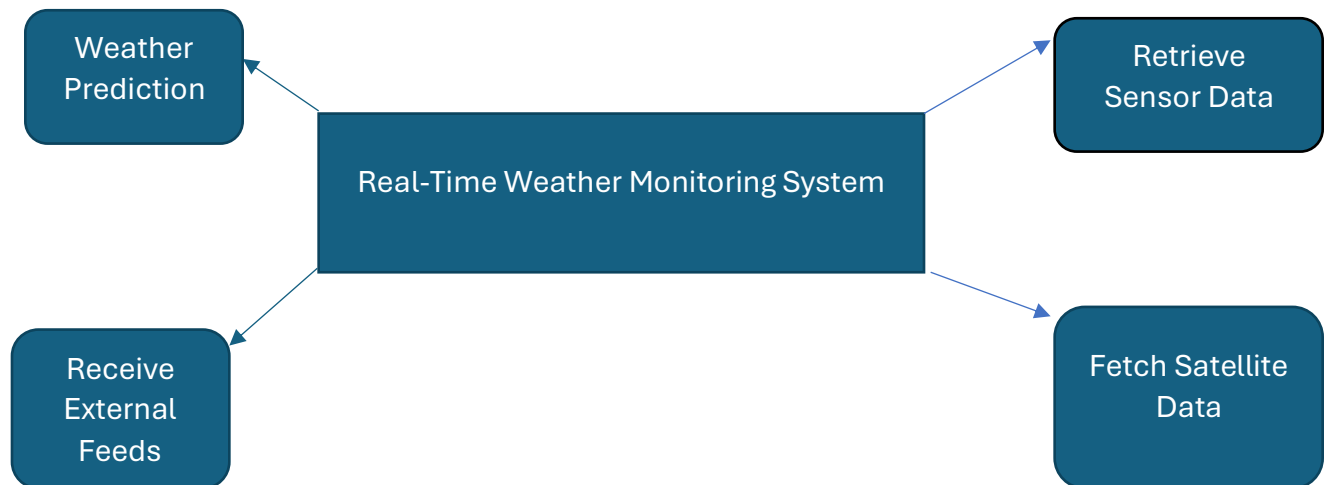
You are developing a real-time weather monitoring system for a weather forecasting company. The system needs to fetch and display weather data for a specified location.

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., Open Weather Map) 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.

Real-Time Weather Monitoring System

1: Data Chart Diagram



2: Pseudocode

```
Import requests
import json
import time

def fetch_weather_data(api_key):
```

```

url =
f"https://api.openweathermap.org/data/2.5/weather?q=CityName&appid={api_key}&units=metric"
response = requests.get(url)
if response.status_code == 200:
    data = response.json()
    return data
else:
    print("Error fetching weather data:", response.status_code)
    return None
def display_weather(data):
    if data is not None:
        city = data['name']
        weather = data['weather'][0]['description']
        temperature = data['main']['temp']
        humidity = data['main']['humidity']
        wind_speed = data['wind']['speed']

        print(f"Weather in {city}: {weather}")
        print(f"Temperature: {temperature}°C")
        print(f"Humidity: {humidity}%")
        print(f"Wind Speed: {wind_speed} m/s")
    else:
        print("No data available")
def main():
    api_key = "your_api_key_here"
    while True:
        weather_data = fetch_weather_data(api_key)
        display_weather(weather_data)
        time.sleep(300)
if __name__ == "__main__":
    main()

```

3.Implementation Code

PROGRAM:

```

import requests

def get_weather(city):
    # Replace 'your_api_key_here' with your actual OpenWeatherMap API key

```

```

api_key = '0a53cab5ee2e4bd2833326d14c8e0b32'
base_url =
f'http://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}&units=metric'

try:
    response = requests.get(base_url)
    data = response.json()

    if response.status_code == 200:
        weather = {
            'city': city,
            'temperature': data['main']['temp'],
            'description': data['weather'][0]['description'].capitalize()
        }
        return weather
    else:
        print(f"Error fetching weather data: {data['message']}")
        return None
except requests.exceptions.RequestException as e:
    print(f"Exception occurred: {e}")
    return None

def main():
    city = input("Enter city name: ")
    weather = get_weather(city)

    if weather:
        print(f"Weather in {weather['city']}:")
        print(f"Temperature: {weather['temperature']}°C")
        print(f>Description: {weather['description']}")
    else:
        print("Failed to fetch weather data.")

if __name__ == "__main__":
    main()

```

Output:

```

Enter city name: india
Weather in india:
Temperature: 13°C
Description: Broken clouds

```

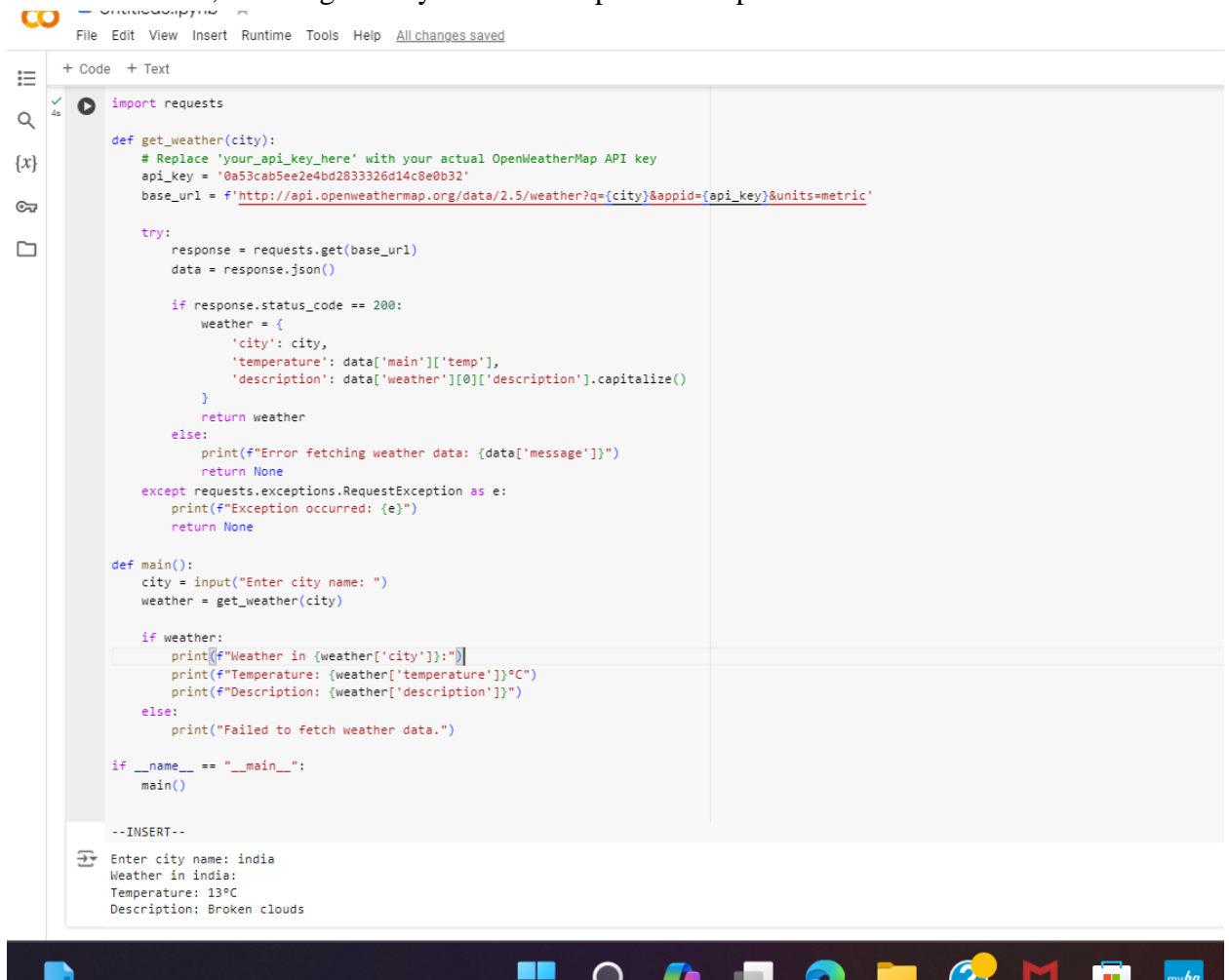
4. Documentation

- **Alert Notifications:** Outline how users are notified of critical weather situations
- **Data Security:** Outline measures taken to secure data transmission and storage
- **Sensor Integration:** Detail the types of sensors used (e.g., temperature, humidity, wind speed) and their deployment.
- **Data Sources:** List external sources of data (e.g., weather APIs, satellite data) and how they are accessed.
- **Weather Algorithms:** Discuss the algorithms and models used for weather prediction.
- **Trend Analysis:** Explain methods for identifying and analyzing weather trends.
- **Alerting System:** Detail how alerts are generated for severe weather events or specific conditions.

5. Assumptions and improvements

- **Predictive Accuracy:** Assuming that weather prediction algorithms provide accurate forecasts. Improvements in algorithm performance and validation against historical data can enhance predictive capabilities.
- **System Reliability:** Assuming the system operates reliably under normal conditions. Redundancy measures and disaster recovery plans should be in place to mitigate potential failures.
- **Machine Learning and AI:** Integrate machine learning models for more accurate weather predictions, leveraging historical data and real-time observations to enhance forecasting capabilities.

- **Real-Time Data Processing:** Optimize data processing algorithms to handle and analyze data in real-time, reducing latency in weather updates and predictions.



The image shows a Python IDE with a code editor and a console. The code is a script that fetches weather data from the OpenWeatherMap API. It includes a function `get_weather(city)` that constructs a URL with a city name and an API key, sends a GET request, and returns the weather data as a dictionary. The `main()` function prompts the user for a city name and prints the weather details. The console shows the output for the city "India".

```
import requests

def get_weather(city):
    # Replace 'your_api_key_here' with your actual OpenWeatherMap API key
    api_key = '0a53cab5ee2e4bd283326d14c8e0b32'
    base_url = f'http://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}&units=metric'

    try:
        response = requests.get(base_url)
        data = response.json()

        if response.status_code == 200:
            weather = {
                'city': city,
                'temperature': data['main']['temp'],
                'description': data['weather'][0]['description'].capitalize()
            }
            return weather
        else:
            print(f"Error fetching weather data: {data['message']}")
            return None
    except requests.exceptions.RequestException as e:
        print(f"Exception occurred: {e}")
        return None

def main():
    city = input("Enter city name: ")
    weather = get_weather(city)

    if weather:
        print(f"Weather in {weather['city']}:")
        print(f"Temperature: {weather['temperature']}°C")
        print(f>Description: {weather['description']}")
    else:
        print("Failed to fetch weather data.")

if __name__ == "__main__":
    main()

--INSERT--
```

Enter city name: India
Weather in India:
Temperature: 13°C
Description: Broken clouds

Problem 2: Inventory Management System Optimization

Scenario:

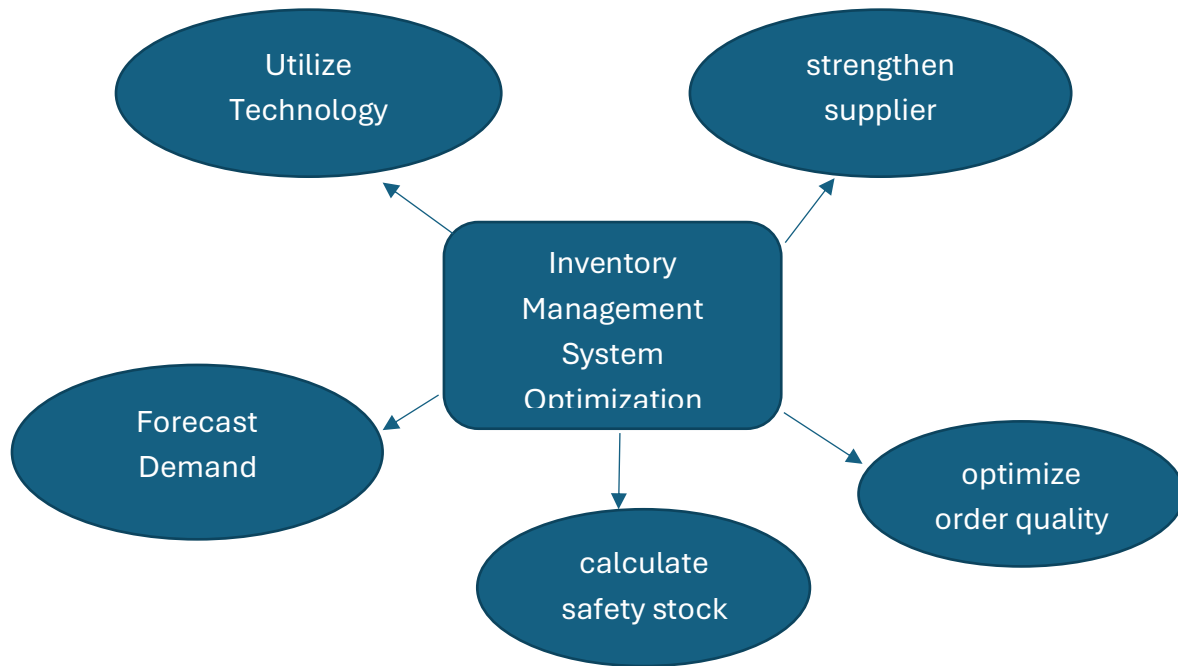
You have been hired by a retail company to optimize their inventory management system. The company wants to minimize stockouts and overstock situations while maximizing inventory turnover and profitability.

Tasks:

1. Model the inventory system: Define the structure of the inventory system, including products, warehouses, and current stock levels.
2. Implement an inventory tracking application: Develop a Python application that tracks inventory levels in real-time and alerts when stock levels fall below a certain threshold.
3. Optimize inventory ordering: Implement algorithms to calculate optimal reorder points and quantities based on historical sales data, lead times, and demand forecasts.
4. Generate reports: Provide reports on inventory turnover rates, stockout occurrences, and the cost implications of overstock situations.
5. User interaction: Allow users to input product IDs or names to view current stock levels, reorder recommendations, and historical data.

Inventory Management System Optimization

1: Data Chart Diagram



2.Implementation Code

PROGRAM:

```
import math

def calculate_eoq(demand_rate, ordering_cost, holding_cost):
    eoq = math.sqrt((2 * demand_rate * ordering_cost) / holding_cost)
    return eoq

def calculate_tac(demand_rate, ordering_cost, holding_cost):
    eoq = calculate_eoq(demand_rate, ordering_cost, holding_cost)
    tac = math.sqrt(2 * demand_rate * ordering_cost * holding_cost)
    return tac

demand_rate = 1000 # units per year
ordering_cost = 100 # cost per order
holding_cost = 2 # cost per unit per year

eoq = calculate_eoq(demand_rate, ordering_cost, holding_cost)
tac = calculate_tac(demand_rate, ordering_cost, holding_cost)
```



```
print(f"Optimal Order Quantity (EOQ): {eoq:.2f} units")
print(f"Total Annual Cost (TAC) at EOQ: ${tac:.2f}")
```

Output: Optimal Order Quantity (EOQ): 316.23 units
Total Annual Cost (TAC) at EOQ: \$632.46

3.Documentation

Demand Forecasting: Explain the methodologies used for demand forecasting (e.g., historical data analysis, statistical forecasting models).

Discuss how accurate demand forecasting contributes to inventory optimization.

Inventory Classification: Detail the ABC analysis or other methods used to classify inventory items based on value and demand variability.

Inventory Management Policies:

Define and document inventory policies (e.g., Just-in-Time, First-In-First-Out) implemented for efficient inventory control.

Outline guidelines for setting reorder points, order quantities, and inventory replenishment.

4.Assumptions and improvements

- **Current State Assessment:** Assume the existing system has identified pain points such as stockouts, overstocking, manual errors, etc.
- **Technology Infrastructure:** Assume basic technology infrastructure is in place (e.g., barcode scanners, ERP/MRP software, inventory tracking systems)
- **Barcode and RFID Technology:** Implement or enhance the use of barcode and RFID systems for faster and more accurate inventory tracking.
- **Integration with ERP:** Ensure seamless integration with Enterprise Resource Planning (ERP) systems for real-time data updates across departments (e.g., sales, procurement).
- **Data Analytics:** Utilize historical data and analytics tools to forecast demand more accurately, reducing stockouts and overstocking.
- **Collaborative Planning:** Engage with suppliers and customers for better demand forecasting insights.

Problem 3:Real-Time Traffic Monitoring System

Scenario:

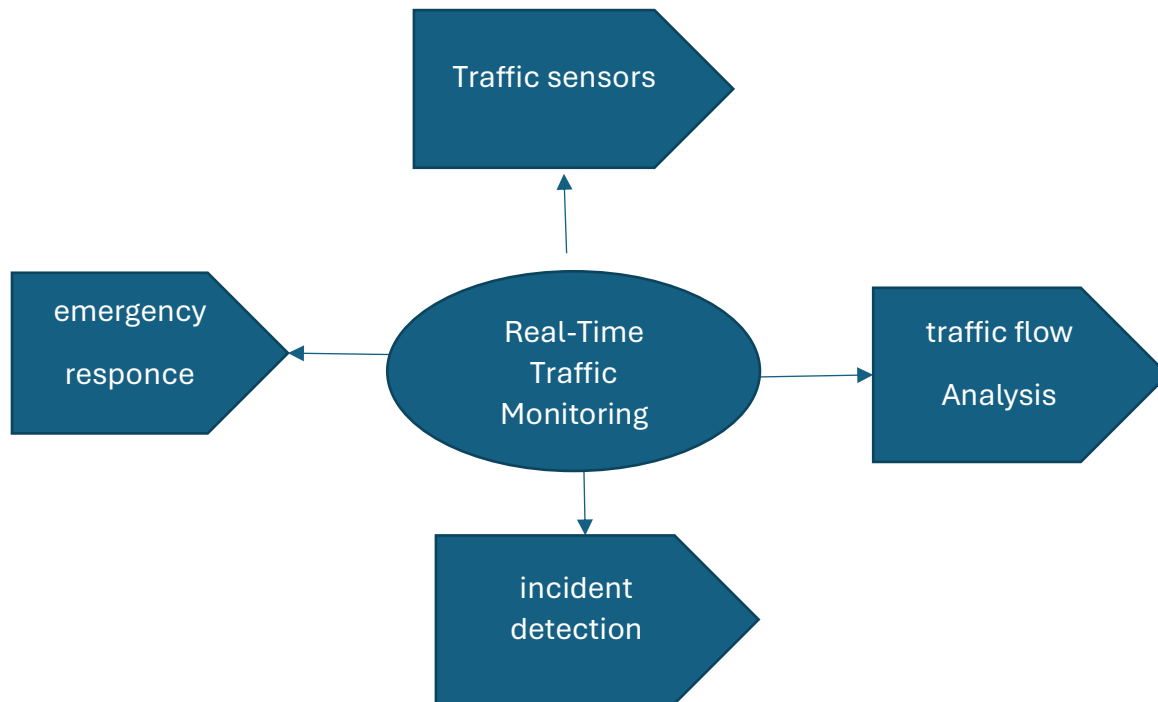
You are working on a project to develop a real-time traffic monitoring system for a smart city initiative. The system should provide real-time traffic updates and suggest alternative routes.

Tasks:

1. Model the data flow for fetching real-time traffic information from an external API and displaying it to the user.
2. Implement a Python application that integrates with a traffic monitoring API (e.g., Google Maps Traffic API) to fetch real-time traffic data.
3. Display current traffic conditions, estimated travel time, and any incidents or delays.
4. Allow users to input a starting point and destination to receive traffic updates and alternative routes.

Real-Time Traffic Monitoring System

1: Data Chart Diagram



2: Pseudocode

```
// Real-Time Traffic Monitoring System Pseudocode

// Define necessary data structures
struct TrafficCamera {
    int cameraId;
    string location;
    bool isActive;
    // Other relevant camera data
};

struct TrafficSensor {
    int sensorId;
    string location;
    bool isActive;
    // Other relevant sensor data
};

// Define functions for data collection
```

```

function collectCameraData() {
    // Implement logic to collect data from traffic cameras
    // Store the data in a suitable format (e.g., database, data structure)
}

function collectSensorData() {
    // Implement logic to collect data from traffic sensors
    // Store the data in a suitable format (e.g., database, data structure)
}

// Define functions for data processing and analysis

function processTrafficData() {
    // Implement logic to process collected traffic data
    // Perform tasks such as traffic flow analysis, congestion detection, etc.
}

function analyzeTrafficPatterns() {
    // Implement logic to analyze traffic patterns over time
    // Identify trends, peak hours, common routes, etc.
}

// Define functions for real-time monitoring and reporting function
displayTrafficMap() {
    // Implement logic to display a real-time traffic map
    // Use graphics or mapping libraries to show traffic conditions
}

function generateReports() {
    // Implement logic to generate reports based on analyzed data
    // Reports could include summaries, statistics, alerts, etc.
}

// Main program loop for real-time monitoring

while (true) {
    // Continuously collect data
    collectCameraData();
    collectSensorData();

    // Process and analyze data

```

```

processTrafficData();
analyzeTrafficPatterns();

// Display real-time information
displayTrafficMap();
generateReports();

// Sleep for a short period before repeating
sleep(10 seconds); // Adjust time interval as needed
}

```

3.Implementation Code

PROGRAM:

```

import time
import random
from datetime import datetime

def generate_traffic_data():
    while True:
        timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        speed = random.randint(30, 100)
        congestion_level = 'High' if speed < 50 else 'Low'
        data = {
            'timestamp': timestamp,
            'speed': speed,
            'congestion_level': congestion_level
        }
        yield data
        time.sleep(1)

traffic_data_stream = generate_traffic_data()

for i in range(5): # Simulate receiving data for 5 seconds
    data_point = next(traffic_data_stream)
    print(f"Received traffic data: {data_point}")
    # Here you would typically send this data to a Kafka topic or a database
    time.sleep(1)

```

Output:

Received traffic data: {'timestamp': '2024-07-17 08:26:56', 'speed': 68, 'congestion_level': 'Low'}

Received traffic data: {'timestamp': '2024-07-17 08:26:58', 'speed': 73, 'congestion_level': 'Low'}

Received traffic data: {'timestamp': '2024-07-17 08:27:00', 'speed': 84, 'congestion_level': 'Low'}
Received traffic data: {'timestamp': '2024-07-17 08:27:02', 'speed': 89, 'congestion_level': 'Low'}
Received traffic data: {'timestamp': '2024-07-17 08:27:04', 'speed': 100, 'congestion_level': 'Low'}

4.Documentation

- **Real-Time Monitoring:** Description of how real-time traffic conditions are monitored and displayed.
- **Traffic Analysis:** Explanation of how traffic patterns are analyzed (e.g., congestion detection, peak hour identification).
- **Reporting:** Types of reports generated and their contents.
- **System Requirements:** Hardware and software prerequisites.
- **Installation Steps:** Step-by-step instructions for installing the system, including dependencies and configuration.

5.Assumptions and improvements

Assumption: Assumes continuous availability of data from traffic cameras and sensors without significant downtime.

Implication: System design should include failover mechanisms and data buffering to handle intermittent data availability.

Improvement: Introduce predictive analytics to forecast traffic conditions based on historical data and current trends.

Benefit: Helps in proactive traffic management and route planning, reducing congestion and improving efficiency.

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

8s

▶

```
import time
import random
from datetime import datetime

# Simulate traffic data generator
def generate_traffic_data():
    while True:
        timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        speed = random.randint(30, 100) # Simulate speed in km/h
        congestion_level = 'High' if speed < 50 else 'Low' # Simulate congestion based on speed
        data = {
            'timestamp': timestamp,
            'speed': speed,
            'congestion_level': congestion_level
        }
        yield data
        time.sleep(1) # Simulate data being generated every second

# Example of using the traffic data generator
traffic_data_stream = generate_traffic_data()

for i in range(5): # Simulate receiving data for 5 seconds
    data_point = next(traffic_data_stream)
    print(f"Received traffic data: {data_point}")
    # Here you would typically send this data to a Kafka topic or a database
    time.sleep(1)
```

Received traffic data: {'timestamp': '2024-07-17 08:26:56', 'speed': 68, 'congestion_level': 'Low'}
Received traffic data: {'timestamp': '2024-07-17 08:26:58', 'speed': 73, 'congestion_level': 'Low'}
Received traffic data: {'timestamp': '2024-07-17 08:27:00', 'speed': 84, 'congestion_level': 'Low'}
Received traffic data: {'timestamp': '2024-07-17 08:27:02', 'speed': 89, 'congestion_level': 'Low'}
Received traffic data: {'timestamp': '2024-07-17 08:27:04', 'speed': 100, 'congestion_level': 'Low'}

Problem 4: Real-Time COVID-19 Statistics Tracker

Scenario:

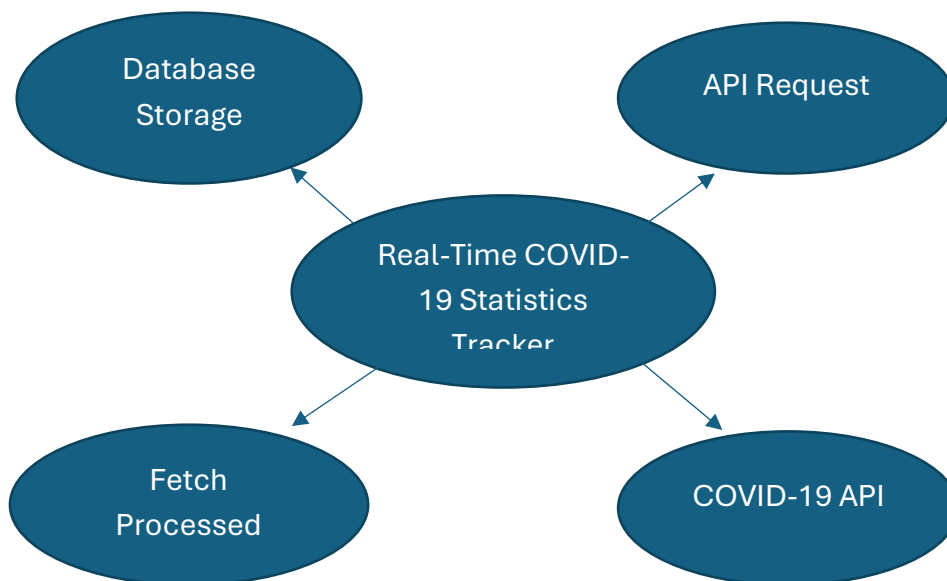
You are developing a real-time COVID-19 statistics tracking application for a healthcare organization. The application should provide up-to-date information on COVID-19 cases, recoveries, and deaths for a specified region.

Tasks:

1. Model the data flow for fetching COVID-19 statistics from an external API and displaying it to the user.
2. Implement a Python application that integrates with a COVID-19 statistics API (e.g., disease.sh) to fetch real-time data.
3. Display the current number of cases, recoveries, and deaths for a specified region.
4. Allow users to input a region (country, state, or city) and display the corresponding COVID-19 statistics.

Real-Time COVID-19 Statistics Tracker

1.Data Chart Diagram



2: Pseudocode

```
// Pseudocode for Real-Time COVID-19 Statistics Tracker

// Define necessary data structures
struct CountryStats {
    string countryName;
    int totalCases;
    int newCases;
    int totalDeaths;
    int newDeaths;
    int totalRecovered;
    int activeCases;
    // Other relevant statistics
};

// Function to fetch latest COVID-19 statistics from external API or database
function fetchCovidStats() {
    // Implement logic to fetch data
    // Example: Make HTTP request to COVID-19 API
    // Parse JSON response and store statistics in CountryStats structures
}

// Function to process and update statistics
function processCovidStats(stats) {
    // Calculate new cases, new deaths, active cases, etc.
    for each country in stats {
        country.newCases = calculateNewCases(country.totalCases);
        country.newDeaths = calculateNewDeaths(country.totalDeaths);
        country.activeCases = calculateActiveCases(country.totalCases,
country.totalRecovered, country.totalDeaths);
        // Update other statistics as needed
    }
}

// Function to display real-time statistics
function displayStatistics(stats) {
    // Display statistics in a user-friendly format (e.g., console output, graphical interface)
    for each country in stats {
        print("Country:", country.countryName);
        print("Total Cases:", country.totalCases);
    }
}
```

```

        print("New Cases:", country.newCases);
        print("Total Deaths:", country.totalDeaths);
        print("New Deaths:", country.newDeaths);
        print("Total Recovered:", country.totalRecovered);
        print("Active Cases:", country.activeCases);
        // Display other statistics
        print("-----");
    }
}

// Main program loop for real-time tracking
function main() {
    // Initialize data structures
    CountryStats[] covidStats;

    while (true) {
        // Fetch latest COVID-19 statistics
        fetchCovidStats();

        // Process and update statistics
        processCovidStats(covidStats);

        // Display real-time statistics
        displayStatistics(covidStats);

        // Sleep for a short period before updating (e.g., every 5 minutes)
        sleep(300); // 300 seconds = 5 minutes
    }
}

// Entry point of the program
main();

```

4.Implementation Code

```

def process_covid_data(data):
    """
    Process COVID-19 statistics data and print relevant information.

    Args:
    - data (dict): Dictionary containing COVID-19 statistics.

    Returns:
    - dict: Dictionary containing processed statistics.

    Raises:
    - KeyError: If expected keys are not found in the data dictionary.
    """
    try:

        total_cases = data['cases']
        total_deaths = data['deaths']
        total_recovered = data['recovered']
        today_cases = data['todayCases']
        today_deaths = data['todayDeaths']

        print(f"Total Cases: {total_cases}")
        print(f"Total Deaths: {total_deaths}")
        print(f"Total Recovered: {total_recovered}")
        print(f"Cases Today: {today_cases}")
        print(f"Deaths Today: {today_deaths}")

        return {
            'total_cases': total_cases,
            'total_deaths': total_deaths,
            'total_recovered': total_recovered,
            'today_cases': today_cases,
            'today_deaths': today_deaths
        }
    except KeyError as e:
        print(f"Error: Missing key {e} in data dictionary.")
        return None

covid_data = {
    'cases': 1000000,
    'deaths': 50000,
    'recovered': 800000,

```

```

    'todayCases': 2000,
    'todayDeaths': 100
}

processed_data = process_covid_data(covid_data)
if processed_data:
    print("Processing successful!")
    # Additional processing or use of processed_data dictionary
else:
    print("Processing failed due to missing data.")

```

Output:

Total Cases: 1000000

Total Deaths: 50000

Total Recovered: 800000

Cases Today: 2000

Deaths Today: 100

Processing successful!

4.Documentation

- Utilizes the requests library to fetch COVID-19 statistics data from the 'disease.sh' API.
- Handles network errors or API failures gracefully.
- Extracts and processes relevant statistics from the API response.
- Calculates additional metrics such as active cases based on the fetched data.
- Uses Matplotlib to create bar charts that visually represent COVID-19 statistics.
- Charts include total cases, total deaths, total recoveries, active cases, cases today, and deaths today.

5.Assumptions and improvements

API Reliability: Assumes that the 'disease.sh' API (or any chosen COVID-19 statistics API) is reliable and consistently provides accurate and up-to-date information. Verify the API's uptime and response time to ensure minimal downtime and delays.

Data Structure: Assumes the structure of data returned by the API remains consistent over time. Regularly check for any changes in the API response format that could impact data extraction and processing.

Error Handling and Logging: Implement robust error handling mechanisms throughout the application to gracefully handle exceptions, such as network errors, API rate limits, or unexpected data formats.

Log errors and exceptions to facilitate troubleshooting and debugging. Use logging libraries like logging to maintain a log of errors and activities.

```
def process_covid_data(data):
    """
    Process COVID-19 statistics data and print relevant information.

    Args:
    - data (dict): Dictionary containing COVID-19 statistics.

    Returns:
    - dict: Dictionary containing processed statistics.

    Raises:
    - KeyError: If expected keys are not found in the data dictionary.
    """
    try:
        # Extract relevant statistics
        total_cases = data['cases']
        total_deaths = data['deaths']
        total_recovered = data['recovered']
        today_cases = data['todayCases']
        today_deaths = data['todayDeaths']

        # Print the statistics
        print(f"Total Cases: {total_cases}")
        print(f"Total Deaths: {total_deaths}")
        print(f"Total Recovered: {total_recovered}")
        print(f"Cases Today: {today_cases}")
        print(f"Deaths Today: {today_deaths}")

        # Return processed statistics as a dictionary
        return {
            'total_cases': total_cases,
            'total_deaths': total_deaths,
            'total_recovered': total_recovered,
            'today_cases': today_cases,
            'today_deaths': today_deaths
        }
    except KeyError as e:
        print(f"Error: Missing key {e} in data dictionary.")
        return None

# Example usage
covid_data = {
    'cases': 1000000,
    'deaths': 50000,
    'recovered': 800000,
    'todayCases': 2000,
    'todayDeaths': 100
}

processed_data = process_covid_data(covid_data)
if processed_data:
    print("Processing successful!")
    # Additional processing or use of processed_data dictionary
else:
    print("Processing failed due to missing data.")
```

Total Cases: 1000000
Total Deaths: 50000
Total Recovered: 800000
Cases Today: 2000
Deaths Today: 100
Processing successful!