Application 1 : Rule Engine with AST

To implement this 3-tier rule engine application with an Abstract Syntax Tree (AST), let's break down each section, including the data structure, data storage, and API design.

1. Data Structure

We will define a Node class that represents the nodes of the AST. This data structure should support dynamic changes to rules and can handle both operators (AND/OR) and conditions.

```
class Node:
    def __init__(self, type, operator=None, left=None, right=None, attribute=None,
comparator=None, value=None):
        """
        Represents a node in the AST.


        :param type: String indicating the node type ("operator" or "operand").
        :param operator: String for logical operators ("AND" or "OR") if the node type is
"operator".
        :param left: Reference to the left child Node (applicable for "operator" nodes).
        :param right: Reference to the right child Node (applicable for "operator" nodes).
        :param attribute: String representing the attribute name (e.g., "age", "department") for
operand nodes.
        :param comparator: String representing the comparison operator (e.g., ">", "==") for
operand nodes.
        :param value: The value to be compared against the attribute for operand nodes.
        """
        self.type = type
        self.operator = operator    # Only for "operator" nodes ("AND"/"OR")
        self.left = left            # Reference to the left child node (for operators)
        self.right = right          # Reference to the right child node (for operators)
        self.attribute = attribute  # Only for "operand" nodes (e.g., "age")
        self.comparator = comparator  # Comparison operator (e.g., ">", "<", "==")
        self.value = value          # The value for comparison (e.g., 30, "Sales")
```

2. Data Storage

Choice of Database

We can use a **NoSQL database like MongoDB** to store rules and application metadata. MongoDB provides flexibility in storing nested data structures like an AST, and JSON-like documents can be easily mapped to the Node structure.

Alternatively, a **relational database like PostgreSQL** can be used with JSONB fields to store ASTs in JSON format.

Database Schema

Here's a sample schema definition for MongoDB:

1. **Rules Collection**

```
{
  "_id": "rule1",
  "name": "Rule 1",
  "description": "Eligibility rule for Sales and Marketing",
  "ast": {
    "type": "operator",
    "operator": "AND",
    "left": {
      "type": "operator",
      "operator": "OR",
      "left": {
        "type": "operator",
        "operator": "AND",
        "left": {
          "type": "operand",
          "attribute": "age",
          "comparator": ">",
          "value": 30
        },
        "right": {
          "type": "operand",
            "attribute": "department",
            "comparator": "==",
            "value": "Sales"
        }
    },
    "right": {
        "type": "operator",
        "operator": "AND",
        "left": {
            "type": "operand",
            "attribute": "age",
            "comparator": "<",
            "value": 25
        },
        "right": {
            "type": "operand",
            "attribute": "department",
            "comparator": "==",
            "value": "Marketing"
        }
    }
},
"right": {
    "type": "operator",
    "operator": "OR",
    "left": {
        "type": "operand",
        "attribute": "salary",
        "comparator": ">",
            "value": 50000
          },
          "right": {
            "type": "operand",
            "attribute": "experience",
            "comparator": ">",
            "value": 5
          }
        }
      }
    }
```

2. **Application Metadata Collection**

```
{
  "_id": "app_meta",
  "last_updated": "2024-10-18T00:00:00Z",
  "rule_count": 2,
  "user_count": 100
}
```

3. **API Design**
   - **create_rule Function**

```
def create_rule(rule_string):
    # Parse the rule string to construct the AST
    # This could involve tokenizing the string and converting it into an AST
    # For simplicity, we will mock this implementation

    # Example hardcoded AST creation for the rule string
    age_condition = Node(type="operand", attribute="age", comparator=">",
value=30)
    department_condition = Node(type="operand", attribute="department",
comparator="==", value="Sales")
    and_node = Node(type="operator", operator="AND", left=age_condition,
right=department_condition)

    return and_node
```

   - **combine_rules Function**

```
def evaluate_rule(ast, data):
   if ast.type == "operand":
      # Evaluate the condition based on the data
      attribute_value = data.get(ast.attribute)
      if ast.comparator == ">":
         return attribute_value > ast.value
      elif ast.comparator == "<":
         return attribute_value < ast.value
      elif ast.comparator == "==":
         return attribute_value == ast.value
      # Add other comparison operations as needed

   elif ast.type == "operator":
      left_result = evaluate_rule(ast.left, data)
      right_result = evaluate_rule(ast.right, data)
      if ast.operator == "AND":
         return left_result and right_result
      elif ast.operator == "OR":
         return left_result or right_result

   return False
```

To perform comprehensive testing of the rule engine and implement bonus functionalities, we'll work through the given test cases step-by-step. Let's cover each of the cases and discuss additional requirements for error handling, validations, and modifications.

Test Case 1: Creating Individual Rules

We will use the create_rule function to generate ASTs for given rules and verify the structure.

```
# Test case 1 - Creating rules

rule1_ast = create_rule("((age > 30 AND department = 'Sales') OR (age < 25 AND department = 'Marketing')) AND (salary > 50000 OR experience > 5)")
```

```
rule2_ast = create_rule("((age > 30 AND department = 'Marketing')) AND (salary
> 20000 OR experience > 5)")


# Function to print the AST in a readable format for verification

def print_ast(node, level=0):

    indent = "  " * level

    if node.type == "operand":

        print(f"{indent}Operand: {node.attribute} {node.comparator} {node.value}")

    elif node.type == "operator":

        print(f"{indent}Operator: {node.operator}")

        if node.left:

            print_ast(node.left, level + 1)

        if node.right:

            print_ast(node.right, level + 1)


# Verifying the AST representation

print("Rule 1 AST:")

print_ast(rule1_ast)


print("\nRule 2 AST:")

print_ast(rule2_ast)
```

Test Case 2: Combining Rules

We will use the combine_rules function to merge rule1 and rule2 and verify if the resulting AST represents the correct combined logic.

```
# Test case 2 - Combining rules

combined_ast = combine_rules(["((age > 30 AND department = 'Sales') OR (age <
25 AND department = 'Marketing')) AND (salary > 50000 OR experience > 5)",

                              "((age > 30 AND department = 'Marketing')) AND (salary > 20000
OR experience > 5)"])


# Verifying the combined AST representation
```

```
print("\nCombined AST:")

print_ast(combined_ast)
```

Test Case 3: Evaluating Rules

We will create some sample JSON data and test evaluate_rule to ensure the rule evaluation logic works as expected.

```
# Test case 3 - Evaluating rules

data1 = {"age": 35, "department": "Sales", "salary": 60000, "experience": 3}

data2 = {"age": 22, "department": "Marketing", "salary": 45000, "experience": 6}

data3 = {"age": 40, "department": "Engineering", "salary": 30000, "experience": 10}


# Evaluating rule1 against the data samples

print("\nEvaluating Rule 1:")

print(f"Data 1 result: {evaluate_rule(rule1_ast, data1)}") # Expected True

print(f"Data 2 result: {evaluate_rule(rule1_ast, data2)}") # Expected True

print(f"Data 3 result: {evaluate_rule(rule1_ast, data3)}") # Expected False


# Evaluating rule2 against the data samples

print("\nEvaluating Rule 2:")

print(f"Data 1 result: {evaluate_rule(rule2_ast, data1)}") # Expected False

print(f"Data 2 result: {evaluate_rule(rule2_ast, data2)}") # Expected True

print(f"Data 3 result: {evaluate_rule(rule2_ast, data3)}") # Expected False
```

Test Case 4: Combining Additional Rules

We will create new rules and test the combination to verify flexibility and correctness.

```
# Test case 4 - Combining additional rules

additional_combined_ast = combine_rules([

    "((age > 30 AND department = 'Sales') OR (age < 25 AND department = 'Marketing')) AND (salary > 50000 OR experience > 5)",

    "(spend > 1000 AND age > 50)",

    "(experience < 3 AND department = 'HR')"
])

# Verifying the combined AST representation with additional rules
print("\nCombined AST with additional rules:")
print_ast(additional_combined_ast)
```

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

To develop a real-time data processing system for weather monitoring, we'll focus on key requirements and functionalities. Here is a detailed breakdown of the solution:

Objective

The goal is to create a system that continuously retrieves weather data, processes it in real time, and provides summarized insights using rollups and aggregates.

Data Source

- **OpenWeatherMap API:** We'll use the OpenWeatherMap API to fetch weather data. The key parameters of interest will include:
  - Temperature
  - Humidity
  - Wind speed
  - Atmospheric pressure
  - Weather conditions (e.g., clear, cloudy, rainy)

Implementation Details

1. Data Retrieval

- **Weather Data API Integration**: Use the OpenWeatherMap API to get the current weather data for multiple cities.
- **Sample API Call**:

  https://api.openweathermap.org/data/2.5/weather?q=CityName&appid=YourAPIKey&units=metric
- **Data Retrieval Logic**:
  - Create a scheduler to fetch data at regular intervals.
  - Parse the JSON response to extract temperature, humidity, wind speed, etc.
  - Handle errors such as invalid API key, rate limit exceeded, or network issues.

2. Data Processing

- **Data Parsing**: Extract relevant data fields from the JSON response.
- **Rollup Aggregation**:
  - Compute hourly, daily, and weekly rollups.
  - Aggregate metrics like average temperature, maximum wind speed, etc.
- **Alert Generation**:
  - Define configurable thresholds for alerts.
  - Monitor real-time data against these thresholds and trigger notifications.

3. Data Storage

- **Database Choice**: Use a database suitable for time-series data, such as InfluxDB, or a traditional RDBMS like PostgreSQL for structured storage.
- **Schema Design**:
  - Weather data table: Store raw weather parameters (city, temperature, humidity, timestamp).
  - Aggregates table: Store computed rollups (average temperature per hour, max wind speed per day).
  - Alerts table: Log alert events with details (city, parameter, threshold exceeded).

4. Data Visualization

- **Visualization Tools**: Use tools like Grafana, Tableau, or a custom web dashboard to display weather trends.
- **Dashboards**:
  - Show graphs for temperature, humidity, and wind speed trends.
  - Include alerts history and current weather conditions.

API Design

1. **get_weather_data(city, start_time, end_time)**: Retrieve weather data for a given city within a time range.
2. **get_aggregates(city, time_period)**: Fetch rollup data for a

To build the real-time data processing system for weather monitoring using the specified weather parameters (main, temp, feels_like, and dt), here's how we can structure the solution:

Step 1: Data Retrieval

Requirements:

- Continuously fetch weather data from the OpenWeatherMap API.

- Focus on the specified parameters: main, temp, feels_like, and dt.

  API Call Example: To get the current weather data for a city, we can use:

  https://api.openweathermap.org/data/2.5/weather?q=CityName&appid=YourAPIKey&units=metric

- Replace CityName with the city you want to monitor.
- YourAPIKey should be replaced with the API key obtained after signing up for OpenWeatherMap.
- Setting units=metric ensures the temperature is in Celsius.

  Response Example: The API will return a JSON response similar to:

```
{
  "weather": [
    {
      "main": "Clear",
      "description": "clear sky"
    }
  ],
  "main": {
    "temp": 26.3,
    "feels_like": 27.1
  },
  "dt": 1605182400,
  "name": "CityName"
}
```

Step 2: Data Processing

Extract Required Data:

- Parse the JSON response to extract:
  - main: From weather[0].main, representing the main weather condition.
  - temp: From main.temp, representing the current temperature in Celsius.
  - feels_like: From main.feels_like, representing the perceived temperature.
  - dt: The Unix timestamp of the data update.

Data Aggregation and Rollups:

- **Hourly Rollup**: Calculate average temperature, average perceived temperature, and most common weather condition.
- **Daily Rollup**: Aggregate the above metrics for the entire day.

Alert Generation:

- Define thresholds for conditions such as:
    - High temperature (e.g., temp > 35°C).
    - Sudden weather changes (e.g., change from Clear to Rain).
- Trigger alerts if any threshold is exceeded.

Step 3: Data Storage

Database Choice:

- **Option 1**: Use a time-series database (e.g., InfluxDB or TimescaleDB) for efficient storage and querying of time-based data.
- **Option 2**: Use a relational database (e.g., PostgreSQL) if more structured data storage is needed.

Schema Design:

- **Weather Data Table**: Store raw weather data.

```
CREATE TABLE weather_data (
   id SERIAL PRIMARY KEY,
   city_name VARCHAR(50),
   main_condition VARCHAR(50),
   temperature FLOAT,
   feels_like FLOAT,
   data_timestamp BIGINT
);
```

- **Aggregates Table**: Store hourly and daily rollups.

```
CREATE TABLE weather_aggregates (
   id SERIAL PRIMARY KEY,
   city_name VARCHAR(50),
   time_period VARCHAR(10),  -- 'hourly' or 'daily'
   start_time BIGINT,
   end_time BIGINT,
   avg_temperature FLOAT,
   avg_feels_like FLOAT,
   most_common_condition VARCHAR(50)
);
```

- **Alerts Table**: Log alerts triggered by threshold violations.

```
CREATE TABLE alerts (
   id SERIAL PRIMARY KEY,
   city_name VARCHAR(50),
   alert_type VARCHAR(50),
   alert_message TEXT,
   alert_timestamp BIGINT
);
```

Step 4: Data Visualization

Dashboard and Visualizations:

- Use a dashboard tool like **Grafana**, **Tableau**, or **a custom-built web interface**.
- Display graphs for:
  - Temperature trends (actual vs. perceived).
  - Weather condition changes over time.
  - Alerts and notifications for extreme weather events.

Step 5: API Design

- **getWeatherData(cityName)**: Retrieve the latest weather data for the specified city.
- **getHourlyRollups(cityName, startTime, endTime)**: Retrieve hourly aggregated weather data.
- **getDailyRollups(cityName, startTime, endTime)**: Retrieve daily aggregated weather data.
- **getAlerts(cityName, startTime, endTime)**: Retrieve alerts logged for a specified time range.

To implement the processing and analysis requirements for continuously retrieving weather data from the OpenWeatherMap API and converting temperature values, follow these steps:

Step 1: Continuous Data Retrieval

1. **Configurable Interval**:
   - Set up a scheduler to call the OpenWeatherMap API at a configurable interval (e.g., every 5 minutes).
   - You can use Java's ScheduledExecutorService to achieve this.
2. **Cities to Monitor**:
   - Retrieve weather data for major metro cities in India: Delhi, Mumbai, Chennai, Bangalore, Kolkata, and Hyderabad.
   - Create an array of these city names and iterate over them to fetch weather data.

Step 2: Convert Temperature Values from Kelvin to Celsius

1. **Conversion Formula**:
   - The temperature from the OpenWeatherMap API is provided in Kelvin by default, unless specified otherwise.
   - To convert from Kelvin to Celsius: Celsius=Kelvin−273.15\text{Celsius} = \text{Kelvin} - 273.15Celsius=Kelvin−273.15
2. **User Preference**:
   - Allow the user to specify their preferred temperature unit (Kelvin, Celsius, or Fahrenheit).
   - Convert temperatures accordingly.

Java Code

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class WeatherMonitoringSystem {
   private static final String API_KEY = "YourAPIKey"; // Replace with your actual API
key
   private static final String BASE_URL =
"https://api.openweathermap.org/data/2.5/weather";
   private static final String[] CITIES = {"Delhi", "Mumbai", "Chennai", "Bangalore",
"Kolkata", "Hyderabad"};
   private static final int INTERVAL_MINUTES = 5; // Configurable interval in minutes

   public static void main(String[] args) {
      // Schedule the weather data retrieval task at a fixed interval
      ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

scheduler.scheduleAtFixedRate(WeatherMonitoringSystem::fetchAndProcessWeathe
rData, 0, INTERVAL_MINUTES, TimeUnit.MINUTES);
   }

   public static void fetchAndProcessWeatherData() {
      for (String city : CITIES) {
         String response = getWeatherData(city);
         if (response != null) {
            processWeatherData(response, "Celsius"); // User preference for
temperature unit
         } else {
            System.out.println("Failed to retrieve weather data for " + city);
         }
      }
   }

   public static String getWeatherData(String cityName) {
      try {
         // Construct the full API URL
         String urlString = BASE_URL + "?q=" + cityName + "&appid=" + API_KEY;
         URL url = new URL(urlString);

         // Open a connection to the URL
         HttpURLConnection connection = (HttpURLConnection) url.openConnection();
         connection.setRequestMethod("GET");

         // Check the response code
         int responseCode = connection.getResponseCode();
         if (responseCode == 200) { // HTTP OK
            // Read the response
            BufferedReader in = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
```

```
        String inputLine;
        StringBuilder response = new StringBuilder();

        while ((inputLine = in.readLine()) != null) {
          response.append(inputLine);
        }
        in.close();

        // Return the response as a string
        return response.toString();
      } else {
        System.out.println("GET request failed. Response Code: " + responseCode);
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
    return null;
  }

  public static void processWeatherData(String jsonResponse, String
temperatureUnit) {
    try {
      // Parse the JSON response (using org.json library or any other JSON parsing
library)
      org.json.JSONObject jsonObject = new org.json.JSONObject(jsonResponse);
      double tempInKelvin =
jsonObject.getJSONObject("main").getDouble("temp");
      double feelsLikeInKelvin =
jsonObject.getJSONObject("main").getDouble("feels_like");
      String weatherCondition =
jsonObject.getJSONArray("weather").getJSONObject(0).getString("main");

      // Convert temperatures based on user preference
      double temp = convertTemperature(tempInKelvin, temperatureUnit);
      double feelsLike = convertTemperature(feelsLikeInKelvin, temperatureUnit);

      // Output the processed weather data
      System.out.println("Weather Condition: " + weatherCondition);
      System.out.println("Temperature: " + temp + "° " + temperatureUnit);
      System.out.println("Feels Like: " + feelsLike + "° " + temperatureUnit);
      System.out.println("-----------------------------------");

    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static double convertTemperature(double tempInKelvin, String unit) {
    switch (unit.toLowerCase()) {
      case "celsius":
        return tempInKelvin - 273.15;
      case "fahrenheit":
        return (tempInKelvin - 273.15) * 9/5 + 32;
      default: // Kelvin
        return tempInKelvin;
    }
  }
}
```

To implement rollups, aggregates, alerting thresholds, and visualizations for the weather monitoring system, the following approach can be used:

Step 1: Daily Weather Summary

1. **Roll Up Weather Data for Each Day**
   - Store individual weather updates throughout the day in a database or in-memory structure.
   - Aggregate these updates at the end of the day or periodically to generate daily summaries.
2. **Daily Aggregates**
   - **Average Temperature**: Compute the average of all temperature readings recorded for the day.
   - **Maximum Temperature**: Track the highest temperature recorded during the day.
   - **Minimum Temperature**: Track the lowest temperature recorded during the day.
   - **Dominant Weather Condition**:
     - Determine the most frequent weather condition of the day (e.g., "Clear", "Rain", "Cloudy").
     - If there's a tie, consider factors such as duration or severity of the conditions.
3. **Storing Daily Summaries**
   - Use a database (e.g., SQLite, MySQL, or NoSQL databases like MongoDB) to store the daily weather summaries.
   - Schema example:

```
CREATE TABLE DailyWeatherSummary (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  city VARCHAR(50),
  date DATE,
  avg_temperature DOUBLE,
  max_temperature DOUBLE,
  min_temperature DOUBLE,
  dominant_condition VARCHAR(50)
);
```

Step 2: Alerting Thresholds

1. **Configurable Thresholds**
   - Allow the user to define thresholds for temperature or weather conditions (e.g., temperature above 35°C for two consecutive updates).
   - Store these thresholds in a configuration file or database table.
2. **Tracking Weather Data Against Thresholds**
   - Continuously compare the latest weather data with the configured thresholds.

    - If a threshold is breached, log the alert or trigger an action.

3. **Alert Handling**
    - Alerts can be displayed on the console, stored in a database, or sent as email notifications.
    - Example alert schema:

```
CREATE TABLE WeatherAlerts (
   id INTEGER PRIMARY KEY AUTOINCREMENT,
   city VARCHAR(50),
   alert_message TEXT,
   alert_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 3: Visualizations

1. **Displaying Daily Weather Summaries and Historical Trends**
    - Use a visualization library like **JavaFX**, **JFreeChart** for desktop applications, or **Chart.js**, **D3.js** for web-based visualizations.
    - Plot data such as daily average temperature, max/min temperature, and weather conditions over time.
2. **Displaying Alerts**
    - Show alerts in a separate section of the visualization (e.g., as a list or notification pop-up).
    - Include details such as the city, condition that triggered the alert, and the time.
3. **Java Implementation Example**
    1. **Rollup Weather Data and Calculate Aggregates**

```
import java.util.*;



class WeatherData {

    String city;

    double temperature;

    String condition;

    Date timestamp;



    public WeatherData(String city, double temperature, String condition, Date timestamp) {

        this.city = city;

        this.temperature = temperature;

        this.condition = condition;

        this.timestamp = timestamp;
    }
}


class DailySummary {
    String city;
    double avgTemperature;
    double maxTemperature;
    double minTemperature;
    String dominantCondition;


    public DailySummary(String city, double avgTemperature, double
maxTemperature, double minTemperature, String dominantCondition) {
        this.city = city;
        this.avgTemperature = avgTemperature;
        this.maxTemperature = maxTemperature;
        this.minTemperature = minTemperature;
        this.dominantCondition = dominantCondition;
    }
}


public class WeatherRollup {
    public static DailySummary calculateDailySummary(List<WeatherData>
dailyWeatherData) {
        double totalTemp = 0;
        double maxTemp = Double.MIN_VALUE;
        double minTemp = Double.MAX_VALUE;
        Map<String, Integer> conditionCount = new HashMap<>();
        for (WeatherData data : dailyWeatherData) {

            // Update temperature totals

            totalTemp += data.temperature;

            maxTemp = Math.max(maxTemp, data.temperature);

            minTemp = Math.min(minTemp, data.temperature);



            // Count occurrences of weather conditions

            conditionCount.put(data.condition,
conditionCount.getOrDefault(data.condition, 0) + 1);

        }



        // Calculate averages

        double avgTemp = totalTemp / dailyWeatherData.size();



        // Determine the dominant weather condition

        String dominantCondition = conditionCount.entrySet().stream()

                .max(Map.Entry.comparingByValue())

                .get()

                .getKey();



        // Create the daily summary

        return new DailySummary(dailyWeatherData.get(0).city, avgTemp,
maxTemp, minTemp, dominantCondition);

    }

}
```

2. **Alerting Mechanism**

```
public class WeatherAlert {
    private static final double TEMPERATURE_THRESHOLD = 35.0;
    private static int consecutiveBreaches = 0;
    private static final int REQUIRED_CONSECUTIVE_BREACHES = 2;

    public static void checkForAlerts(WeatherData data) {
        if (data.temperature > TEMPERATURE_THRESHOLD) {
            consecutiveBreaches++;
            if (consecutiveBreaches >= REQUIRED_CONSECUTIVE_BREACHES) {
                triggerAlert(data.city, "Temperature exceeded " +
TEMPERATURE_THRESHOLD + "°C for " +
REQUIRED_CONSECUTIVE_BREACHES + " consecutive updates.");
            }
        } else {
            consecutiveBreaches = 0; // Reset the counter if threshold is not
breached
        }
    }

    private static void triggerAlert(String city, String message) {
        System.out.println("ALERT: " + city + " - " + message);
        // Additional code to log the alert or send notifications can be added
here
    }
}
```

Step 4: Visualization Approach

1. **JavaFX Example for Line Chart Visualization**:
   - JavaFX can be used to create a line chart displaying the daily average temperature over time.
2. **Web-Based Visualization Using Chart.js**:
   - Use a web server (e.g., Spring Boot) to serve a web page containing a Chart.js visualization that displays daily weather trends.

To verify the setup and functionality of the weather monitoring system, we can develop a series of test cases that cover each of the specified requirements. Below is a structured approach to implement the tests for each component of the system.

Step 1: System Setup Verification

Test Case: System Initialization

- **Objective**: Verify that the system starts successfully and connects to the OpenWeatherMap API using a valid API key.
- **Implementation**:
  - Create a test class to initialize the weather monitoring system.
  - Attempt to fetch weather data using the API key.
- **Expected Result**: The system should not throw any exceptions, and a successful connection message should be logged.

```
public class SystemSetupTest {
  public static void main(String[] args) {
    String apiKey = "YourAPIKey"; // Replace with your actual API key
    WeatherMonitoringSystem weatherSystem = new
WeatherMonitoringSystem(apiKey);
    boolean isConnected = weatherSystem.testConnection();
    assert isConnected : "Failed to connect to OpenWeatherMap API";
    System.out.println("System setup verified: Successfully connected to
OpenWeatherMap API.");
  }
}
```

Step 2: Data Retrieval Testing

Test Case: Simulate API Calls

- **Objective**: Ensure the system retrieves weather data for the specified location and parses the response correctly.
- **Implementation**:
  - Mock the API response for specified locations.
  - Verify that the weather data is parsed correctly.
- **Expected Result**: Parsed weather data should match the mocked response.

```
public class DataRetrievalTest {
  public static void main(String[] args) {
    WeatherMonitoringSystem weatherSystem = new
WeatherMonitoringSystem("YourAPIKey");
    String mockedResponse = "{...}"; // Mocked JSON response
    weatherSystem.setMockedResponse(mockedResponse); // Method to
set a mocked response for testing
    weatherSystem.fetchAndProcessWeatherData(); // Simulate fetching
weather data
    // Assert conditions for expected data
    WeatherData data = weatherSystem.getLatestWeatherData(); //
Assuming a method to get the latest data
    assert data != null : "Weather data should not be null";
    System.out.println("Data retrieval verified: Weather data parsed
correctly.");
  }
}
```

Step 3: Temperature Conversion Testing

Test Case: Temperature Conversion

- **Objective**: Test conversion of temperature values from Kelvin to Celsius or Fahrenheit based on user preference.
- **Implementation**:
  - Test both conversion methods and verify the output.
- **Expected Result**: The converted temperatures should match the expected values.

```
public class TemperatureConversionTest {
  public static void main(String[] args) {
    double kelvin = 300.0; // Example Kelvin value
    double expectedCelsius = 26.85; // Expected Celsius value
    double expectedFahrenheit = 80.33; // Expected Fahrenheit value
```

```
        double celsius = WeatherMonitoringSystem.convertTemperature(kelvin,
"Celsius");
        double fahrenheit =
WeatherMonitoringSystem.convertTemperature(kelvin, "Fahrenheit");

        assert Math.abs(celsius - expectedCelsius) < 0.01 : "Celsius conversion
failed.";
        assert Math.abs(fahrenheit - expectedFahrenheit) < 0.01 : "Fahrenheit
conversion failed.";

        System.out.println("Temperature conversion verified: Celsius and
Fahrenheit values are correct.");
    }
}
```

Step 4: Daily Weather Summary Testing

Test Case: Calculate Daily Weather Summary

- **Objective**: Simulate a sequence of weather updates and verify that daily summaries are calculated correctly.
- **Implementation**:
    - Create a list of weather data for several days.
    - Call the summary calculation method and check the results.
- **Expected Result**: The daily summary should reflect accurate average, maximum, minimum temperatures, and the dominant weather condition.

```
public class DailyWeatherSummaryTest {
    public static void main(String[] args) {
        List<WeatherData> weatherDataList = new ArrayList<>();
        // Simulate weather updates over several days
        weatherDataList.add(new WeatherData("Delhi", 30.0, "Clear", new
Date()));
        weatherDataList.add(new WeatherData("Delhi", 32.0, "Cloudy", new
Date()));
        weatherDataList.add(new WeatherData("Delhi", 28.0, "Rain", new
Date()));

        DailySummary summary =
WeatherRollup.calculateDailySummary(weatherDataList);

        assert Math.abs(summary.avgTemperature - 30.0) < 0.01 : "Average
temperature calculation failed.";
        assert summary.maxTemperature == 32.0 : "Maximum temperature
calculation failed.";
        assert summary.minTemperature == 28.0 : "Minimum temperature
calculation failed.";
        assert summary.dominantCondition.equals("Clear") : "Dominant
condition calculation failed.";

        System.out.println("Daily weather summary verified: Summary values
are correct.");
```

```
    }
}
```

Step 5: Alerting Thresholds Testing

Test Case: Threshold Alerts

- **Objective**: Define user thresholds and verify that alerts are triggered only when a threshold is violated.
- **Implementation**:
    - Set a threshold for temperature.
    - Simulate weather data exceeding the threshold.
    - Verify that the alert is triggered correctly.
- **Expected Result**: Alerts should be logged or triggered only when the conditions are met.

```
public class AlertingThresholdTest {
    public static void main(String[] args) {
        WeatherData data1 = new WeatherData("Delhi", 34.0, "Clear", new
Date());
        WeatherData data2 = new WeatherData("Delhi", 36.0, "Clear", new
Date());

        WeatherAlert.checkForAlerts(data1); // Should not trigger alert
        WeatherAlert.checkForAlerts(data2); // Should trigger alert

        // Validate that alert is triggered
        // Assuming we have a way to check if an alert was logged
        assert WeatherAlert.getAlertCount() == 1 : "Alert should have been
triggered.";

        System.out.println("Alerting thresholds verified: Alerts triggered
correctly.");
    }
}
```

To extend the weather monitoring system to support additional weather parameters from the OpenWeatherMap API, such as humidity and wind speed, and to incorporate functionalities like weather forecasts retrieval, we can follow these steps:

Step 1: Extend Data Model

1.1 Update WeatherData Class

Add new fields to the WeatherData class to include humidity and wind speed.

```
class WeatherData {
    String city;
    double temperature;
    double humidity; // New field
    double windSpeed; // New field
    String condition;
    Date timestamp;

    public WeatherData(String city, double temperature, double humidity,
double windSpeed, String condition, Date timestamp) {
        this.city = city;
        this.temperature = temperature;
        this.humidity = humidity;
        this.windSpeed = windSpeed;
        this.condition = condition;
        this.timestamp = timestamp;
    }
}
```

Step 2: Modify API Call to Retrieve Additional Parameters

2.1 Update API Call

Modify the API call to retrieve additional parameters (humidity and wind speed).

```
public WeatherData fetchWeatherData(String city) {
    String apiUrl =
String.format("https://api.openweathermap.org/data/2.5/weather?q=%s&a
ppid=%s&units=metric", city, apiKey);
    // Make API call and parse JSON response
    // Extract temperature, humidity, wind speed, and weather condition
    JSONObject jsonResponse = //... your JSON parsing logic
    double temperature =
jsonResponse.getJSONObject("main").getDouble("temp");
    double humidity =
jsonResponse.getJSONObject("main").getDouble("humidity");
    double windSpeed =
jsonResponse.getJSONObject("wind").getDouble("speed");
    String condition =
jsonResponse.getJSONArray("weather").getJSONObject(0).getString("main")
;

    return new WeatherData(city, temperature, humidity, windSpeed,
condition, new Date());
}
```

Step 3: Update Daily Weather Summary

3.1 Modify DailySummary Class

Update the DailySummary class to include aggregates for humidity and wind speed.

```
class DailySummary {
    String city;
    double avgTemperature;
    double maxTemperature;
    double minTemperature;
    double avgHumidity; // New field
    double maxWindSpeed; // New field
    String dominantCondition;

    public DailySummary(String city, double avgTemperature, double
maxTemperature, double minTemperature, double avgHumidity, double
maxWindSpeed, String dominantCondition) {
        this.city = city;
        this.avgTemperature = avgTemperature;
        this.maxTemperature = maxTemperature;
        this.minTemperature = minTemperature;
        this.avgHumidity = avgHumidity; // Initialize new field
        this.maxWindSpeed = maxWindSpeed; // Initialize new field
        this.dominantCondition = dominantCondition;
    }
}
```

3.2 Update Summary Calculation

In the WeatherRollup class, modify the calculateDailySummary method to compute aggregates for humidity and wind speed.

```
public static DailySummary calculateDailySummary(List<WeatherData>
dailyWeatherData) {
    double totalTemp = 0, totalHumidity = 0;
    double maxTemp = Double.MIN_VALUE, minTemp = Double.MAX_VALUE;
    double maxWindSpeed = Double.MIN_VALUE;
    Map<String, Integer> conditionCount = new HashMap<>();

    for (WeatherData data : dailyWeatherData) {
        totalTemp += data.temperature;
        totalHumidity += data.humidity; // Update humidity totals
        maxTemp = Math.max(maxTemp, data.temperature);
        minTemp = Math.min(minTemp, data.temperature);
        maxWindSpeed = Math.max(maxWindSpeed, data.windSpeed); //
Update wind speed

        conditionCount.put(data.condition,
conditionCount.getOrDefault(data.condition, 0) + 1);
    }

    double avgTemp = totalTemp / dailyWeatherData.size();
    double avgHumidity = totalHumidity / dailyWeatherData.size(); //
Calculate average humidity

    String dominantCondition = conditionCount.entrySet().stream()
            .max(Map.Entry.comparingByValue())
            .get()
            .getKey();

    return new DailySummary(dailyWeatherData.get(0).city, avgTemp,
maxTemp, minTemp, avgHumidity, maxWindSpeed, dominantCondition);
}
```

Step 4: Weather Forecasts Retrieval

To incorporate weather forecast retrieval and summaries based on predicted conditions, follow these steps:

4.1 API Call for Forecasts

Use the OpenWeatherMap 5-day/3-hour forecast API endpoint to fetch forecast data.

```
public List<WeatherData> fetchWeatherForecast(String city) {
    String apiUrl =
String.format("https://api.openweathermap.org/data/2.5/forecast?q=%s&a
ppid=%s&units=metric", city, apiKey);
    // Make API call and parse JSON response
    List<WeatherData> forecastDataList = new ArrayList<>();

    JSONObject jsonResponse = //... your JSON parsing logic
    JSONArray forecastList = jsonResponse.getJSONArray("list");

    for (int i = 0; i < forecastList.length(); i++) {
        JSONObject forecast = forecastList.getJSONObject(i);
        double temperature =
forecast.getJSONObject("main").getDouble("temp");
        double humidity =
forecast.getJSONObject("main").getDouble("humidity");
        double windSpeed =
forecast.getJSONObject("wind").getDouble("speed");
        String condition =
forecast.getJSONArray("weather").getJSONObject(0).getString("main");
        Date timestamp = new Date(forecast.getLong("dt") * 1000); // Convert
UNIX timestamp to Date

        forecastDataList.add(new WeatherData(city, temperature, humidity,
windSpeed, condition, timestamp));
    }

    return forecastDataList;
}
```

Step 5: Generate Summaries Based on Forecasted Conditions

5.1 Create Forecast Summaries

Create a method to summarize the forecasted weather data.

```
public DailySummary generateForecastSummary(List<WeatherData>
forecastData) {
    // Similar logic to calculateDailySummary but using forecastData
    return calculateDailySummary(forecastData);
}
```

Step 6: Integration into the System

6.1 Update Main Monitoring Loop

Integrate the new functionalities into the main loop of your weather monitoring system, ensuring that both real-time weather data and forecasts are retrieved and summarized.

```
public void runWeatherMonitoring() {
    while (true) {
        // Retrieve current weather data
        WeatherData currentData = fetchWeatherData("Delhi"); // Example for
Delhi
        // Store current data and calculate daily summaries

        // Retrieve weather forecast
        List<WeatherData> forecastData = fetchWeatherForecast("Delhi");
        DailySummary forecastSummary =
generateForecastSummary(forecastData);

        // Output summaries to console or store in database
        System.out.println("Forecast Summary: " + forecastSummary.toString());

        // Sleep for a configurable interval before next retrieval
        try {
            Thread.sleep(300000); // Sleep for 5 minutes
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```