

Weather Tracking App:



Class: Software Engineering CSCI 665

Professor: Altion Simo

Group Members: Pietro Gagliano, Blair Murray, Kyle Bruckner, Edward Ostuni, John Zhang

Introduction and Project Requirements:

The concept of our project is to design a system which uses a hardware based temperature and humidity and have this data displayed on our web app in real time. This documentation will provide a comprehensive overview of the entirety of our system from hardware design and embedded firmware structure that is running on the hardware up to the software data acquisition from aws to the backend and the visualization of the sensor values in the front end.

The requirements of our design can be broken down into a few simple layers of abstraction. These are listed below and these are the fundamental components and requirements of our system.

1. We first require hardware which can both obtain temperature and humidity data and connect to wifi and interface AWS.
2. Then we require firmware which allows our hardware to connect to the internet, connect to aws and send this sensor data and values to AWS.
3. Then we require the AWS tools to acquire this data and send it to our application.
4. Then we require a front end which allows us to visualize this data so that the end user can use our application properly.

In our technical design and architecture section we will outline the detailed design processes and final outputs for each of the requirements. It will become clear how exactly we are satisfying each of the requirements. We will start from hardware and firmware which is our lowest abstraction layer and gradually elevate our scope up to the application code and front end which is closest to the user experience.

Technical Design and Architecture:

In this section, we will explain how our code is used in order to run the application. Arduino was used on our firmware for this early revision of our code. The Arduino platform allows one to prototype firmware functionalities across wide hardware platforms using very intuitive coding styles that are very approachable to beginners. This is what provided a starting point for us before having further code developed in the more finalized softwares for our hardware.

Beginning with the initialization of our firmware, we first started off by including our necessary libraries. These included our libraries for connecting to wifi as well as our library for the DHT11 sensor. These needed to be included in the Arduino IDE and these libraries allow for the development of very efficient code. We then initialized the SSID and password for our 2.4GHz wifi network. Next, we could then begin the requirement of initializing our API gateway URL. This API gateway URL can be accessed in AWS. Then we set our timer delays to space the sending of data to our S3 bucket to approximately every 5 seconds. We also have an uptime iterator variable so that tracks the data logging. In our setup section, we initialize the serial port at 115200 baud rate, connect to the wifi, check the wifi connection before proceeding, and finally we initialize the DHT11 sensor so it can start gathering temperature and humidity data. This primary setup stage is shown below in Figure 1.

AWS_IoT_API_Gateway_Send

```
#include <WiFi.h>
#include <HTTPClient.h>

#include "DHT.h"
#define DHTPIN 14    // Digital pin connected to the DHT sensor
#define DHTTYPE DHT11 // DHT 11
DHT dht(DHTPIN, DHTTYPE);

const char* ssid = "*****";
const char* password = "*****";

//Your Domain name with URL path or IP address with path
String serverName = "*****"; //fill this out with your own API Gateway deployment

// the following variables are unsigned longs because the time, measured in
// milliseconds, will quickly become a bigger number than can be stored in an int.
unsigned long lastTime = 0;
// Timer set to 10 minutes (600000)
//unsigned long timerDelay = 600000;
// Set timer to 5 seconds (5000)
unsigned long timerDelay = 5000;
unsigned long Uptime;

void setup() {
  Serial.begin(115200);

  WiFi.begin(ssid, password);
  Serial.println("Connecting");
  while(WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.print("Connected to WiFi network with IP Address: ");
  Serial.println(WiFi.localIP());
  dht.begin();
  Serial.println("Timer set to 5 seconds (timerDelay variable), it will take 5 seconds before publishing the first reading.");
}
```

Figure 1: Initialization of libraries and setup stage for the Arduino code.

For the main loop of our firmware, we define our humidity and temperature variables, *h* and *t*, using the *dht.readHumidity()* and *dht.readTemperature()* as our functions. The uptime is monitored as the time since the code has started logging to AWS. This uptime gets logged as well along with the temperature and humidity. We then create a server path variable which

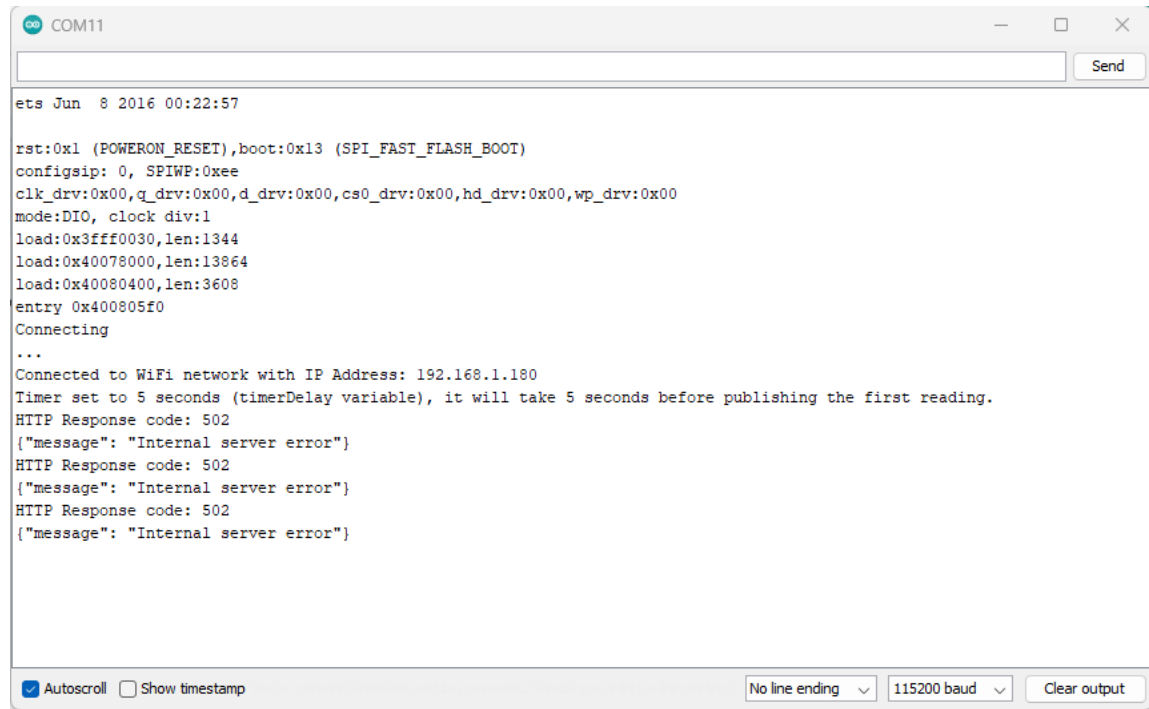
includes our server name, the API gateway URL with deployment name, as well as the temperature, humidity and uptime. We hit our API gateway with this server path as long as our wifi remains connected which we are consistently monitoring. We also send HTTP get requests and log the response codes so from the serial port, firmware side we can validate that things are working the way they should. This main loop stage of our firmware is shown in Figure 2 below.

```
void loop() {  
    float h = dht.readHumidity();  
    float t = dht.readTemperature();  
    Uptime = millis()/1000;  
    //Send an HTTP POST request every 10 minutes  
    if ((millis() - lastTime) > timerDelay) {  
        //Check WiFi connection status  
        if(WiFi.status() == WL_CONNECTED){  
            HTTPClient http;  
  
            String serverPath = serverName + "?uptime=" + (String) Uptime  
                + "&temperature=" + (String) t + "&humidity=" + (String) h;  
  
            // Your Domain name with URL path or IP address with path  
            http.begin(serverPath.c_str());  
  
            // Send HTTP GET request  
            int httpResponseCode = http.GET();  
  
            if (httpResponseCode>0) {  
                Serial.print("HTTP Response code: ");  
                Serial.println(httpResponseCode);  
                String payload = http.getString();  
                Serial.println(payload);  
            }  
            else {  
                Serial.print("Error code: ");  
                Serial.println(httpResponseCode);  
            }  
            // Free resources  
            http.end();  
        }  
        else {  
            Serial.println("WiFi Disconnected");  
        }  
        lastTime = millis();  
    }  
}
```

Figure 2: Main loop of Arduino firmware.

In the software section of our hardware and software testing, we will detail more about how we set up the AWS and software elements including and beyond the API Gateway to actually display our data on the app. When this is all done we can easily test to see if our

hardware and firmware are working by looking at our serial port which we have used to flash the hardware and see what logs appear. These log outputs are shown in figure 3 below.



```
ets Jun  8 2016 00:22:57

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0030,len:1344
load:0x40078000,len:13864
load:0x40080400,len:3608
entry 0x400805f0
Connecting
...
Connected to WiFi network with IP Address: 192.168.1.180
Timer set to 5 seconds (timerDelay variable), it will take 5 seconds before publishing the first reading.
HTTP Response code: 502
{"message": "Internal server error"}
HTTP Response code: 502
{"message": "Internal server error"}
HTTP Response code: 502
{"message": "Internal server error"}

Autoscroll Show timestamp No line ending 115200 baud Clear output
```

Figure 3: Arduino serial output logging from ESP32 serial port.

We can clearly observe from the logs that not only do we have wifi connecting properly, as seen by the fact that we obtain an IP address, but we then are able to see that our HTTP response codes verify that we are obtaining data properly. The *internal server error* response code is what we are looking for as in our particular software setup we have not programmed a particular response code for the ESP32 HTTP GET requests. The diagram below essentially shows what workflow we have established so far in our hardware/firmware setup. The API gateway has already been set up in this testing stage and further details of exactly how that was done are explained in the AWS and Software section.

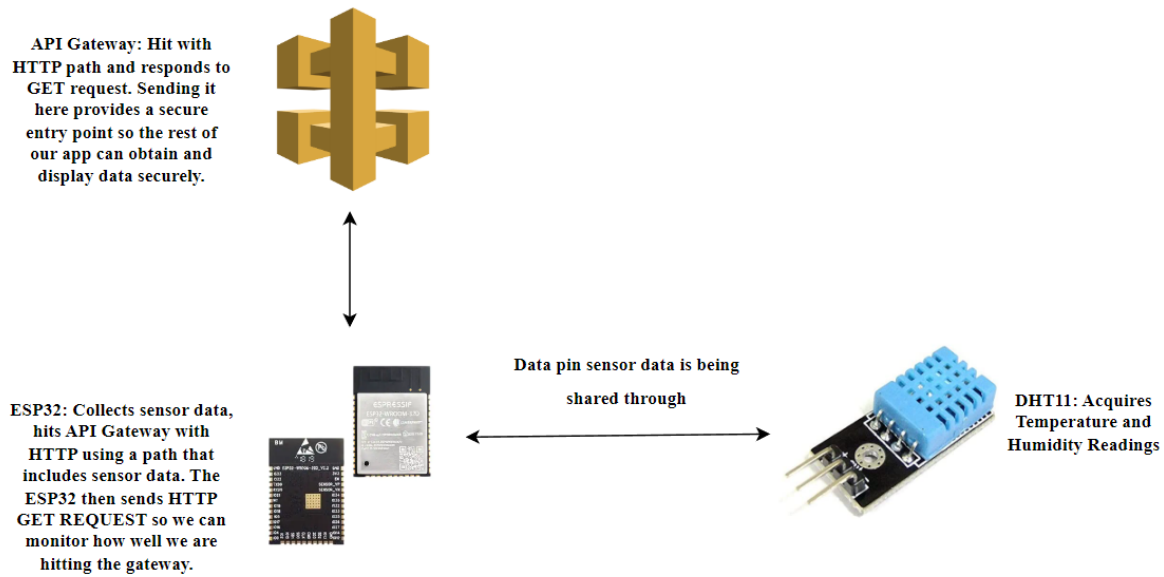


Figure 4: Workflow diagram of the functionality of hardware and firmware in our system.

AWS and Software Implementation:

The intended workflow of our AWS and software combination is shown in the figure below.

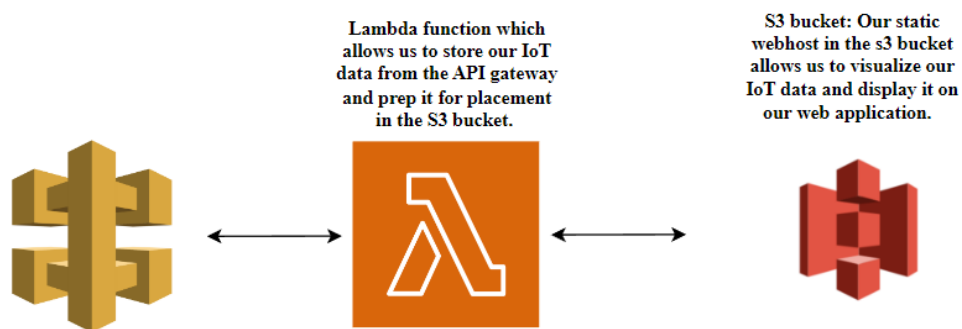
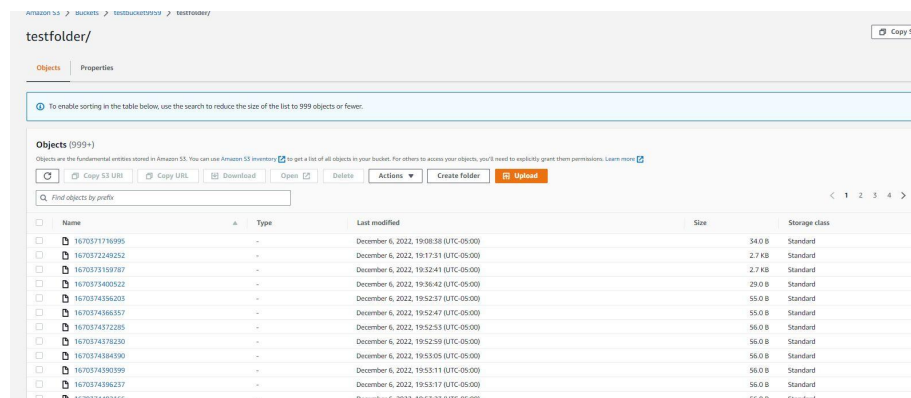


Figure 4: Workflow diagram of the functionality of AWS and software combination.

We wanted to store our MQTT data in a data folder within an S3 bucket so that we could access this bucket using JavaScript code within our HTML code on our web page. We also wanted to have a second bucket, that held our HTML code and acted as a static host for our webpage. To get started on this end, we had to create our S3 bucket and create a data folder in that bucket. Our current JSON files are inside our bucket's IoT Data folder. To get the data in the folder, we needed to use an AWS Lambda function. Below we can see in Figure 5 what our files with our data will look like when they are populated in the S3 bucket.



Name	Type	Last modified	Size	Storage class
1670571716095	-	December 6, 2022, 19:08:58 (UTC-05:00)	54.0 B	Standard
1670572248252	-	December 6, 2022, 19:17:31 (UTC-05:00)	2.7 KB	Standard
1670573169787	-	December 6, 2022, 19:52:41 (UTC-05:00)	2.7 KB	Standard
1670573409532	-	December 6, 2022, 19:56:43 (UTC-05:00)	29.0 B	Standard
1670574396203	-	December 6, 2022, 19:52:57 (UTC-05:00)	55.0 B	Standard
1670574398357	-	December 6, 2022, 19:52:47 (UTC-05:00)	55.0 B	Standard
1670574372285	-	December 6, 2022, 19:52:53 (UTC-05:00)	56.0 B	Standard
1670574378230	-	December 6, 2022, 19:52:59 (UTC-05:00)	56.0 B	Standard
1670574384390	-	December 6, 2022, 19:53:05 (UTC-05:00)	56.0 B	Standard
1670574390399	-	December 6, 2022, 19:53:11 (UTC-05:00)	56.0 B	Standard
1670574396237	-	December 6, 2022, 19:53:17 (UTC-05:00)	56.0 B	Standard
1670574396237	-	December 6, 2022, 19:53:17 (UTC-05:00)	56.0 B	Standard

Figure 5: IoT data files in our test folder in the S3 bucket.

In AWS, we used the Lambda service to directly send our ESP32 MQTT data from the firmware sensor to our AWS S3 Bucket. Using the AWS-SDK, we are able to export all of the JSON data files that are being sent to AWS IoT directly into our S3 Bucket. We used the (event.queryStringParameters) to make sure that our data was being formatted properly so that we could read it into our web page charts. We then run an `s3.putObjects` command to successfully transfer this JSON data to our S3 bucket. Below in Figure 6, our lambda function is displayed.

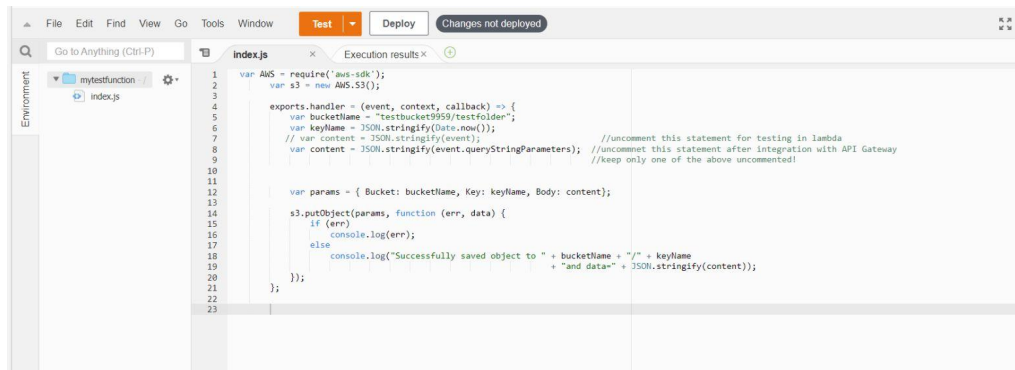
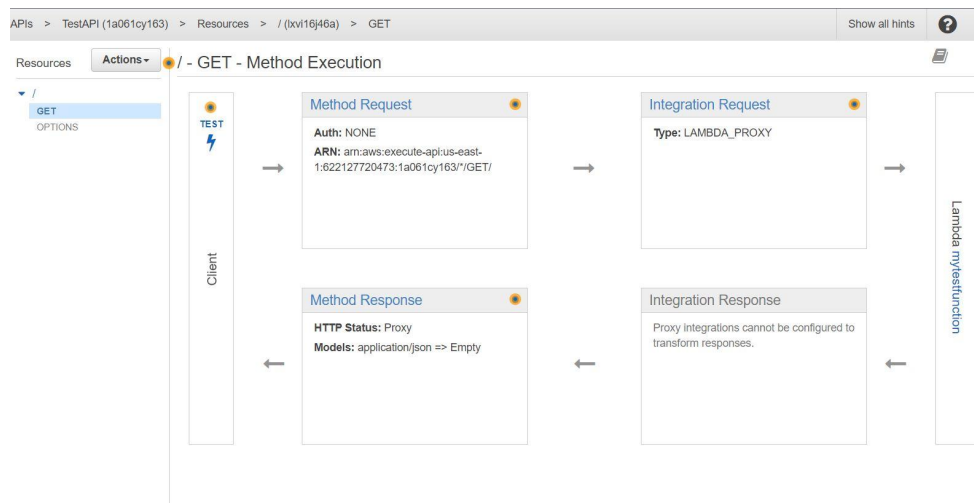


Figure 6: Lambda function JSON code.

In order for us to run our Lambda function to continuously take in the IoT data from the ESP32 sensor, we had to create a REST API within Amazon AWS' API Gateway. In the REST API, we create a new GET Method, which runs our Lambda function and returns our JSON data values and places them in the S3 bucket. Shown below in figure 7 is the REST API GET METHOD creation dashboard where we created our GET METHOD.



We then create a second S3 bucket, where we upload the HTML and CSS style codes that will be used to display our data on a web page. We then go into the “Properties” tab of the bucket and scroll down to “Static website hosting” and enable it and choose the “index.html” as our source code file. The two files are in our bucket and it states that it is “Publicly accessible”. We have to make both our data and hosting buckets public in order for our data to be shown. Below in figures 8 and 9, we have displayed our secondary S3 bucket for hosting the static webpage as well as our static web hosting settings which we have enabled to allow for static web hosting.

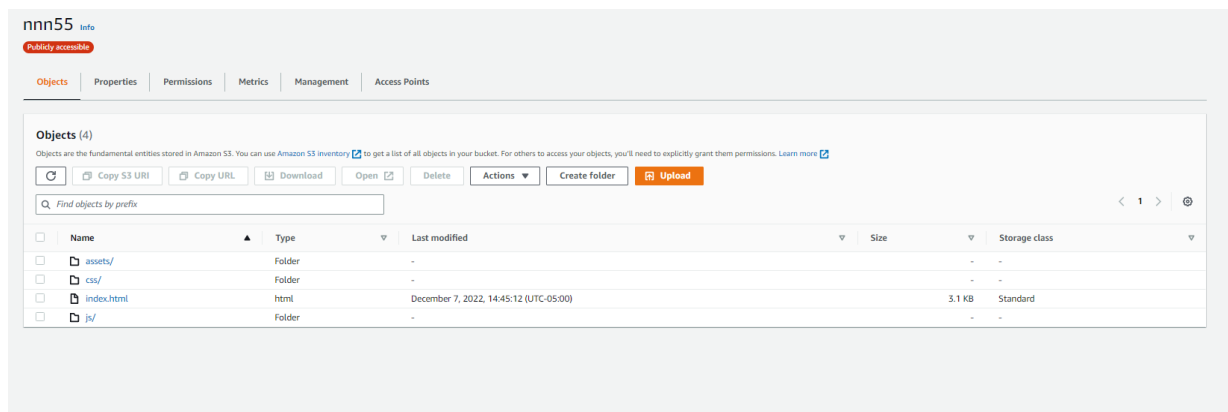


Figure 8: Secondary S3 bucket where html file and css and javascript files are being hosted.

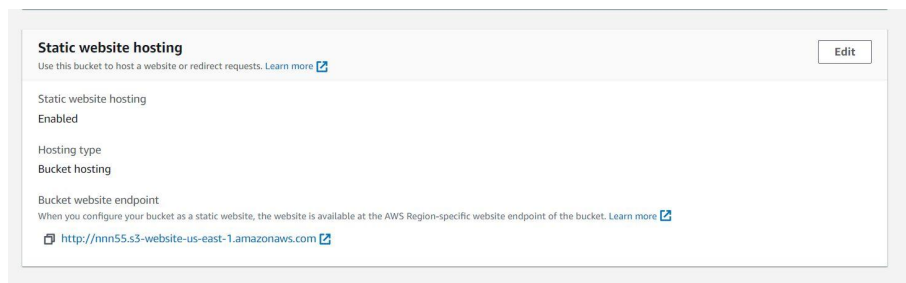


Figure 9: Static web hosting settings.

To make both buckets public, we first have to turn off “Block all public access” settings and enable Access Control Lists (ACLs) and give everyone list or read access to our bucket objects. You also see that we added Bucket Policies to both buckets that allow for anyone to publicly getObject from our respective S3 Buckets. We then also had to add a Cross-origin

resources sharing, CORS, policy to both buckets which allows an Authorization header, GET methods, and all objects to be shared from our buckets.

The screenshot displays the AWS IAM console interface for configuring bucket permissions. It is divided into two main sections: 'Permissions overview' and 'Access control list (ACL)'.

Permissions overview:

- Block public access (bucket settings):** A section with an 'Edit' button. Below it, a toggle for 'Block all public access' is set to 'Off'. A link for 'Individual Block Public Access settings for this bucket' is provided.
- Bucket policy:** A section showing a JSON policy. The policy grants 's3:GetObject' permission to 'arn:aws:s3:::*' (all S3 objects).

Access control list (ACL):

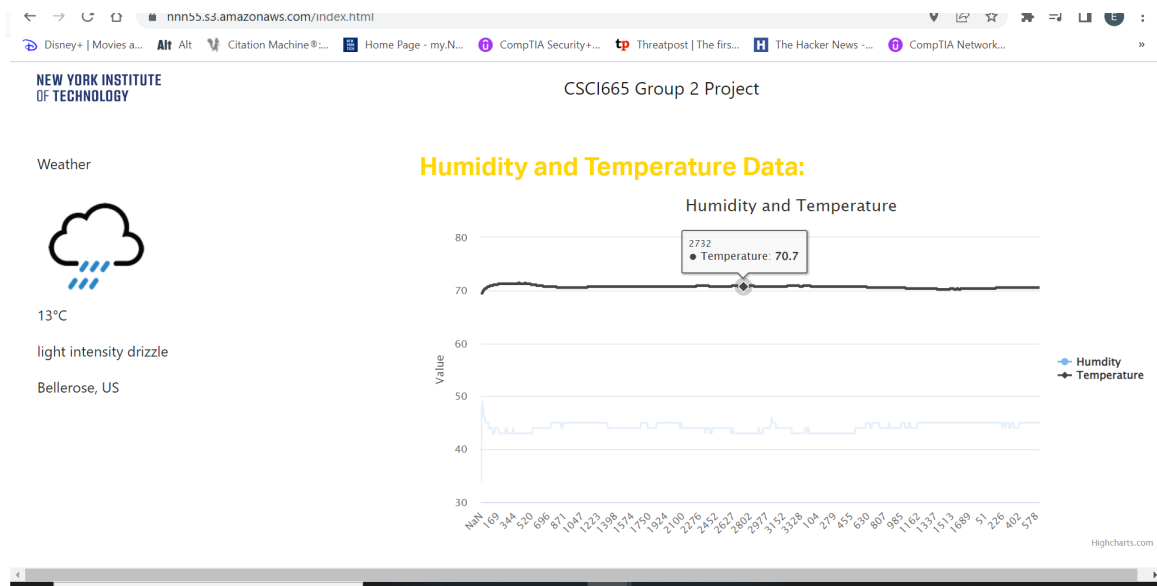
- A warning message states: 'The console displays combined access grants for duplicate grantees. To see the full list of ACLs, use the Amazon S3 REST API, AWS CLI, or AWS SDKs.'
- A warning message states: 'AWS doesn't recommend granting access to the Everyone grantee. Anyone in the world can access the objects in this bucket.'
- A table lists the grantees and their permissions:

Grantee	Objects	Bucket ACL
Bucket owner (your AWS account) Canonical ID: BAAED378A6B4795A45454688B4C742055W1620W90995W500174d8B243c46	List, Write	Read, Write
Everyone (public access) Group: http://acs.amazonaws.com/groups/global/AllUsers	List	Read
Authenticated users group (anyone with an AWS account) Group: http://acs.amazonaws.com/groups/global/AuthenticatedUsers	-	-
S3 log delivery group Group: http://acs.amazonaws.com/groups/S3LogDelivery	-	-

Cross-origin resource sharing (CORS):

- A section showing a JSON CORS configuration. A 'Copy' button is visible.

Now that all our data is inserted into our bucket, our buckets now allow public access, and our second bucket is hosting our static webpage, we can now open up our webpage URL from within the hosting buckets index.html file and see our webpage is loaded up with the graph of our data. The following figure shows our webpage actually running and displaying the necessary data.



Our HTML code is very simple on the design end, but we implement HighCharts as a source to help us chart and graph our data into a Line Chart that shows the Humidity and Temperature along with the uptime that was taken by the sensor. We also created a bar graph to tabulate the total amount of readings we received in the S3 Bucket. We used an XMLHttpRequest GET method to publicly access the bucket and data folder. After accessing the folder, the code then inserts our JSON data into multiple arrays based on which type of value it is. After these arrays are created and filled, it then uses these arrays as the data values for the HighCharts that we created.

The website itself is based on Bootstrap. Bootstrap is a powerful, feature-packed frontend toolkit. This enables us to create websites and web applications that are consistent, modern, and user-friendly. It is a popular framework among web developers because it makes it easier to create responsive, mobile-first websites that are cross-browser compatible.

Our CSS is a very bare-bone CSS. We imported Aktiv Grotesk from Adobe font since it looks like Arial but It looks more aesthetically pleasing.

The JavaScript is used to fetch and process the weather data from the API, and to update the app's display based on the current weather conditions. It

End User:

In this section, we will explain how the Temperature and Humidity Tracking App works.

ESP32 Board will interface a dht11 temperature and humidity sensor and it will collect the temperature and humidity of the environment a person is in and this data will display on an app.

The website is divided into a 4:8 Ratio. The left side is a weather app that will first validate if the browser supports geolocation API. After the validation, the app will request the browser to provide

HTML is used to create the structure and layout of the app, including the various elements such as text, images, and buttons.

The CSS is used to style the app, giving it a cohesive and visually appealing look.

The JavaScript is used to fetch and process the weather data from the API, and to update the app's display based on the current weather conditions.

The app may also use additional libraries or frameworks, such as jQuery or AngularJS, to help with the implementation and functionality of the app.

When the user accesses the app, they can enter their location or zip code and the app will use the API to retrieve the current weather data for that location. The app will then display the current temperature, weather conditions, and other relevant information. The app may also provide additional features, such as the ability to view the forecast for the next few days or to see the current weather in multiple locations.

Marketing:

As our product stands currently, it tracks the temperature and humidity of any location or environment. We plan to, in addition to these features, also integrate a GPS module that allows

for very precise tracking of location as well. This would allow the hardware design to become more of a wearable device in market application.

Communication:

The section includes the various softwares and platforms our team used to communicate with one another as well as collaborate and complete elements of the firmware and code implementation.

WhatsApp:

A messaging app used for quick communication between all the team members and used to decide when to hold our weekly meetings on Google meetings/Zoom sessions. We also WhatsApp to send documentation and files quickly and easily before our github repository was in place so we could test our software in stages.



Google Meetings and Zoom Sessions:

We used both Google Meeting and Zoom to have long form discussions about the things we needed to do at every stage of the project. This format also allowed us to share screens and

control each other's screens at different steps of the process so we could help each debug code in real time beyond just WhatsApp descriptions.



AWS Accounts:

Our free AWS accounts were necessary to set up our API and S3 buckets so that we could obtain information from the ESP32 sensors and send them to the application to be displayed. We also hosted our static web page on an S3 bucket.



Google Drive:

Our Google documents were shared through Google Drive which allowed us to efficiently outline, organize, and document our project.



GitHub:

We used Github to host our various software and firmware files so that we could collaborate more effectively on code changes and optimizations at the final stages of the project.

