Historical Environmental Tracker

Josiah Sweeney & Neima Kashani

Overview

This project builds on the concepts utilized and introduced in Project 2 with the basics of MQTT and utilizing a temperature and humidity sensor and adds additional sensors including barometric pressure, ambient light, CO2, and TVOC(Total Volatile Organic Compounds) to create a device that records environmental data and sends it, using the MQTT protocol, registered as an AWS IoT "Thing". The messages from the registered IoT device, sent in a JSON format, are gathered and stored in an Amazon DynamoDB table, where they can be queried and scanned from an Android application, which displays the data that comes in. Figure 1 shows an overall high level look at what this looks like.

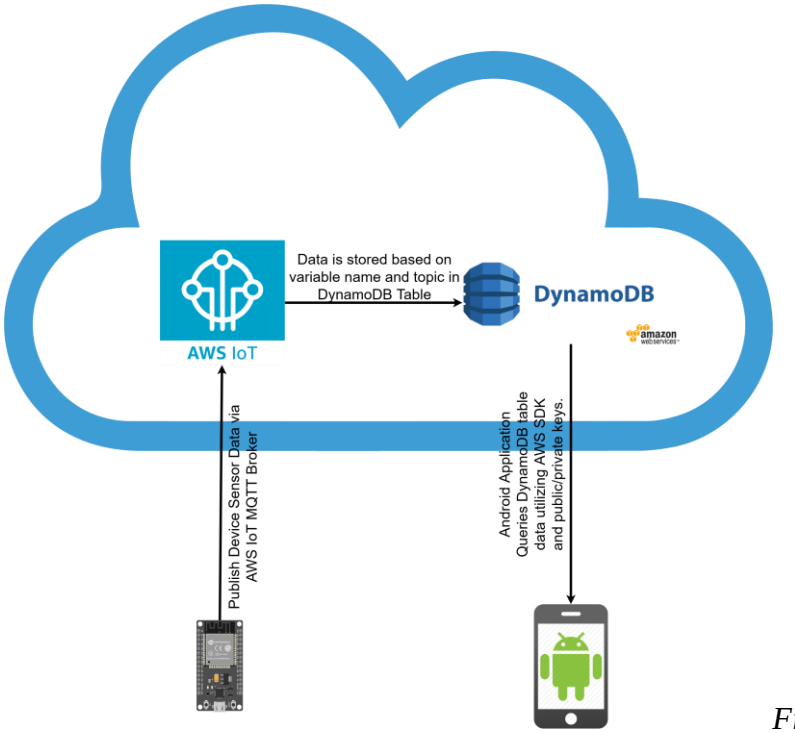

Figure 1:

Hardware and Micro-controller Development

Hardware

In addition to the AHT20 temperature and humidity sensor that was used with project 2, we added two additional I2C sensors, the SGP30, and the DPS310, as well as a small photo-transistor to measure relative light levels. The SGP30 is an environmental gas model that measures the ppm and ppb of CO2, TVOC, H2, and Ethanol in the air. It requires calibration and temperature and humidity readings for higher accuracy. The DPS310 is a barometric pressure sensor that is also I2C based. This means that you can easily measure altitude changes and weather patterns. Finally, the photo-transistor is a simple photo-transistor attached to a 4.7k Ohm resistor to detect changes in ambient light. In figure 2, on the next page, the breadboard set-up for this project is seen.

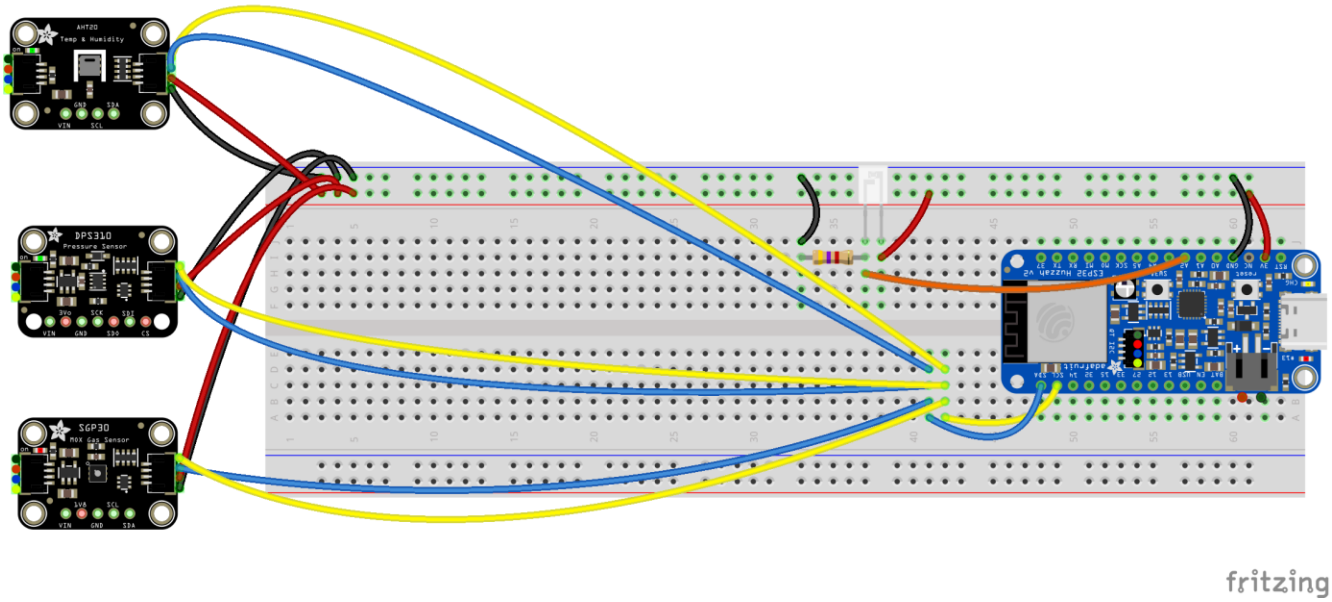

Figure 2: Wiring for Microcontroller

Micro-controller Programming

For programming the micro-controller, we used Arduino IDE, and the available libraries for the sensors. These were fairly straightforward. However, difficulty came when connecting to the AWS IoT, which will be gotten into more detail later. On the micro-controller side, it was very difficult to get the AWS certificates to be recognized. This was solved by using the pgmspace library in Arduino, where the ROOT CA, private certificate, and the private key were stored in memory. Separately from connecting to an AWS IoT instance, I used a different MQTT library, the MQTTClient library as it ended up being cleaner. Also, in order for the MQTT messages to be read easily by the AWS backend rules that were set-up, the ArduinoJson library was used, which allowed for very easy JSON formatting for the MQTT messages.

Setting Up AWS

This part of the project was very time-consuming, and a large learning curve. In order to keep data both private and available, we decided to use AWS to use as an MQTT broker. For this, I had to create an AWS account, and set up an AWS IoT "Thing" to register for the ESP32 device. This required the creation of certificates and policies allowing the ESP32 to actually connect to AWS. The policies were formatted in JSON and had to be very specific to allow for publishing limited topics. The following is a snippet for a small part of the policy allowing the device to publish.

```
{
  "Effect": "Allow",
  "Action": "iot:Publish",
  "Resource": "arn:aws:iot:us-east-1:154723643083:topic/device/1/data"
},
```

Once this was set-up with proper certificates on the ESP32, I was able to publish. Next was the challenge of getting that MQTT data stored. AWS has rules that can be set-up that allow actions to be taken upon triggering by a received MQTT message. In this project, a rule was set up that put items automatically into a pre-created AWS DynamoDB table. This rule was a very short SQL script, which was another learning experience. After all of this was set up, and the micro-controller programmed, the device hooked up to the backend, and sent the sensor data once a minute, where it was stored in the table. Here is a small snapshot of the table from the AWS console in Figure 3.

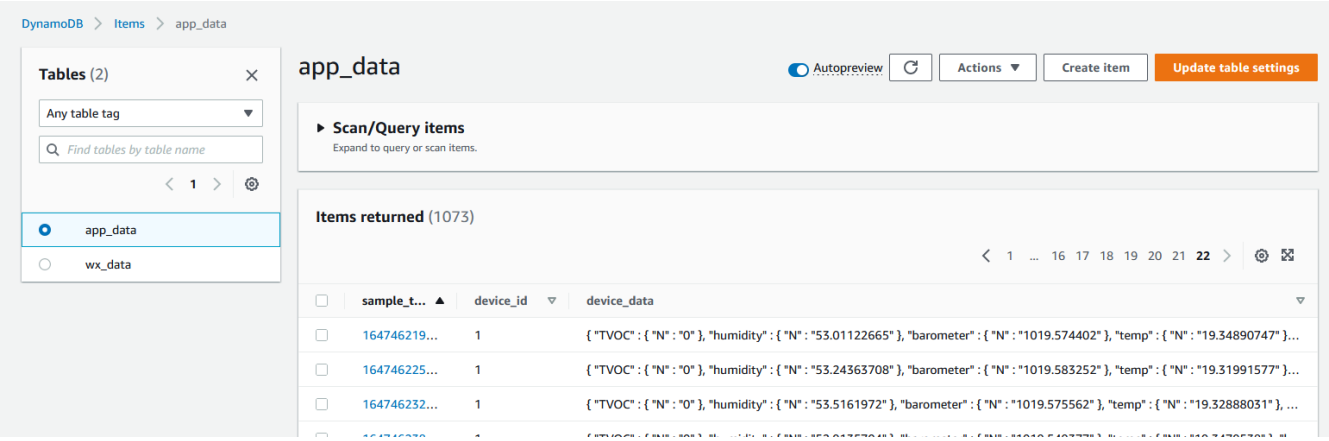

Figure 3: AWS Console Look at app_data table

Android Application Development

The Android application for this project was made using two fragments, along with a View Model for handling the data and a lot of new libraries. The first fragment that is seen is the Chart Fragment, which shows a graph of the data, while the second one allows you to control what is seen on the graph itself. The graph was created using the aachartmodel library, which utilizes the WebView functionality in a layout to display data in a graphical format. Figures 4 and 5 below show what these two fragments look like.

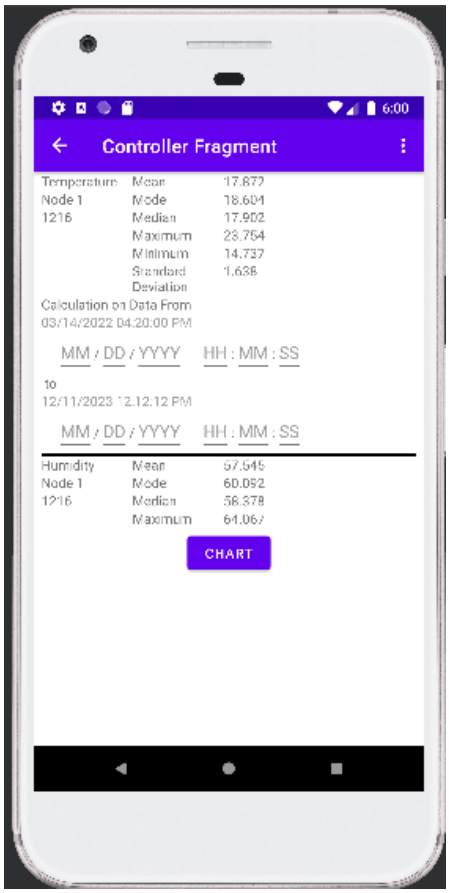

Figure 4: Controller Fragment

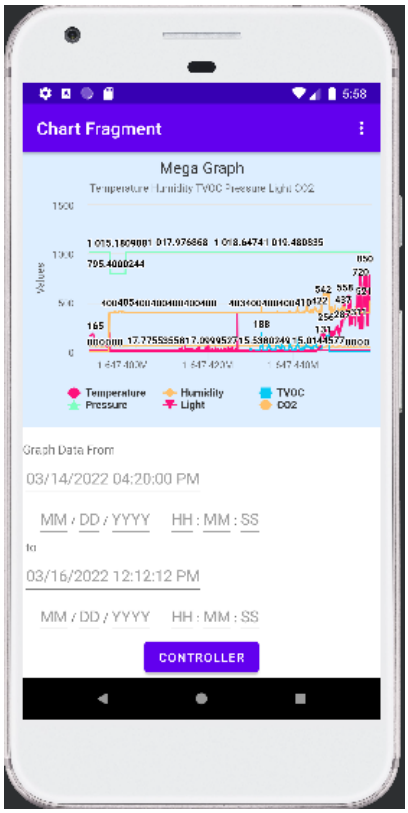

Figure 5: Chart Fragment

The navigation between these two fragments is very simple, just going in-between them based on the options selected and a button press. As said before, the Chart Fragment is the first one seen, and the simple navigation can be seen between the two charts here in Figure 6.

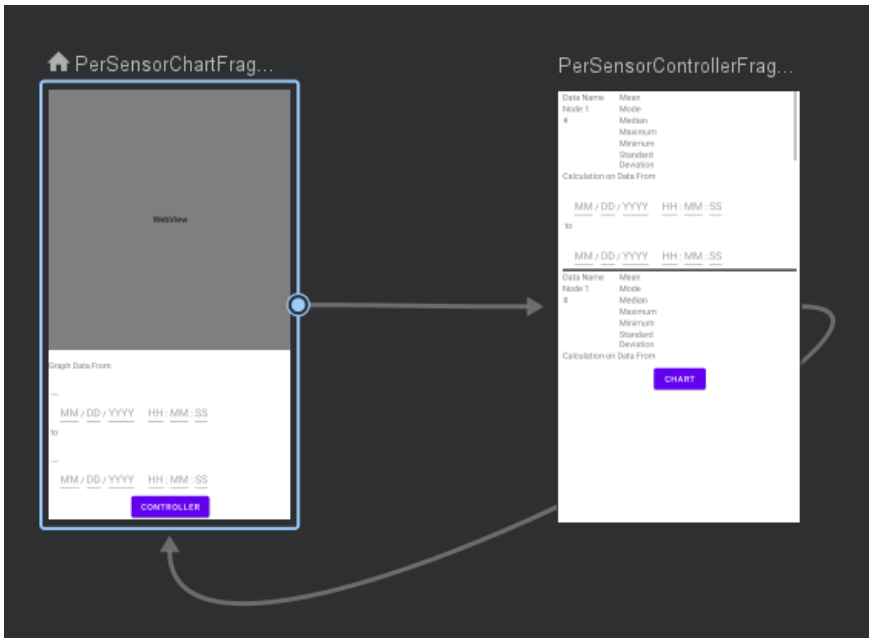

Figure 6: Nav Graph for Android

In order to get data onto the graph, we had to use the AWS SDK, which despite good documentation provide extremely tricky. This was for several reasons, the first being that until just 4 months ago, the AWS Kotlin SDK was not being used in most capacities for Android development and the most deprecated Java SDK is what was being used in Android development. The Kotlin SDK is still mostly in pre-release, with almost no examples of building in Android. That being said, this was eventually overcome, and the AWS Kotlin SDK was utilized in the project, where we used the DynamoDB Scan and Query functionalities to get values off of the table to fill the graphs. As this is a query, this has to be done in a co-routine as to not disrupt overall functionality of the application.

Each time either the Chart Button or the Controller Button are pressed, the application checks each of the Time and Date Inputs to see if it has been given any new valid values for the calculation sample time maximum, calculation sample time minimum, graph sample time maximum, or graph sample time minimum to find any updates to the calculation sample time range and graph sample time range. The time inputs are done in 24-hour military time.

When the new Fragment calls OnViewCreated() upon being switched to, the ViewModel queries the server for new data and processes thew data. To prevent the ViewModel from downloading the whole data base each time, the sample time of the latest data point gotten from the server is kept track of. The AWS request includes a rule to only include data that is more recent than that data point. If new data is received, that tracked latest sample time is update. To process the data, the data is copied into a calculation data list and a graphing data list and the 2 data lists are then sorted and filtered based on the calculation sample time range and the graph sample time range. Sorting base on sample time is needed as the data is not given by the server in terms of sample time. The ViewModel then calculates the mean, median, mode, maximum, minimum, and standard deviant for data that is within the calculation sample time range and makes a new Graph with the data in the graph sample time range.

Results

Overall, we put together something that is what I would call a good start to something that could be useful. The backend work was a lot more than expected and took a long time getting things up and running. On the micro-controller and backend side, I believe there isn't much more that could be done to make things better. To add new devices, it would be the process of making a new “Thing” that shares the same rules, but with a different device_id. The DynamoDB table is setup to accept a wildcard for the device id, so this could be scaled easily. On the Android side, it would be cool to see a lot of features, which would just take time, like selecting a specific device if you have multiple registered, or even comparing two different devices. The good is that we accomplished something that works, the bad was that we couldn't get as far as I would like on features, and that documentation for some of the functionality was very difficult to come by for Kotlin.

Using the Final Project

As all of the data for this project is stored until I(Josiah) delete it or my AWS account, it is fairly easy to test this out. The apk can be installed, and the IAM Amazon credentials are actually stored in code so just by using the app, you can look at the data. I probably will turn off the ESP32 logging data at some point. If you want to build the software, the gradle settings in the build.gradle file should make this very straightforward as well for Android. The ESP32 software, does require a bit more set-up, as several libraries must be installed on the Arduino IDE. These libaries are the WifiClientSecure, MQTTClient, ArduinoJson, Adafruit_AHTX0, Adafruit_DPS310, and Adafruit_SGP30 libraries. Once those are installed, the software should build to be written onto the ESP32.

Individual Contributions

Neima:
- Android Application Development

Josiah:
- AWS IoT & DynamoDB Setup
- Micro-controller code
- AWS SDK Android Troubleshooting

Additional Information

AWS IoT Developer Guide:
https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html

AWS DynamoDB Developer Guide:
https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

AWS SDK for Kotlin Developer Guide:
https://docs.aws.amazon.com/sdk-for-kotlin/latest/developer-guide/home.html