Department of Electronic & Telecommunication Engineering
University of Moratuwa

EN3251 Internet of Things

Laboratory Exercise 2
Information transfer with MQTT and HTTP using JSON

KURUPPU M.P.	210325M
PEIRIS D.L.C.J.	210454G
PERERA L.C.S.	210463H

Date - 2024.09.13

Contents

1 Step 01

1.1 Introduction for Json

JSON (JavaScript Object Notation) is a lightweight, text-based data interchange format that is easy for both humans and machines to read and write. It's widely used for data exchange in various applications, including the Internet of Things (IoT).

1.2 Key Features of JSON

- Text-based Format: It's primarily used to represent structured data as text.
- Human-readable: Data is easy to understand and manually create.
- Language-independent: Although it originated from JavaScript, it can be used with many programming languages.
- Lightweight: It has minimal overhead, making it efficient in terms of network traffic and processing.

1.3 Benefits of JSON in IoT

1. Readability: JSON's clear structure makes it easy to read and understand. This is beneficial for debugging and data interpretation.
2. Lightweight: JSON's simple format minimizes the amount of data being transferred, which is crucial for IoT devices with limited bandwidth and processing power.
3. Compatibility: JSON is supported by many programming languages, making it a versatile choice for data interchange across different systems.
4. Ease of Parsing: Most programming languages offer libraries or built-in functions to parse JSON data efficiently, simplifying data handling in IoT applications.

1.4 Uses of JSON in IoT

1. **Data Exchange Between IoT Devices**:
   IoT devices often need to transmit data from sensors to a central server or cloud platform. JSON is commonly used to encode sensor readings, such as temperature, humidity, and GPS coordinates, for easy transmission.
   Example:

```
{
  "sensor_id": "A1B2C3",
  "temperature": 22.5,
  "humidity": 60,
  "location": {
    "latitude": 51.5074,
    "longitude": 0.1278
  }
}
```

2. **Configuration Management**:
   IoT devices require configuration parameters (e.g., network settings, device behavior, thresholds for sensors). JSON is used to send configuration data from the server to the IoT device or to retrieve settings from the device.

3. **Control Commands**:
   JSON is used to send control messages (e.g., turn on/off, set a parameter) to IoT devices from a central system or mobile application. Devices can interpret JSON messages to perform actions.
   Example:

```
{
  "command": "turn_on",
  "device": "light",
  "brightness": 80
}
```

4. **MQTT with JSON Payloads**:
   JSON is often used as the message payload in MQTT (Message Queuing Telemetry Transport) messages. Devices publish/subscribe to topics with JSON-formatted data, making communication efficient and structured.
   Example MQTT payload:

```
{
  "topic": "home/living_room/temperature",
  "temperature": 21.0,
  "unit": "C"
}
```

5. **Data Storage and Logging**:
   JSON is often used to store data generated by IoT devices. For example, historical

sensor data can be stored in JSON format in databases, making it easy to analyze later.
Example:

```
[
  {"timestamp": "2024-09-12T08:00:00Z", "temperature": 21.0},
  {"timestamp": "2024-09-12T09:00:00Z", "temperature": 22.5}
]
```

6. **IoT Dashboard Updates**:
JSON is used to update dashboards with real-time data from IoT devices. JSON-formatted data streams from the devices to the dashboard, where it is displayed to users.
Example:

```
{
  "dashboard": "SmartHome",
  "data": {
    "temperature": 22,
    "humidity": 45,
    "status": "online"
  }
}
```

2 Step 02

Done

3 Step 03

3.1 Publisher Code

The below code represents the MQTT publisher which reads an object containing multiple string-value pairs from a file called data, and publish it to a Broker on Topic of `IOT_JSON`

```
from paho.mqtt import client as mqtt_client
import paho.mqtt.client as mqtt
import time
import json
read_file_name = "data.json"

# Function to read data from a JSON file
def read_json_file(file_path):
    with open(read_file_name) as json_file:
            file_data= json.load(json_file)
    data_out=json.dumps(file_data) #encode object to JSON
    return data_out


# Callback when the client connects to the MQTT broker
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("Connected to MQTT broker\n")
    else:
        print("Connection failed with code {rc}")


# Create an MQTT client instance
client = mqtt.Client(mqtt_client.CallbackAPIVersion.VERSION1,"PythonPub"
    )

# Set the callback function
client.on_connect = on_connect

broker_address = "test.mosquitto.org"  # broker's address
broker_port = 1883
keepalive = 60
qos = 2
publish_topic = "IOT_JSON"

# Connect to the MQTT broker
client.connect(broker_address, broker_port, keepalive)

# Start the MQTT loop to handle network traffic
client.loop_start()

# Publish loop

try:
    while True:
```

```
        # Publish a message to the send topic

        value = read_json_file(read_file_name )
        client.publish(publish_topic,value, qos=qos)
        print(f"Published message '{value}' to topic '{publish_topic}'\n
            ")

        # Wait for a moment to simulate some client activity
        time.sleep(6)

except KeyboardInterrupt:
    # Disconnect from the MQTT broker
    pass
client.loop_stop()
client.disconnect()

print("Disconnected from the MQTT broker")
```

Wireshark Packet Capture

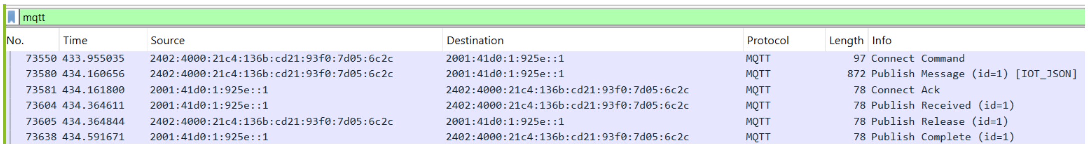

Figure 1: Packet Capturing at Publisher

Observations

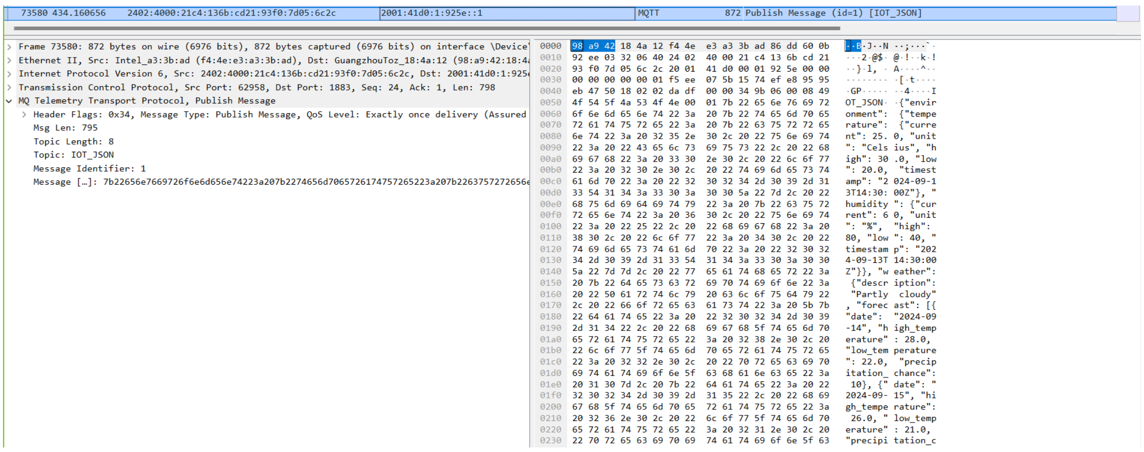

Figure 2: Payload of the Published Message

1. Payload Format: The payload is successfully transmitted as a JSON object. The IOT_JSON topic contains structured data with key-value pairs, making it useful for

IoT monitoring or control applications.

2. Correct Functionality: From the packet capture, it seems your MQTT client is functioning as expected—reading the JSON object, publishing it to the topic, and ensuring that it reaches the broker with no packet errors.

3.2 Subscriber Code

The below code represents the MQTT subscriber which receives an object containing multiple string-value pairs that we have published and writes it to a file called `data_received.json`.

```
from paho.mqtt import client as mqtt_client
import paho.mqtt.client as mqtt
import time
write_file_name = "data_received.json"
import json

# Callback when the client connects to the MQTT broker
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("Connected to MQTT broker")
        client.subscribe(subscribe_topic, qos)  # Subscribe to the
            receive topic
    else:
        print("Connection failed with code {rc}")

# Callback when a message is received from the subscribed topic
def on_message(client, userdata, msg):
    print ("Message received " + "on "+ subscribe_topic + ": "  + str(
       msg.payload.decode("utf-8")))
    value = str(msg.payload.decode("utf-8"))
    with open(write_file_name, 'w') as json_file:
        json.dump(value, json_file, indent=4)  # The 'indent' parameter
            adds pretty formatting
        print("Data has been written to", write_file_name)



# Create an MQTT client instance
client = mqtt.Client(mqtt_client.CallbackAPIVersion.VERSION1,"PythonSub"
    )

# Set the callback functions
client.on_connect = on_connect
client.on_message = on_message

# Connect to the MQTT broker
broker_address = "test.mosquitto.org"  # broker's address
broker_port = 1883
keepalive = 60
qos = 2

subscribe_topic = input ('Enter the topic to subscribe to: ')
client.connect(broker_address, broker_port, keepalive)

# Start the MQTT loop to handle network traffic
client.loop_start()

# Subscribe loop
```

```python
try:
    while True:
        time.sleep(6)

except KeyboardInterrupt:
    # Disconnect from the MQTT broker
    pass
client.loop_stop()
client.disconnect()

print("Disconnected from the MQTT broker")
```

Wireshark Packet Capture

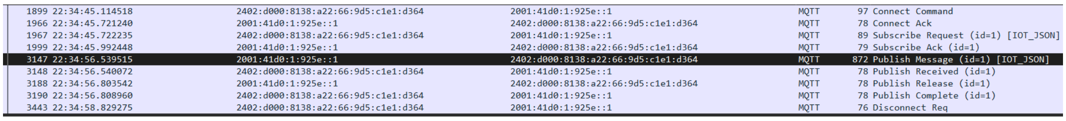

Figure 3: Packet Capturing at Subscriber

Observations

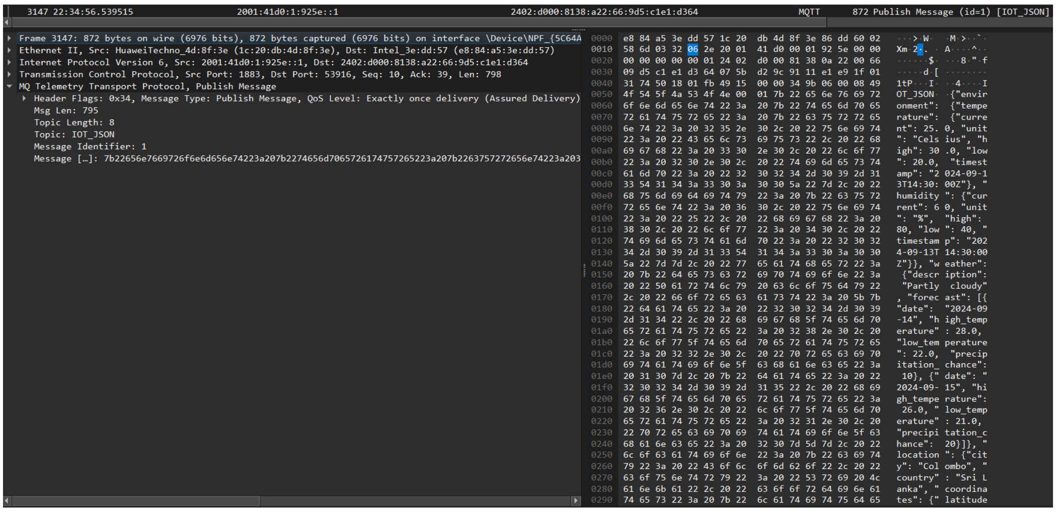

Figure 4: Payload Received at Subscriber

1. The packet flow follows the typical sequence for MQTT with QoS 2 (Exactly Once Delivery). Wireshark confirms the MQTT operations such as Connect, Publish, Publish Received (PUBREC), Publish Release (PUBREL), Publish Complete (PUBCOMP), and the receipt of messages. This ensures that the message is delivered exactly once, preventing duplicates.

2. The payload being transmitted is structured as a JSON object. Wireshark successfully captured and decoded the payload, showing its contents clearly in the hex view.

3. Since the payload matches the expected JSON structure, we can confirm that the data integrity is preserved across the MQTT communication, even with the additional steps involved in QoS 2 to ensure reliable message delivery.

3.3 Discussion

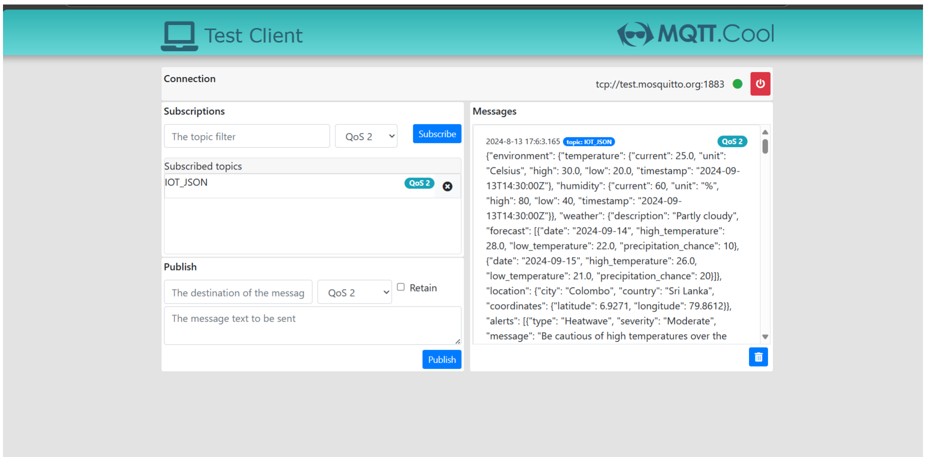

Figure 5: Test Client

The above tasks demonstrated the successful use of MQTT for publishing and subscribing to JSON data. The JSON format's flexibility and readability made it an ideal choice for transmitting structured sensor data. The use of Wireshark confirmed that the MQTT messages, including JSON payloads, were correctly transmitted and received. Both publishing and subscribing processes successfully handled the JSON format, showing its suitability for transmitting complex data structures in IoT scenarios.(Objects containing multiple string-value pairs)

4 Step 04

4.1 Open Weather Code

```
import requests
import json

write_file_name = "openweathermap_data.json"

def get_weather(city_name, api_key):
    base_url = "http://api.openweathermap.org/data/2.5/weather"

    # Parameters for the API request
    params = {
        'q': city_name,         # City name
        'appid': api_key,        # API key
        'units': 'metric'        # Units for temperature (metric,
            imperial, or standard)
    }

    # Make the HTTP GET request to the OpenWeather API
    response = requests.get(base_url, params=params)

    # Check if the request was successful
    if response.status_code == 200:
        # Parse JSON data from the response
        data = response.json()

        # Write the JSON data to a file
        with open(write_file_name, 'w') as json_file:
            json.dump(data, json_file, indent=4)  # The 'indent'
                parameter adds pretty formatting

        print("Data has been written to", write_file_name)

        # Extract relevant information from the response
        main = data['main']
        weather = data['weather'][0]
        wind = data['wind']

        # Print weather information
        print(f"Weather in {city_name.capitalize()}:")
        print(f"Temperature: {main['temp']} C ")
        print(f"Humidity: {main['humidity']}%")
        print(f"Condition: {weather['description'].capitalize()}")
        print(f"Wind Speed: {wind['speed']} m/s")
    else:
        # Print error message if request was unsuccessful
        print(f"Error: Could not retrieve weather for '{city_name}' (
            HTTP {response.status_code})")
        print(f"Response Text: {response.text}")

if __name__ == "__main__":
```

```
    # Your OpenWeather API key
    api_key = "17c3824f1382e7309370c91d8173294c"

    # Get city name from user
    city_name = input("Enter the name of the city: ")

    # Fetch and display weather information
    get_weather(city_name, api_key)
```

The above Python code fetches current weather data for a user-specified city using the OpenWeather API. It saves the data in a JSON file and prints the weather details, including temperature, humidity, weather conditions, and wind speed. If the API request fails, it displays an error message.

The results obtained:

Figure 6: Openweather Results

We have written a json file containing the obtained results as follows:

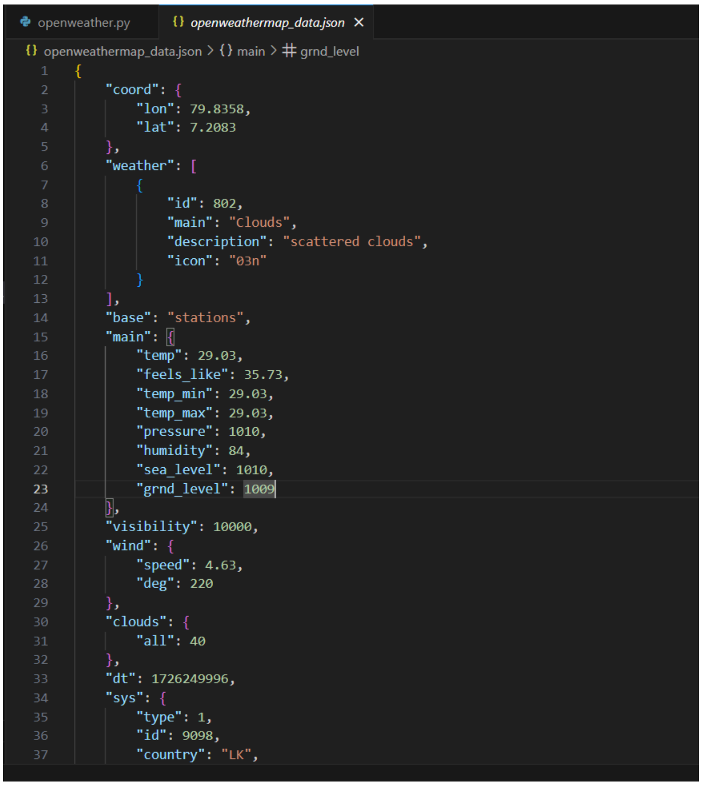
openweather.py   openweathermap_data.json
openweathermap_data.json > {} main > # grnd_level
```
{
    "coord": {
        "lon": 79.8358,
        "lat": 7.2083
    },
    "weather": [
        {
            "id": 802,
            "main": "Clouds",
            "description": "scattered clouds",
            "icon": "03n"
        }
    ],
    "base": "stations",
    "main": {
        "temp": 29.03,
        "feels_like": 35.73,
        "temp_min": 29.03,
        "temp_max": 29.03,
        "pressure": 1010,
        "humidity": 84,
        "sea_level": 1010,
        "grnd_level": 1009
    },
    "visibility": 10000,
    "wind": {
        "speed": 4.63,
        "deg": 220
    },
    "clouds": {
        "all": 40
    },
    "dt": 1726249996,
    "sys": {
        "type": 1,
        "id": 9098,
        "country": "LK",
```

Figure 7: Generated Json file data obtained by Open Weather

5 Homework

5.1 Using Publisher Message for Node-Red

By using the following python code we can obtained the weather data from `https://openweathermap.org` and publish the message to MQTT In in the Red-Note.

```
import requests
import json
from paho.mqtt import client as mqtt_client
import paho.mqtt.client as mqtt
import time

def get_weather(city_name, api_key):
    base_url = "http://api.openweathermap.org/data/2.5/weather"

    # Parameters for the API request
    params = {
        'q': city_name,          # City name
        'appid': api_key,        # API key
        'units': 'metric'        # Units for temperature (metric, imperial
            , or standard)
    }
```

```
    # Make the HTTP GET request to the OpenWeather API
    response = requests.get(base_url, params=params)

    print(f"Response Status Code: {response.status_code}")

    # Check if the request was successful
    if response.status_code == 200:
        # Parse JSON data from the response
        data = response.json()

        # Extract relevant information from the response
        main = data['main']
        weather = data['weather'][0]
        wind = data['wind']

        # Create a dictionary with weather data for the city
        weather_data = {
            'city': city_name.capitalize(),
            'temperature': main['temp'],
            'humidity': main['humidity'],
            'condition': weather['description'].capitalize(),
            'wind_speed': wind['speed']
        }

        return weather_data
    else:
        # Return an error message if the request was unsuccessful
        print(f"Error: Could not retrieve weather for '{city_name}' (
            HTTP {response.status_code})")
        return None

def main():
    #OpenWeather API key
    api_key = "79d08796032858d8504f2f893c77115f"

    # Get weather data for 3 cities
    cities = ["Ratnapura", "Nugegoda", "Negombo"]

    # Initialize an empty list to hold weather data for all cities
    weather_data_list = []

    for city in cities:
        city_weather = get_weather(city, api_key)
        if city_weather:
            weather_data_list.append(city_weather)

    # Store weather data directly
    weather_data_json = json.dumps(weather_data_list, indent=4)

    # MQTT client setup
    def on_connect(client, userdata, flags, rc):
        if rc == 0:
            print("Connected to MQTT broker\n")
        else:
```

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

    client = mqtt.Client(mqtt_client.CallbackAPIVersion.VERSION1, "
        PythonPub")
    client.on_connect = on_connect

    broker_address = "test.mosquitto.org"  # Broker's address
    broker_port = 1883
    keepalive = 60
    qos = 2
    publish_topic = "IOT_JSON"

    # Connect to the MQTT broker
    client.connect(broker_address, broker_port, keepalive)

    # Start the MQTT loop to handle network traffic
    client.loop_start()

    try:
        while True:
            # Publish the weather data directly
            client.publish(publish_topic, weather_data_json, qos=qos)
            print(f"Published message '{weather_data_json}' to topic '{
                publish_topic}'\n")

            # Wait for a moment to simulate some client activity
            time.sleep(6)

    except KeyboardInterrupt:
        # Disconnect from the MQTT broker
        pass

    client.loop_stop()
    client.disconnect()

    print("Disconnected from the MQTT broker")

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

5.2 Direct Implementation within Node-Red

Node-Red Flow Diagram

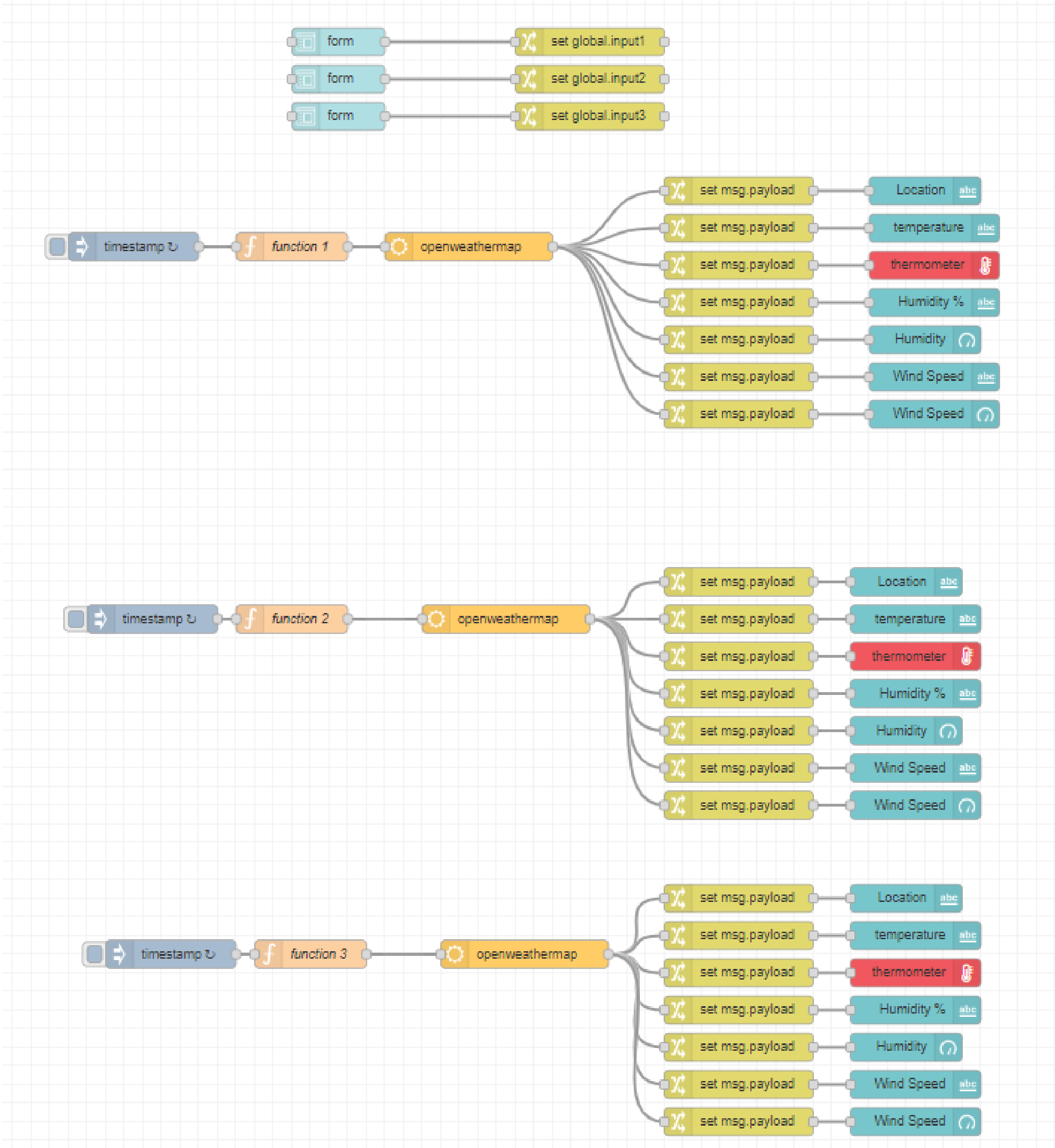

Figure 8: Node-Red Flow

Dash Board

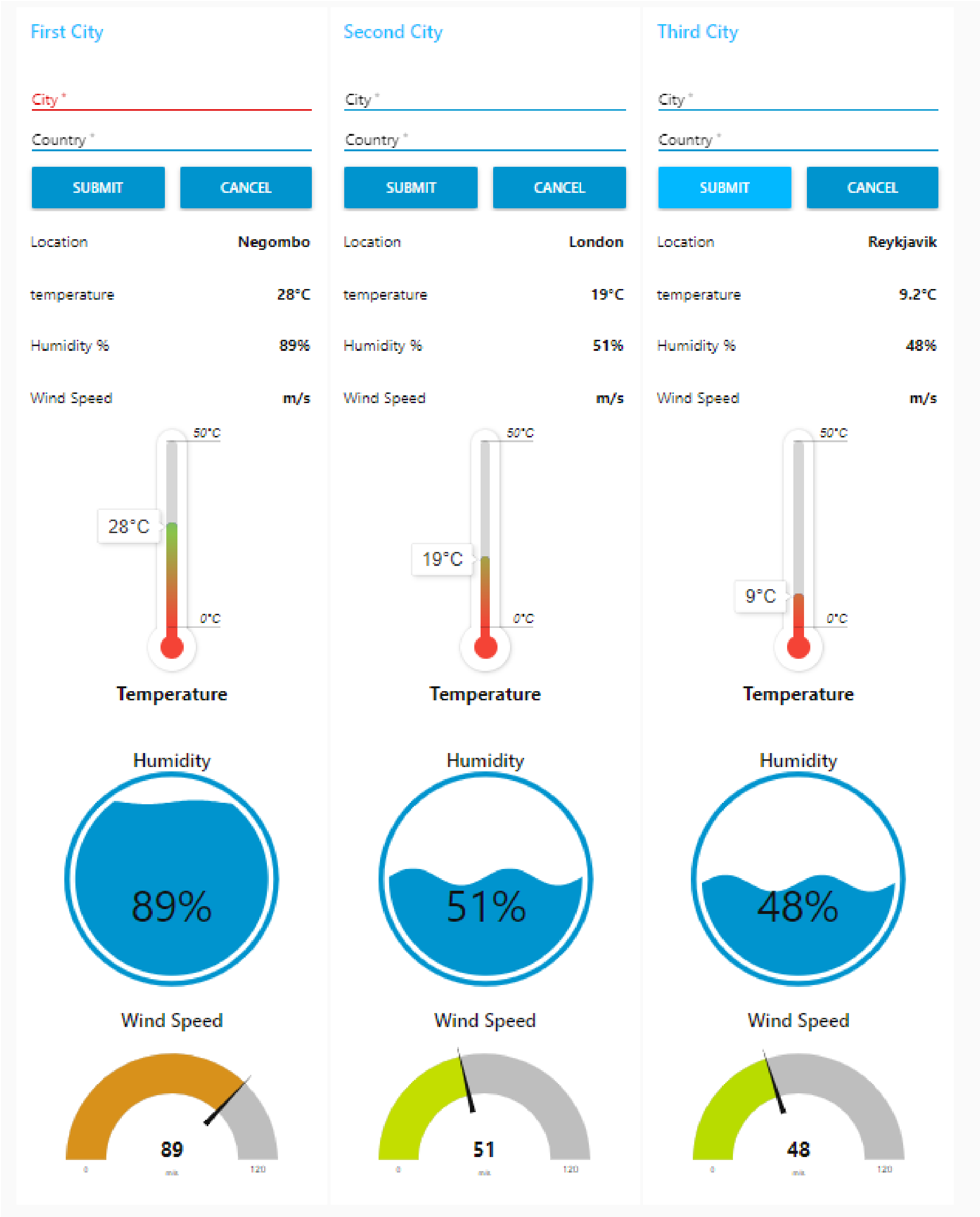

Figure 9: Dash Board

Implementation Summery

Our Node-RED dashboard integrated with the OpenWeatherMap API, designed to display real-time weather data for multiple cities. The dashboard provides a clear, graphical representation of temperature, humidity, and wind speed, utilizing interactive elements such as thermometers, humidity gauges, and wind speed dials. The underlying Node-RED flow connects input forms to the API, where user-specified city data is processed. Timestamp nodes initiate API requests, and function nodes handle the retrieval and formatting of weather data, which is then routed to various output widgets on the dashboard. This setup allows for dynamic weather monitoring, offering users an intuitive interface for accessing environmental conditions from the OpenWeatherMap service.