Documentation for AirU-GCP (Google Cloud Project)

Last updated: 30 October 2019 by Scott Gale

Purpose: To serve as a how to guide in administering and updating the AirU-GCP system and to provide insight into implementation decisions. This is a living document and resides in the Web App repository.

Table of Contents

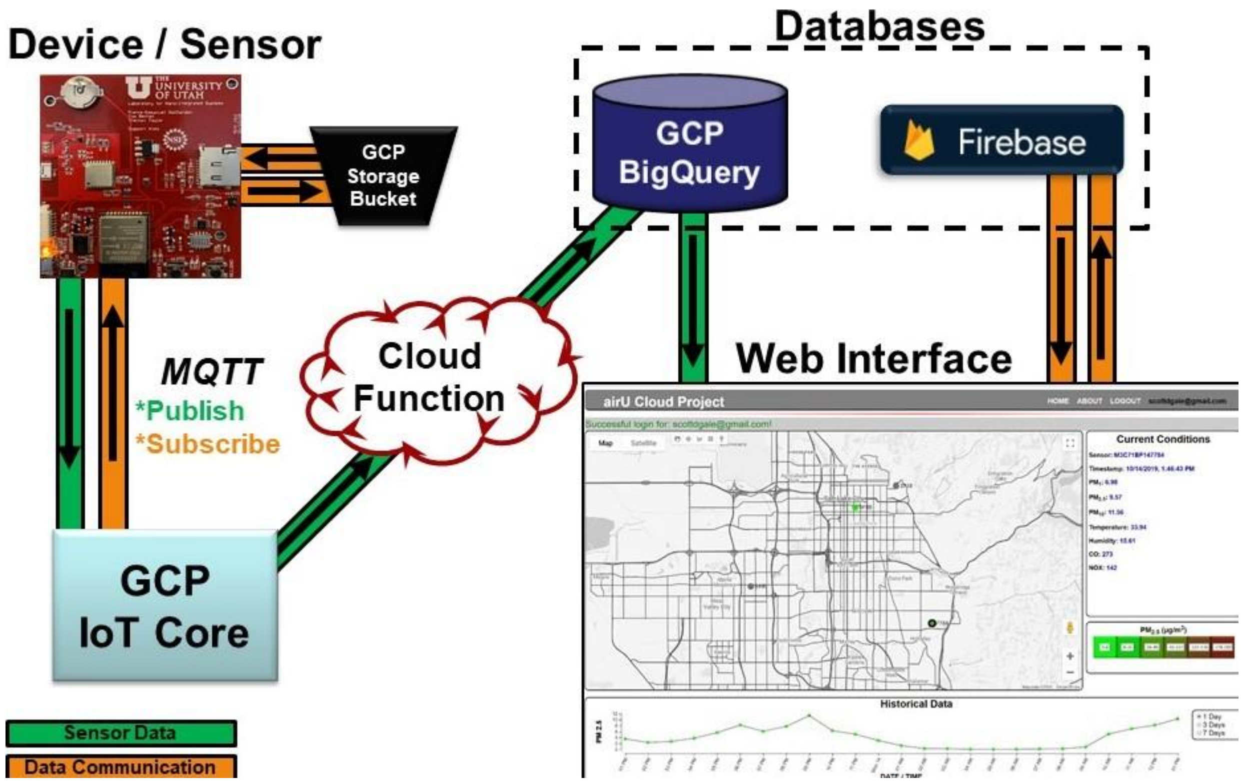
1. Functional Block Diagram
2. Device Firmware
3. Creating Devices in Google IoT Core
4. Managing Devices within the Device Registry (Config, State, OTA updates)
5. IoT Stackdriver Logging (Debug logs for each device)
6. Cloud Function
7. BigQuery Database (Sensor data repository)
8. Firebase Authentication and Database
9. Web Application

Repository locations

Firmware: https://github.com/aqandu/airu-v2-fw/tree/mqtt_cloud

Web App: https://github.com/LNIS-Projects/AirU-GCP

FUNCTIONAL BLOCK DIAGRAM



DEVICE FIRMWARE

Repository: https://github.com/aqandu/airu-v2-fw (branch mqtt_cloud)

The device is engineered around an ESP-32 microprocessor and Free RTOS (free real time operating system). It has multiple embedded sensors that measure characteristics of air quality such as temperature, humidity, particulates in the air (PM1, PM2.5, PM10), etc as well as a GPS. It has both Bluetooth and WIFI capabilities.

To program the firmware, you must install the ESP-IDF. Instructions can be found at the following link: https://docs.espressif.com/projects/esp-idf/en/latest/get-started/index.html#step-2-get-esp-idf

It is possible to program via the Windows environment; however, we have found this to be problematic (had a hard time getting it to work properly). Since I have a Windows machine, I used VMWare, installed a Linux VM and did my programming there. It worked great.

We are currently using ESP-IDF releaseV3.3 (the "stable" version). As time goes on newer versions will become stable and updates will be required.

RTOS utilizes a "task" based system for execution and control flow. Code execution begins in the app_main() function located in main.c. The app_main() function essentially initializes the sensors / peripherals, and then creates and starts several tasks.

Discuss Each Task Here

mqtt_task: As soon as the device connects to WiFi the wifi_manager task executes MQTT_Initialize() (found in mqtt_if.c) which creates the mqtt_task. The mqtt_task is subsequently destroyed anytime the WiFi is disconnected.

The purpose of the mqtt_task is to establish and maintain a connection with Google IoT Core, which serves as the MQTT broker, and publish packets containing sensor data. All functionality is coded in the mqtt_if.c file except for the over the air update functionality found in ota_if.c. We use the ESP MQTT library to manage the MQTT connections, subscriptions, publication of data, and OTA updates.

ESP MQTT Documentation: https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/protocols/mqtt.html

Prior to connecting with IoT Core a JSON Web Token (JWT) is created which is used for the password parameter in the mqtt configuration. The mqtt configuration (see getMQTT_Config() in mqtt_if.c) contains all the details to establish a connection with IoT Core. The JWT is just one part of this configuration. There are many settings that can be adjusted. The current settings have been determined to provide the most stable connection; however, future updates may require fine tuning. Note that the username parameter is not used by Google. The value is currently set to “unused.”

Remark: The JWT (JSON Web Token) contains two time references. First, the time of initialization and second, the expiration time. The maximum lifespan of a JWT is 24 hours. This means that the connection will be dropped at 24 hours unless another JWT is created and the connection is reinitialized.

IoT reference for JWT: https://cloud.google.com/iot/docs/how-tos/credentials/jwts

Connection and Reconnection Logic: Once the client configuration is complete, we can connect with IoT Core. Connection is a two-step process. First, we call the esp_mqtt_client_init() function and pass in the configuration structure. This function returns a handle that we use for all future functions associated with this configuration. Second, we start the connection with the esp_mqtt_client_start() function.

Two likely issues will close the connection with IoT Core. First, if the device disconnects from WiFi. Obviously, if there is no WiFi connection than we can’t communicate with IoT. Google will disconnect any device that is idle for 15 minutes. For this reason if WiFi is interrupted we destroy the mqtt_task and wait for the WiFi connection to be restored. When the WiFi is restored, the wifi_manager will call the MQTT_Initialize() function again which will create the mqtt_task and start the process working again.

The second likely cause that will close the connection with IoT is expiration of the JWT. To avoid this problem, we preempt JWT expiration after 23 hours (remember JWT expires at 24 hours). When a JWT is created we get the time and add 23 hours and save it as reconnect_time. Within the infinite while loop that publishes data we check to see if current_time > reconnect_time. When this condition evaluates to true, then we destroy the client and reconnect to IoT Core with a new JWT.

Subscriptions: ESP MQTT uses an event handler to capture events such as Connected, Disconnected, Subscribed, etc. We handle subscriptions within the MQTT_EVENT_CONNECTED event. We subscribe to two topics: config and command. Config is used to receive the name of the binary file for OTA updates. The command subscription is used to send commands to the device. Currently, the only command that is implemented is the restart command. When the device receives a restart command it restarts. See the Managing Devices within the Device Registry to see how configurations and commands are sent to the device through the GCP console.

IoT reference for Config and State: https://cloud.google.com/iot/docs/concepts/devices
IoT reference for Command: https://cloud.google.com/iot/docs/how-tos/commands#commands_compared_to_configurations

Publish Algorithm: The publish algorithm is designed to minimize the number of packets transmitted to IoT and saved in the BigQuery DB. The idea behind the algorithm is only publish data when the data has changed. The mqtt_task contains a while() loop that continues as long as the WIFI is connected. At the end of the loop is a five minute delay; therefore, the loop is executed once every five minutes.

Every time a data packet is sent the corresponding sensor value are saved in variable that begin with the prefix “pub” meaning published. Variables that do not begin with “pub” are used every five minutes to save current sensor values. These are the variables used for storing published and current sensor values:

```
static pm_data_t pub_pm_dat, pm_dat;
static double pub_temp, temp, pub_hum, hum;
static uint16_t pub_co, co, pub_nox, nox;
static esp_gps_t pub_gps, gps;
```

“pub_temp” for instance stores the last published value for temperature. While “temp” is used for the current value.

The publish algorithm checks to see if the current sensor value is different (see thresholds below) than the last published value. If the value is within a certain threshold tolerance than no data is published. Each sensor value is checked to see if it exceeds the threshold of previously published data. If any of the sensor data DOES exceed the threshold then the “publish_flag” is set and the program with publish a packet.

Here are the current thresholds. These can be easily adjusted and tuned.

```
float pm_delta = 0.25;                  // All PM data
float minor_delta = 1.0;                // Temp, Humidity, NOX
float co_delta = 30.0;                  // CO
float gps_delta = 0.05;                 // GPS
```

Note that if temperature met the change threshold but all other sensor values did not, a complete packet would still be published. Currently, only complete packets are published. Therefore, packets are published any time at least one sensor value has exceeded the change threshold.

There is also a maximum time set between publications. For data processing and visualization purposes the algorithm ensures that at least one packet is published every hour. If a sensor is completely static (no change to any sensor values) the algorithm will still publish at least one packet per hour. This is accomplished through a simple “loop_counter” variable. Recall the publication loop executes every five minutes. Each time within the loop that data is NOT published we increment the “loop_counter.” If the “loop_counter” reaches 12 then we publish. Whenever a packet is published we reset the “loop_counter.”

OTA Updates: OTA updates are accomplished in two parts. This section will cover how the firmware handles OTA updates. Refer to the MANAGING DEVICES IN THE DEVICE REGISTRY for information on how to initiate an OTA update from the GCP console.

The device flash memory is divided into two partitions. One of the partitions contains the firmware that the device is currently running. The other partition is used to store another firmware version. These settings are configured in menuconfig on the ESP32.

Any time the device receives data the MQTT_EVENT_DATA is executed. This code resides within the mqtt_event_handler function. When data is received it is parsed and checked to see if it is an OTA event. This is done by check the tokens of the string for “ota” and “.bin” substrings. An OTA update string has a fixed format that begins with “ota” followed by a binary filename like this: “ota version2_1.bin” If received data does not follow this format it is discarded.

When a valid configuration string is passed the “ota_trigger()” function is called which transfers execution to code in the ota_if.c file. The “_ota_commence()” function is executed which contains the following critical line of code:

```
sprintf(fn_path, "http://storage.googleapis.com/ota_firmware_updates/%s", ota_file_basename);
```

This line sets the path of the http site (GCP storage bucket) where the firmware .bin file is saved. The file is located and downloaded to the free partition within flash memory. Once the file is downloaded, we check to see if it is different than the current .bin file running on the device. If the .bin is different then the device resets itself and begins running the new .bin file.

Note: Every time the device is reset, one of the first things it does after connecting to the MQTT broker is subscribe to the “configuration” topic. Once subscribed to the configuration topic, the device will receive the most current configuration. In our case this is the ota update command. Therefore, when a device is reset it will always check for the latest OTA update. This can cause problems for debugging – if you make changes to firmware and upload it to a device that firmware will automatically be overwritten by whatever firmware file is set in the device configuration. The OTA can be bypassed by commenting out this line of code within the mqtt_event_handler() function:

```
ota_trigger();                              // Comment out to avoid doing OTA updates during development
```

This is found in the MQTT_EVENT_DATA case.

CREATING DEVICES IN GOOGLE IOT CORE

Remark: Before any device can connect to IoT Core it must be created within the GCP. IoT Core acts as the MQTT broker for devices and will NOT except any traffic from devices that are not contained in the device registry. Therefore, PRIOR to connecting or deploying a device these steps must be taken.

1. Login to Google Cloud Platform (https://cloud.google.com/)

2. From the navigation menu (upper left corner) select “IoT Core.”

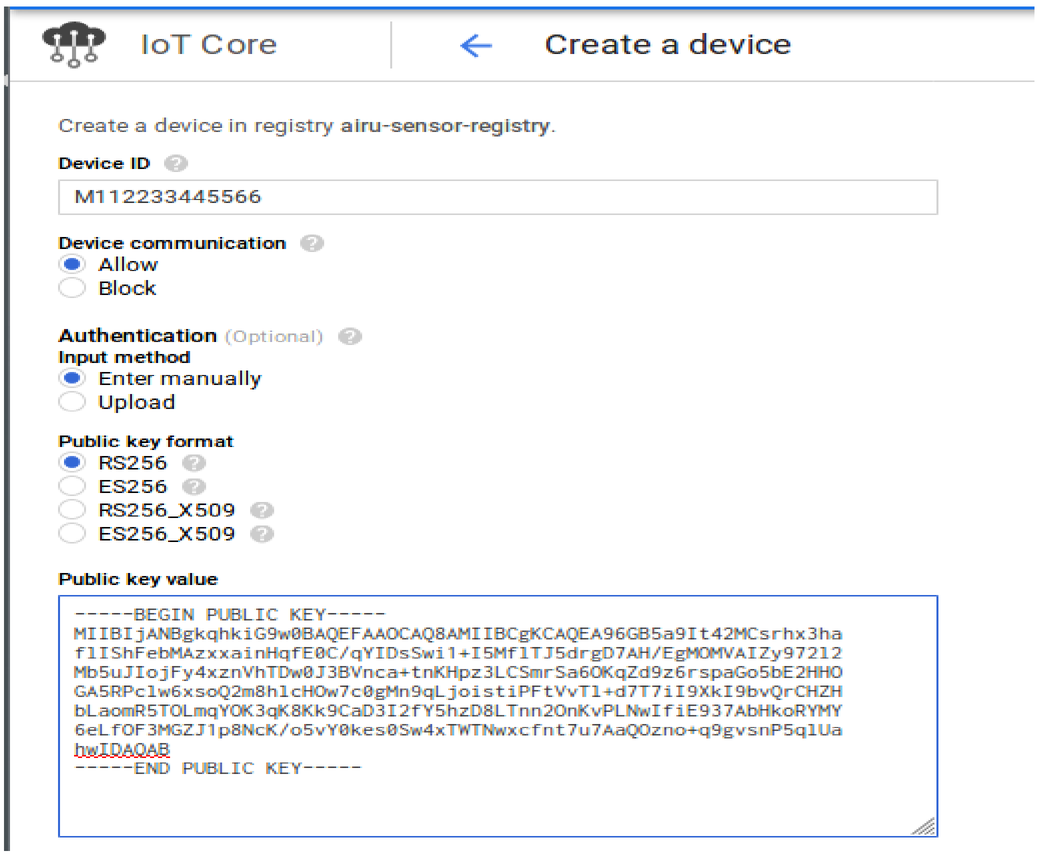
3. Select “airu-sensor-registry.”

4. From the menu along the left side, select “Devices.”

5. From the top of the page, select “+ CREATE A DEVICE.”

6. Fill out the form using the template below. The “Device ID” will be the character ‘M’ followed by the MAC address (replace 112233445566 with the MAC of the device). Do not include any colons when entering the MAC address.

The letter ‘M’ is required because a Device ID cannot start with a number. Therefore, as a design decision throughout GCP the device ID will always be represented by the letter ‘M’ followed by the MAC address in this manner: M112233445566.



Create a device in registry airu-sensor-registry.

Device ID
M112233445566

Device communication
[✓ Allow]
Block

Authentication (Optional)
Input method
[✓ Enter manually]
Upload

Public key format
[✓ RS256]
ES256
RS256_X509
ES256_X509

Public key value

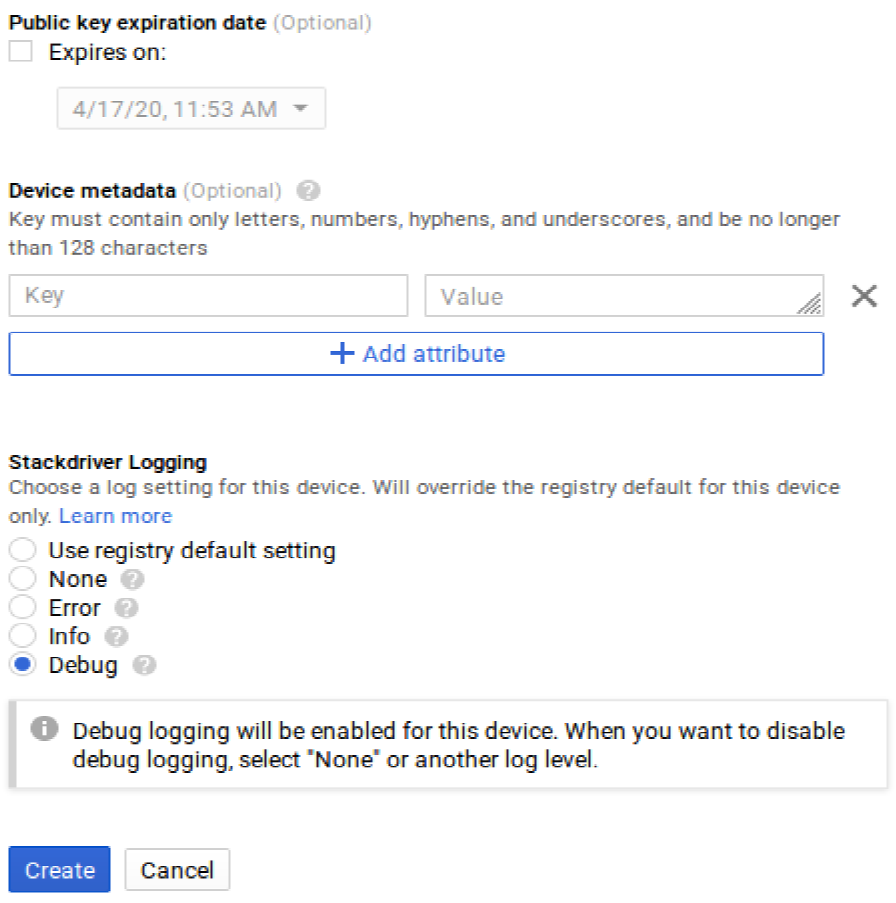
```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA96GB5a9It42MCsrhx3ha
flIShFebMAzxxainHqfE0C/qYIDsSwi1+I5MflTJ5drgD7AH/EgMOMVAIZy97212
Mb5uJIojFy4xznVhTDw0J3BVnca+tnKHpz3LCSmrSa6OKqZd9z6rspaGo5bE2HHO
GA5RPclw6xsoQ2m8hlcHOw7c0gMn9qLjoistiPFtVvTl+d7T7iI9XkI9bvQrCHZH
bLaomR5TOLmqYOK3qK8Kk9CaD3I2fY5hzD8LTnn2OnKvPLNwIfiE937AbHkoRYMY
6eLfOF3MGZJ1p8NcK/o5vY0kes0Sw4xTWTNwxcfnt7u7AaQOzno+q9gvsnP5qlUa
hwIDAQAB
-----END PUBLIC KEY-----
```

7. Copy the public key EXACTLY as found below into the “Public key value” section:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA96GB5a9It42MCsrhx3ha
flIShFebMAzxxainHqfE0C/qYIDsSwi1+I5MflTJ5drgD7AH/EgMOMVAIZy972l2
Mb5uJIojFy4xznVhTDw0J3BVnca+tnKHpz3LCSmrSa6OKqZd9z6rspaGo5bE2HHO
GA5RPclw6xsoQ2m8hlcHOw7c0gMn9qLjoistiPFtVvTl+d7T7iI9XkI9bvQrCHZH
bLaomR5TOLmqYOK3qK8Kk9CaD3I2fY5hzD8LTnn2OnKvPLNwIfiE937AbHkoRYMY
6eLfOF3MGZJ1p8NcK/o5vY0kes0Sw4xTWTNwxcfnt7u7AaQOzno+q9gvsnP5qlUa
hwIDAQAB
-----END PUBLIC KEY-----
```

8. Finish filling out the form using the settings provided below and then select “Create.”

Remark: Selecting “Debug” as the logging setting allows the administrator access to a robust set of logs that can be viewed from within GCP. These logs are very useful in debugging problems that may be occurring at the sensor (firmware) or within GCP itself.



Public key expiration date (Optional)
☐ Expires on:
4/17/20, 11:53 AM

Device metadata (Optional)
Key must contain only letters, numbers, hyphens, and underscores, and be no longer than 128 characters

Key	Value

Stackdriver Logging
Choose a log setting for this device. Will override the registry default for this device only. Learn more

Use registry default setting / None / Error / Info / [✓ Debug]

Debug logging will be enabled for this device. When you want to disable debug logging, select "None" or another log level.

MANAGING DEVICES IN THE DEVICE REGISTRY

Firmware updates occur in two parts. First, create and upload the .bin file to a Google Data Bucket. Second, tell the device to go and get the update and start using it.

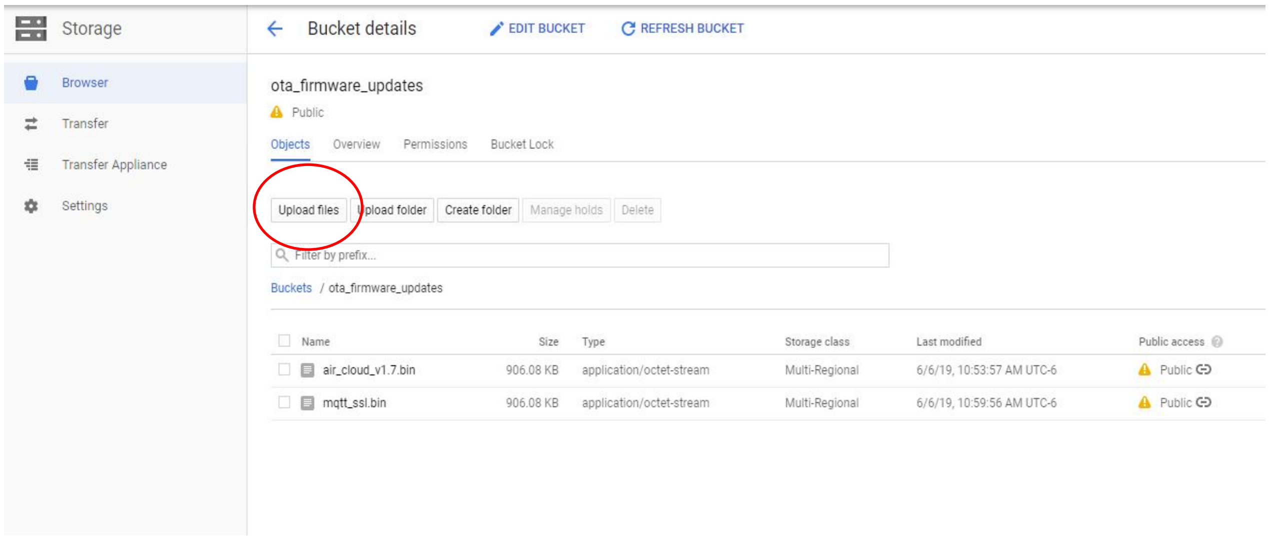
Part 1 – Uploading the .bin file to Google Storage Bucket.

1. Login to Google Cloud Platform (GCP) https://cloud.google.com/

2. From the navigation menu (upper left corner) select “Storage” under the subheading “STORAGE.”

3. Select the appropriate data bucket – default is set to “ota_firmware_updates.”

4. From the button menu along the top of the screen, select “Upload files.”



Name	Size	Type	Storage class	Last modified	Public access
air_cloud_v1.7.bin	906.08 KB	application/octet-stream	Multi-Regional	6/6/19, 10:53:57 AM UTC-6	Public
mqtt_ssl.bin	906.08 KB	application/octet-stream	Multi-Regional	6/6/19, 10:59:56 AM UTC-6	Public

5. Browse to the appropriate .bin file and save to the bucket.

Note: The default setting for this bucket is public. The allows the device access to download the specified file without any additional requirement for authentication.

Part 2 – Setting device configuration to upload and run on new firmware.

1. Login to Google Cloud Platform console (https://cloud.google.com/).

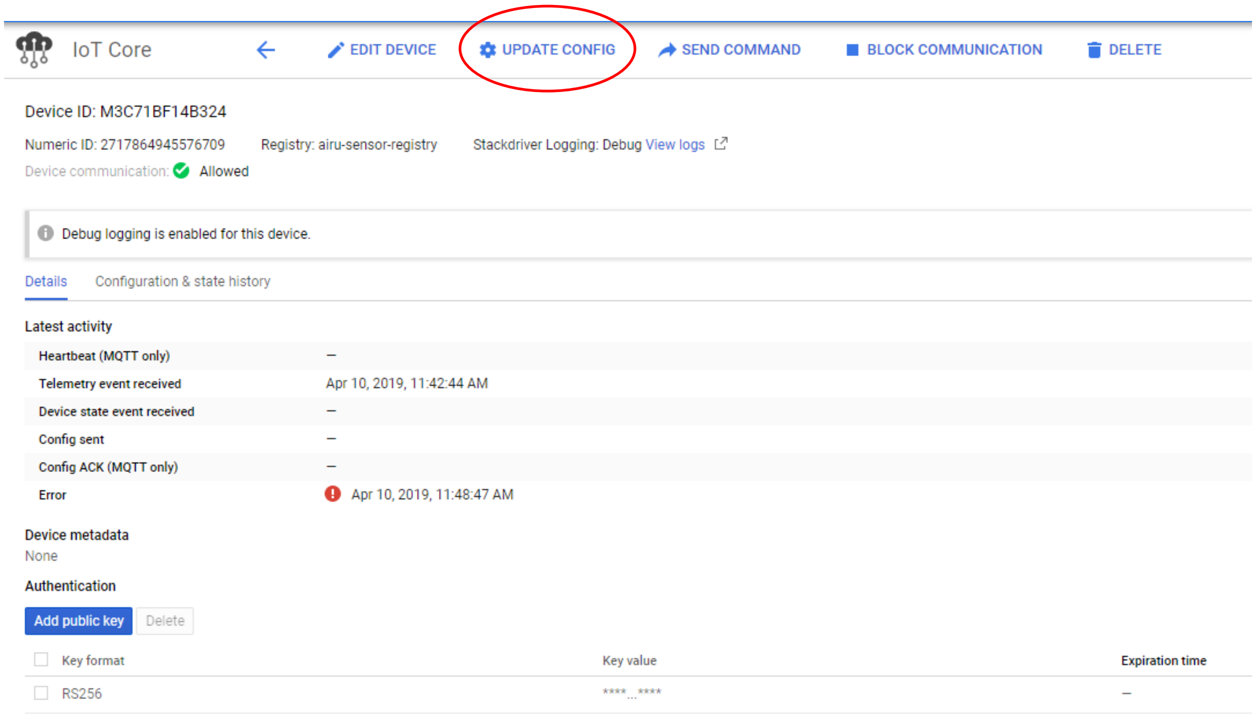
2. From the navigation menu (upper left corner) select “IoT Core.”

3. Select “airu-sensor-registry” or applicable device registry.

4. From the menu along the left side, select “Devices.”

5. Click on the “Device ID” of the desired device to update.

6. Select “UPDATE CONFIG” from along the top part of the screen.



Device ID: M3C71BF14B324

Numeric ID: 2717864945576709 Registry: airu-sensor-registry Stackdriver Logging: Debug View logs

Device communication: ✓ Allowed

Debug logging is enabled for this device.

Details Configuration & state history

Latest activity

Heartbeat (MQTT only)	–
Telemetry event received	Apr 10, 2019, 11:42:44 AM
Device state event received	–
Config sent	–
Config ACK (MQTT only)	–
Error	Apr 10, 2019, 11:48:47 AM

Device metadata

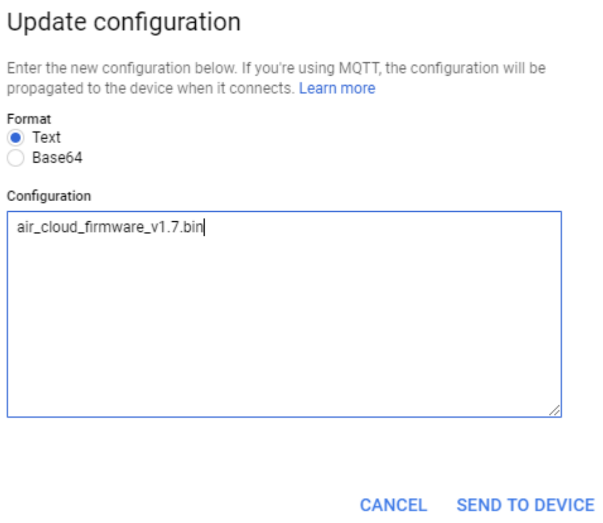
None

Authentication

Add public key Delete

Key format	Key value	Expiration time
RS256	****...****	–

7. Within the update configuration dialog box, select “Text” as the format and enter the name of the .bin firmware file in the Configuration box. This file name must match exactly with the file name uploaded in part 1.



Update configuration

Enter the new configuration below. If you're using MQTT, the configuration will be propagated to the device when it connects. Learn more

Format

[✓ Text]

Base64

Configuration

air_cloud_firmware_v1.7.bin

8. Click “SEND TO DEVICE” to finalize the process. The device will immediately upload the new firmware and restart.

Note: To validate that a device correctly updated and is running on new firmware – check the “State” of the device. The “State” contains the firmware file the device is currently using. See Checking State section of the documentation.

Checking device “State” through Google IoT Core
Last updated: 10 June 2019 by Scott Gale

Each device within IoT contains a State. State is used to indicate the firmware the sensor is currently running.

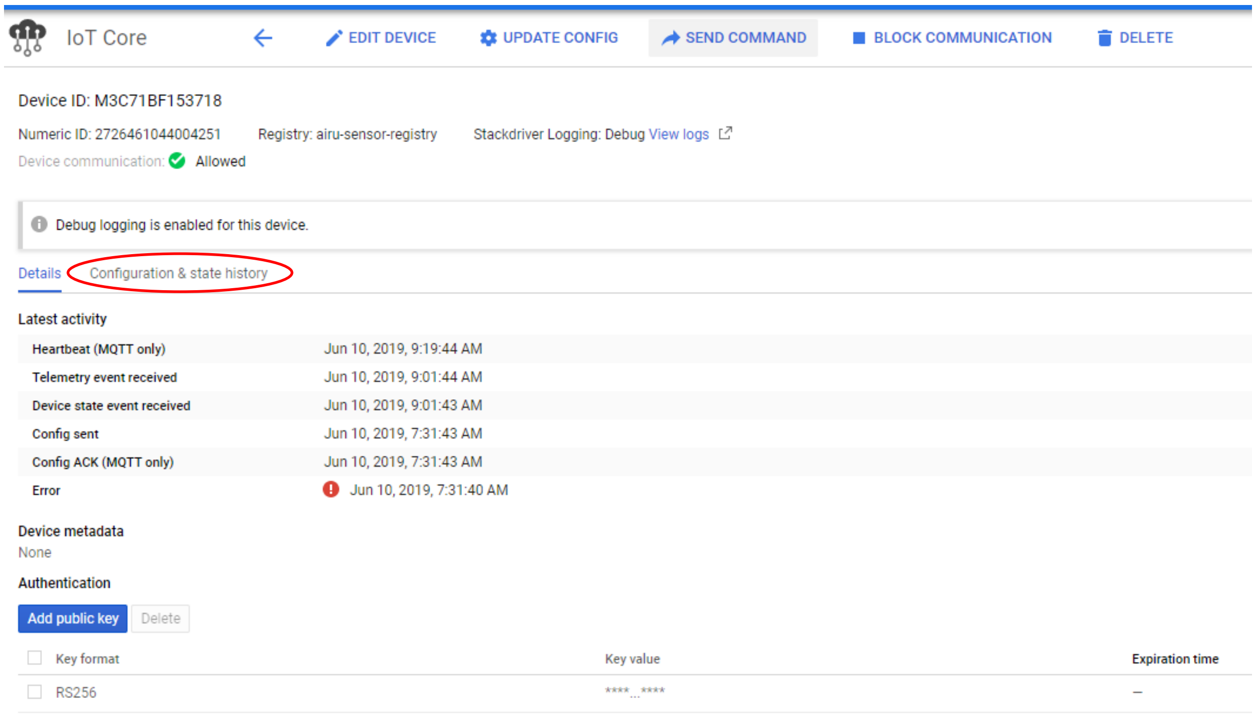
1. Login to Google Cloud Platform (https://cloud.google.com/)

2. From the navigation menu (upper left corner) select “IoT Core.”

3. Select “airu-sensor-registry” or applicable device registry.

4. From the menu along the left side, select “Devices.”

5. Click on the “Device ID” of interest.



Device ID: M3C71BF153718

Numeric ID: 2726461044004251 Registry: airu-sensor-registry Stackdriver Logging: Debug View logs

Device communication: Allowed

Debug logging is enabled for this device.

Details Configuration & state history

Latest activity

Heartbeat (MQTT only)	Jun 10, 2019, 9:19:44 AM
Telemetry event received	Jun 10, 2019, 9:01:44 AM
Device state event received	Jun 10, 2019, 9:01:43 AM
Config sent	Jun 10, 2019, 7:31:43 AM
Config ACK (MQTT only)	Jun 10, 2019, 7:31:43 AM
Error	Jun 10, 2019, 7:31:40 AM

Device metadata
None

Authentication

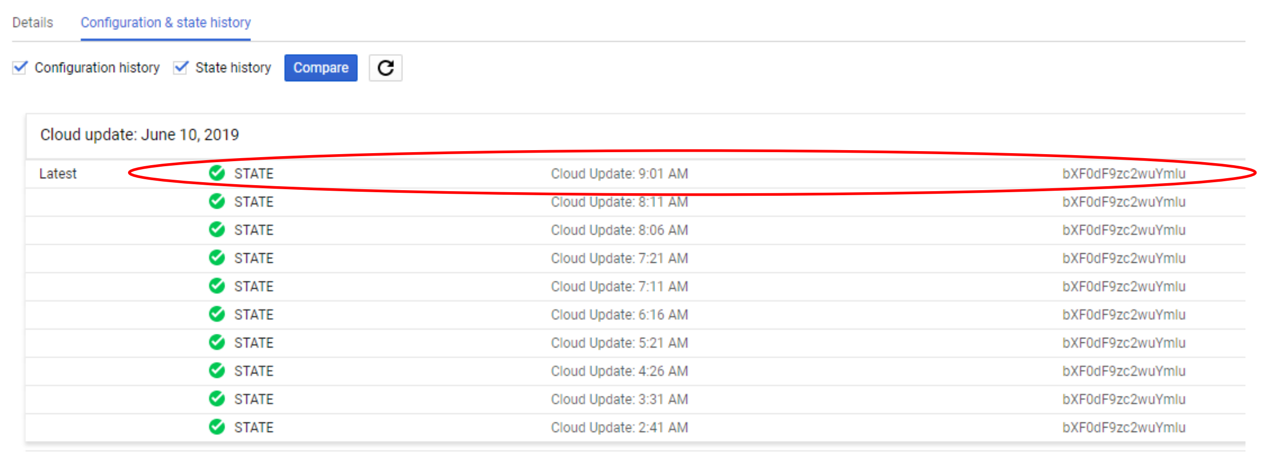
Add public key Delete

Key format	Key value	Expiration time
RS256	****...****	–

Note: The above screen will appear after you select a device. Many useful pieces of information can be obtained here such as the last telemetry event published, when the last configuration was sent, and when the most recent state was reported.

6. To see the State of the device, click on “Configuration & state history.”

7. Then click on the latest STATE publication.

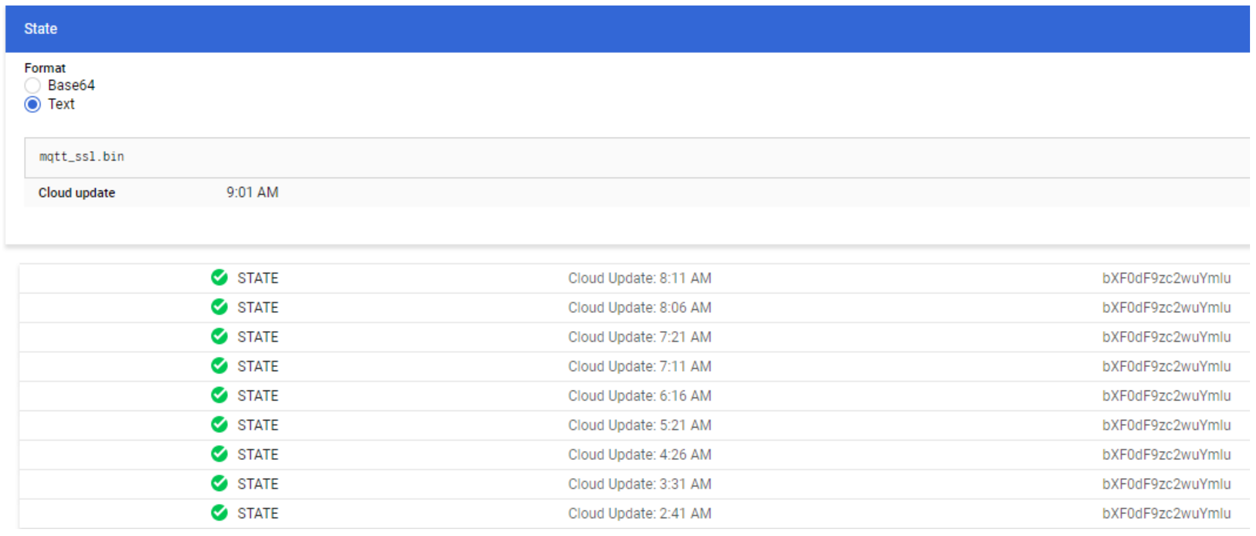

Details | Configuration & state history

✓ Configuration history ✓ State history Compare

Cloud update: June 10, 2019

Latest	STATE	Cloud Update: 9:01 AM	bXF0dF9zc2wuYmlu
	STATE	Cloud Update: 8:11 AM	bXF0dF9zc2wuYmlu
	STATE	Cloud Update: 8:06 AM	bXF0dF9zc2wuYmlu
	STATE	Cloud Update: 7:21 AM	bXF0dF9zc2wuYmlu
	STATE	Cloud Update: 7:11 AM	bXF0dF9zc2wuYmlu
	STATE	Cloud Update: 6:16 AM	bXF0dF9zc2wuYmlu
	STATE	Cloud Update: 5:21 AM	bXF0dF9zc2wuYmlu
	STATE	Cloud Update: 4:26 AM	bXF0dF9zc2wuYmlu
	STATE	Cloud Update: 3:31 AM	bXF0dF9zc2wuYmlu
	STATE	Cloud Update: 2:41 AM	bXF0dF9zc2wuYmlu

8. Next, select “Text” to view the State in ascii. The text value will indicate the firmware that is currently running on the sensor. If you recently updated the configuration of the device – this value should reflect the new .bin file that was uploaded.


State

Format
Base64 [✓ Text]

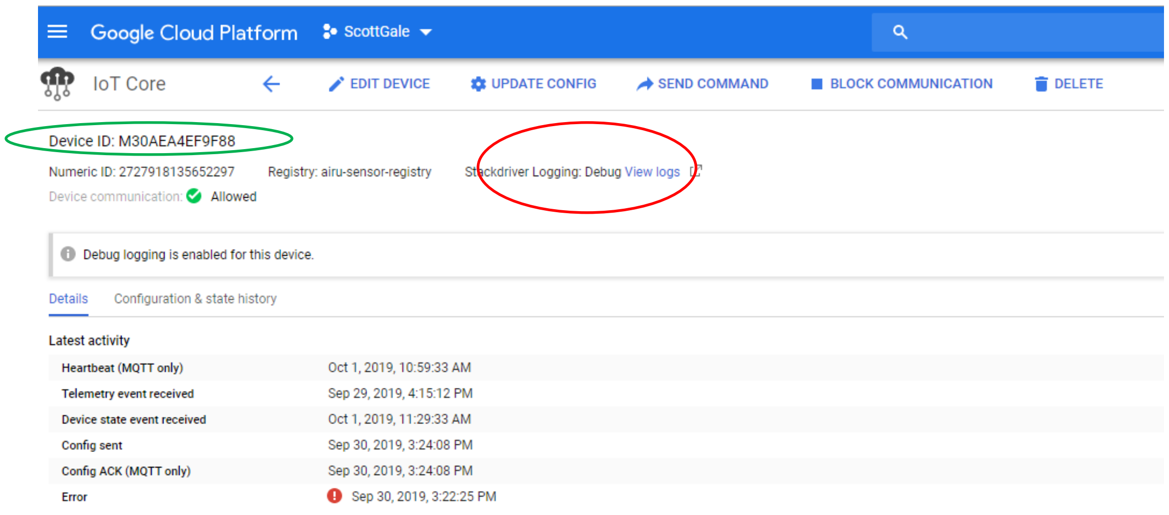
```
mqtt_ssl.bin
```

Cloud update 9:01 AM

STATE	Cloud Update: 8:11 AM	bXF0dF9zc2wuYmlu
STATE	Cloud Update: 8:06 AM	bXF0dF9zc2wuYmlu
STATE	Cloud Update: 7:21 AM	bXF0dF9zc2wuYmlu
STATE	Cloud Update: 7:11 AM	bXF0dF9zc2wuYmlu
STATE	Cloud Update: 6:16 AM	bXF0dF9zc2wuYmlu
STATE	Cloud Update: 5:21 AM	bXF0dF9zc2wuYmlu
STATE	Cloud Update: 4:26 AM	bXF0dF9zc2wuYmlu
STATE	Cloud Update: 3:31 AM	bXF0dF9zc2wuYmlu
STATE	Cloud Update: 2:41 AM	bXF0dF9zc2wuYmlu

IoT STACKDRIVER LOGGING (Debug logs for each device)

Stackdriver logging provides very useful debugging capabilities when working with devices. Logging allows an administrator to view details about the device and its connection with GCP. Every time a device connects, subscribes, publishes, disconnects, etc there is a log created. To view the logs navigate to the device registry and select the device you want to view. In this example we are looking at the device highlighted by the green circle. Select the “View logs” link shown by the red circle.



Device ID: M30AEA4EF9F88

Numeric ID: 2727918135652297 Registry: airu-sensor-registry Stackdriver Logging: Debug View logs

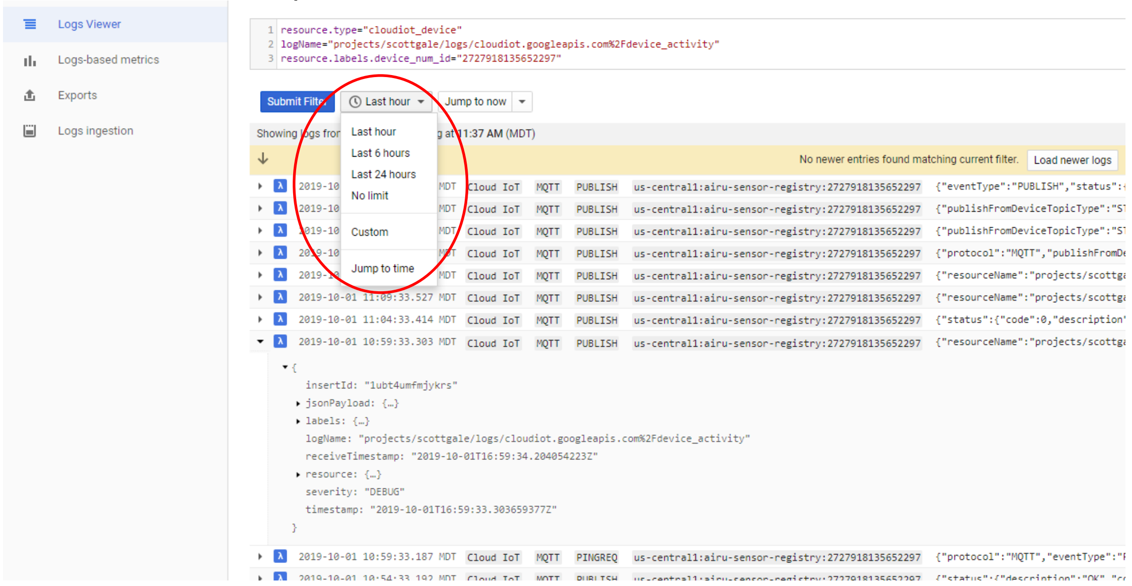
Device communication: Allowed

Debug logging is enabled for this device.

Latest activity

Heartbeat (MQTT only)	Oct 1, 2019, 10:59:33 AM
Telemetry event received	Sep 29, 2019, 4:15:12 PM
Device state event received	Oct 1, 2019, 11:29:33 AM
Config sent	Sep 30, 2019, 3:24:08 PM
Config ACK (MQTT only)	Sep 30, 2019, 3:24:08 PM
Error	Sep 30, 2019, 3:22:25 PM

After selecting the “View logs” link the actual logging window will appear. Logs within the last hour will load as a default; however, logs from any period can be queried and reviewed. There is a drop-down menu that will allow you specify a custom time period and review those specific logs (useful to select the time the sensor went offline to determine the cause).



```
resource.type="cloudiot_device"
logName="projects/scottgale/logs/cloudiot.googleapis.com%2Fdevice_activity"
resource.labels.device_num_id="2727918135652297"
```

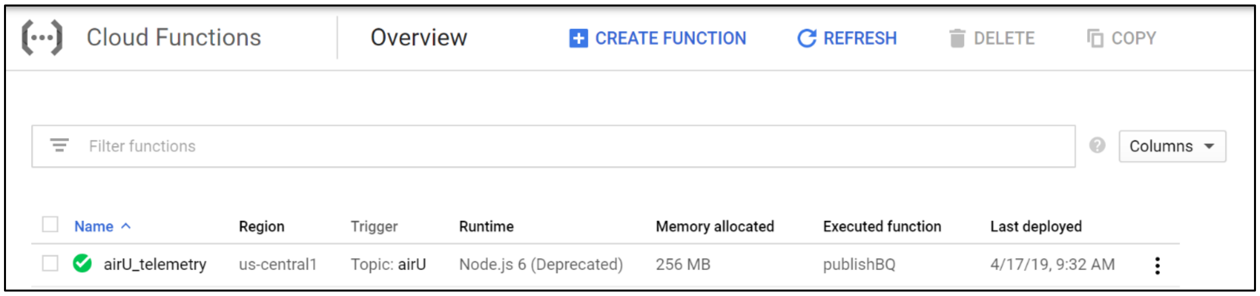
Showing logs fro… …g at 11:37 AM (MDT)

```
2019-10…                    MDT Cloud IoT MQTT PUBLISH us-central1:airu-sensor-registry:2727918135652297 {"eventType":"PUBLISH","status":
2019-10…                    MDT Cloud IoT MQTT PUBLISH us-central1:airu-sensor-registry:2727918135652297 {"publishFromDeviceTopicType":"S
2019-10…                    MDT Cloud IoT MQTT PUBLISH us-central1:airu-sensor-registry:2727918135652297 {"publishFromDeviceTopicType":"S
2019-10…                    MDT Cloud IoT MQTT PUBLISH us-central1:airu-sensor-registry:2727918135652297 {"protocol":"MQTT","publishFromDe
2019-10…                    MDT Cloud IoT MQTT PUBLISH us-central1:airu-sensor-registry:2727918135652297 {"resourceName":"projects/scottga
2019-10-01 11:09:33.527 MDT Cloud IoT MQTT PUBLISH us-central1:airu-sensor-registry:2727918135652297 {"resourceName":"projects/scottga
2019-10-01 11:04:33.414 MDT Cloud IoT MQTT PUBLISH us-central1:airu-sensor-registry:2727918135652297 {"status":{"code":0,"description"
2019-10-01 10:59:33.303 MDT Cloud IoT MQTT PUBLISH us-central1:airu-sensor-registry:2727918135652297 {"resourceName":"projects/scottga
  {
    insertId: "1ubt4umfmjykrs"
    jsonPayload: {…}
    labels: {…}
    logName: "projects/scottgale/logs/cloudiot.googleapis.com%2Fdevice_activity"
    receiveTimestamp: "2019-10-01T16:59:34.204054223Z"
    resource: {…}
    severity: "DEBUG"
    timestamp: "2019-10-01T16:59:33.303659377Z"
  }
2019-10-01 10:59:33.187 MDT Cloud IoT MQTT PINGREQ us-central1:airu-sensor-registry:2727918135652297 {"protocol":"MQTT","eventType":"F
2019-10-01 10:54:33.192 MDT Cloud IoT MQTT PUBLISH us-central1:airu-sensor-registry:2727918135652297 {"status":{"description":"OK","c
```

CLOUD FUNCTION

The cloud function is simply a forwarding mechanism that takes sensor data published to IoT Core and inserts it into the BigQuery database. It is an event driven stand-alone function that is triggered each time sensor data is published to IoT Core. The function simply converts the sensor data to JSON format and then inserts into the database.

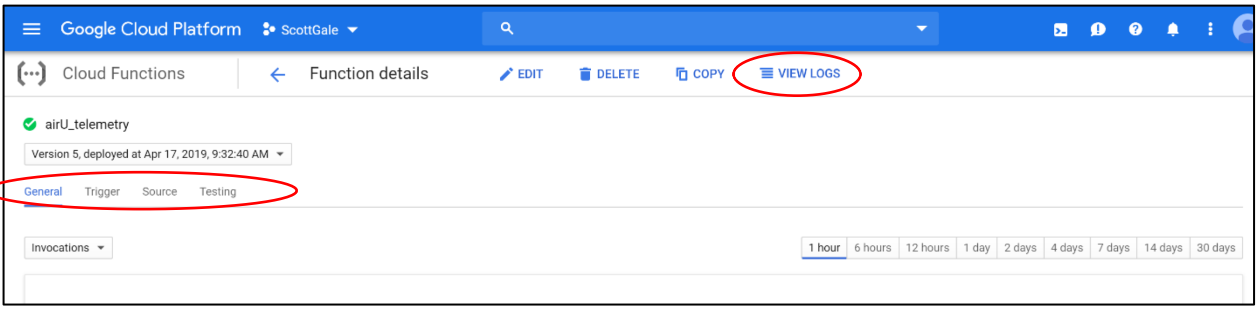
To view or make changes to the function goto the GCP console and click on the Navigation Menu icon in the upper left corner. From the dropdown list scroll down the to the compute section and select Cloud Functions.



Name	Region	Trigger	Runtime	Memory allocated	Executed function	Last deployed
airU_telemetry	us-central1	Topic: airU	Node.js 6 (Deprecated)	256 MB	publishBQ	4/17/19, 9:32 AM

This overview screen shows the name of the function, the trigger (MQTT topic to which sensor data is published) and the executed function. The executed function is the function that is called when the trigger is met. The function itself is written in JavaScript.

By clicking on the function, you can get detailed information.



airU_telemetry

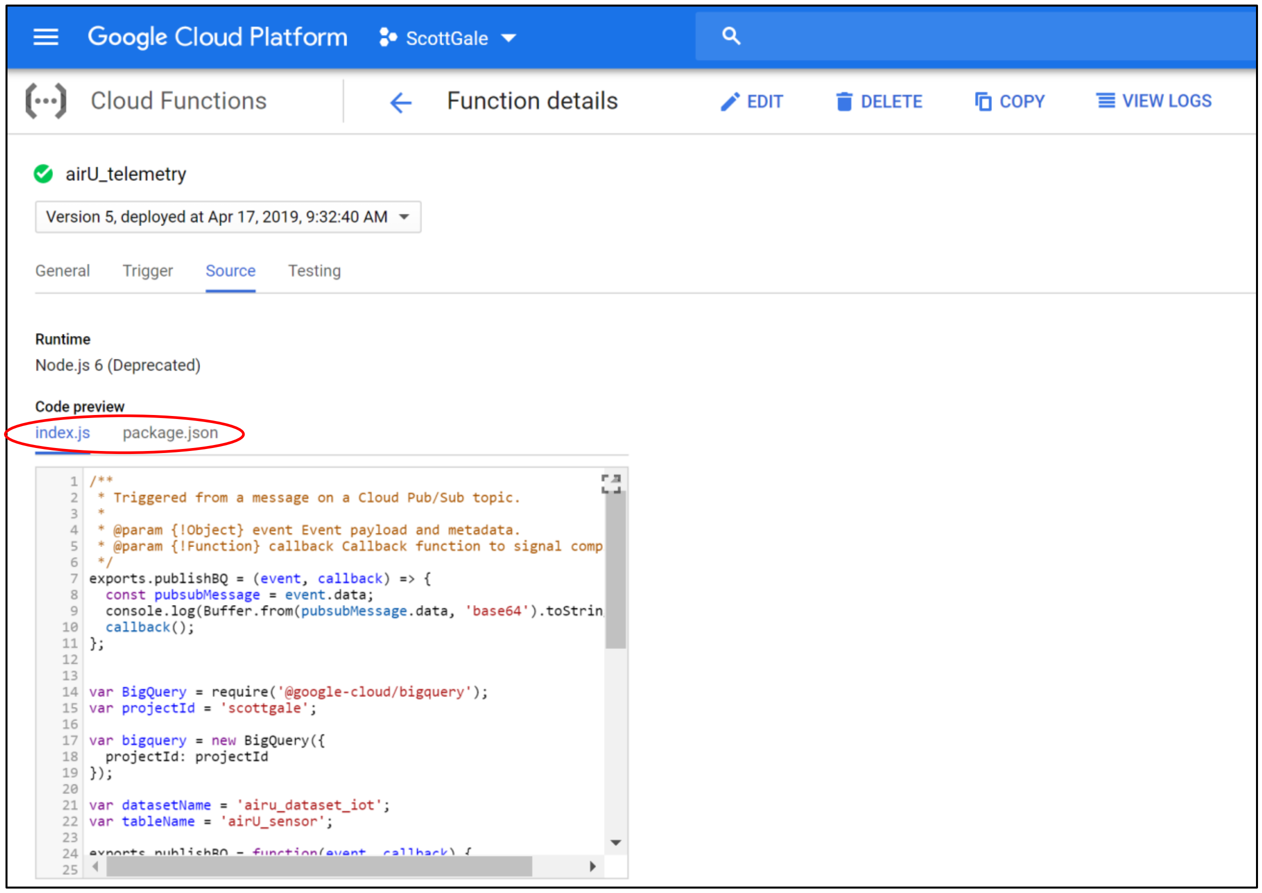
Version 5, deployed at Apr 17, 2019, 9:32:40 AM

You can view / edit the source code by clicking on "Source." Note that any time you change the source code the function must be deployed again. Additionally, for debugging purposed you can insert console.log("insert string here") statements within the code similar to what you might do if developing a web application. The logs can then be view by clicking on the "View Logs" menu button. This is very helpful to view and understand the content of variables.

The code is only about 50 lines of JavaScript. Be very careful when making changes. The reason this function works and can insert published data into the database is because the labels within the MQTT packet MATCH the field names in the database

EXACTLY. This is important because if you change the database table schema or change the packet data string (char[ ]) in the firmware this function will not work.

Lastly, there is a separate file called “package.json” that contains configuration information. Be sure to check and update this if changes are made to the source code; otherwise the function may not work.



```
/**
 * Triggered from a message on a Cloud Pub/Sub topic.
 *
 * @param {!Object} event Event payload and metadata.
 * @param {!Function} callback Callback function to signal comp
 */
exports.publishBQ = (event, callback) => {
  const pubsubMessage = event.data;
  console.log(Buffer.from(pubsubMessage.data, 'base64').toStrin
  callback();
};


var BigQuery = require('@google-cloud/bigquery');
var projectId = 'scottgale';

var bigquery = new BigQuery({
  projectId: projectId
});

var datasetName = 'airu_dataset_iot';
var tableName = 'airU_sensor';

exports.publishBQ = function(event, callback) {
```

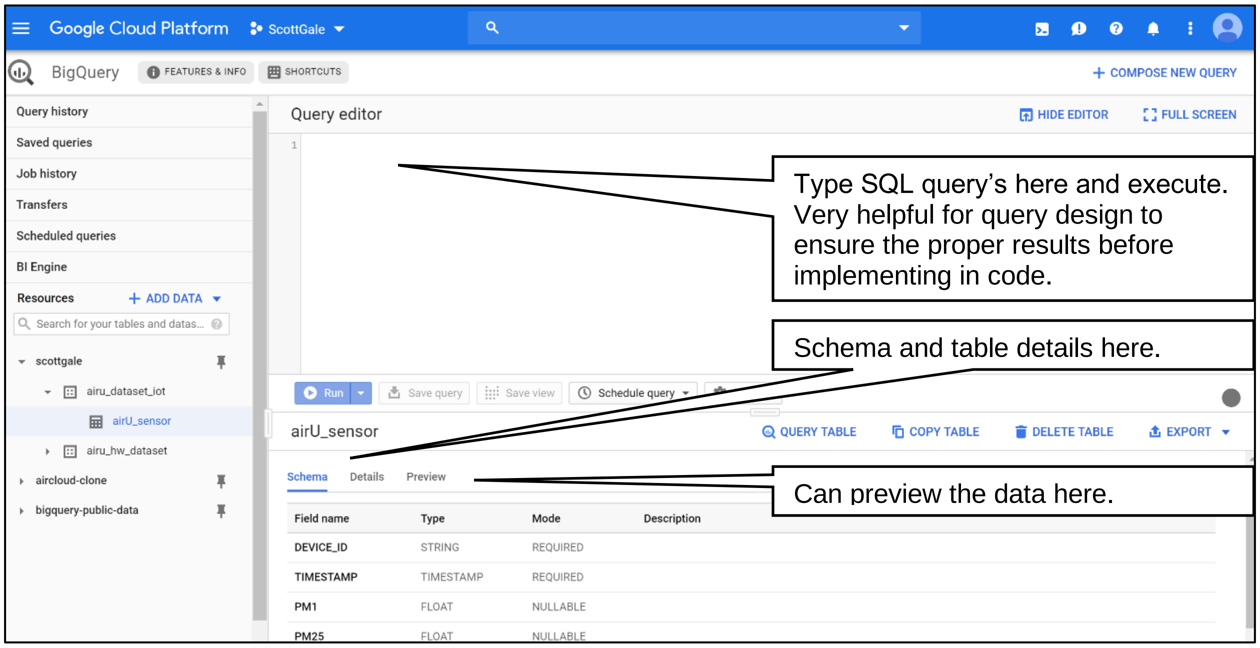
BIGQUERY DATABSE

Overview: https://cloud.google.com/bigquery/

The BigQuery database was selected because of its design features and ability to handle large amounts of data. It scales to the petabyte size, handles replicating data across multiple geographic locations, and facilitates data backup all automatically. BigQuery is a relational database with traditional tables and fields. It is also compatible with standard SQL which makes querying very convenient.

Navigate to the BigQuery console by clicking on the Navigation menu in the GCP console in the upper left-hand corner. Scroll down to the Big Data section and select BigQuery.

Available datasets are located along the left-hand side of the screen. Each datasets can contain multiple tables. Locate the airu_dataset_iot under the scottgale project and click on the airU_sensor table. This is where all the sensor data is stored.



Field name	Type	Mode	Description
DEVICE_ID	STRING	REQUIRED	
TIMESTAMP	TIMESTAMP	REQUIRED	
PM1	FLOAT	NULLABLE	
PM25	FLOAT	NULLABLE	

One major limitation with BigQuery is that records CANNOT be edited. Once a recorded is saved in the database it cannot be changed. The only way to make a change is to create a new record with the change and delete the old record.

The Cloud Function writes sensor data to BigQuery and the Web App reads data from BigQuery. These are the only functional blocks that interact with the database. An example of how to query the database is detailed in the WEB APPLICATION section of this documentation.

FIREBASE AUTHENTICATION AND DATABASE

Authentication: Firebase provides a very flexible set of tools to manage user credentials and data. Firebase falls under the Google umbrella; however, they have their own console that can be accessed here: https://console.firebase.google.com. The authentication and database tools / API are separate and require separate objects to access. Firebase authentication uses the firestore_auth and pyrebase_auth objects and the Firebase database uses the firestore_client object.

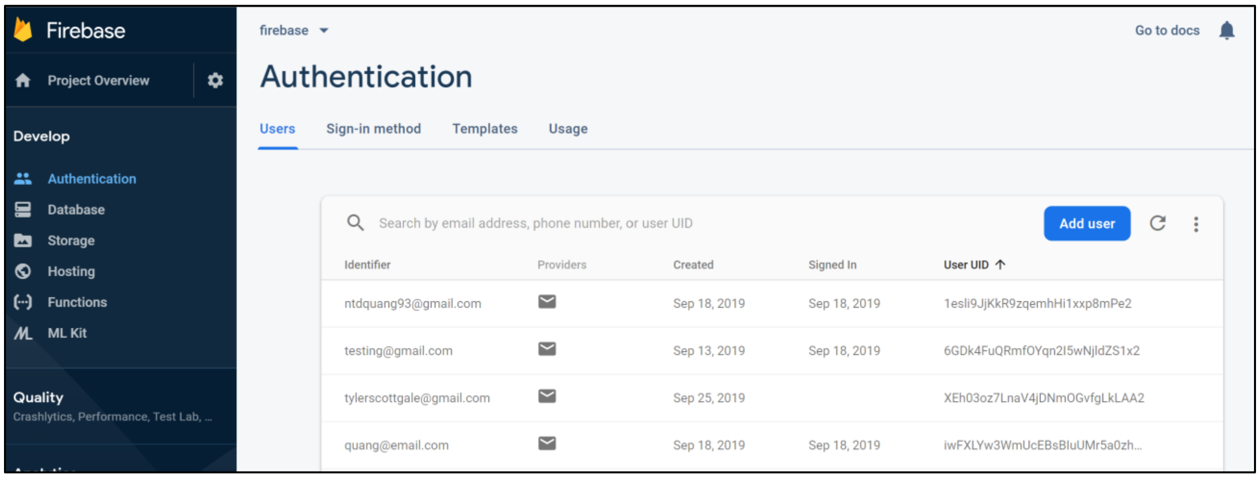
The pyrebase_auth object is essentially a wrapper API for the standard Firebase authentication API that firestore_auth utilizes. The wrapper API provides some nice and convenient functions, so I use both objects depending on what I want to do.

API: https://firebase.google.com/docs/reference/admin/python/firebase_admin.auth
Pyrebase4 wrapper API: https://github.com/nhorvath/Pyrebase4

Firebase authentication allows us to take a very hands-off approach to storing user passwords. Firebase does the work for us. Once a user registers through the Registration Form, we take their email and password and create a new Firebase user. The code looks like this from routes.py:

```
# Create FIREBASE AUTHENTICATION CREDENTIALS
firestore_auth.create_user(
    email=form.email.data,
    password=form.password.data)
```

Once a user is created you can view the information in the Firebase console by selecting Authentication on the left side of the screen. Here you can manage current user profiles. Once a user authentication is created, Firebase sends an email to the user to validate their account. The account IS NOT ACTIVE until the user clicks a link in their email.



Identifier	Providers	Created	Signed In	User UID ↑
ntdquang93@gmail.com	✉	Sep 18, 2019	Sep 18, 2019	1esli9JjKkR9zqemhHi1xxp8mPe2
testing@gmail.com	✉	Sep 13, 2019	Sep 18, 2019	6GDk4FuQRmfOYqn2I5wNjldZS1x2
tylerscottgale@gmail.com	✉	Sep 25, 2019		XEh03oz7LnaV4jDNmOGvfgLkLAA2
quang@email.com	✉	Sep 18, 2019	Sep 18, 2019	iwFXLYw3WmUcEBsBluUMr5a0zh...

Database: Firebase database is a NOSQL or non-relational database. The structure uses collections and documents. A collection is like a table and a document is like an entry into that table. There is no fixed schema so within the same collection we could have completely different information. The database is accessed through the firestore_client object created in _init_.py.

API: https://firebase.google.com/docs/reference/admin/python/firebase_admin.db

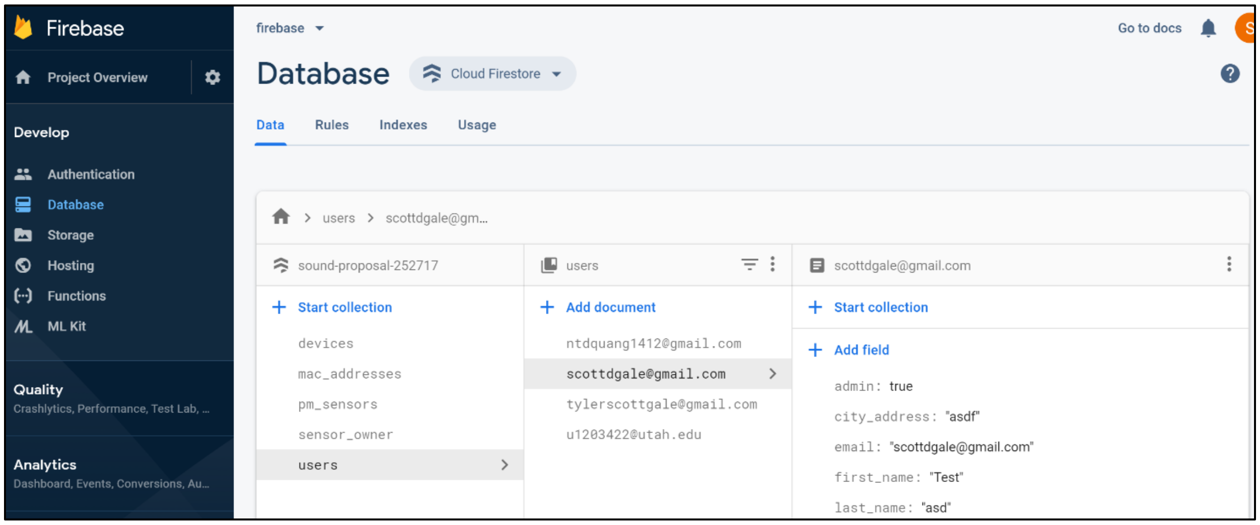
We store all user information (name, address, email, etc) in the users collection. Although we are not required to adhere to a fixed schema – we do anyway so that we can accurately query data by agreed upon names and get consistent results. All documents in the users collection are named by the email address of the user they represent. A document within the users collection has the following attributes:

admin: a boolean value that is either “true” or “false”
first_name: Users first name
last_name: User’s last name
email: User’s email address (used as the name of the document)
city_address: Street address
state_address: State
zip_code: Zip code

The admin field is used if you want to grant a user access to all sensors. When a user clicks on a sensor after logging into the web application, the program checks first to see if the user is an admin (admin == true). If not, then the program checks to see if the user is the owner of the sensor.

Sensor owner information is contained in the sensor_owner collection. Each document in the sensor_owner collection is named after the MAC address of the device it represents. No two documents can be the same (that would mean that a sensor was owned by two different people). Documents within the sensor_owner collection contain the following attributes:

email: Email address of the owner – must match email in the users collection
mac_address: MAC address of sensor owned by user with the associated email



devices
mac_addresses
pm_sensors
sensor_owner
users

ntdquang1412@gmail.com
scottdgale@gmail.com
tylerscottgale@gmail.com
u1203422@utah.edu

admin: true
city_address: "asdf"
email: "scottdgale@gmail.com"
first_name: "Test"
last_name: "asd"

WEB APPLICATION

Repository: https://github.com/LNIS-Projects/AirU-GCP

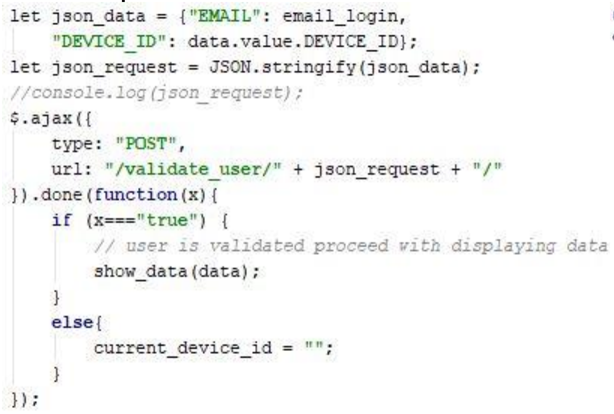
The GCP web app is built around the Python – Flask framework and utilizes JavaScript (D3 library) for the custom visualizations. Firebase is used for user authentication and provides the database for storing user information. Firebase authentication is a service provided by GCP and handles password management and email authentication.

The web app provides the following functionality:

1. Visualization of all deployed sensors color coded with current air quality.
1a. Historical sensor data for authenticated users.
2. User account creation / registration and sensor requests.
3. User login (authentication) to view detailed sensor information.
4. Password management – reset password functionality.

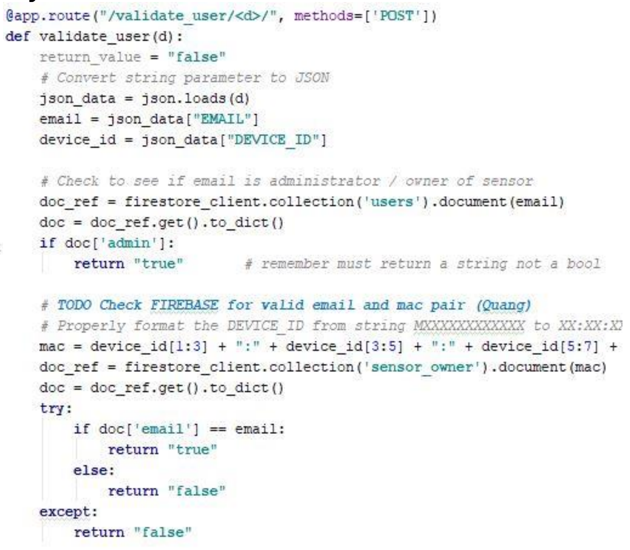
JavaScript – Python Communication: Most of the functionality is handled in Python; however, because the visualization computation is done in Javascript there needs to be a clean way for the Javascript to communicate to the Python server. Here is a basic framework to make Javascript Requests to the Python server. Both parameters and return values are transmitted as JSON strings. Both Javascript and Python have JSON classes that simply transforming JSON objects into string and vice versa.

Javascript Code:



```
let json_data = {"EMAIL": email_login,
    "DEVICE_ID": data.value.DEVICE_ID};
let json_request = JSON.stringify(json_data);
//console.log(json_request);
$.ajax({
    type: "POST",
    url: "/validate_user/" + json_request + "/"
}).done(function(x){
    if (x==="true") {
        // user is validated proceed with displaying data
        show_data(data);
    }
    else{
        current_device_id = "";
    }
});
```

Python Code:

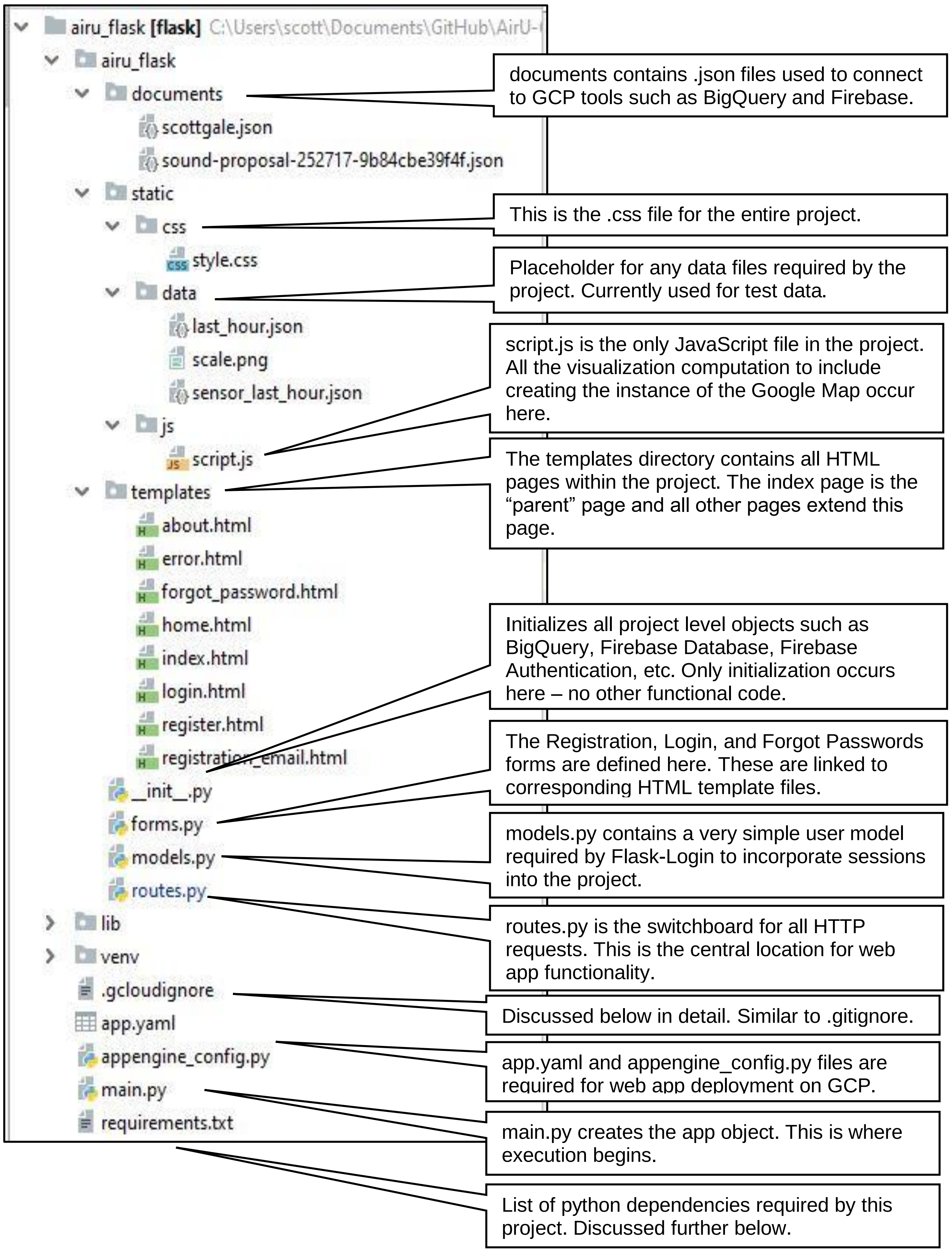


```
@app.route("/validate_user/<d>/", methods=['POST'])
def validate_user(d):
    return_value = "false"
    # Convert string parameter to JSON
    json_data = json.loads(d)
    email = json_data["EMAIL"]
    device_id = json_data["DEVICE_ID"]

    # Check to see if email is administrator / owner of sensor
    doc_ref = firestore_client.collection('users').document(email)
    doc = doc_ref.get().to_dict()
    if doc['admin']:
        return "true"       # remember must return a string not a bool

    # TODO Check FIREBASE for valid email and mac pair (Quang)
    # Properly format the DEVICE_ID from string MXXXXXXXXXXXX to XX:XX:X
    mac = device_id[1:3] + ":" + device_id[3:5] + ":" + device_id[5:7] +
    doc_ref = firestore_client.collection('sensor_owner').document(mac)
    doc = doc_ref.get().to_dict()
    try:
        if doc['email'] == email:
            return "true"
        else:
            return "false"
    except:
        return "false"
```

File Structure: Below is a representation of the file structure for the web app. The file structure is deliberately organized in this fashion to support best practices while working within the flask framework. If you are familiar with flask then this structure should be intuitive. I will provide a brief explanation of critical files and directories and how they are used.



```
airu_flask [flask] C:\Users\scott\Documents\GitHub\AirU-
  airu_flask
    documents
      scottgale.json
      sound-proposal-252717-9b84cbe39f4f.json
    static
      css
        style.css
      data
        last_hour.json
        scale.png
        sensor_last_hour.json
      js
        script.js
    templates
      about.html
      error.html
      forgot_password.html
      home.html
      index.html
      login.html
      register.html
      registration_email.html
    __init__.py
    forms.py
    models.py
    routes.py
  lib
  venv
  .gcloudignore
  app.yaml
  appengine_config.py
  main.py
  requirements.txt
```

documents contains .json files used to connect to GCP tools such as BigQuery and Firebase.

This is the .css file for the entire project.

Placeholder for any data files required by the project. Currently used for test data.

script.js is the only JavaScript file in the project. All the visualization computation to include creating the instance of the Google Map occur here.

The templates directory contains all HTML pages within the project. The index page is the "parent" page and all other pages extend this page.

Initializes all project level objects such as BigQuery, Firebase Database, Firebase Authentication, etc. Only initialization occurs here – no other functional code.

The Registration, Login, and Forgot Passwords forms are defined here. These are linked to corresponding HTML template files.

models.py contains a very simple user model required by Flask-Login to incorporate sessions into the project.

routes.py is the switchboard for all HTTP requests. This is the central location for web app functionality.

Discussed below in detail. Similar to .gitignore.

app.yaml and appengine_config.py files are required for web app deployment on GCP.

main.py creates the app object. This is where execution begins.

List of python dependencies required by this project. Discussed further below.

The .gcloudignore file is very important as it defines files and directories that should not be uploaded to the cloud and included as part of the web app. This file works that same as a .ignore file. If you have additional files or directories that are not part of the web app your working directory they need to be included in this file. Google limits the number of files (10,000 I think) that can be uploaded to support the web app. For instance, my virtual environment folder is part of the file structure should not be included in my web app upload. If I fail to annotate my virtual environment folder in the .gcloudignore my web app will not work.

Environmental Variables

Deploying an update to GCP:

1. Open a terminal window and navigate to the root directory of the project. On my computer it looks like this: C:\Users\scott\Documents\GitHub\AirU-GCP\airu_flask>

2. Ensure that the requirements.txt file is up to date and that there are not uses libraries in this file. Simply doing a $pip freeze command and copying the results into this file will likely yield a lot of overhead and copying of unused modules.

3. Install / update / copy all requirements into the /lib folder through the following command: $pip install --upgrade -t lib -r requirements.txt

This will copy or upgrade all the requirements contained in requirements.txt to the lib directory (used for app deployment).

4. Once the requirements.txt have been updated, run $gcloud app deploy from the same project root directory. This process takes about 5 minutes and will deploy your updated project to GCP.

Querying sensor data from BigQuery: To access the BigQuery API we create a BigQuery object with the name bq. This is done in the init.py file. As noted earlier, I always engineer my queries using the BigQuery console to ensure I am getting expected results. Remember, if you don’t need the data then exclude those fields from your SELECT statement to minimize the data transmission over the Internet.

Querying data is one of the most expensive (in terms of $$$) in this project. We are not charged by the query but by the data each query process. The first 1TB of data query is free and after that we are charged. The amount of data that is queried can quickly add up. This will likely be an area for optimization in the future once we deploy a large number of sensors.

Below is a code example of a BigQuery query that is used within the routes.py file to get the most recent data information for every deployed sensor. Once the data is retrieved from the query I iterate through each row and create a JSON object that is used to send to the browser for visualization in JavaScript.

```
@app.route("/request_data_flask/<d>/", methods=['POST'])
def request_data_flask(d):
    sensor_list = []
    # get the latest sensor data from each sensor
    q = ("SELECT `scottgale.airu_dataset_iot.airU_sensor`.DEVICE_ID, TIMESTAMP, PM1, PM25, PM10, 
LAT, LON, TEMP, CO, NOX, HUM "
         "FROM `scottgale.airu_dataset_iot.airU_sensor` "
         "INNER JOIN (SELECT DEVICE_ID, MAX(TIMESTAMP) as maxts "
         "FROM `scottgale.airu_dataset_iot.airU_sensor` GROUP BY DEVICE_ID) mr "
         "ON `scottgale.airu_dataset_iot.airU_sensor`.DEVICE_ID = mr.DEVICE_ID AND TIMESTAMP = 
maxts;")

    query_job = bq_client.query(q)
    rows = query_job.result()  # Waits for query to finish
    for row in rows:
        sensor_list.append({"DEVICE_ID": str(row.DEVICE_ID),
                            "LAT": row.LAT,
                            "LON": row.LON,
                            "TIMESTAMP": str(row.TIMESTAMP),
                            "PM1": row.PM1,
                            "PM25": row.PM25,
                            "PM10": row.PM10,
                            "TEMP": row.TEMP,
                            "HUM": row.HUM,
                            "NOX": row.NOX,
                            "CO": row.CO})

    json_sensors = json.dumps(sensor_list, indent=4)
    return json_sensors
```

This function is called from the browser. Values passed to and from the browser must be strings. Therefore, the JSON objects are very useful to transform JSON objects into strings.