WeatherI/O

IoT project for displaying hourly weather to a user on web app or RPi

Joe Crichton

CMP408: IoT and Cloud Secure Development

BSc Ethical Hacking Year 4

2023/24

Note that Information contained in this document is for educational purposes.

.

Contents

.

1 INTRODUCTION

Collection and presentation of data is important to businesses and individuals alike. Digestible, informative data is often a benefit to user and provider alike to allow for engagement or actionable information (Analytics Training Hub, no date). Automated collection and handling of data from API's is no different.

This project aims to collect hourly weather data from an API and present it to the user. The collection will be done automatically every hour by AWS services, and the data transmitted to a Raspberry Pi Zero W (RPi) using MQTT traffic. The RPi will then display the current precipitation data to the user via LEDs on a breadboard. The user can also view collected weather data on a web application.

1.1 RELEVANCE TO IOT & CLOUD SECURITY

This project is relevant to some areas of IoT and cloud security.

First, the use of a LKM (Loadable Kernel Module) on Linux allows for the controlling of the RPi's GPIO pins. Due to the privileged running of LKMs, they can cause system instability or vulnerabilities within the IoT device: therefore, the implementation of the LKM needs to not cause either of these issues.

Second, secure communication systems with the cloud are a relevant security aspect. This project uses secure communications to and from the cloud. This makes internet traffic far harder to tamper with (Cloudflare, no date).

1.2 OBJECTIVES

The objective of this project is to create an IoT project with a cloud, software and hardware component to display weather information to a user. The objectives of the project are to:

- Use the cloud to provide an automated weather data collection every hour
- Host a web application on the cloud which will allow a user to view this weather data
- Establish communications between the cloud and an IoT device
- Use the IoT device's hardware to display or indicate weather data from the cloud

2 PROCEDURE

2.1 PROJECT STRUCTURE

The implementation of the project was split into phases as shown below in table 1-1, with the components for each show in tables 1-2, 1-3 and 1-4. Figure 1-1 below shows the overall project structure.

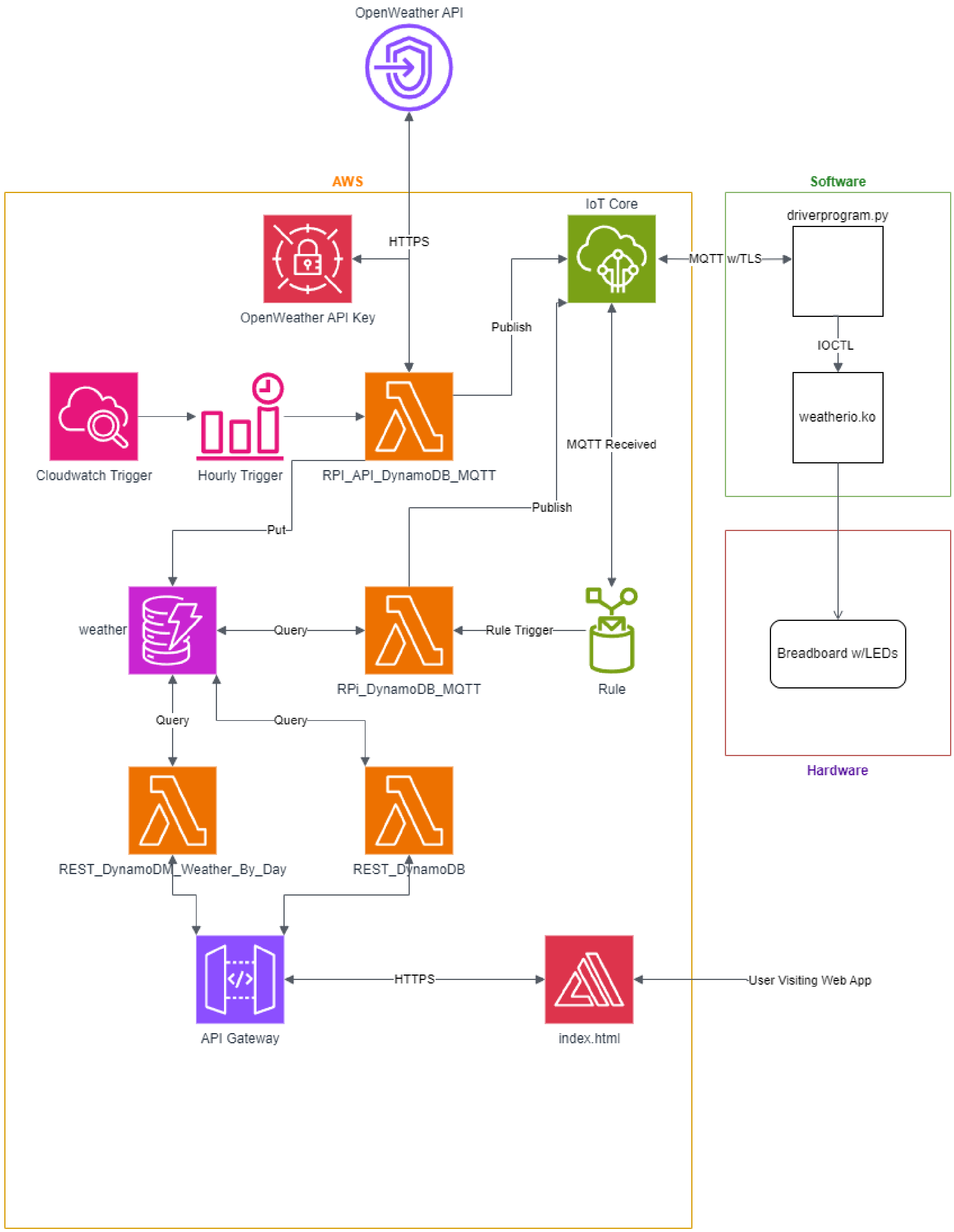

Figure 1-1: Overall Project Diagram

The project was developed by first implementing the overall phases of the project, and then developing the integrations between the components of each separate phase e.g IoT core to driverprogram.py.

Code for components can be found in the relevant appendix at the end of the report.

Table 1-1: Implementation Phases

Section	Phase
2.2	AWS (Cloud)
2.3	Software
2.4	Hardware

Table 1-2: AWS (Cloud) Components

AWS Service	Purpose
IoT Core	Broker for MQTT traffic between RPi and AWS
DynamoDB	NoSQL, key- value database used for storing weather data
Lambda	Serverless code execution for fetching/ retrieving data for DynamoDB across multiple functions
Secrets Manager	Hides sensitive information within code by calling the real information from the Secrets Manager service
Cloudwatch	Hourly trigger for a Lambda function
Amplify	Serverless web hosting for WeatherI/O web application
API Gateway	API management service used with Amplify to call Lambda functions with HTTPS requests

Table 1-3: Software Components

Software Component	Purpose
driverprogram.py	Userspace program for making system calls to ioctl of device driver after receiving MQTT data from AWS IoT Core
weatherio.ko	LKM device driver for interacting with GPIO pins of RPi

Table 1-4: Hardware Components

Hardware Components	Purpose
Breadboard wired to RPi GPIO pins	LEDs on breadboard light up for the used to indicate the data received from AWS IoT Core

2.2 AWS (CLOUD)

All the following steps in this section were done in the AWS Learner Lab environment

A DynamoDB database was used to store weather data, as it features low latency queries and is NoSQL (Amazon, no date. A) for the JSON data that will be passed to it.

Four Lambda functions would be used for various operations, as they are a serverless on-demand service (Amazon, no date. B), which suits the stipulations of hourly or client instantiated operations. The Lambda functions were written in Node.js for its event driven, I/O operations (freeCodeCamp, 2017) which were again suited to the demands of a service which runs at scheduled times. The first of these functions, RPI_API_DynamoDB_MQTT, requests the current weather from the OpenWeather API using HTTPS. It then formats the response, inserts the data into the DynamoDB table, and sends the JSON data over MQTT to the topic (this was the function that was run hourly). The second, RPI_DynamoDB_MQTT, queries the latest instance if the weather from the DynamoDB table and sends the JSON over MQTT. This was triggered by an IoT Core rule, as shown below in figure 2-1.

SQL statement

SQL statement

```
SELECT * FROM 'mqtt/weather408' WHERE message = "REQ"
```

Figure 2-1: IoT Core Rule

The third, REST_DynamoDB, functions the same as the second except instead of MQTT, the JSON is sent in a HTTPS response. The fourth (REST_DynamoDM_Weather_By_Day) is again the same, but queries for all weather items using a user supplied date and returns JSON over HTTPS.

Amplify was used to host a web page for the project, which featured a form that a user could submit a date which would return the recorded weather for that date using AWS API Gateway with AWS Lambda. The current weather would also be retrieved. Amplify allowed for serverless hosting.

AWS IoT core would be used as the MQTT broker for the project, ensuring secure communications with IoT devices.

2.3 SOFTWARE

A python userspace program was used to call driver functions. Python was chosen for the userspace program as an AWS SDK for interacting with IoT devices utilises python, and python libraries are available for interacting with IOCTL function calls.

The userspace program, when launched, is supplied with 3 pin numbers as arguments. The program will create pin structs using ctypes (for compatibility with the LKM, written in C), which will be the same structure found in the character driver.

The userspace program subscribes to the MQTT topic and sends a request for the latest weather item. It then listens for returning MQTT messages. As the program is subscribed to the MQTT topic, it will also receive hourly weather messages.

The program will then parse the JSON received from these MQTT messages and initiate an IOCTL call to the LKM driver weatherio to set the pins on the hardware according to the precipitation levels in the MQTT message that was received. The program does this by passing the pin and the function to be run to an IOCTL call.

An LKM char device driver allows for setting the RPi pins high and low. A char device was chosen as an efficient way of controlling devices. The source for this LKM is lab code but has been modified to allow for memory allocation of multiple pins, which allows for proper cleanup of kernel memory.

2.4 HARDWARE

A breadboard with three LEDs connects to the RPi GPIO pins as shown below in figure 2-3. Note the specific GPIO pins are set by the user when driverprogram.py is called.

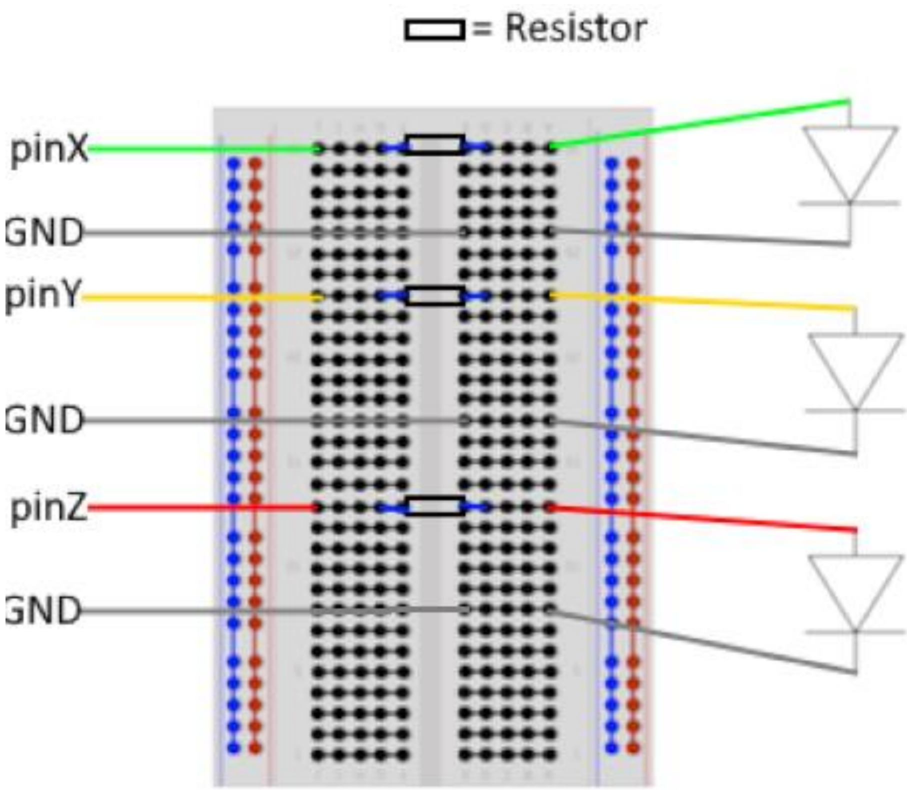

Figure 2-3: Breadboard Layout

2.5 SECURITY HIGHLIGHTS

The LKM driver, weatherio, does not modify any existing system calls for kernel safety.

The MQTT data transmitted by the RPi makes use of a TLS encrypted connection with AWS. This makes intercepted data harder to decode (HiveMQ, 2015).

As the web application is hosted on AWS amplify, all HTTP traffic to and from the web app is HTTPS. This is more secure than using HTTP.

The web application makes use of input filtering when the API calls the Lambda function. The Lambda will not return successfully if an unexpected character is detected. Similarly, if the website receives a response with an unexpected character, it will drop the response. These measures should help prevent Cross-Site-Scripting or Man-in-the-Middle attacks.

AWS Secrets Manager was used to encrypt the API key used for OpenWeather in code.

3 DISCUSSION

3.1 SUMMARY

The project uses AWS to perform weather data collection from an API and display that information to the user through a web application. This meets two objectives of the project with additional security measures identified previously to the project's benefit.

AWS communicates with a python program running on an RPi using MQTT with TLS, which meets another of the objectives of the project with the positive factor of added security.

Finally, the RPi indicates precipitation data visually for the user which meets another project objective. This is done using a userspace program and an LKM driver, which allows for efficient low-level control of the RPi’s GPIO pins.

3.2 FUTURE WORK

All the project objectives were met, but several could be improved by progressing the work further.

The hardware output for the user indicates the current precipitation level for the user. This could be explored by implementing other systems, such as a button that when pressed changes which part of the received data is indicated by the RPi LEDs.

The web application for the user could also be improved by using a more responsive UI rather than a simple form that returns the requested data: improvements such as dropdown menus, calendars or other systems.

REFERENCES

Amazon (no date, A) *Amazon DynamoDB.* Available at: https://aws.amazon.com/dynamodb/ (Accessed: 13 December 2023)

Amazon (no date, B) *AWS Lambda.* Available at: https://aws.amazon.com/lambda/ (Accessed: 13 December 2023)

Analytics Training Hub (no date) *Data Presentation – Types & Its Importance in Data Analytics.* Available at: https://analyticstraininghub.com/data-presentation-types-importance/ (Accessed: 12 December 2023)

Cloudflare (no date) *Why use HTTPS?.* Available at: https://www.cloudflare.com/learning/ssl/why-use-https/ (Accessed: 12 December 2023)

freeCodeCamp (2017) *What exactly is Node.js and why should you use it?.* Available at: https://www.freecodecamp.org/news/what-exactly-is-node-js-and-why-should-you-use-it-8043a3624e3c/ (Accessed: 12 December 2023)

HiveMQ (2015) *TLS/SSL - MQTT Security Fundamentals.* Available at: https://www.hivemq.com/blog/mqtt-security-fundamentals-tls-ssl/ (Accessed: 13 December 2023)

APPENDICES PART 1- LAMBDA FUNCTIONS

RPi_API_DynamoDB_MQTT

```
// DynamoDB Setup for Node.js
import { DynamoDBClient, PutItemCommand } from '@aws-sdk/client-dynamodb';
// Module for MQTT Stream
import { IoTDataPlaneClient, PublishCommand } from "@aws-sdk/client-iot-data-plane";
const REGION = 'us-east-1';
// New DynamoDB client
const dynamo = new DynamoDBClient({
  region: REGION,
});
// New IoTDataPlaneClient
const client = new IoTDataPlaneClient({
  region: 'us-east-1',
});
// Secret client import
import {
  SecretsManagerClient,
  GetSecretValueCommand,
} from "@aws-sdk/client-secrets-manager";

// Lambda handler
export const handler = async (event, context) => {

    const secret_name = "OpenWeatherAPI-Key";

    const secretClient = new SecretsManagerClient({
        region: "us-east-1",
    });

    let response;

    try {
        response = await secretClient.send(
        new GetSecretValueCommand({
            SecretId: secret_name,
            VersionStage: "AWSCURRENT",
            })
        );
        } catch (error) {
            throw error;
        }
    // Parse API key from JSON secret
    const secret = response.SecretString;
```

```
const secretJson = JSON.parse(secret);
const apiKey = secretJson.OpenWeatherAPI;

// Use fetch to make HTTPS request
const json = await (
  await fetch(
    apiKey,
    {
      headers: {
        "Content-Type": "application/json"
      }
    }
  )
).json();
console.log("From fetch", json);
console.log(json.dt);
console.log("Generating date");
// Generate current day-month-year
const date = new Date();
const day = date.getDate();
const month = date.getMonth() + 1;
const year = date.getFullYear();
const currentDate = `${day}-${month}-${year}`;
console.log("Date generated");

console.log("Making Dynamo parameters");
// Use the JSON values for DynamoDB parameters
const params = {
  TableName: 'weather',
  Item: {
  'date': {S: `${currentDate}`},
  'dt': {N: `${json.dt}`},
  'id': {N: `${json.weather[0].id}`},
  'temperature': {N: `${json.main.temp}`},
  'humidity': {N: `${json.main.humidity}`},
  'wind': {N: `${json.wind.speed}`},
  'desc': {S: `${json.weather[0].description}`},
  }
};
console.log("Parameters made");
console.log("Sending to dynamo");
// Place item into DynamoDB

dynamo.send(new PutItemCommand(params), (err, data) => {
  if (err) {
    console.log('Down here!');
    console.error(err);
```

```
        } else {
        console.log(data);
        }
    });


    console.log("Dynamo sent");
    console.log("Gerating JSON");
    // Generate new JSON payload
    const json_payload = {

        'dt' : json.dt,
        'id': json.weather[0].id,
        'temperature': json.main.temp,
        'humidity': json.main.humidity,
        'wind': json.wind.speed,
        'desc': json.weather[0].description,

    };
    console.log("JSON Generated");
    console.log("Sending to MQTT");
    // Send to MQTT Stream
    const input = {
        topic: "mqtt/weather408",
        payload: JSON.stringify(json_payload),
        payloadFormatIndicator: "UNSPECIFIED_BYTES" || "UTF8_DATA",
        responseTopic: "mqtt/testing",
    };
    // Send payload
    const command = new PublishCommand(input);
    await client.send(command);
    console.log("MQTT sent");

    // End request
    return;
};
```

RPi_DynamoDB_MQTT

```
// DynamoDB Setup for Node.js
import { DynamoDBClient, QueryCommand} from "@aws-sdk/client-dynamodb";
// Module for MQTT Stream
import { IoTDataPlaneClient, PublishCommand } from "@aws-sdk/client-iot-data-plane";
const REGION = 'us-east-1';
// New DynamoDB client
const dynamo = new DynamoDBClient({
  region: REGION,
```

```
});
//New IoTDataPlaneClient
const client = new IoTDataPlaneClient({
  region: 'us-east-1',
});

// Lambda handler
export const handler = async (event, context) => {

    // Generate current day-month-year
    const date = new Date();
    const day = date.getDate();
    const month = date.getMonth() + 1;
    const year = date.getFullYear();
    const currentDate = `${day}-${month}-${year}`;


    // Query weather table using partition key (date) to find largest 'dt' in that partition
    const params = {
        TableName: "weather",
        KeyConditionExpression: "#date = :date",
        // Have to use this expression to alias 'date' to '#date' as 'date' is a reseved keyword- will throw
error otherwise
        ExpressionAttributeNames: {
            "#date": "date",
        },
        ExpressionAttributeValues: {
            ":date": { S: currentDate },
        },
        // ScanIndexForward: false will return the partition key, sorted by sort key, in descending order
        // Therefore, the largest 'dt' (which will be the latest record of the weather) will be the first item
returned
        ScanIndexForward: false,
        // This limits the return to 1 item; which will be the latest weather
        Limit: 1,
    };

    // Send Query command
    const query = new QueryCommand(params);
    const data = await dynamo.send(query);
    console.log(data);

    // Generate JSON payload from return data
    const json_payload = {

        'dt' : data.Items[0].dt.N,
        'id': data.Items[0].id.N,
        'temperature': data.Items[0].temperature.N,
```

```
      'humidity': data.Items[0].humidity.N,
      'wind': data.Items[0].wind.N,
      'desc': data.Items[0].desc.S,

    };

    // Send to MQTT Stream
    const input = {
      topic: "mqtt/weather408",
      payload: JSON.stringify(json_payload),
      payloadFormatIndicator: "UNSPECIFIED_BYTES" || "UTF8_DATA",
      responseTopic: "mqtt/testing",
    };
    // Send payload
    const command = new PublishCommand(input);
    await client.send(command);

    // End request
    return;
};
```

REST_DynamoDB

```
// DynamoDB Setup for Node.js
import { DynamoDBClient, QueryCommand} from "@aws-sdk/client-dynamodb";
const REGION = 'us-east-1';
// New DynamoDB client
const dynamo = new DynamoDBClient({
  region: REGION,
});

// Lambda handler
export const handler = async (event, context) => {

    // Generate current day-month-year
    const date = new Date();
    const day = date.getDate();
    const month = date.getMonth() + 1;
    const year = date.getFullYear();
    const currentDate = `${day}-${month}-${year}`;


    // Query weather table using partition key (date) to find largest 'dt' in that partition
    const params = {
      TableName: "weather",
      KeyConditionExpression: "#date = :date",
```

```
        // Have to use this expression to alias 'date' to '#date' as 'date' is a reseved keyword- will throw
error otherwise
        ExpressionAttributeNames: {
          "#date": "date",
        },
        ExpressionAttributeValues: {
          ":date": { S: currentDate },
        },
        // ScanIndexForward: false will return the partition key, sorted by sort key, in descending order
        // Therefore, the largest 'dt' (which will be the latest record of the weather) will be the first item
returned
        ScanIndexForward: false,
        // This limits the return to 1 item; which will be the latest weather
        Limit: 1,
    };

    // Send Query command
    const query = new QueryCommand(params);
    const data = await dynamo.send(query);
    console.log(data);

    // Generate JSON payload from return data
    const json_payload = {

        'dt' : data.Items[0].dt.N,
        'id': data.Items[0].id.N,
        'temperature': data.Items[0].temperature.N,
        'humidity': data.Items[0].humidity.N,
        'wind': data.Items[0].wind.N,
        'desc': data.Items[0].desc.S,

    };


    // Return the JSON as a HTTP respons
    return {
        statusCode: 200,
        body: JSON.stringify(json_payload),
    };

};
```

REST_DynamoDM_Weather_By_Day

```

// DynamoDB Setup for Node.js
import { DynamoDBClient, QueryCommand} from "@aws-sdk/client-dynamodb";
const REGION = 'us-east-1';
```

```
// New DynamoDB client
const dynamo = new DynamoDBClient({
  region: REGION,
});

// Lambda handler
export const handler = async (event, context) => {

   // Generate target day-month-year, which will have been passed to lambda
   // Return an invalid response if request contained anything other than numbers
   // Prevents potential code inejection attacks
   if(/[^0-9]/.test(event.day) || /[^0-9]/.test(event.month) || /[^0-9]/.test(event.year)){
   console.log("Invalid character detected in request");
   console.log(event);
      return {
            statusCode: 400,
            body: JSON.stringify({
            message: "Invalid character in request",
            }),
         };
}else{
   console.log("No invalid characters found");
}

   const day = event.day;
   const month = event.month;
   const year = event.year;
   const targetDate = `${day}-${month}-${year}`;

   // Query weather table using partition key (date) to find largest 'dt' in that partition
   const params = {
      TableName: "weather",
      KeyConditionExpression: "#date = :date",
      // Have to use this expression to alias 'date' to '#date' as 'date' is a reseved keyword- will throw
error otherwise
      ExpressionAttributeNames: {
         "#date": "date",
      },
      ExpressionAttributeValues: {
         ":date": { S: targetDate },
      },
      // ScanIndexForward: false will return the partition key, sorted by sort key, in descending order
      // Therefore, the largest 'dt' (which will be the latest record of the weather) will be the first item
returned
      ScanIndexForward: false,
   };

   // Send Query command
```

```
    const query = new QueryCommand(params);
    const data = await dynamo.send(query);
    // Handle no return data
    if (data.Items.length == 0) {
      return {
            statusCode: 404,
            body: JSON.stringify({
              message: "No weather data for this date",
            }),
          };
      }

    // Generate JSON payload from return data, using map to iterate through the returned items
    const json_payload = data.Items.map(item => ({

      'dt' : item.dt.N,
      'id': item.id.N,
      'temperature': item.temperature.N,
      'humidity':item.humidity.N,
      'wind': item.wind.N,
      'desc': item.desc.S,

    }));

    // Return the JSON as a HTTP respons
    return {
      statusCode: 200,
      body: JSON.stringify(json_payload),
    };

  };
```

APPENDICES PART 2- AMPLIFY APP

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<!-- viewport and style tags, disable use in iframe -->
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>WeatherI/O</title>
  <style>
    h1, h2, h3, p, label {
      font-family: Arial, sans-serif;
      margin-left: 20px;
    }
    button {
      color: #ABB8C3
      font-family: Arial, sans-serif;
      font-size: 20px;
      font-weight: bold;
      margin-left: 30px;
      margin-top: 20px;
      width: 140px;
    }
    input {
      color: #222629;
      font-family: Arial, sans-serif;
      font-size: 20px;
      margin-left: 10px;
      margin-top: 20px;
      width: 100px;
    }
  </style>
</head>
<script>
  // getCurrentWeather
  async function getCurrentWeatherRequest() {
try{
      // Headers
      var myHeaders = new Headers();
      // Specify content type
      myHeaders.append("Content-Type", "application/json");

      // create a JSON object with parameters for API call and store in a variable
      var requestOptions = {
        method: 'GET',
```

```
            headers: myHeaders,
            redirect: 'follow'
          };

          // make API call with parameters and use promises to get response
          const response = await fetch("https://ha2kz5qsb5.execute-api.us-east-
1.amazonaws.com/beta/current_weather", requestOptions);
          const data = await response.json();
const dataBody = JSON.parse(data.body);

          // Display weather information
// Use += to not overwrite
          const currentWeatherDisplayDiv = document.getElementById('currentWeatherDisplay');
          currentWeatherDisplayDiv.innerHTML += `
<p>Description: ${dataBody.desc}</p>
<p>Temperature: ${dataBody.temperature}C</p>
<p>Wind: ${dataBody.wind}m/s</p>
            <p>Humidity: ${dataBody.humidity}%</p>
          `;
       } catch (error) {
          console.error('Error fetching weather data:', error);
       }
    }
    window.onload = function () {
       getCurrentWeatherRequest();
    };
    </script>
<script>
// getTargetWeatherRequest
async function getTargetWeatherRequest(day,month,year){
try{
// Use repalce() to knock out leading 0 (as the input is limited to 2 digits for day and month)
// Have to use regex to target leading 0's only
day = day.replace(/^0/, '');
month = month.replace(/^0/, '');
// Headers
var myHeaders = new Headers();
// Specify content Type
myHeaders.append("Content-Type", "application/json");
// Generate JSON for sending
var raw = JSON.stringify({"day":day,"month":month,"year":year});
// create a JSON object with parameters for API call and store in a variable
          var requestOptions = {
            method: 'POST',
            headers: myHeaders,
body: raw,
            redirect: 'follow'
          };
```

```
// make API call with parameters and use promises to get response
        const response = await fetch("https://ha2kz5qsb5.execute-api.us-east-
1.amazonaws.com/beta/all_weather", requestOptions);
        const data = await response.json();
const dataBody = JSON.parse(data.body);
console.log(dataBody);
// Detect if no weather has been returned and return function to avoid errors
if(dataBody.message == "No weather data for this date" || dataBody.message == "Invalid character in
request"){
return;
}
// Display weather information
// Use += to not overwrite
        const dayWeatherDisplayDiv = document.getElementById('dayWeatherDisplay');
// Clear div
dayWeatherDisplayDiv.innerHTML = '';
//Iterate through returned JSON object and find data
dataBody.forEach(item =>{
// Check each element for anything that is not 0-9 for numerics, expected characters for descripton,
0-9 points and hypens for others
// return if there are any
// This mitigates proxy/MITM for incoming data
// Can use .test() from built in JS
if(/[^0-9]/.test(item.dt) || /[^a-zA-Z ]/.test(item.desc) || /[^0-9.-]/.test(item.temperature) || /[^0-
9.]/.test(item.wind) || /[^0-9]/.test(item.humidity)){
console.log("Invalid character detected in response, cancelling!");
console.log(item);
return;
}else{
console.log("No invalid characters found");
}
// Set timestamp for Date()
// * 1000 for milliseconds
var timestamp = item.dt * 1000;
// Get date and times using dt item that has been recieved
// Get dd/mm/yyyy from Date()
// Add 0's and slice to format nicely
var date = new Date(timestamp)
var year = date.getFullYear();
var month = ('0' + (date.getMonth() + 1)).slice(-2);
var day = ('0' + date.getDate()).slice(-2);
// Get hours and minutes from dates
var hours = ('0' + date.getHours()).slice(-2);
var minutes = ('0' + date.getMinutes()).slice(-2);
// Place data into div
dayWeatherDisplayDiv.innerHTML += `
<h3 style="background-color: #ABB8C3;">${hours}:${minutes} on ${day}/${month}/${year}</h3>
```

```
<p>Description: ${item.desc}</p>
<p>Temperature: ${item.temperature}C</p>
<p>Wind: ${item.wind}m/s</p>
<p>Humidity: ${item.humidity}%</p>
`;
});

} catch (error) {
        console.error('Error fetching weather data:', error);
      }
}
</script>
<body>
  <h1>WeatherI/O</h1>
<!--Div for current weather -->
  <div id="currentWeatherDisplay">
<h2 style="background-color: #ABB8C3;">Current Weather Information:</h2>
</div>
<!--Div for requesting weather -->
<div id="dayWeatherForm">
<h2 style="background-color: #ABB8C3;">Retrieve Weather Information:</h2>
<p>Please enter the date of the day you wish to retrieve below:</p>
<form>
<!-- Provide some input security by limiting characters. Will do the rest in Lambda in case of altered
requests -->
      <label>Day:</label>
      <input type="text" id="day" maxlength="2">
      <label>Month:</label>
      <input type="text" id="month" maxlength="2">
<label>Year:</label>
      <input type="text" id="year" maxlength="4">
      <!-- Set button onClick method to call function for reqeusting weather -->
      <button type="button"
onclick="getTargetWeatherRequest(document.getElementById('day').value,document.getElementByI
d('month').value,document.getElementById('year').value)">RETRIEVE</button>
  </form>
</div>
<!--Div for inserting requested weather -->
<div id="dayWeatherDisplay"></div>


</body>
</html>
```

APPENDICES PART 3- SOFTWARE

driverprogram.py

```
import sys
# Import for interacting with IOCTL using c struct, 1 per pin
import ctypes
# Import for ICTOL in Python
import fcntl
# Time for delays
import time
# Import json for handling incoming stream
import json
# Import SDK Package
from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTClient

# Define IOCTL command values
IOCTL_WEATHERIO_GPIO_HIGH = 0x65
IOCTL_WEATHERIO_GPIO_LOW = 0x66

# Path to device file
deviceFile = "/dev/weatheriodev"

# C struct for GPIO pins for driver
# As IOCTL in driver expects a C struct (or size of a c struct) buffer to be passed in
# Can use Python CTypes to make one
class gpio_pin(ctypes.Structure):
_fields_ = [

("label", ctypes.c_char * 16),
("pin", ctypes.c_uint)]

# Function to handle incoming message
def listener_start(topic, qos, message):
    json_message = json.loads(message.payload.decode('utf-8'))

    # Open device file for driver
    try:
            with open(deviceFile, "r") as file:
                    # File descriptor
                fd = file.fileno()
                print("FILE DESCRIPTOR ACCESSED")

                # Beginning of message processing
                print("\n")
                # Check for request message and do not activate if detected
                if 'message' in json_message:
```

```
            print("Request method detected, exiting callback")
            print("--------------")
            return

        # Print incoming message
        print("Received a new message: ")
        print(message.payload)
        print("from topic: ")
        print(message.topic)
        print()
        print("Preparing to set GPIO Pins accoring to weather ID...")

        # Reset all pins
        ret = fcntl.ioctl(fd, IOCTL_WEATHERIO_GPIO_LOW, pin1Struct)
        ret = fcntl.ioctl(fd, IOCTL_WEATHERIO_GPIO_LOW, pin2Struct)
        ret = fcntl.ioctl(fd, IOCTL_WEATHERIO_GPIO_LOW, pin3Struct)

        # Check for ID in incoming payload and set accordingly
        if 'desc' in json_message:
            print("Weather ID detected, proceeding...")
            desc_value = json_message.get("desc")

                # If else chain for weather descriptors, taken from OpenWeathers site
                # Elif so it will 'break' when condition detected and not fire all of them
                # Heavy precipitations from various weather groups
            if 'heavy' in desc_value:
                    print("Heavy intensity precipitation called")
                    ret = fcntl.ioctl(fd, IOCTL_WEATHERIO_GPIO_HIGH, pin1Struct)
                    ret = fcntl.ioctl(fd, IOCTL_WEATHERIO_GPIO_HIGH, pin2Struct)
                    ret = fcntl.ioctl(fd, IOCTL_WEATHERIO_GPIO_HIGH, pin3Struct)
                # Light precipitations from various weather groups
            elif 'light' in desc_value:
                print("light intensity precipitation called")
                ret = fcntl.ioctl(fd, IOCTL_WEATHERIO_GPIO_HIGH, pin1Struct)
                # No precipitation from various weather groups
            elif 'clouds' in desc_value or 'clear' in desc_value:
                print("No precipitation called")
                # Medium precipitation does not have any descriptors, so no if needed
            else:
                ret = fcntl.ioctl(fd, IOCTL_WEATHERIO_GPIO_HIGH, pin1Struct)
                ret = fcntl.ioctl(fd, IOCTL_WEATHERIO_GPIO_HIGH, pin2Struct)
                print("Medium precipitation or atmospheric conditions called")

        print("--------------")
    # If device file could bot be accessesd or found, alert user
    except FileNotFoundError:
        print(f"Error: The file '{deviceFile}' was not found.")
    except Exception as e:
```

```
        print(f"An error occurred: {e}")

def main():
    # Check if the correct number of arguments have been provided
    if len(sys.argv) != 4:
        print("Usage: python script.py lowPin medPin highPin, where any pin uses the BCM numbering")
        return

    # Retrieve command line arguments
    pin1 = int(sys.argv[1])
    pin2 = int(sys.argv[2])
    pin3 = int(sys.argv[3])

    # Print the provided arguments for debug
    print("Low pin:", pin1)
    print("Medium pin:", pin2)
    print("High pin:", pin3)

    # Global GPIO pin structs so listener callback can access data
    global pin1Struct
    global pin2Struct
    global pin3Struct

    pin1Struct = gpio_pin()
    pin2Struct = gpio_pin()
    pin3Struct = gpio_pin()

    # Assign low,medium and high pins (1,2,3 respectivley)
    pin1Struct.pin = pin1
    pin2Struct.pin = pin2
    pin3Struct.pin = pin3

    # Terminal code for debug
    print("IMPORT SUCCESS")
    myMQTTClient = AWSIoTMQTTClient("cmp408_pi")
    print("CLIENT CREATED")
    myMQTTClient.configureEndpoint("a14x7b3xxl15gx-ats.iot.us-east-1.amazonaws.com", 8883)
    print("ENDPOINT CONFIGURED")
    myMQTTClient.configureCredentials("root-CA.crt", "RPi.private.key", "RPi.cert.pem")
    print("CREDS LOADED")
    try:
        myMQTTClient.connect()
    except Exception as e:
        print(f"Connection failed: {e}")
        return
    print("CONNECTED")
```

```python
    # Subscribe to topic then post request
    myMQTTClient.subscribe("mqtt/weather408", 0, listener_start)

    # JSON for weather update
    weatherRequest = {
        "message": "REQ"
    }

    # Publish as JSON
    weatherRequest_json = json.dumps(weatherRequest)
    myMQTTClient.publish("mqtt/weather408", weatherRequest_json, 0)

    # Run program until keyboard interrupt

    try:
        while True:
            pass
    except KeyboardInterrupt:
        print("Exiting")
        myMQTTClient.disconnect()
        return

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

weatherio.c

```c
/*
Name: weatherio.c
Author: Joe Crichton
Description: A lkm driver for use with the CMP408 weather project by Joe Crichton
*/
#include "weatherio.h"

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/gpio.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/slab.h>

static int DevBusy = 0;
static int MajorNum = 100;
static struct class*  ClassName  = NULL;
static struct device* DeviceName = NULL;
// Keep track of requested pins for module cleanup so all pins can be gpio_free'd
```

```
static gpio_pin* allocated_pins[3] = {NULL, NULL, NULL};

gpio_pin targetPin;


static int device_open(struct inode *inode, struct file *file)
{
printk(KERN_INFO "weatherio: device_open(%p)\n", file);

if (DevBusy)
return -EBUSY;

DevBusy++;
try_module_get(THIS_MODULE);
return 0;
}

static int device_release(struct inode *inode, struct file *file)
{
printk(KERN_INFO "weatherio: device_release(%p)\n", file);
DevBusy--;

module_put(THIS_MODULE);
return 0;
}

// Function to check if a pin already exists
// Will return 1 if pin already exists and 0 if not
static int pin_check(int pinNumber, const char *label)
{
// Iterate through allocated_pins array
int i;
for(i = 0; i < 3; i++){
// If pin exists, and has the same number and same label, return 1
if(allocated_pins[i] && allocated_pins[i]->pin == pinNumber && strcmp(allocated_pins[i]->label, label)
== 0){
return 1;
}
}
// Pin did not exist
return 0;
}

// Function to make a copy of a pin
static int make_pin(gpio_pin *pinInput)
{
// If pin check returns 0
if(!pin_check(pinInput->pin, pinInput->label)){
```

```
// Create a new pin in kernel memory
gpio_pin *new_pin = kmalloc(sizeof(gpio_pin), GFP_KERNEL);
if(!new_pin){
// Memory allocation failure
return -ENOMEM;
}
// Set new variables
new_pin->pin = pinInput->pin;
strcpy(new_pin->label, pinInput->label);

// Add pin to the array
int i;
for(i=0; i<3; i++){
// If there is a free space in the array
if(!allocated_pins[i]){
allocated_pins[i] = new_pin;
printk("weatherio: new pin created - pin:%u\n" , new_pin->pin);
return 0;
}
}
// If array is full, free memory
kfree(new_pin);
return -ENOMEM;
}

return 0;

}

static long device_ioctl(struct file *file, unsigned int cmd, unsigned long arg){

printk("weatherio: device_ioctl - Device IOCTL invoked : 0x%x - %u\n" , cmd , cmd);

switch (cmd) {
case IOCTL_WEATHERIO_GPIO_LOW:
copy_from_user(&targetPin, (gpio_pin *)arg, sizeof(gpio_pin));
make_pin((gpio_pin *)arg);
gpio_request(targetPin.pin, targetPin.label);
gpio_direction_output(targetPin.pin, 0);
gpio_set_value(targetPin.pin, 0);
printk("weatherio: IOCTL_PIIO_GPIO_LOW - pin:%u - val:%s - desc:%s\n" , targetPin.pin , "0" ,
targetPin.label);
break;
case IOCTL_WEATHERIO_GPIO_HIGH:
copy_from_user(&targetPin, (gpio_pin *)arg, sizeof(gpio_pin));
make_pin((gpio_pin *)arg);
gpio_request(targetPin.pin, targetPin.label);
gpio_direction_output(targetPin.pin, 0);
```

```
gpio_set_value(targetPin.pin, 1);
printk("weatherio: IOCTL_PIIO_GPIO_HIGH - pin:%u - val:%s - desc:%s\n" , targetPin.pin , "1" ,
targetPin.label);
break;
default:
printk("weatherio: FD accessed\n");
}

return 0;
}


struct file_operations Fops = {
.unlocked_ioctl = device_ioctl,
.open = device_open,
.release = device_release,
};


static int __init weatherio_init(void){

  printk(KERN_INFO "weatherio: initializing the dd\n");
  MajorNum = register_chrdev(0, DEVICE_NAME, &Fops);
    if (MajorNum<0){
      printk(KERN_ALERT "weatherio: failed to register with major number\n");
      return MajorNum;
    }
  printk(KERN_INFO "weatherio: registered with major number %d\n", MajorNum);


  ClassName = class_create(THIS_MODULE, CLASS_NAME);
  if (IS_ERR(ClassName)){
    unregister_chrdev(MajorNum, DEVICE_NAME);
    printk(KERN_ALERT "weatherio: failed to register device class\n");
    return PTR_ERR(ClassName);
  }
  printk(KERN_INFO "weatherio: device class registered\n");


  DeviceName = device_create(ClassName, NULL, MKDEV(MajorNum, 0), NULL, DEVICE_NAME);
  if (IS_ERR(DeviceName)){
    class_destroy(ClassName);
    unregister_chrdev(MajorNum, DEVICE_NAME);
    printk(KERN_ALERT "weatherio: failed to create the device\n");
    return PTR_ERR(DeviceName);
  }
  printk(KERN_INFO "weatherio: device class created\n");
```

```
return 0;

}

static void __exit weatherio_exit(void){
  // Iterate through allocated pins and free them
  // No need to gpio_free targetPin as the pin (not address) will have been freed in the loop
  int i;
  for(i=0;i<3;i++){
  if(allocated_pins[i]){
  printk("weatherio: GPIO Free on pin:%u\n" , allocated_pins[i]->pin);
  gpio_set_value(allocated_pins[i]->pin, 0);
  gpio_free(allocated_pins[i]->pin);
  kfree(allocated_pins[i]);
  }
  }
  printk("weatherio: beginning exit...\n");
  device_destroy(ClassName, MKDEV(MajorNum, 0));
  printk("weatherio: destroyed device\n");
  class_unregister(ClassName);
  printk("weatherio: unregistered class\n");
  class_destroy(ClassName);
  printk("weatherio: destroyed class\n");
  unregister_chrdev(MajorNum, DEVICE_NAME);
  printk("weatherio: unregistered chrdev\n");
  printk(KERN_INFO "weatherio: Module removed\n");
}
module_init(weatherio_init);
module_exit(weatherio_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("CMP408 - Joe Crichton");
MODULE_DESCRIPTION("RPi Weather GPIO Driver");
MODULE_VERSION("0.2");
```

weatherio.h

```
/*
Name: weatherio.h
Author: Joe Crichton
Description: Header file for CMP408 Weather project driver
*/
#ifndef WEATHER_H
#define WEATHER_H

#include <linux/ioctl.h>
```

```
typedef struct gpio_pin {
char label[16];
unsigned int pin;
} gpio_pin;

#define IOCTL_WEATHERIO_GPIO_HIGH 0x65
#define IOCTL_WEATHERIO_GPIO_LOW 0x66


#define  DEVICE_NAME "weatheriodev"
#define  CLASS_NAME  "weatheriocls"

#endif
```