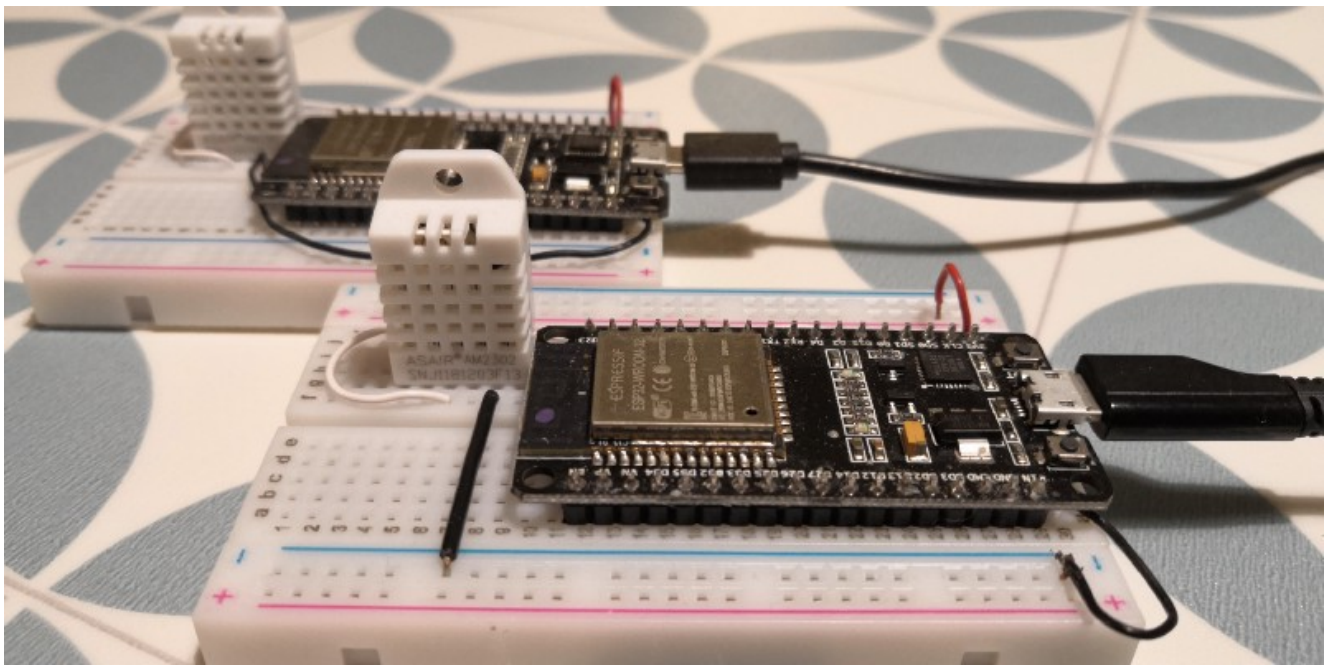
28 FEBRUARY 2019 / #IOT

How to check the weather using GCP-Cloud IoT Core with ESP32 and Mongoose OS

Olivier LOURME



This post on freecodecamp.org is not maintained. The most up to date version is on Medium:

cloudiotcore-esp32-mongooseos-1st-5c88d8134ac7

This post is a step-by-step tutorial for newbies to **Google Cloud Platform-Cloud IoT Core**. The devices are **ESP32 Wifi chips** running **Mongoose OS**. To go through his tutorial, the concepts and then the setup of a simple **IoT system measuring weather data** are described.

Live demo is here: https://hello-cloud-iot-core.firebaseapp.com/

GitHub for last section *(Logging, storing and visualizing weather data with Firebase)* ***is here:*** https://github.com/olivierlourme/iot-store-display

This post is completed by a second one: see here.

Introduction

1) History

In a previous 3-post series [link, link, link], we used an **ESP8266 Wifi chip** to regularly measure luminosity and feed a database with the obtained data. The data set was ultimately lively plotted to a web app (see live plot here: [link]). We massively used **Firebase products** (Realtime Database, Cloud Functions, SDK and Hosting) to meet our goals.

This project works fine, it draws very little power and we enjoyed developing it — but:

- **This project was okay to handle just a few connected sensors**. Setting up a set of a hundred sensors would require a lot of (rigorous) manual intervention and monitoring them would be challenging as well. Indeed, there is no central place where we can manage our system.

discovering ESP8266 but they are **quickly insufficient**: The IDE file management is really basic, there is only one program in the chip, and **there is no Operating System providing useful APIs for IoT**.

- **FirebaseArduino library**, allowing an ESP8266 to push data to a Firebase Realtime Database, **was experimental**. Some features like authentication should be improved. For now, the "secret" type authentication we used gives ESP8266 admin rights over the whole database!
- Eventually, **ESP8266 SPI flash memory was not designed to be encrypted**. In our first post [link], we showed how easy it was to recover a Wifi password when reading this memory.

> In a word, this past project couldn't be used in an industrial context. It was more a prototype for a Proof of Concept. We learnt a lot with it but today **we'd like to develop a professional and fully secured solution capable of managing in a simple way a lot of connected sensors**.

This is why we decided to:

- **investigate Google Cloud Platform-Cloud IoT Core** [link] to manage our system : devices setup, provision, authentication and monitoring;
- **move from ESP8266 to ESP32**, which offers memory encryption;
- **run Mongoose OS** [link] in our ESP32s. This OS accepts programs written in Javascript(JS) and provides a lot of APIs to deal with time, MQTT protocol, sensors, provisioning, etc. It is easy to interface with the main IoT platforms, including Google Cloud Platform-Cloud IoT Core.

ESP32 Wifi chip is a successor of the famous ESP8266 we described here: [link]. Compared to it, every feature is enhanced (speed up to 240 MHz, two cores, 520 kiB RAM, number of GPIOs, variety of peripherals, etc.) and there are some new ones (Bluetooth: legacy/BLE, **4 MiB-flash memory encryption capability**, **cryptographic hardware acceleration**: AES, SHA-2, RSA, ECC, RNG). There are a lot of resources on the web concerning ESP32. The following one deals with the **ESP32 DEVKIT V1 development board** that we will use and gives its pinout: [link].

There is also this extensive resource concerning the wide variety of ESP32 chips and development kits : http://esp32.net/ . On their home page, searching for "ESP32 DevKit" or "GeekCreit" leads to a link to the schematic of our ESP32 DEVKIT V1. This development board embeds an official Espressif ESP32-WROOM-32 chip and costs about 6€ at Banggood.

Basic IoT concepts explained through our use case

So, what will be our playground for test all these new tools?

> To illustrate IoT concepts through Cloud IoT Core, we chose to build **a weather station reporting humidity and temperature from different places**.

For simplicity we'll handle only 2 places: inside our house ("indoor") and outside our house ("outdoor"). It's up to you to deal with many more places.

1) Project hardware : ESP32 & DHT22

At each of theses places, we'll install a connected sensor (**"a device"**) constituted by a **DHT22 humidity/temperature sensor**

an **ESP32 DEVKIT V1** development board. DHT22 observes a kind of "1-Wire" protocol. Each ESP32 will house **Mongoose OS** for operating system. Its installation on an ESP32, a Hello, World! and a test with a DHT22 are given in the following section below.

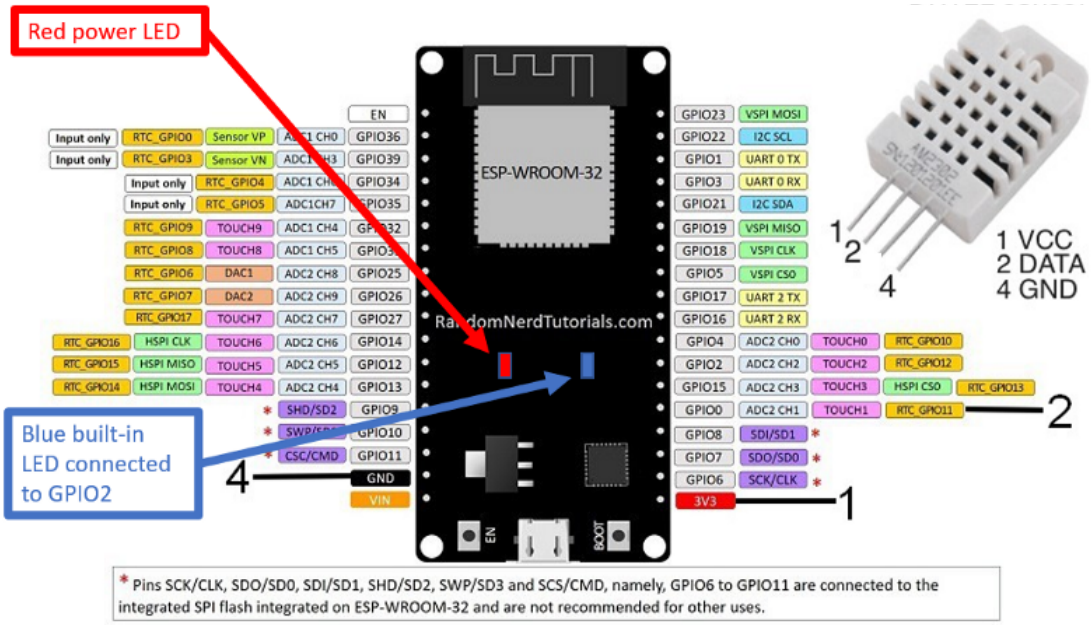
Just below are given DHT22 specifications. Afterwards, we think the accuracy figures are a bit optimistic but that's not our concern today.

DHT22

- Low cost
- 3 to 5V power and I/O
- 2.5mA max current use during conversion (while requesting data)
- Good for 0-100% humidity readings with 2-5% accuracy
- Good for -40 to 80°C temperature readings ±0.5°C accuracy
- No more than 0.5 Hz sampling rate (once every 2 seconds)
- Body size 15.1mm x 25mm x 7.7mm
- 4 pins with 0.1" spacing

DHT22 sensor characteristics ([link])

We can already build the following assembly twice (one for indoor and one for outdoor). For now, power will come from the USB connector connected to our host machine. In production, power may come from a power bank.



Assembly Diagram — ESP32 DEVKIT V1 and DHT22 sensor constitute a “device”. Pinout is here : [link]

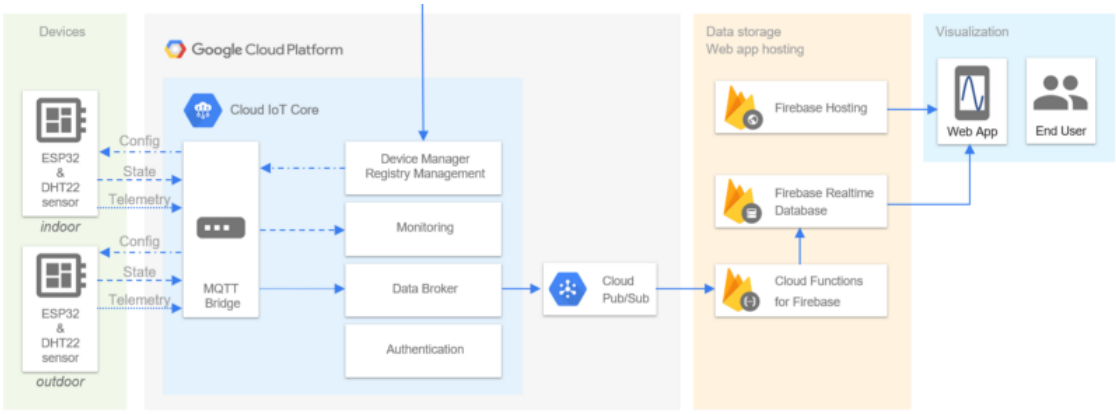
That’s all for hardware! The rest of the project uses **serverless solutions** from Google. We describe them now…

2) Project architecture : Cloud IoT Core & Firebase

All this “Project architecture” section is theoretical, there is no step to perform. Its aim is to introduce vocabulary and notions related to IoT, more specifically when this domain involves Google Cloud solutions.

Here is the general architecture of our project:

Donate



Project architecture

Note: There is no **gateway** between our devices and Cloud IoT Core because they "speak" MQTT.

Note: Devices can also communicate with Cloud IoT Core via its **HTTP bridge**. As it is less performant than the **MQTT bridge** (see a comparison : [link]), we will disallow this communication later during registry configuration. Limiting access just to what is necessary is a good practice.

Let's explain this architecture in three sections:

- "From Devices to Cloud Pub/Sub" describes the classical Google IoT architecture.
- "From Cloud Pub/Sub to data storage and visualization", describes the choices we made to exploit data.
- "Additional config and state topics" completes this architecture presentation.

From Devices to Cloud Pub/Sub

- Cloud IoT Core

Cloud IoT Core is the Google Cloud Platform service to which each of our **registered devices** will send temperature/humidity data.

telemetry event (sometimes also called a “telemetry message”). *Note:* Pricing is detailed here : [link]. For small projects with a few devices, there’s little chance you get charged.

- *MQTT*

This publication is done through a **MQTT connection**. MQTT is a publish/subscribe-based message protocol; most of the time it lies over TCP [link] (or better: over TLS, itself being over TCP). The telemetry message has to be published by the device (a MQTT client) to the Cloud Iot Core “MQTT bridge” (a MQTT server) in a **MQTT topic** whose name imperatively respects this format:

```
/devices/{device-id}/events
```

Note: Sub-folders in the topic name are possible. We won’t need this feature here but see [link], as it can sometimes be useful.

`{device-id}` is unique to each device. In our case, Mongoose OS creates it from the last 3 bytes of the MAC address of the ESP32. For example it could be `esp32_ABB3B4` .

- *Quality of Service (QoS)*

The MQTT specification describes three **Quality of Service (QoS)** levels, when publishing to a topic ([link]):

> QoS 0, the message is delivered at most once;

> QoS 1, the message is delivered at least once;

Cloud IoT Core does not support QoS 2. And QoS 1 is better than QoS 0. So **QoS 1 is the one we will adopt**. Mongoose OS can do that.

- *Security*

Concerning **security**, in our Mongoose OS/Cloud IoT Core context, MQTT communications are made over **TLS** ([link]), so (1) the device is assured to be connected to Cloud IoT Core MQTT server (CA's certificates are stored in Mongoose OS `ca.pem` file), (2) the data exchange will be private and (3) data integrity will be checked. On the other way, **device authentication** ([link]) with Cloud IoT Core is performed with a per-device public/private key authentication using **JSON Web Tokens (JWT)**. The device performs the signature part of the JWT with its private key and Cloud IoT Core validates it using the related public key. Mongoose OS tools handles this keys generation and distribution, we'll see that soon in the section called "Device registration within the Cloud IoT Core project" lying a few paragraphs below. In this section, we'll see also how to store securely the private key on the device by performing memory encryption (preventing as well reverse engineering).

Note: Beyond JWT device authentication, for additional security, it's possible to impose TLS from Cloud IoT Core to devices (so each device has also a public key certificate, etc.). It is an option we won't use but it's described here for Mongoose OS side (see "mutual TLS") and here for Cloud Iot Core side. It's good to know that AWS IoT imposes this mutual TLS, unconditionally ([link]).

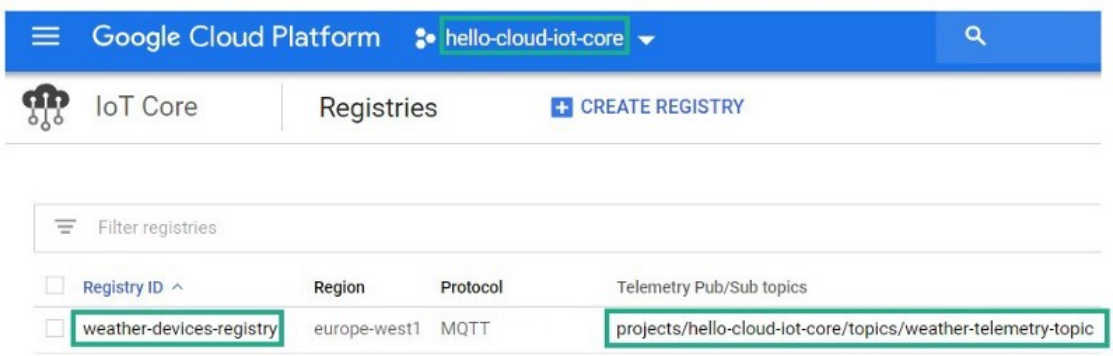
- *Registry*

Devices sharing the same purpose are regrouped within a **registry**.

Telemetry data from all devices belonging to the same registry is then *forwarded* to a Cloud Pub/Sub topic (Cloud Pub/Sub is a GCP product [link], not specifically a Cloud IoT Core one). The name of the Cloud Pub/Sub topic follows this pattern:

```
projects/id-of-google-cloud-project/topics/name-of-telemetry-topi
```

So, if we call our Google Cloud project `hello-cloud-iot-core` , if we choose `weather-telemetry-topic` for the name of our Pub/Sub telemetry topic and if finally our registry is called `weather-devices-registry` , we'll get sooner or later that kind of view in **Google Cloud Console** :



Project ID, registry ID and telemetry Pub/Sub topic name in Google Cloud Console

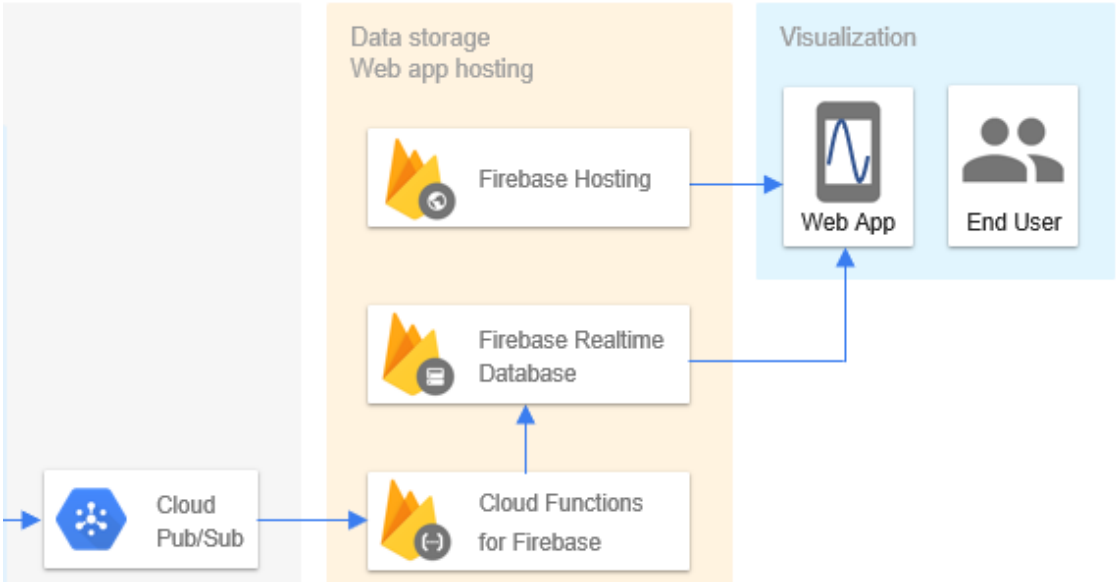
But no stress, everything will be explained step by step to reach that.

Note: As it is said here ([link]), each message in the Cloud Pub/Sub topic contains a copy of the telemetry message published by the device but also some **message attributes**, the most important being

the device that published it.
Note: We talk a lot about **Pub(lish)**, but where is the **Sub(scribe)**? In fact, we'll create quickly with Google Cloud Command Line Interface a Cloud Pub/Sub subscription (a "pull" one) in order to view the messages published to the telemetry topic. Later in this post, we'll create a Firebase Cloud Function reacting to each publication and this will automatically create another subscription (a "push" one this time).

From Cloud Pub/Sub to data storage and visualization

We're following the right part of the project architecture diagram given at the beginning of this post:



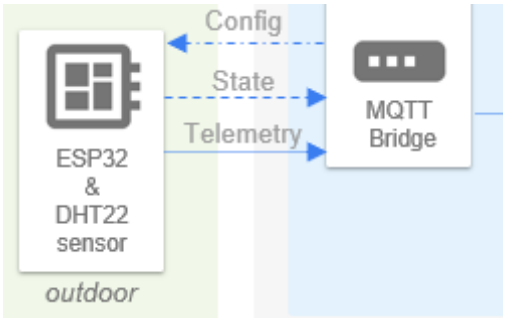
Project Architecture — Weather data storage and visualization

A publication to the Cloud Pub/Sub topic will **trigger a Firebase Cloud Function** that will itself **fulfill a Firebase Realtime Database** with the new data. A web app hosted by **Firebase Hosting** will lively plot data from the Firebase Realtime Database, in the same way as we did in a previous post: [link].

store/treat/visualize data. Alvaro Viebrantz's really good post [link] that helped us uses **Big Query** ([link]) and **Data Studio** ([link]).

Additional "config" and "state" topics

On the project architecture diagram given at the beginning of this post, we see besides telemetry two other data flows: **Config** ([link]) and **State** ([link]):



Config and State data flows

Indeed, the Cloud IoT Core service may publish **configuration update messages** to a special topic the device has subscribed to ([link]). It is useful when we need the device to go to a new state, *e.g.* by updating a parameter of its associated sensor, by changing a deep sleep period, moving a servomotor, etc.

For efficiency, there shouldn't be more than one message of that type per second per device. Such a message is an arbitrary user-defined blob (we'll use JSON), up to 64 kiB. At last, the name of this special MQTT topic is imperatively:

```
/devices/{device-id}/config
```

Cloud IoT Core has automatically subscribed to — **messages concerning its state** ([link]), *e.g.* quantity of RAM available, state of a button, etc. It is often used to see if the previous config message sent to the device had the desired effect.

For efficiency, this kind of publication shouldn't be done more than once per second per device. Such a message is an arbitrary user-defined blob (we'll use JSON), up to 64 kiB. At last, the topic to which the device publishes its state data has imperatively this name:

```
/devices/{device-id}/state
```

Note: Sending **commands** to devices is also possible from Cloud IoT Core: see [link] but we won't illustrate it.

> But for the moment, we will focus on telemetry. After this journey, in a "coming soon" post we will show how to handle `config` and `state` special topics.

UPDATE March 29, 2019: This post about `config` and `state` special topics is out: [link].

Mongoose OS installation on devices

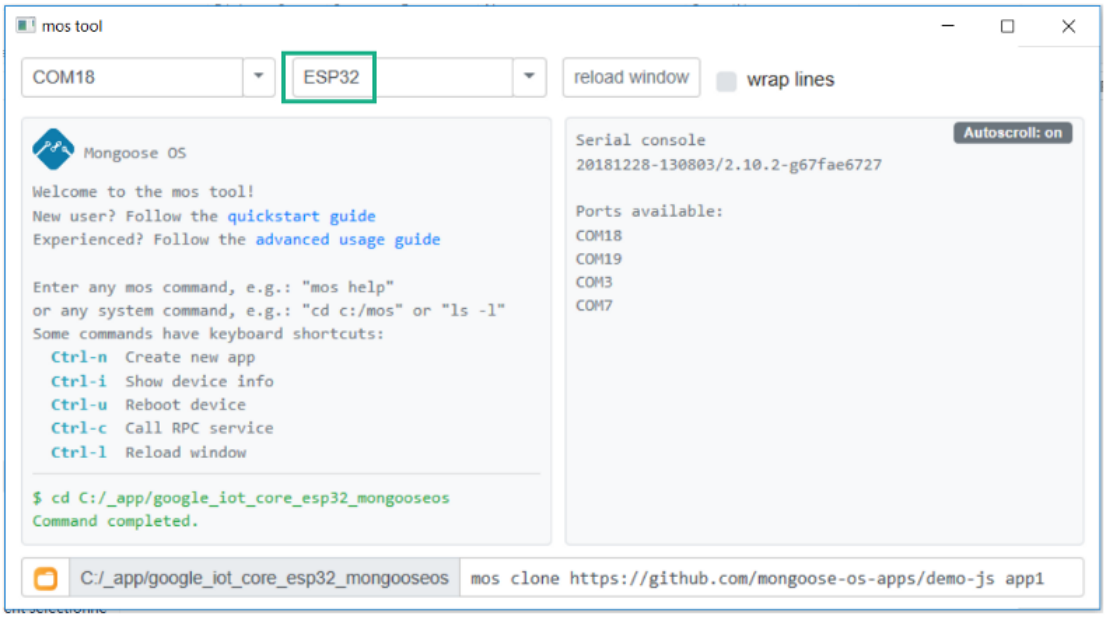
1) A short description of Mongoose OS

Mongoose OS ([link], [link]) is a smart IoT-oriented OS, runnable on several chips, including ESP8266 and ESP32. Mongoose OS is in partnership with the major actors in IoT ([link]). It comes with a development tool called **mos**, working either in a UI or with a Command Line Terminal (like `cmd.exe` in Windows). In either cases, we'll write `mos` commands. There is also a device management app called mDash but we didn't try it. **Numerous APIs dealing with most**

Step #1, **Step #2** and **Step #3** are trivial. At Step #3 don't forget to connect the device to the host machine via a USB cable.

For **Step #4** "Create new app", we choose to call the app `app1`. When `mos clone https://github.com/mongoose-os-apps/demo-js app1` indicated on the web site is completed, mos tool automatically goes to the just created `app1/` folder.

In `app1/fs/` folder, there is a source file called `init.js`. It is a demo file capable to communicate with different IoT platforms (if they are configured of course). We will basically test it and soon simplify it for our purposes.



mos tool launched in a UI ; ESP32 selected; Serial console (on the right) is by default at 115200 bds, ESP32's default speed.

Step #5 "Build app firmware" is launched with `mos build` command (add `--arch esp32` to this command if you launch it from a Command Line Terminal, not from mos tool). It may take a while but normally we have to perform this build only once. After it, we have

binaries of the OS and `init.js`. It will be flashed to ESP32 in the next step.

Step #6 "Flash firmware" is launched with `mos flash` command. It has normally to be done just once. Even if later we change some files (like `init.js` for instance), we will use a `mos put` command to upload a file from the host machine to the **local device's file system**. Of course this command is only available after the flash process.

Note: Firmware flash step can be tricky with a brand new ESP32. With our ESP32 DEVKIT V1, we had messages in console (that's a first good point!) reporting issues about failing to connect to ESP32 ROM. Retrying to flash by pressing the BOOT button (closed to USB connector) finally turned out to a successful flash. Though, be ready to wait for one minute or two.

Then, the device automatically reboots and executes `init.js`. We obtained every second the following information in mos console (or in any serial terminal @115200 bds) :

```
[Feb 10 22:22:30.950] online: false {"ram_free":105736,"uptime":91.200081,"btnCount":0,"on":false}
```

Console after initial ESP32 flash firmware with Mongoose OS

In **Step #7** we connect ESP32 to our wifi network (we use mos tool):

```
mos wifi WIFI_NETWORK_NAME WIFI_PASSWORD
```

The device will reboot by itself after getting an IP address and synchronizing time by contacting a SNTP server. We then ping our device to check its internet connexion.

CTRL+i in mos tool, or by typing `mos call Sys.GetInfo`.

Note: We reset the device by hitting **CTRL+u** in mos tool, or by typing `mos call Sys.Reboot`.

Note: Steps #5, #6 and #7 could be the beginning of a "provisioning script", useful if we have many devices to setup. It is optional to rerun Step #5 if all devices are the same, *e.g.* only ESP32.

3) ESP32 "Hello, World!" program with Mongoose OS

To get used to Mongoose OS JS programming style and mos tool, let's write a small program whose aim is to make the blue built-in LED blink and print messages on console. **This led is connected to GPIO2 pin** of ESP32 DEVKIT V1 (see Assembly Diagram at the beginning of this post). On our host machine, let's replace the content of `app1/fs/init.js` by this one:

```
/*
 ESP32 DEVKIT V1 - Mongoose OS
 Built-in LED blink and console log
 This blue LED is connected to GPIO2.
 See:
 - https://mongoose-os.com/docs/mos/api/core/mgos_timers.h.md
*/

load('api_config.js');
load('api_gpio.js');
load('api_timer.js');

let pin = 2;

GPIO.set_mode(pin, GPIO.MODE_OUTPUT);

// Call every 2 seconds
Timer.set(2000, Timer.REPEAT, function() {
  let value = GPIO.toggle(pin);
```

From mos tool or from a Command Line Terminal, we upload this file to Mongoose OS file system and finally we reboot the device:

```
mos put fs/init.js
mos call Sys.Reboot
```

The blue led should blink and we should see alternatively `Tick` and `Tock` printed on console.

4) DHT22 test with Mongoose OS

At the beginning of this post, there is an Assembly Diagram showing how to connect DHT22 sensor with ESP32 DEVKIT V1. We chose to connect **DHT22 data pin to GPIO0 of ESP32 DEVKIT V1**.

So, here is another short `init.js` program. This one prints periodically to serial console DHT22 measures (temperature and humidity - as an object in JSON, no MQTT publication yet):

```
/*
 ESP32 DEVKIT V1 - Mongoose OS
 DHT22 sensor measures are sent to console.
 DHT22 data pin is connnected to GPIO0.
 See:
 - https://mongoose-os.com/docs/quickstart/develop-in-js.md
 - https://mongoose-os.com/docs/mos/api/drivers/dht.md
*/

load('api_config.js');
load('api_dht.js');
load('api_timer.js');

let pin = 0;
```

```
Timer.set(5000, true, function() { // timer period is in ms
  let msg = JSON.stringify({temperature: dht.getTemp(), humidity: dht.
  print(msg);
}, null);
```

Then:

```
mos put fs/init.js
mos call Sys.Reboot
```

And this is the related console we get after uploading the `init.js` program and rebooting the device. Humidity is in % and temperature is in Celsius degrees:

```
[Feb 22 08:16:57.096] {"humidity":45.299999,"temperature":24.100000}
[Feb 22 08:17:02.095] {"humidity":45.299999,"temperature":24.100000}
[Feb 22 08:17:07.096] {"humidity":45.299999,"temperature":24.100000}
```

DHT22 measures as printed to console. A bit too accurate, isn't it ?

Numbers seem to have a long decimal part but this will be fixed later within a Cloud Function.

Note: For pedagogical purposes, we chose all over this post explicit long key names like `temperature` or `humidity` . This will have consequences on the volume of data stored later in a NoSQL database (Firebase Realtime Database) as those keys will be repeated for each measure. Shorter key names could be a good idea.

5) Let's publish data to the MQTT

This is our last program, the one ready to work with Cloud IoT Core! On the previous program, we just add a publication to the telemetry topic we already talked about: `/devices/{device-id}/events`.

Note that messages are published in JSON as it will facilitate later their content retrieval with the Firebase Cloud Function reacting to messages publication.

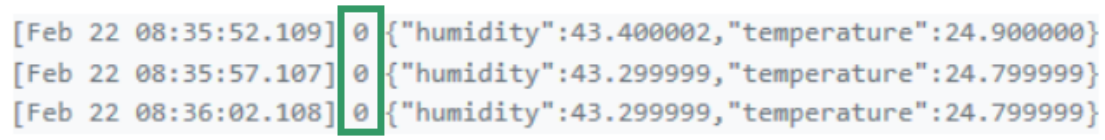
```
/*
 ESP32 DEVKIT V1 - Mongoose OS
 DHT22 sensor measures are sent to console.
 DHT22 data pin is connnected to GPIO0.
 Publishes weather data to the appropriate topic.

 See:
 - https://mongoose-os.com/docs/quickstart/develop-in-js.md
 - https://mongoose-os.com/docs/mos/api/drivers/dht.md
 - https://mongoose-os.com/docs/mos/api/net/mqtt.md
*/

load('api_config.js');
load('api_dht.js');
load('api_timer.js');
load('api_mqtt.js');

// Telemetry topic must have this name:
let topic = '/devices/' + Cfg.get('device.id') + '/events';

let pin = 0;
let dht = DHT.create(pin, DHT.DHT22);

Timer.set(5000, true, function() { // timer period is in ms
  let msg = JSON.stringify({temperature: dht.getTemp(), humidity: dht.
  // Publish message with a QoS 1
  // MQTT.pub() returns 1 in case of success, 0 otherwise.
  let ok = MQTT.pub(topic, msg, 1);
  print(ok, msg);
}, null);
```

provoke a reset:

```
mos put fs/init.js
mos call Sys.Reboot
```

Note: These commands could be appended to the "provisioning script" we mentioned earlier.

When running, this last program prints data to console but it fails to publish data to the MQTT bridge of Cloud IoT Core (`MQTT.pub()` returns 0):



Telemetry data can still not be published as we haven't set up a Google Cloud project yet.

Indeed, we haven't set up any Google Cloud project yet, neither *a fortiori* registered a single device to it. Let's do it now!

Cloud IoT Core project setup

1) Google Cloud SDK installation

Firstly we need to **install Google Cloud SDK** because we will have to type some `gcloud` **commands** in a Command Line Terminal. At the time of writing, it requires Python 2.7. It won't work with Python 3.5. The Google Cloud SDK download page ([link]) offers versions of the SDK with Python bundled inside (if you're sure you don't have

point).
Then, Cloud IoT Core requires some **Beta versions of** `gcloud` **commands**. So in a Command Line Terminal, from any folder, we type:

```
gcloud components install beta
```

These two previous steps have to be done just one time!

Note: Most of the following actions on Google IoT Core can be performed in three ways:

- with **Google Cloud Console** (on the web)
- with **some APIs** in different languages, and
- with **Command Line Interface** in a terminal, typing `gcloud` commands.

We will use the latter to configure things and we'll check facts with Google Cloud Console (on the web).

2) Google Cloud project setup

We follow now this guide from Mongoose OS web site : [link].

```
# Commands indicated in this grey frame have to be done just onc

# Get authenticated with Google Cloud
gcloud auth login

# Create cloud project. We chose hello-cloud-iot-core as PROJECT
gcloud projects create hello-cloud-iot-core
```

```
# Set default project for gcloud
gcloud config set project hello-cloud-iot-core

# Create Pub/Sub topic for device telemetry
gcloud beta pubsub topics create weather-telemetry-topic

# Create a Pub/Sub subscription to the just created topic
gcloud beta pubsub subscriptions create --topic weather-telemetr

# Create devices registry (we call it weather-devices-registry)

# Precise Pub/Sub topic name for event notifications

# Disallow device connections to the HTTP bridge
gcloud beta iot registries create weather-devices-registry --reg

# Say 'yes' to enable API (if prompted).

# But the last command may not work all the same

# if you don't enable billing.

# So, follow the link to enable billing and retry last command.

# It should end up to "Created registry [weather-devices-registr
```

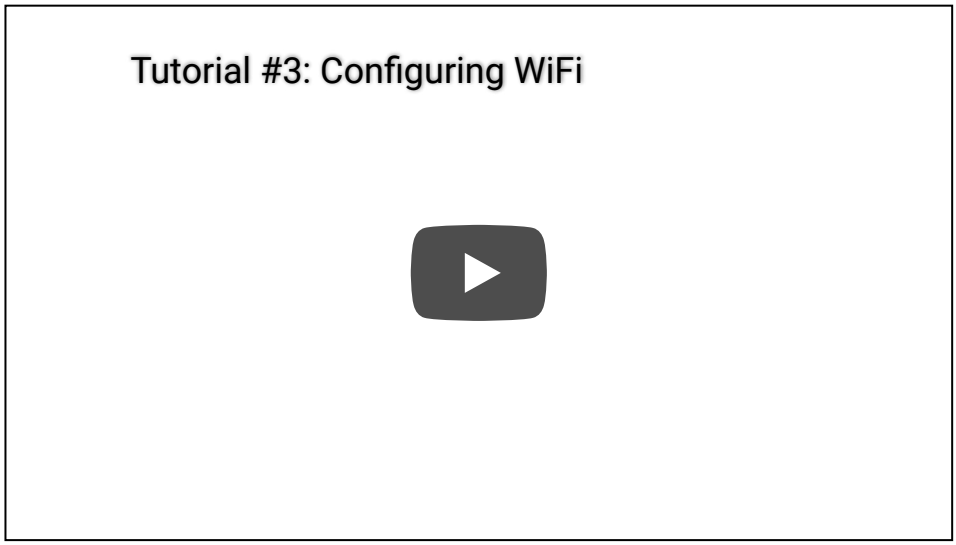
3) Device registration within the Cloud IoT Core project

Let's now register the devices to the project! One at a time of course. **mos tool is really helpful for this task.** From mos tool launched in its UI or from Command Line Terminal, placed in our `app1` folder, we type the following command (project id and registry name are involved, as you see):

```
# Register device with Cloud IoT Core (do it for each device!)
mos gcp-iot-setup --gcp-project hello-cloud-iot-core --gcp-regio
```

be written in both C/C++ and JS.

At last, there is a 12-tutorial series on YouTube, really useful:

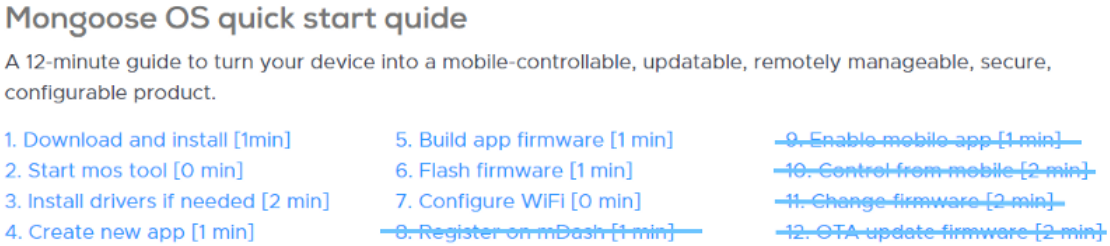


Note: We used Mongoose OS Community Edition, which is free, licensed under Apache 2.0.

2) Mongoose OS installation on ESP32

This installation has to be performed on each device.

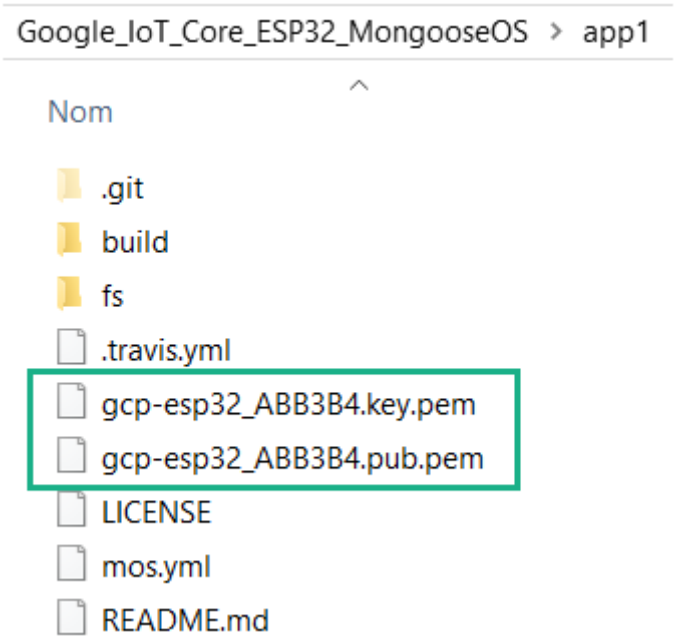
We head to the **developers section** of Mongoose OS web site ([link]) in order to perform the **first seven steps** of the list given in this resource:



Note: This command could be the last one of the "provisioning script" we mentioned already twice.

This command is a `mos` command that will itself use `gcloud` commands. The device about to be registered must be connected via the serial port to our host computer because some information will be uploaded to it just like keys, MQTT bridge address, etc.

Indeed, we see on mos console that **two keys (one private, one public) are generated**. We can inspect them in `app1` project folder. The private one is for ESP32 and the public one is for Google IoT Core. They are used during the authentication process involving the JSON Web Token we mentioned earlier.



Pair of keys just generated (private and public)

Note concerning security: **The private key shouldn't be stored in plain text in ESP32 flash memory**. This is why we describe in the post following this one ([link]) how to encrypt this memory. Also, **the private key file shouldn't be stored in plain text on the host development computer**. At least, protect access to its content with a password.

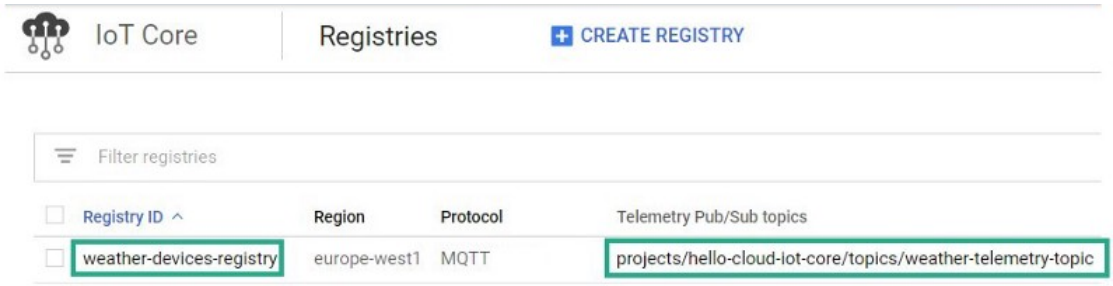
When the device reboots, we see in the console that it successfully connects to the Google MQTT bridge and publishes telemetry messages (`MQTT.pub()` returns 1):

```
[Feb 22 08:16:57.096] 1 {"humidity":45.299999,"temperature":24.100000}
[Feb 22 08:17:02.095] 1 {"humidity":45.299999,"temperature":24.100000}
[Feb 22 08:17:07.096] 1 {"humidity":45.299999,"temperature":24.100000}
```

Publications to MQTT bridge are successful. Nice job!

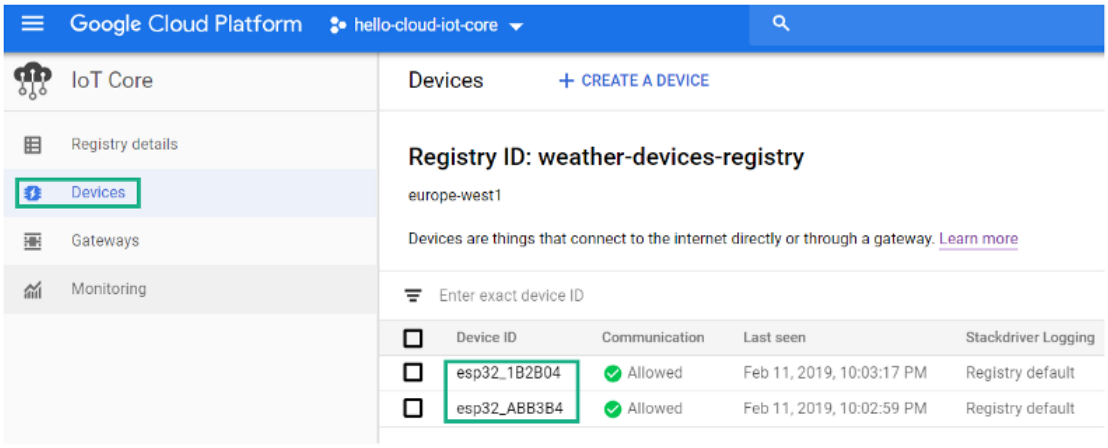
4) Checking the project setup in Google Cloud Console

We head to https://console.cloud.google.com/iot/ to check that everything was well configured:



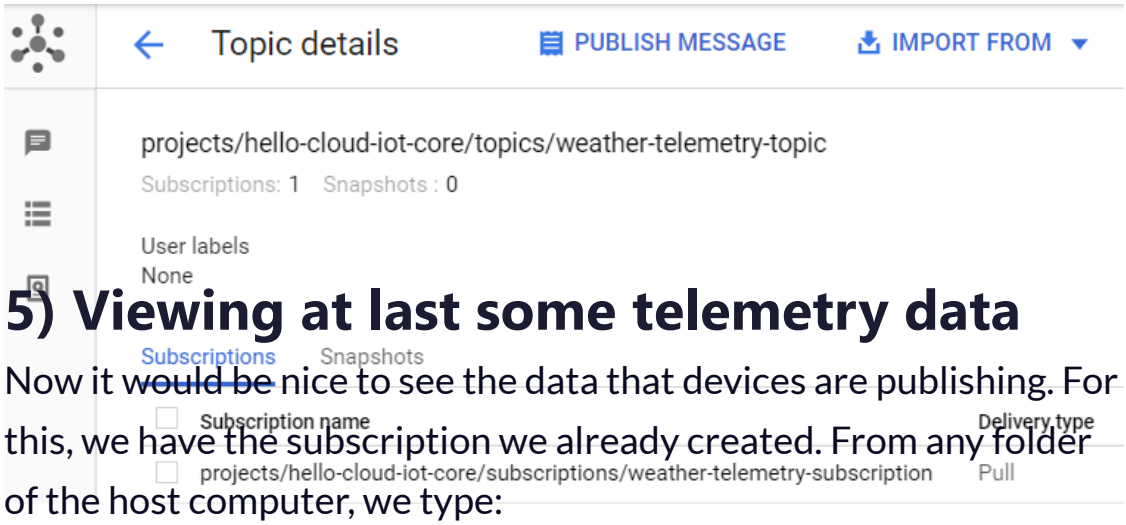
Project ID, registry ID and telemetry Pub/Sub topic name in Google Cloud Console

Clicking on the **Registry ID** `weather-devices-registry` reaches another screen. Clicking on "Devices" on this new screen lists provisioned devices and gives details like the last time they were seen (but this is not a live update, we have to refresh the page):



Devices View in Google Cloud Console

Clicking on the **Telemetry Pub/Sub topic** name goes to **Pub/Sub console** to show the subscription we created before, *i.e.* the one related to the telemetry topic:



Google Cloud Console (Pub/Sub) — names of topic and related subscription

5) Viewing at last some telemetry data

Now it would be nice to see the data that devices are publishing. For this, we have the subscription we already created. From any folder of the host computer, we type:

```
gcloud beta pubsub subscriptions pull --auto-ack weather-telemet
```

This command ([link]) pulls until *2* Pub/Sub messages from our `weather-telemetry-subscription` subscription. We can see data in JSON, messages ids and a list of attributes for each message. Among them the `deviceId` attribute is present. Unfortunately there are no timestamps, we'll see how to get them later.

```
+----------------------------------------+-----------------+----------------------------------------+
|                  DATA                  |    MESSAGE_ID   |               ATTRIBUTES               |
+----------------------------------------+-----------------+----------------------------------------+
| {"humidity":46.599998,"temp":22.600000} | 386738013568252 | deviceId=esp32_ABB3B4                  |
|                                        |                 | deviceNumId=2967231588148233           |
|                                        |                 | deviceRegistryId=weather-devices-registry |
|                                        |                 | deviceRegistryLocation=europe-west1    |
|                                        |                 | projectId=hello-cloud-iot-core         |
|                                        |                 | subFolder=                             |
| {"humidity":46.299999,"temp":20.500000} | 388929138921924 | deviceId=esp32_1B2B04                  |
|                                        |                 | deviceNumId=2832514993433952           |
|                                        |                 | deviceRegistryId=weather-devices-registry |
|                                        |                 | deviceRegistryLocation=europe-west1    |
|                                        |                 | projectId=hello-cloud-iot-core         |
|                                        |                 | subFolder=                             |
+----------------------------------------+-----------------+----------------------------------------+
```

Result of pulling telemetry messages from a subscription

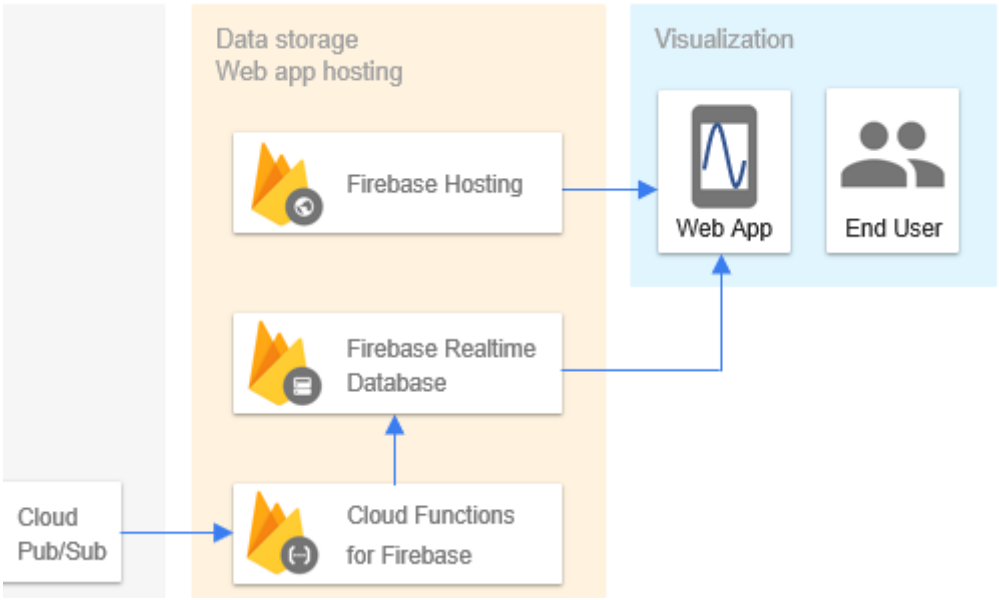
> If you have reached this milestone, congrats! We're now ready to write a Firebase Cloud Function reacting to each

Donate

Logging, storing and visualizing weather data with Firebase

1) Introduction

We're now tackling this part of the project:



The part we study now

On that diagram, we see that our project needs 3 Firebase products :

A **Firebase Cloud Function** (more exactly "a Cloud Function for Firebase") must react to any publication to the telemetry topic in order to store the weather data of this publication to a **Firebase Realtime Database**. This storage allows weather data persistence and is used to feed a web app hosted by **Firebase Hosting**. This web app draws live plots of this weather data across time.

The good new is that it's possible to configure all these products with one command.

2) Firebase configuration, GitHub repository

We are still working on the same Google Cloud project called `hello-cloud-iot-core`. Firebase will just "enhance" this project with its products.

We made a GitHub repository for the Firebase aspects of our project:

olivierlourme/iot-store-display
*Contribute to olivierlourme/iot-store-display development by creating an account on GitHub.*github.com

Clone this repository in your favorite development folder and head to the newly created directory:

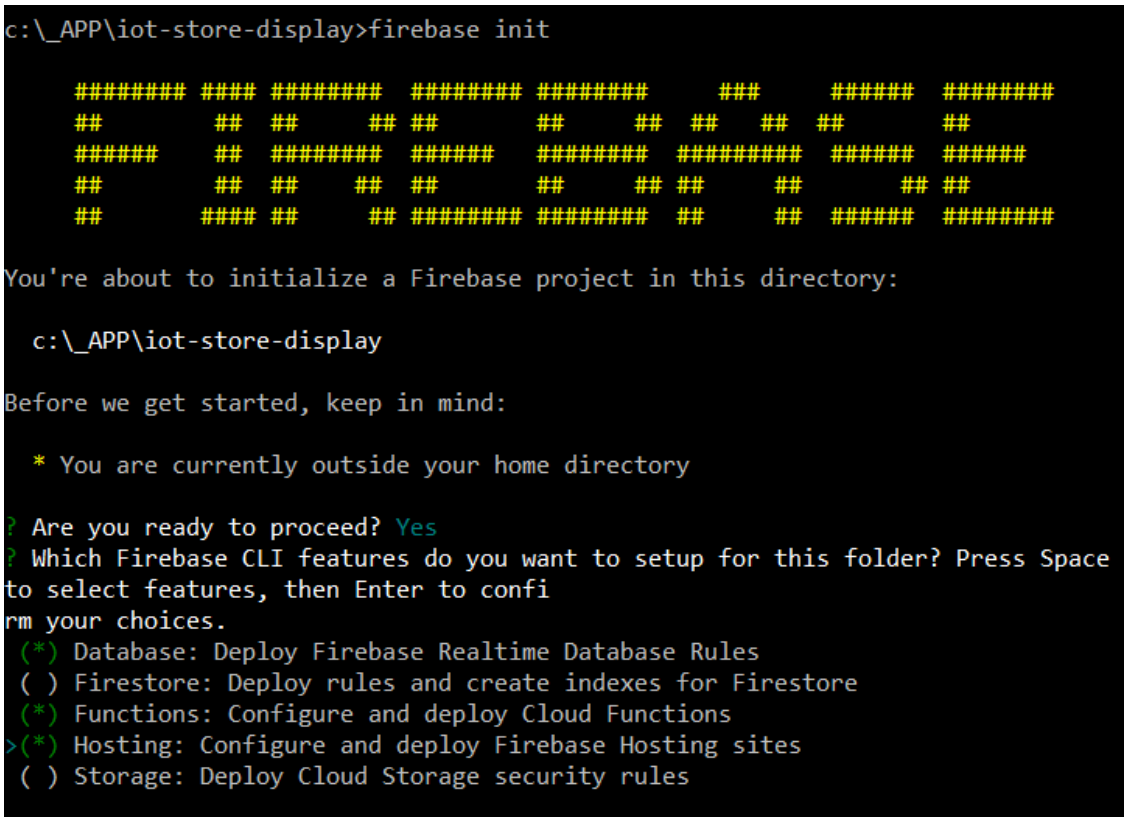
```
c:\_app>git clone https://github.com/olivierlourme/iot-store-dis
c:\_app>cd iot-store-display
```

Global Firebase configuration

Note: We suppose you have **Firebase tools** installed (*i.e.* Node.js installed and `npm install -g firebase-tools` was run, see [link] for details).

Let's perform the Firebase initializations:

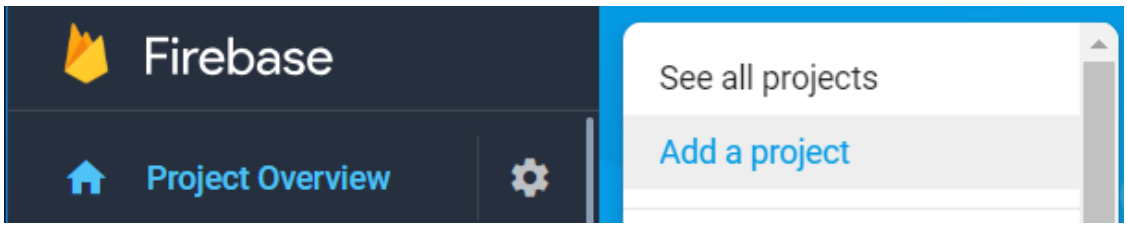
```
c:\_app\iot-store-display>firebase init
```



Choosing Firebase products

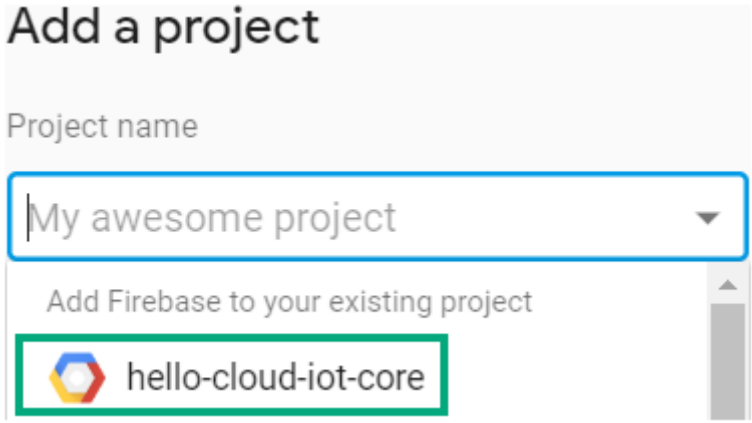
We are then prompted to associate the current directory (`iot-store-display`) with one of the listed Firebase projects. The problem is that our project `hello-cloud-iot-core` doesn't appear in the list because before being a Firebase project it's also a Google Cloud

Firebase and Google Cloud: [link] and [link].
To overcome this, first we hit CTRL+C to stop this initialization process and then we go to **Firebase Console** at https://console.firebase.google.com. We choose “Add a project”:



Firebse Console — Add a project

And we can see our project (with Google Cloud logo) and choose it:



Fierbase Console — Even Google Cloud projects are listed.

Note: You might then be asked to confirm Firebase billing plan if the Google Cloud project itself has a billing plan.

Great! We restart the Firebase initialization with `firebase init` command and this time our Google Cloud project `hello-cloud-iot-core` is listed. We choose it:

```
  blagues-animaux (blagues-animaux)
  fir-codelab-757eb (firebase-codelab)
> hello-cloud-iot-core (hello-cloud-iot-core)
```

Our `iot-store-display` directory is associated with **hello-cloud-iot-core** project.

Note: If you still don't see your project you might be logged to Firebase without the correct Google account. In this case, type `firebase logout` followed by `firebase login`.

Realtime Database configuration

Then the wizard asks a single question about the Realtime Database and its rules: the name of the file where they will be saved. We maintain the default name. It's more practical to have these rules in a file lying in the project directory than to go to Firebase Console as we did in past posts. We will detail these rules later.

```
=== Database Setup

Firebase Realtime Database Rules allow you to define how your data should be
structured and when your data can be read from and written to.

? What file should be used for Database Rules? (database.rules.json)
```

Name of file storing Realtime Database rules

Cloud Functions configuration

Here are the answers we made to the wizard concerning Functions Setup:

```
A functions directory will be created in your project with a Node.js
package pre-configured. Functions can be deployed with firebase deploy.

? What language would you like to use to write Cloud Functions? JavaScript
? Do you want to use ESLint to catch probable bugs and enforce style? No
+  Wrote functions/package.json
? File functions/index.js already exists. Overwrite? No
i  Skipping write of functions/index.js
? Do you want to install dependencies with npm now? Yes
```

Cloud Functions setup

Of course, we choose not to overwrite the `functions/index.js` file obtained from GitHub.

Firebase Hosting configuration

And here are the answers we made to the wizard concerning Hosting Setup:

```
=== Hosting Setup

Your public directory is the folder (relative to your project directory) that
will contain Hosting assets to be uploaded with firebase deploy. If you
have a build process for your assets, use your build's output directory.

? What do you want to use as your public directory? public
? Configure as a single-page app (rewrite all urls to /index.html)? No
+  Wrote public/404.html
? File public/index.html already exists. Overwrite? No
i  Skipping write of public/index.html

i  Writing configuration info to firebase.json...
i  Writing project information to .firebaserc...
i  Writing gitignore file to .gitignore...

+  Firebase initialization complete!
```

Hosting setup

Of course we choose not to overwrite the `public/index.html` file obtained from GitHub.

Deploying (not now!)

Later, if we want to deploy some updates we made to our 3 products, we can type globally:

```
c:\_app\iot-store-display>firebase deploy
```

But if we want to deploy only, respectively:

- updated database rules,
- updated cloud functions,
- updated web app,

we type, respectively:

```
firebase deploy --only database
firebase deploy --only functions
firebase serve --only hosting (local deployment) OR firebase dep
```

3) Pub/Sub trigger Cloud Function

Introduction

In a past post, we explained that we could write **Firebase Cloud Functions** triggered on some events happening to some of the Google products.

Post 2 of 3. Our IoT journey through ESP8266, Firebase and Plotly.js
*A Firebase Cloud Function appends a timestamp to each value pushed to a Firebase Realtime Database.*medium.com

Cloud Pub/Sub is one of these products and so it is possible to trigger a function each time a message is published to a Pub/Sub

So if the Firebase Cloud Function is triggered on each publication to the `weather-telemetry-topic` topic, watching its log will allow us to watch the telemetry topic's activity.

The code of the Cloud Function has to store each new published data to the Firebase Realtime Database associated with our project.

Cloud Function source code

The beginning of the source code looks like this:

```
exports.detectTelemetryEvents = functions.pubsub.topic('weather-
    (message, context) => {...
```

The full Cloud Function source code lies in the file named `index.js`. This file is in the `functions` folder of our `iot-store-display` directory on GitHub. It is fully commented, so run and study it, it's short and not complicated.

Cloud Function deployment

It's time to deploy the Cloud Function:

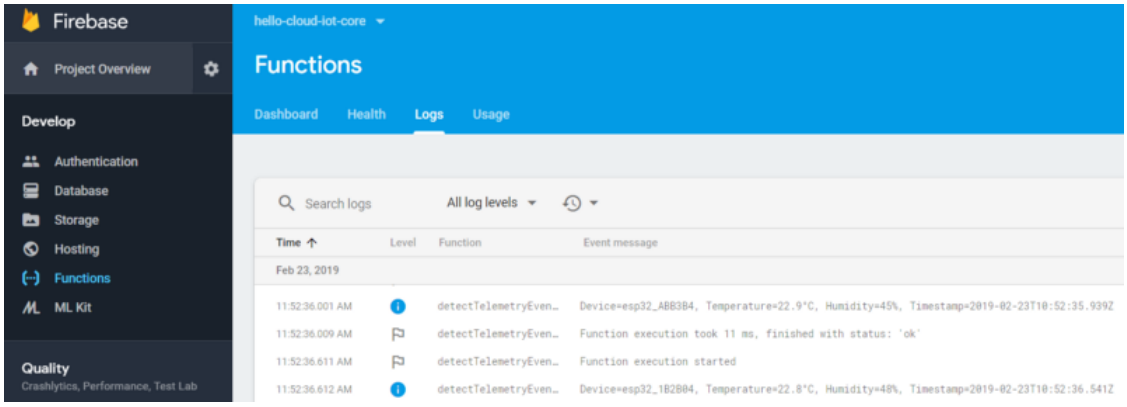
```
c:\_app\iot-store-display>firebase deploy --only functions
```

Cloud Function validation

Once Cloud Function is deployed, we can watch the Cloud Function logs and among other things, we'll see the results of the `console.lo`

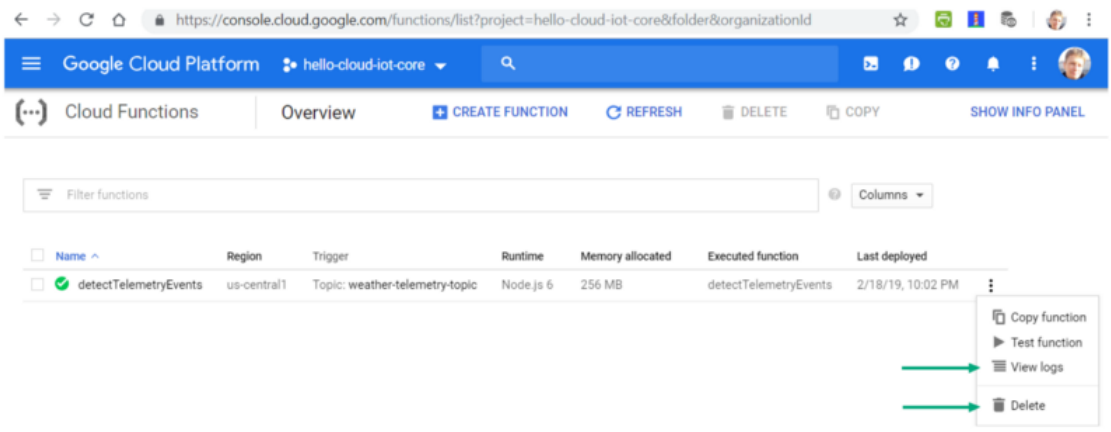
Where to see those logs? We have two opportunities:

- in Firebase Console (https://console.firebase.google.com):



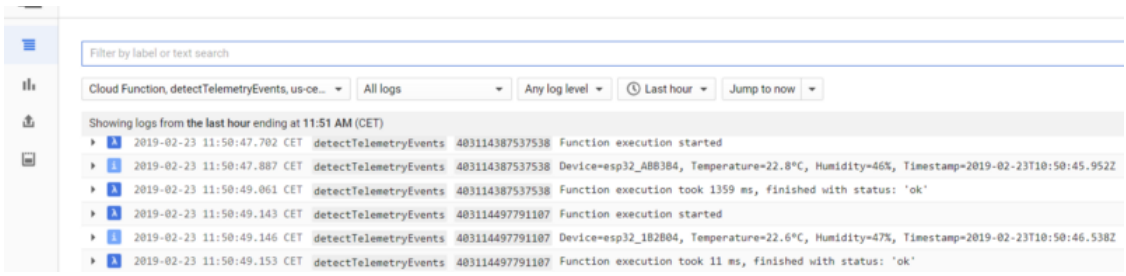
Firebase Console — Firebase Cloud Functions logs

- in Google Cloud Console (https://console.cloud.google.com/functions/):



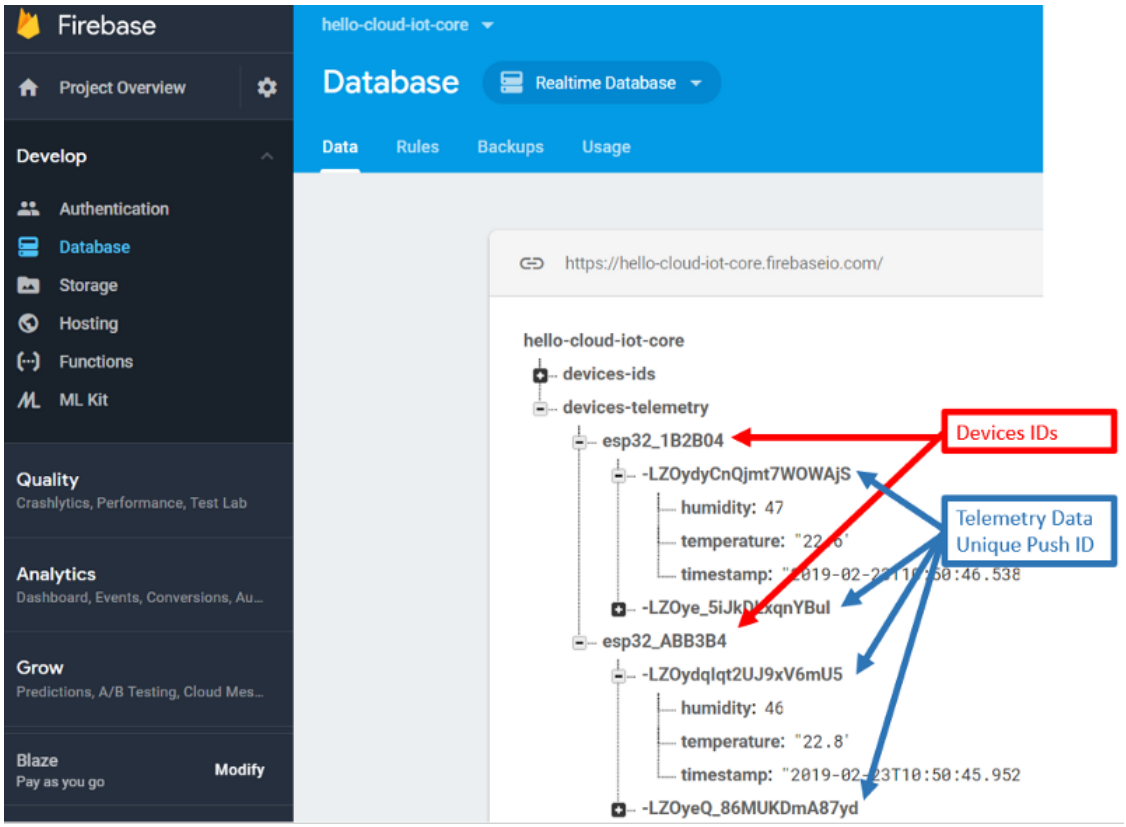
Google Cloud Console — Access to **Cloud Function logs** and to **Cloud Function deletion**

We prefer this latter solution, as logs are clearer:



Google Cloud Console- Cloud Functions logs

Concerning storage, here is what lies in Firebase Realtime Database after each device has made 2 telemetry data publication. Data is of course sorted by device as we specified it in `index.js` :



Firebase Console — Realtime Database after 2 telemetry data publications per device

Note: Don't forget to **delete your Cloud Function** on Google servers if you don't use it, otherwise you might either reach the invocation quota or pay for service you don't use, as indicated here: [link]. Function deletion is to be performed on Google Cloud Console (see "Delete", 3 screenshots above).

Note: The Cloud Function has admin rights over the database, whatever is the content of the `database.rules.json` file. At this step, the `database.rules.json` file can still be very restrictive. Don't forget to deploy them, once edited.

```
{
  "rules": {
    ".read": false,
    ".write": false
  }
}
```

4) A web app using Firebase and plotlys.js to visualize weather data

Introduction

Note: We're now building a "homemade" (and satisfying) data visualization solution. For enhanced UI (dashboard, etc.), maybe you should investigate Data Studio we already mentioned elsewhere in this post.

This web app **lively plots** the data stored in the Firebase Realtime Database. We used plotly (https://plot.ly/javascript/) for the plotting library. We are familiar with that work as we already undertook a similar one in a previous post:

Post 3 of 3. Our IoT journey through ESP8266, Firebase and Plotly.js
*A web app hosted by Firebase Hosting susbscribes to the data stream coming from a Firebase Realtime Database and plot…*medium.com

What's different today is that we have to:

- draw several charts: temperature *vs* time and humidity *vs* time,
- inside each chart, we have one plot per device.

Database rules & devices-ids node

Concerning the database rules, what should be now the `database.rules.json` file? The `device-telemetry` node needs to be **read** by the web app. And if you looked attentively at the Realtime Database screenshot given a few screenshots before, there is another node called `devices-ids`.

Firebase Console — Detail of devices-ids node in Realtime Database

You need to create **manually in the Firebase Console** this `devices-ids` node and fill it appropriately for the web app to work properly. It is a simple mean to declare to the web app the devices we want plots for and also to give aliases to the devices. Its role and necessity are fully explained in comments of the `public/script.js` file given in GitHub.

Note: An improvement could be a form (accessed through authentication) that, once fulfilled, calls a script to generate this `devices-ids` node.

This `devices-ids` node also needs to be read by the web app. So the `database.rules.json` file should eventually become:

```
{
  "rules": {
    "devices-ids": {
      ".read": true,
```

```
    "devices-telemetry": {
      ".read": true,
      ".write": false
    }
  }
}
```

These new rules, once edited and saved, must be deployed with:

```
c:\_app\iot-store-display>firebase deploy --only database
```

Web app source code

The web app source code lies in the `public` directory of our `hello-cloud-iot-core` folder or in GitHub. The content of the folder, especially `script.js` , is fully commented so you know where to study (and improve!) it.

Note: we have only two devices for this demo but the source code is okay for *x* devices as long as you declare them in the `devices-ids` node.

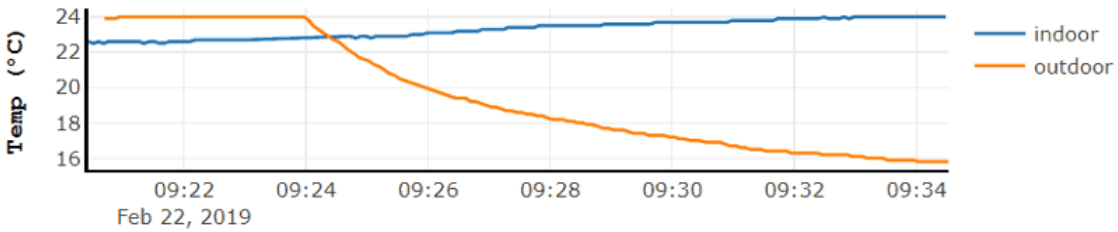
Web app local deployment and validation

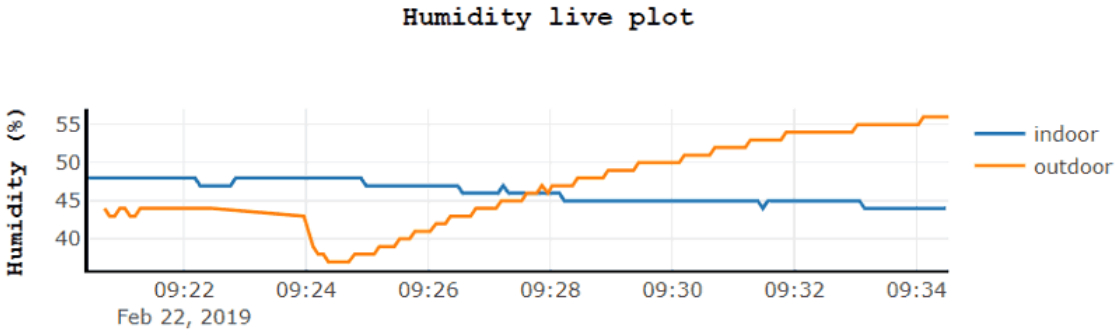
For testing purposes, Firebase Hosting can launch a local live server:

```
c:\_app\iot-store-display>firebase serve --only hosting
```

We head to `http://localhost:5000` and we're happy to get this:

Donate



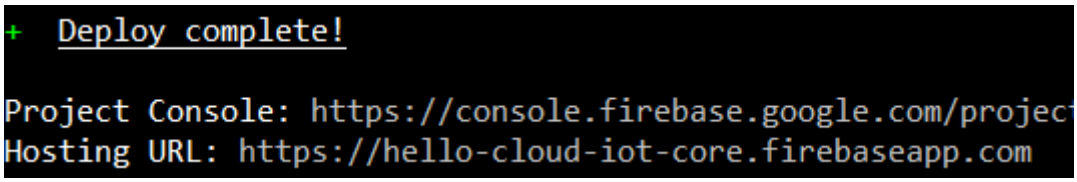


At last the charts we wanted!

Web app remote deployment

At last, Firebase offers us the hosting of our web app and an access to it via https:

We quickly get the public URL of our web app : https://hello-cloud-iot-core.firebaseapp.com



Web app deployment is complete!

Note: If you have your own domain, you can connect your Firebase web app to it. See [link].

Conclusion

In this post we discovered how to combine **ESP32**, **Mongoose OS** and **Cloud IoT Core**, obtaining a serious, secure and **professional IoT project**. Now that we know, it can go really fast to provision 10, 100… 1000 devices acquiring weather data all over an area, as long as they can get a Wifi connection. Now, devices are centrally managed, it is easy to provision and monitor them. But we can go further!

Indeed, in addition to this post, there is a second one ([link]). Inside it:

- We'll focus on ESP32 flash **memory encryption**, to achieve a fully secured system.
- We'll see how to use the `config` special topic, allowing us to **trig an action on the device from the Google Cloud Console**.
- We'll see how to use the `state` special topic, allowing **the device to communicate to Google Cloud Console**

Donate

We hope you enjoyed this really long post and that you learnt something! Don't hesitate to ping me if you have any questions or improvement suggestions...

Olivier LOURME

Read more posts by this author.

If this article was helpful, tweet it.

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started