# MySubscriber (Intermediate):

Write a script that: **Subscribes to the topic home/temperature** and prints any message it receives.

The script should **publish a message "28°C" to the same topic after 5 seconds of receiving a message**.

import paho.mqtt.client as PahoMQTT

import time

class MySubscriber():

    def \_\_init\_\_(self, clientID, topic, broker):

        self.clientID = clientID

        self.topic = topic

        self.messageBroker = broker

        self.\_paho\_mqtt = PahoMQTT.Client(clientID, True)

        self.\_paho\_mqtt.on\_connect = self.myOnConnect

        self.\_paho\_mqtt.on\_message = self.myOnMessage

    def start(self):

        self.\_paho\_mqtt.connect(self.messageBroker, 1883)

        self.\_paho\_mqtt.loop\_start()

        self.\_paho\_mqtt.subscribe(self.topic, 2)

    def stop(self):

        self.\_paho\_mqtt.unsubscribe(self.topic)

        self.\_paho\_mqtt.loop\_stop()

        self.\_paho\_mqtt.disconnect()

    def myOnConnect(self, paho\_mqtt, userdata, flags, rc):

        print("Connected with result code " + str(rc))

    def myOnMessage(self, paho\_mqtt, userdata, msg):

        print(f"Received message: '{msg.payload.decode()}' on topic '{msg.topic}' with QoS {msg.qos}")

        # Publish a new message after 5 seconds

        time.sleep(5)

        self.\_paho\_mqtt.publish(self.topic, "28°C", 2)

        print("Published message: '28°C'")

if \_\_name\_\_ == '\_\_main\_\_':

    client\_id = 'mamad'  # Corrected to a string

    broker = 'mqtt.eclipseprojects.io'

    topic = "home/temperature"

    subscribe = MySubscriber(client\_id, topic, broker)

    subscribe.start()

    try:

        while True:

            time.sleep(1)  # Keep the script running to allow message handling

    except KeyboardInterrupt:

        print("Stopping the subscriber...")

        subscribe.stop()

# Design a topic structure for the smart home system

Scenario: You are developing an IoT system for a smart home, where various sensors and devices communicate using the MQTT protocol. The system includes temperature sensors, humidity sensors, and lights. Each device publishes data to specific MQTT topics, and a central controller subscribes to these topics to make decisions based on the received data.

MQTT Topic Structure: Design a topic structure for the smart home system, considering the following devices:

Temperature sensors located in different rooms (e.g., living room, bedroom). Humidity sensors located in different rooms.

Lights located in different rooms, which can be turned on/off and dimmed.

MQTT Client Implementation: Write a Python script using the paho-mqtt library to implement an MQTT client that subscribes to all topics related to lights in the smart home. The client should: Print the received messages to the console.

Respond to a specific command ("lights/all/set") to turn off all lights when a message with the payload "off" is received.

import paho.mqtt.client as mqtt

# Define the topic structure for your smart home system

TOPIC\_TEMPERATURE = "home/+/temperature"

TOPIC\_HUMIDITY = "home/+/humidity"

TOPIC\_LIGHTS = "home/+/lights"

TOPIC\_LIGHTS\_ALL\_SET = "home/lights/all/set"

class SmartHomeController:

    def \_\_init\_\_(self, clientID, broker):

        """

        Initializes the SmartHomeController.

        Parameters:

        - clientID: Unique identifier for the MQTT client.

        - broker: MQTT broker address.

        """

        self.clientID = clientID

        self.broker = broker

        self.mqtt\_client = mqtt.Client(self.clientID)

        self.mqtt\_client.on\_connect = self.on\_connect

        self.mqtt\_client.on\_message = self.on\_message

    def on\_connect(self, client, userdata, flags, rc):

        """

        Handles connection to the MQTT broker and subscribes to topics.

        """

        if rc == 0:

            print("Connected to broker.")

            # Subscribe to all relevant topics

            self.mqtt\_client.subscribe(TOPIC\_TEMPERATURE)

            self.mqtt\_client.subscribe(TOPIC\_HUMIDITY)

            self.mqtt\_client.subscribe(TOPIC\_LIGHTS)

            self.mqtt\_client.subscribe(TOPIC\_LIGHTS\_ALL\_SET)

            print(f"Subscribed to topics: {TOPIC\_TEMPERATURE}, {TOPIC\_HUMIDITY}, {TOPIC\_LIGHTS}, {TOPIC\_LIGHTS\_ALL\_SET}")

        else:

            print(f"Connection failed with code {rc}")

    def on\_message(self, client, userdata, msg):

        """

        Handles incoming messages and processes them.

        """

        payload = msg.payload.decode()

        print(f"Message received on {msg.topic}: {payload}")

        if msg.topic == TOPIC\_LIGHTS\_ALL\_SET and payload.lower() == "off":

            print("Command received: Turn off all lights")

    def start(self):

        """

        Starts the MQTT client and connects to the broker.

        """

        self.mqtt\_client.connect(self.broker, 1883)

        self.mqtt\_client.loop\_start()

    def stop(self):

        """

        Stops the MQTT client and disconnects from the broker.

        """

        self.mqtt\_client.loop\_stop()

        self.mqtt\_client.disconnect()

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    broker\_address = "mqtt.eclipseprojects.io"

    client\_id = "smart\_home\_controller"

    controller = SmartHomeController(client\_id, broker\_address)

    controller.start()

    try:

        while True:

            pass  # Keep running

    except KeyboardInterrupt:

        controller.stop()

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# Create a class DeviceController

You are building an MQTT-based system to monitor and control various devices in a smart building. Each device sends periodic status updates to a specific topic, and the controller subscribes to these topics to take appropriate actions based on the device status.

Create a class DeviceController that subscribes to a topic based on device type and location.

The controller should print out a custom message when it receives a status update from any device.

Implement a method process\_message(self, msg) inside the DeviceController class to handle the message received.

The method should decode the message payload, extract the status (e.g., "ON", "OFF"), and print a message that includes the device type, location, and status.

Use parameters (params) to dynamically change the subscribed topic based on the device type and location.

Example Scenario:

Device: Light

Location: Living Room

Topic: building/smart\_home/light/living\_room

The controller should subscribe to this topic and, upon receiving a status update (e.g., "ON"), print: "Light in Living Room is ON"

Hints:

Use the on\_message callback to trigger the process\_message method.

Make use of MQTT parameters to dynamically build the topic for different devices and locations.

ANSWER:

import paho.mqtt.client as mqtt

class DeviceController:

    def \_\_init\_\_(self, clientID, broker, device\_type, location):

        """

        Initializes the DeviceController.

        Parameters:

        - clientID: Unique identifier for the MQTT client.

        - broker: MQTT broker address.

        - device\_type: Type of the device (e.g., "light", "temperature").

        - location: Location of the device (e.g., "living\_room").

        """

        self.clientID = clientID

        self.broker = broker

        self.device\_type = device\_type

        self.location = location

        self.topic = f"building/smart\_home/{device\_type}/{location}"

        self.mqtt\_client = mqtt.Client(self.clientID)

        self.mqtt\_client.on\_connect = self.on\_connect

        self.mqtt\_client.on\_message = self.on\_message

    def on\_connect(self, client, userdata, flags, rc):

        """

        Handles connection to the MQTT broker and subscribes to the topic.

        """

        if rc == 0:

            print(f"Connected to broker. Subscribing to {self.topic}")

            self.mqtt\_client.subscribe(self.topic)

        else:

            print(f"Connection failed with code {rc}")

    def on\_message(self, client, userdata, msg):

        """

        Handles incoming messages and processes them.

        """

        self.process\_message(msg)

    def process\_message(self, msg):

        """

        Processes the received message by decoding the payload and printing a custom message.

        """

        payload = msg.payload.decode()

        device\_name = self.device\_type.capitalize()

        location\_name = self.location.replace('\_', ' ').title()

        print(f"{device\_name} in {location\_name} is {payload}")

    def start(self):

        """

        Starts the MQTT client and connects to the broker.

        """

        self.mqtt\_client.connect(self.broker, 1883)

        self.mqtt\_client.loop\_start()

    def stop(self):

        """

        Stops the MQTT client and disconnects from the broker.

        """

        self.mqtt\_client.loop\_stop()

        self.mqtt\_client.disconnect()

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    device\_type = "light"

    location = "living\_room"

    clientID = "device\_controller\_light\_livingroom"

    broker = "mqtt.eclipseprojects.io"

    controller = DeviceController(clientID, broker, device\_type, location)

    controller.start()

    try:

        while True:

            pass  # Keep running

    except KeyboardInterrupt:

        controller.stop()

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# subscribe to three MQTT topics

You are tasked with developing a **monitoring system for a smart greenhouse**. The system will have multiple sensors (e.g., temperature, humidity, soil moisture) that publish data to different MQTT topics. You need to design a Python class that can subscribe to these topics, collect the sensor data, and calculate the average values over a 5-minute window. The results should be published to a summary topic every 5 minutes.

Requirements:

Subscription to Multiple Topics:

The class should subscribe to three MQTT topics: "greenhouse/temperature", "greenhouse/humidity", and "greenhouse/soil\_moisture".

Data Collection and Averaging:

The class should store the incoming sensor data and calculate the average values for each sensor over the last 5 minutes.

The data should be stored in a list for each sensor, with each entry containing the sensor reading and the timestamp.

Periodic Publishing:

Every 5 minutes, the class should calculate the average values for the temperature, humidity, and soil moisture sensors and publish the results to the topic "greenhouse/summary" in JSON format.

MQTT Implementation:

Implement the MQTT client using the paho-mqtt library.

Ensure the MQTT client handles reconnections in case of a connection loss.

Questions:

How would you structure the class to handle subscriptions to multiple topics?

What data structures would you use to store the sensor data for averaging?

How would you ensure the class calculates the averages and publishes them exactly every 5 minutes?

How would you handle MQTT connection issues, such as disconnects or timeouts?

Bonus:

Can you add a feature to detect anomalies (e.g., temperature above 40°C) and immediately publish an alert to a topic "greenhouse/alerts"?

ANSWER:

import paho.mqtt.client as mqtt

import time

import json

class SmartGreenhouse:

    def \_\_init\_\_(self, clientID, broker):

        """

        Initializes the SmartGreenhouse class.

        """

        self.clientID = clientID

        self.broker = broker

        self.mqtt\_client = mqtt.Client(self.clientID)

        self.mqtt\_client.on\_connect = self.on\_connect

        self.mqtt\_client.on\_message = self.on\_message

        self.data = {

            "greenhouse/temperature": [],

            "greenhouse/humidity": [],

            "greenhouse/soil\_moisture": []

        }

        self.start\_time = time.time()

    def on\_connect(self, client, userdata, flags, rc):

        """

        Handles connection to the MQTT broker and subscribes to sensor topics.

        """

        if rc == 0:

            print("Connected to broker.")

            for topic in self.data.keys():  # Subscribe to all topics in self.data

                self.mqtt\_client.subscribe(topic)

                print(f"Subscribed to: {topic}")

        else:

            print(f"Connection failed: {rc}")

    def on\_message(self, client, userdata, msg):

        """

        Handles incoming sensor data and stores it.

        """

        topic = msg.topic

        reading = float(msg.payload.decode())  # Convert payload to float

        self.data[topic].append(reading)

        print(f"{topic} -> {reading}")

        # Check for anomaly (temperature > 40)

        if topic == "greenhouse/temperature" and reading > 40:

            self.mqtt\_client.publish(

                "greenhouse/alerts",

                json.dumps({"sensor": "temperature", "value": reading, "message": "Anomaly detected"})

            )

            print(f"Published alert for {topic}: {reading}")

        # Publish summary every 5 minutes

        if time.time() - self.start\_time >= 300:

            self.publish\_summary()

    def publish\_summary(self):

        """

        Calculates averages for each sensor and publishes the summary.

        """

        summary = {}

        for topic, readings in self.data.items():

            if readings:  # Only calculate if there are readings

                sensor\_type = topic.split("/")[-1]  # Extract sensor type

                avg\_value = sum(readings) / len(readings)  # Calculate average

                summary[sensor\_type] = avg\_value

                self.data[topic] = []  # Clear data for the topic

        self.mqtt\_client.publish("greenhouse/summary", json.dumps(summary))  # Publish the summary

        self.start\_time = time.time()  # Reset the timer

        print(f"Published summary: {summary}")

    def start(self):

        """

        Starts the MQTT client and connects to the broker.

        """

        self.mqtt\_client.connect(self.broker, 1883)

        self.mqtt\_client.loop\_start()

    def stop(self):

        """

        Stops the MQTT client and disconnects from the broker.

        """

        self.mqtt\_client.loop\_stop()

        self.mqtt\_client.disconnect()

        print("Disconnected from broker.")

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    greenhouse = SmartGreenhouse("greenhouse\_monitor", "mqtt.eclipseprojects.io")

    greenhouse.start()

    try:

        while True:

            time.sleep(1)  # Keep running

    except KeyboardInterrupt:

        greenhouse.stop()

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# MQTT sensor sub-pub

Temperature and Humidity Monitoring System

You are tasked with creating a temperature and humidity monitoring system using MQTT. The system must satisfy the following requirements:

Data Collection:

A sensor generates random temperature and humidity values every 15 seconds.

The temperature should be between -10°C and 50°C.

The humidity should be between 0% and 100%.

Data Storage:

The system should store the last 20 readings for both temperature and humidity.

Data Analysis:

Every 2 minutes, the system should calculate the average temperature and humidity.

If the average temperature exceeds 30°C or the average humidity exceeds 70%, it should trigger an alert by publishing a message to the "alerts" topic.

Data Publishing:

Every 2 minutes, the system should publish the average temperature and humidity values to the "monitoring/temperature" and "monitoring/humidity" topics in JSON format.

MQTT Configuration:

Use mqtt.eclipseprojects.io as the broker.

Use QoS level 1 for publishing and subscribing.

Tasks:

Implement the MQTT client class that handles all the above functionalities.

Ensure the program runs indefinitely, collecting, analyzing, and publishing data as described.

The output should include messages for when alerts are triggered.

import paho.mqtt.client as mqtt

import random

import time

import json

from collections import deque

# Global topic definitions

TEMP\_TOPIC = "monitoring/temperature"

HUMIDITY\_TOPIC = "monitoring/humidity"

ALERTS\_TOPIC = "alerts"

class MonitoringSystem:

    def \_\_init\_\_(self, clientID, broker):

        """

        Initializes the MonitoringSystem class.

        """

        self.clientID = clientID

        self.broker = broker

        self.mqtt\_client = mqtt.Client(self.clientID)

        self.mqtt\_client.on\_connect = self.on\_connect

        # Store the last 20 readings using deque

        self.temperature\_readings = deque(maxlen=20)

        self.humidity\_readings = deque(maxlen=20)

    def on\_connect(self, client, userdata, flags, rc):

        """

        Handles connection to the MQTT broker.

        """

        if rc == 0:

            print("Connected to broker.")

        else:

            print(f"Connection failed with code {rc}")

    def generate\_readings(self):

        """

        Simulates sensor data generation every 15 seconds.

        """

        temperature = round(random.uniform(-10, 50), 2)  # Generate random temperature

        humidity = round(random.uniform(0, 100), 2)  # Generate random humidity

        self.temperature\_readings.append(temperature)

        self.humidity\_readings.append(humidity)

        print(f"Generated readings: Temperature={temperature}°C, Humidity={humidity}%")

    def calculate\_averages\_and\_publish(self):

        """

        Calculates averages, publishes to MQTT topics, and triggers alerts if necessary.

        """

        # Check if there are readings before calculating averages

        if len(self.temperature\_readings) > 0 and len(self.humidity\_readings) > 0:

            avg\_temp = sum(self.temperature\_readings) / len(self.temperature\_readings)

            avg\_humidity = sum(self.humidity\_readings) / len(self.humidity\_readings)

            # Publish averages

            self.mqtt\_client.publish(TEMP\_TOPIC, json.dumps({"average\_temperature": avg\_temp}), qos=1)

            self.mqtt\_client.publish(HUMIDITY\_TOPIC, json.dumps({"average\_humidity": avg\_humidity}), qos=1)

            print(f"Published: Average Temperature={avg\_temp}°C, Average Humidity={avg\_humidity}%")

            # Trigger alerts for temperature

            if avg\_temp > 30:

                temp\_alert = {

                    "sensor": "temperature",

                    "value": avg\_temp,

                    "alert": "High Temperature"

                }

                self.mqtt\_client.publish(ALERTS\_TOPIC, json.dumps(temp\_alert), qos=1)

                print(f"Temperature alert published: {temp\_alert}")

            # Trigger alerts for humidity

            if avg\_humidity > 70:

                humidity\_alert = {

                    "sensor": "humidity",

                    "value": avg\_humidity,

                    "alert": "High Humidity"

                }

                self.mqtt\_client.publish(ALERTS\_TOPIC, json.dumps(humidity\_alert), qos=1)

                print(f"Humidity alert published: {humidity\_alert}")

        else:

            print("Not enough data to calculate averages.")

    def start(self):

        """

        Starts the MQTT client and connects to the broker.

        """

        self.mqtt\_client.connect(self.broker, 1883)

        self.mqtt\_client.loop\_start()

    def stop(self):

        """

        Stops the MQTT client and disconnects.

        """

        self.mqtt\_client.loop\_stop()

        self.mqtt\_client.disconnect()

        print("Disconnected from broker.")

# Main

if \_\_name\_\_ == "\_\_main\_\_":

    monitoring\_system = MonitoringSystem("temp\_humidity\_monitor", "mqtt.eclipseprojects.io")

    monitoring\_system.start()

    last\_time = time.time()  # Initialize timer

    try:

    while True:

        monitoring\_system.generate\_readings()  # Generate new readings first

        if time.time() - last\_time >= 120:  # Check if 2 minutes have passed

            monitoring\_system.calculate\_averages\_and\_publish()  # Calculate averages and publish

            last\_time = time.time()  # Reset the timer

        time.sleep(15)  # Wait for the next reading

except KeyboardInterrupt:

    monitoring\_system.stop()

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# SmartHomeController

Scenario: You are developing a smart home system that controls and monitors various devices like lights, fans, and door locks. You need to create a class SmartHomeController that manages the following tasks:

Data Collection:

Simulate data collection from sensors every 30 seconds. The data could be the state of a light (on or off), the state of a fan (on or off), and the status of the door (locked or unlocked).

Store the last 10 states of each device.

Data Publishing: Every minute, publish the last two collected states of each device to their respective MQTT topics in a JSON format.

Alerts: If a door is found to be "unlocked" for more than 1 minute (i.e., two consecutive states are "unlocked"), publish an alert message.

Control Commands:

Allow the system to subscribe to a "commands" topic, where it can receive control commands for the devices (e.g., "turn\_on\_light", "lock\_door").

Upon receiving a command, update the device state accordingly and publish the new state to its corresponding topic.

Tasks:

Implement the SmartHomeController class with the following methods:

collect\_data: Collect sensor states every 30 seconds.

publish\_data: Publish the last two states every minute.

process\_alerts: Check and publish alerts based on the device states.

handle\_command: Handle incoming commands and update the device states.

start and stop methods for starting and stopping the MQTT client.

Run the system in a loop that continuously collects data, publishes it, and processes any commands received.

ANSWER:

import paho.mqtt.client as mqtt

import time

import json

from collections import deque

import random

class SmartHomeController:

    def \_\_init\_\_(self, clientID, broker):

        """

        Initializes the SmartHomeController class.

        """

        self.clientID = clientID

        self.broker = broker

        self.mqtt\_client = mqtt.Client(self.clientID)

        self.mqtt\_client.on\_connect = self.on\_connect

        self.mqtt\_client.on\_message = self.handle\_command

        # Device states (store the last 10 states)

        self.light\_states = deque(maxlen=10)

        self.fan\_states = deque(maxlen=10)

        self.door\_states = deque(maxlen=10)

        # MQTT Topics

        self.light\_topic = "home/lights"

        self.fan\_topic = "home/fans"

        self.door\_topic = "home/doors"

        self.commands\_topic = "home/commands"

        self.alerts\_topic = "home/alerts"

    def on\_connect(self, client, userdata, flags, rc):

        """

        Handles connection to the MQTT broker and subscribes to the commands topic.

        """

        if rc == 0:

            print("Connected to broker.")

            self.mqtt\_client.subscribe(self.commands\_topic)

        else:

            print(f"Connection failed with code {rc}")

    def generate\_data(self):

        """

        Simulates generating data from devices every 30 seconds.

        """

        light\_state = random.choice(["on", "off"])

        fan\_state = random.choice(["on", "off"])

        door\_state = random.choice(["locked", "unlocked"])

        self.light\_states.append(light\_state)

        self.fan\_states.append(fan\_state)

        self.door\_states.append(door\_state)

        print(f"Generated states - Light: {light\_state}, Fan: {fan\_state}, Door: {door\_state}")

    def publish\_data(self):

        """

        Publishes the last two states of each device every minute.

        """

        data = {

            "light\_states": list(self.light\_states)[-2:],

            "fan\_states": list(self.fan\_states)[-2:],

            "door\_states": list(self.door\_states)[-2:]

        }

        self.mqtt\_client.publish(self.light\_topic, json.dumps({"light": data["light\_states"]}), qos=1)

        self.mqtt\_client.publish(self.fan\_topic, json.dumps({"fan": data["fan\_states"]}), qos=1)

        self.mqtt\_client.publish(self.door\_topic, json.dumps({"door": data["door\_states"]}), qos=1)

        print(f"Published device states: {data}")

    def process\_alerts(self):

        """

        Checks for alerts and publishes them if necessary.

        """

        if len(self.door\_states) >= 2 and self.door\_states[-1] == "unlocked" and self.door\_states[-2] == "unlocked":

            alert\_message = {"alert": "Door has been unlocked for more than 1 minute!"}

            self.mqtt\_client.publish(self.alerts\_topic, json.dumps(alert\_message), qos=1)

            print(f"Alert published: {alert\_message}")

    def handle\_command(self, client, userdata, msg):

        """

        Handles incoming commands and updates device states.

        """

        command = msg.payload.decode()

        print(f"Received command: {command}")

        if command == "turn\_on\_light":

            self.light\_states.append("on")

            self.mqtt\_client.publish(self.light\_topic, json.dumps({"light": "on"}), qos=1)

        elif command == "turn\_off\_light":

            self.light\_states.append("off")

            self.mqtt\_client.publish(self.light\_topic, json.dumps({"light": "off"}), qos=1)

        elif command == "turn\_on\_fan":

            self.fan\_states.append("on")

            self.mqtt\_client.publish(self.fan\_topic, json.dumps({"fan": "on"}), qos=1)

        elif command == "turn\_off\_fan":

            self.fan\_states.append("off")

            self.mqtt\_client.publish(self.fan\_topic, json.dumps({"fan": "off"}), qos=1)

        elif command == "lock\_door":

            self.door\_states.append("locked")

            self.mqtt\_client.publish(self.door\_topic, json.dumps({"door": "locked"}), qos=1)

        elif command == "unlock\_door":

            self.door\_states.append("unlocked")

            self.mqtt\_client.publish(self.door\_topic, json.dumps({"door": "unlocked"}), qos=1)

    def start(self):

        """

        Starts the MQTT client and connects to the broker.

        """

        self.mqtt\_client.connect(self.broker, 1883)

        self.mqtt\_client.loop\_start()

    def stop(self):

        """

        Stops the MQTT client and disconnects.

        """

        self.mqtt\_client.loop\_stop()

        self.mqtt\_client.disconnect()

        print("Disconnected from broker.")

# Main

if \_\_name\_\_ == "\_\_main\_\_":

    controller = SmartHomeController("smart\_home\_controller", "mqtt.eclipseprojects.io")

    controller.start()

    last\_publish\_time = time.time()

    try:

        while True:

            controller.generate\_data()  # Generate new data first

            if time.time() - last\_publish\_time >= 60:  # Check if 1 minute has passed

                controller.publish\_data()

                controller.process\_alerts()

                last\_publish\_time = time.time()  # Reset the timer

            time.sleep(30)  # Wait 30 seconds for the next data generation

    except KeyboardInterrupt:

        controller.stop()

# EXAM Q: MQTT Temperature sensor

You need to **create a temperature sensor** class that will Simulate Temperature Data:

The sensor should generate a random temperature value between 20°C and 30°C every 10 seconds.

Store Data: The sensor should **store the last 12 temperature readings**.

Publish Data: Every minute, the sensor should publish all the stored data (the last 12 readings) to an MQTT topic in SenML format.

Use of SenML: SenML (Sensor Markup Language) is a format for representing simple sensor measurements and device parameters. The data should be published in JSON format, using the SenML structure.

MQTT Class Usage: You should make use of the MyPublisher class we developed earlier for MQTT communication.

**Steps to** Consider**:**

Create a Sensor Class: The class should handle data generation, storage, and MQTT publishing.

Generate Random Temperature: Use Python’s random module to simulate temperature data.

Store Last 12 Readings: Maintain a list to keep track of the last 12 temperature readings.

Publish Data Every Minute: Use time.sleep(10) for the temperature readings and time.sleep(60) for publishing.

SenML Format: The data should be published in JSON with the SenML format, including fields like "bn" (base name), "bt" (base time), "e" (events), and "v" (value).

import random

import time

import json

from collections import deque

import paho.mqtt.client as mqtt

class TemperatureSensorPublisher:

    def \_\_init\_\_(self, clientID, broker, topic):

        """

        Initializes the TemperatureSensorPublisher class.

        """

        self.clientID = clientID

        self.broker = broker

        self.topic = topic

        self.mqtt\_client = mqtt.Client(self.clientID)

        # Register the on\_connect callback

        self.mqtt\_client.on\_connect = self.on\_connect

        # Store the last 12 temperature readings

        self.temperature\_readings = deque(maxlen=12)

        self.base\_name = "temperature\_sensor"

        self.base\_time = int(time.time())  # Base time for SenML

    def on\_connect(self, client, userdata, flags, rc):

        """

        Callback function when the client connects to the broker.

        """

        if rc == 0:

            print("Connected to MQTT broker.")

        else:

            print(f"Failed to connect, return code {rc}")

    def start(self):

        """

        Starts the MQTT client and connects to the broker.

        """

        self.mqtt\_client.connect(self.broker, 1883)

        self.mqtt\_client.loop\_start()

    def stop(self):

        """

        Stops the MQTT client and disconnects.

        """

        self.mqtt\_client.loop\_stop()

        self.mqtt\_client.disconnect()

    def generate\_temperature(self):

        """

        Simulates temperature data generation.

        """

        temperature = round(random.uniform(20, 30), 2)  # Random temperature between 20°C and 30°C

        self.temperature\_readings.append(temperature)

        print(f"Generated Temperature: {temperature}°C")

    def publish\_data(self):

        """

        Publishes the stored data in SenML format.

        """

        # Create the SenML data dictionary

        senml\_data = {

            "bn": self.base\_name,

            "bt": self.base\_time,

            "e": []  # Initialize an empty list for events

        }

        # Populate the "e" field with temperature readings

        for idx, temp in enumerate(self.temperature\_readings):

            event = {

                "v": temp,  # Temperature value

                "t": idx \* 10  # Time offset relative to base time

            }

            senml\_data["e"].append(event)

        # Convert the dictionary to a JSON string

        message = json.dumps(senml\_data)

        # Publish the message to the MQTT topic

        self.mqtt\_client.publish(self.topic, message, qos=1)

        print(f"Published SenML Data: {message}")

# Main

if \_\_name\_\_ == "\_\_main\_\_":

    # Create an instance of the temperature sensor publisher

    sensor = TemperatureSensorPublisher("temperature\_sensor", "mqtt.eclipseprojects.io", "home/sensors/temperature")

    sensor.start()

    last\_publish\_time = time.time()  # Track the last publish time

    try:

        while True:

            # Generate a new temperature reading

            sensor.generate\_temperature()

            # Check if 1 minute has passed for publishing data

            if time.time() - last\_publish\_time >= 60:

                sensor.publish\_data()

                last\_publish\_time = time.time()  # Reset the timer

            time.sleep(10)  # Wait for the next temperature generation

    except KeyboardInterrupt:

        # Gracefully stop the sensor when interrupted

        sensor.stop()

        print("Sensor stopped.")

# EXAM Q: RESTful wordfinder

Develop a RESTful-style program in CherryPy to find a word from a list of words. If the word is present in the list, the program should return its position in the list and the list **without** the considered word. Otherwise, it should return a “not found" message.

The inputs of the program should be: The word to be found & the list of words

Choose the most suitable HTTP method among GET/POST/PUT/DELETE and the data-format. The solution MUST use Object Oriented Programming.

## Answer:

Continue here from chatgpt last response that I responded to repeat! And use POSTMAN for POST.

WordFinderService

**Explanation**

1. **Input Handling**:
   * The input is expected as a JSON payload in the body of the POST request.
   * Example Input:

{

"word": "apple",

"word\_list": ["banana", "apple", "cherry"]

}

1. **Validation**:
   * Ensures that the word is a string and word\_list is a list.
   * Returns a 400 Bad Request error for invalid input formats.
2. **Logic**:
   * Checks if the word exists in the list:
     + If found: Returns its position and a new list excluding the word.
     + If not found: Returns a "not found" message.
3. **Response**:
   * The response is returned in JSON format.
   * Example Successful Output:

{

"message": "Word 'apple' found at position 1",

"updated\_list": ["banana", "cherry"]

}

* + Example Not Found Output:

{

"message": "Word not found."

}

1. **Error Handling**:
   * Handles JSON decoding errors and returns meaningful error messages.

**How to Test**

1. Save the script in a file (e.g., word\_finder.py).
2. Run the script: python word\_finder.py.
3. Use a tool like **Postman** or curl to send a POST request:
   * URL: http://localhost:8080/
   * Headers: Content-Type: application/json
   * Body:

{

"word": "apple",

"word\_list": ["banana", "apple", "cherry"]

}

# EXAM Q: RESTfull Students registration and grades

A close-up of a paper

Description automatically generated

Students= {name:ali, surname:alavi, ID:1}

{ID:1, math:[25, 30, 29], literature: [22, 27], scientce:[30, 28]}

{subject:””, grades:[]}

import cherrypy

class student:

def \_\_init\_\_(self, name, surname, id):

student.name=name

student.surname=surname

student.id=ID

def POST(self, \*uri, \*\*params):

if len(uri)==0:

return “Please provide the endpoints: <b> /students or /students/grades. </b>”

else:

if \_\_main\_\_==”main”:

conf={

“/”: {

“request.dispatch”: cherrypy.dispatch.MethodDispatcher(),

“tools.session.on”: True

}

}

Cherrypy.quickstart(student(), “/” , conf)

# EXAM Q: RESTful NBA scores

A paper with text on it

Description automatically generated

# EXAM Q: RESTful fake sensor

A paper with text on it

Description automatically generated

CODING QUESTION (max 16 points)

Develop a RESTful-style webservice in cherrypy to return fake sensors values. The program must be able to return values of two type of sensors depending on the request it has received. The two type of sensors are a temperature one and a light sensor. Both the sensor values will be randomly extracted depending on the typology of sensor:

Temperature sensor values [0, 40]

Light sensor values [0, 1000]

In addition, the program must be able to check if the extracted value exceeds a certain hard-coded threshold

temperature threshold: 30

light threshold: 500

the webservice will return the sensed values and a different message depending on the fact that the sensed value is above or below the threshold (e.g. "temperature: 35 degrees, It is too hot!" or "light: 200, what a dark day!" etc..)

The program receives as input the type of sensor and if no sensor is specified it will return a response for both of them. Choose the most suitable HTTP methods among GET/POST/PUT/DELETE and the data-format. The solution must use the Object-Oriented Programming.

Provide an example of the URL that can be used to interact with this web-service.

## Answer:

import random

import cherrypy

class Sensor:

    exposed = True

    def \_\_init\_\_(self):

        self.light = None

        self.temp = None

    def lightInq(self):

        self.light = random.randint(0, 1000)

    def tempInq(self):

        self.temp = random.randint(0, 40)

    def GET(self, \*uri, \*\*params):

        if len(uri) > 0:

            if uri[0] == "light" and len(uri) == 1:

                self.lightInq()

                if self.light > 500:

                    return f"Light is {self.light} units, what a bright day!"

                else:

                    return f"Light is {self.light} units, what a dark day!"

            elif uri[0] == "temperature" and len(uri) == 1:

                self.tempInq()

                if self.temp > 30:

                    return f"Temperature is {self.temp} degrees, what a hot day!"

                else:

                    return f"Temperature is {self.temp} degrees, what a cool day!"

            else:

                return "Invalid Request"

        else:

            self.tempInq()

            self.lightInq()

            return (

                f"Please use one of the following endpoints:<br> <b>/light</b> or <b>/temperature</b>"

            )

if \_\_name\_\_ == "\_\_main\_\_":

    conf = {

        "/": {

            "request.dispatch": cherrypy.dispatch.MethodDispatcher(),

            "tools.sessions.on": True,

        }

    }

    cherrypy.quickstart(Sensor(), '/', conf)

    # cherrypy.tree.mount(Sensor(), "/", conf)

    # cherrypy.engine.start()

    # cherrypy.engine.block()

# EXAM definitions:

### Q: Briefly describe the XML data format with properties and structure providing a simple example

**Extensible** **markup language** used for **data exchange**.

Properties:

- self descriptive (using tags)

- hierarchical structure (root element and child elements)

- extensible (adaptable tags)

Example:

<book>

<title>The Hitchhiker's Guide to the Galaxy</title>

<author>Douglas Adams</author>

<genre>Science Fiction</genre>

</book>

### JSON data format

JSON (JavaScript Object Notation) is a **lightweight**, text-based **data exchange format**.

It's derived from JavaScript but is language-independent, meaning it can be used by various programming languages.

JSON is popular for its simplicity and readability, making it a common choice for exchanging data between web servers and web applications, as well as in APIs.

### XML and JSON Data Exchange Formats comparison

They are **standardized ways of structuring data** able to be read by **humans and machines,** and are used for IoT devices. So, they are **languages of machine-to-machine communication**.

* JSON vs. XML: **JSON is more lightweight** and easier for machines to parse, and therefore faster to process. JSON is a common API output in a wide variety of applications.
* SenML: a dialect of JSON specifically for sensor data
* Markup Languages: SGML, HTML, XML

### Data exchange in the web environment:

* SGML: SGML is a **meta-language**, meaning it's a language for defining other markup languages. (XML is a markup language derived from SGML.) (uses tags)
* HTML: It is the standard **markup language** for creating web pages. (uses tags)
* XML: a **markup language** derived from SGML. It's used to store and transport data. (uses tags)
* JSON (SenML): JSON is a lightweight **data-interchange format**. Widely used for data exchange in web applications and APIs. (uses name/value pairs)

### Gateways / Device connector:

* **Data translation**: Since different devices use different protocols, a gateway acts as a translator converting data from one protocol to another. (Translators between the IoT devices that do **protocol conversion** so devices can communicate with each other.)
* **Data aggregation**: Gateway works as a central hub and collects data from multiple devices, packages it up, and sends it to a central location for processing.
* **Firewall action**: Gateway can block any unauthorized access to the network

### Q: Briefly describe what is CherryPy and it’s usage in terms of IoT platforms

CherryPy is a **lightweight** and flexible **object-oriented web framework** written **in Python**.

It's designed to be easy to use, allowing developers to **build web applications** with a focus on object-oriented principles.

**Making the web services**, which communicate using HTTP, for IoT is done by CherryPy. (using CherryPy instead of raw HTTP code)

CherryPy Usage:

**Create a RESTful API**: This API would allow smart home devices (lights, thermostats, etc.) to send data like temperature, humidity, and usage to a central server.

**Collect and Process Data**: The server, running a CherryPy application, would receive this data, process it, and store it in a database.

**Provide a Web Interface**: The same CherryPy application could also provide a web interface for users to monitor their home's energy usage, control devices, and adjust settings.

### REST and its key principles

**HTTP is stateless 🡪 faster but more complex:** To help manage this complexity we have architectural styles like REST.

**REST**: a set of guidelines for building **web services optimized for HTTP**. So, a key principle of REST is using HTTP methods like GET, POST, PUT, and DELETE to interact with web resources.

A web service built on REST principles **receives** **requests[[1]](#footnote-1) and responds** in a standardized and efficient communication.

To make sure these web services are well designed we use REST rules and guidelines, and it has 4 **key principles**:

1. **addressability**: every resource in the web service has a unique ID called URI which is its specific address (uniform resource identifier)
2. **statelessness**: each request is independent and the server does not remember its data, so it scales much better.
3. **cachability**: storing the responses from the web service for later use
4. **uniform interface**: all interactions with the any web service have to use HTTP methods.

### Discuss and compare both TCP (Transmission control protocol) and UDP (user datagram protocol) highlighting their main properties and functionalities.

Thet are both **transport protocols**: one is fast and less reliable (UDP), and the other slower but very reliable (TCP).

TCP is suitable for applications that require **reliable data delivery**, such as file transfers and web browsing. It **makes sure** your data gets to the right place and in the right order. Each lost packet is **resent**. (For each received packet, the receiver sends an **acknowledgment** (ACK) to the sender.)[[2]](#footnote-2)

UDP is suitable for applications that **prioritize speed and low latency over reliability**, such as real-time audio and video streaming. Lost packets are **NOT resent**.

### TCP-IP

**Computers and smartphones** use communication protocols like TCP-IP for using the internet, which works with **a constant connection** **to the internet**, but **IoT devices** only need to send a message now and then, so TCP is too much for them! More **lightweight protocols** are designed to work with **low-power, low-bandwidth communication**, and they include **Bluetooth, ZigBee, Z-wave, and 6lowpan (An adaptation of IPv6 for low-power wireless networks, enabling IPv6 connectivity for constrained devices.)**.

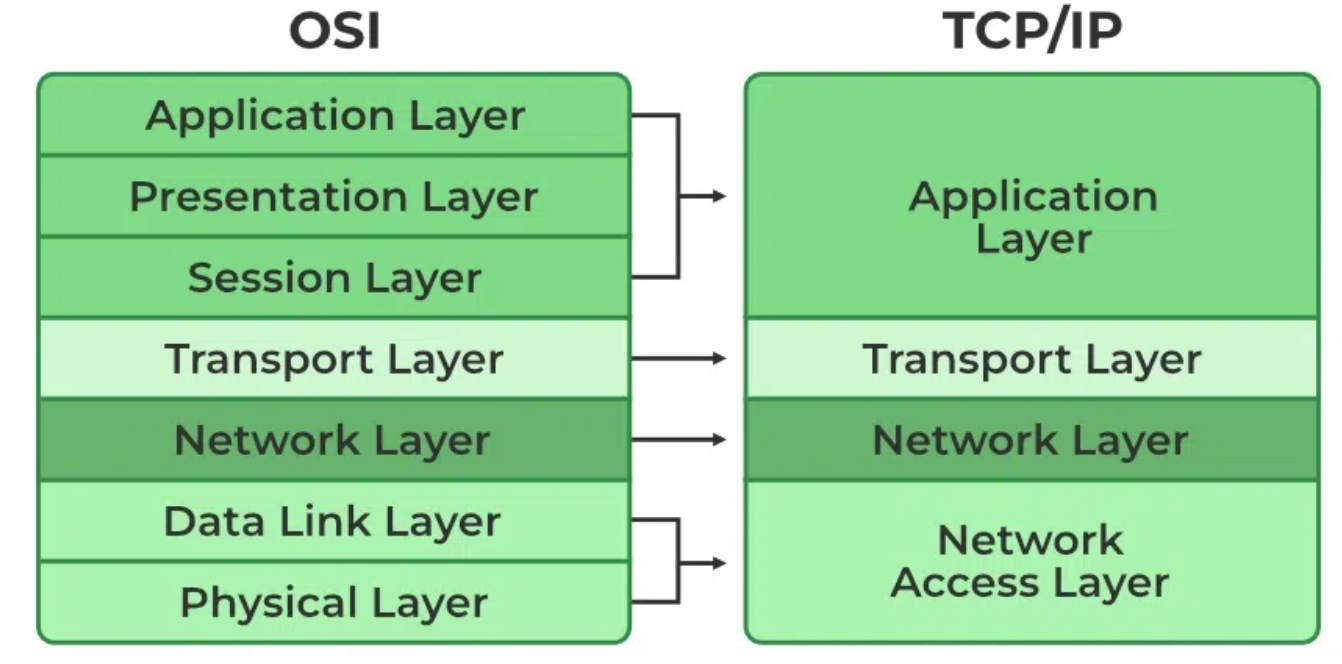
|  |  |  |
| --- | --- | --- |
| Feature | TCP | UDP |
| Reliability | Reliable | Unreliable |
| Connection | Connection-oriented | Connectionless |
| Overhead | High | Low |
| Speed | Slower | Faster |
| Error Handling | Robust error handling | Minimal error handling |
| Order | Guaranteed in-order delivery | No guaranteed order |
| Flow Control | Flow control implemented | No flow control |
| Applications | Web browsing, file transfer, email | Video streaming, online gaming, DNS |

TCP and UDP operate at the **Transport Layer** of the OSI model.

Both TCP and UDP are part of the Internet Protocol Suite (TCP/IP).

### TCP/IP model vs OSI model:

The OSI (Open Systems Interconnection) and TCP/IP models are both frameworks used to describe how data is transmitted across a network.



### ISO/OSI model, used in IoT systems

**TCP/IP Dominance:** The TCP/IP model, with its **four layers**, is more widely used in practice, including many IoT scenarios. It's simpler and more efficient for many IoT applications. (4 vs 7 layers)

### HTTP, CoAP, MQTT, and AMQP:

IoT communication paradigms and protocols= Machine-to-Machine (M2M) messaging or communication protocols suitable for IoT systems:

* HTTP: Well-suited for **web-based interactions** and integrating with existing web services. It's the foundation of the World Wide Web.
  + Connection-oriented: Establishes a connection between client and server for each request.
  + **Request-response**: Client sends a request, and the server sends a response.
  + Stateless: Each request is independent of previous requests.
  + **TCP-based**
* CoAP: Designed for constrained environments, CoAP is optimized for low-power devices and resource-limited networks, like **IoT sensor data collection**.
  + **UDP-based**: Built on top of UDP, making it suitable for unreliable networks.
  + Observational Model: Allows clients to **subscribe** to data streams from servers.
  + CoAP supports both request/response and a variant of publish/subscribe. Therefore, CoAP can interoperate with HTTP and RESTful Web Services through simple gateways (request/response). CoAP uses URI to request/send data.
* MQTT: A lightweight messaging protocol ideal for **resource-constrained devices** and **asynchronous** communication.
  + For **subscribing** to multiple topics at once, **wildcards** are used.
  + **TCP-based**
* AMQP: A **general-purpose** message queuing protocol for reliable and **asynchronous** communication.
  + Resource intensive
  + **TCP-based**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Feature | HTTP | CoAP | MQTT | AMQP |
| Type | Request-Response | Publish-Subscribe  Request-Response | Publish-Subscribe | Message Queuing  Request-Response |
| Connection | Connection-oriented | Connectionless | Connectionless | Connectionless |
| Overhead | Relatively high | Low | Low | Moderate |
| Reliability | Reliable | Best-effort | Variable | Reliable |
| Use Cases | Web services, APIs | IoT devices, sensors | IoT telemetry, M2M | Complex IoT, Enterprise |
| Transport Protocol | TCP | UDP | TCP | TCP |

### Object-oriented programming (OOP) main concepts

Object is a runtime entity of a class. instance is a single occurrence of a class.

* **Object**: It represents a real-world entity with its own set of properties (attributes) and behaviors (methods).
* **Instance**: It's a particular object created from a class.
* **Encapsulation**: the principle of bundling data (attributes) and methods (functions) that operate on the data **within a single unit (the class),** controlling access to an object's data. This protects the data from **unauthorized access or modification**.:
* **Inheritance**: the mechanism of creating new classes (child classes or subclasses) from existing classes (parent classes or superclasses). The child class inherits the properties and methods of the parent class, and can also have its own unique attributes and methods.
* **Polymorphism:** allows objects of different classes to be treated as objects of a common type.

### Wildcards in MQTT

This allows devices **to subscribe to a range of topics** rather than specific ones.

1. **Single-Level Wildcard (#)**

* **sport/#** will match sport/football, sport/basketball (only at the end)

1. **Multi-Level Wildcard (+)**

* **sport/+/championship** will match sport/football/championship and sport/basketball/championship

### Push and Pull behavior ???

1. Params help customize the requests to the web service. [↑](#footnote-ref-1)
2. TCP/IP is also used as a communications protocol in a private computer network -- an intranet or extranet. [↑](#footnote-ref-2)