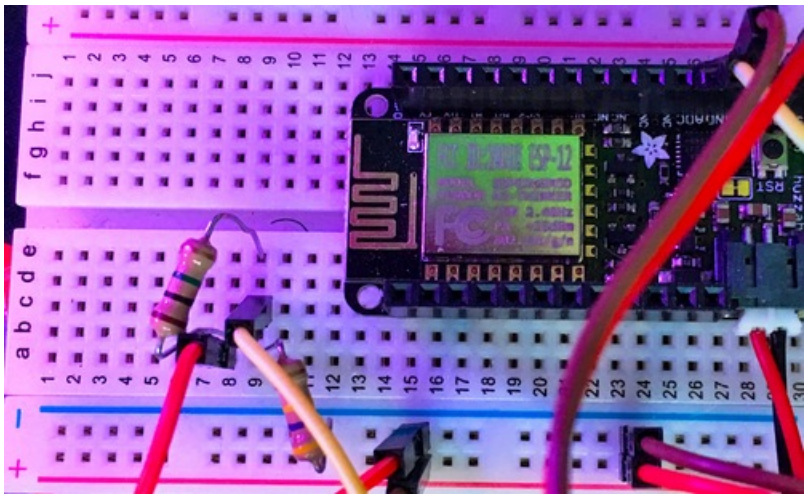
adafruit learning system

Manually bridging MQTT to Adafruit.IO

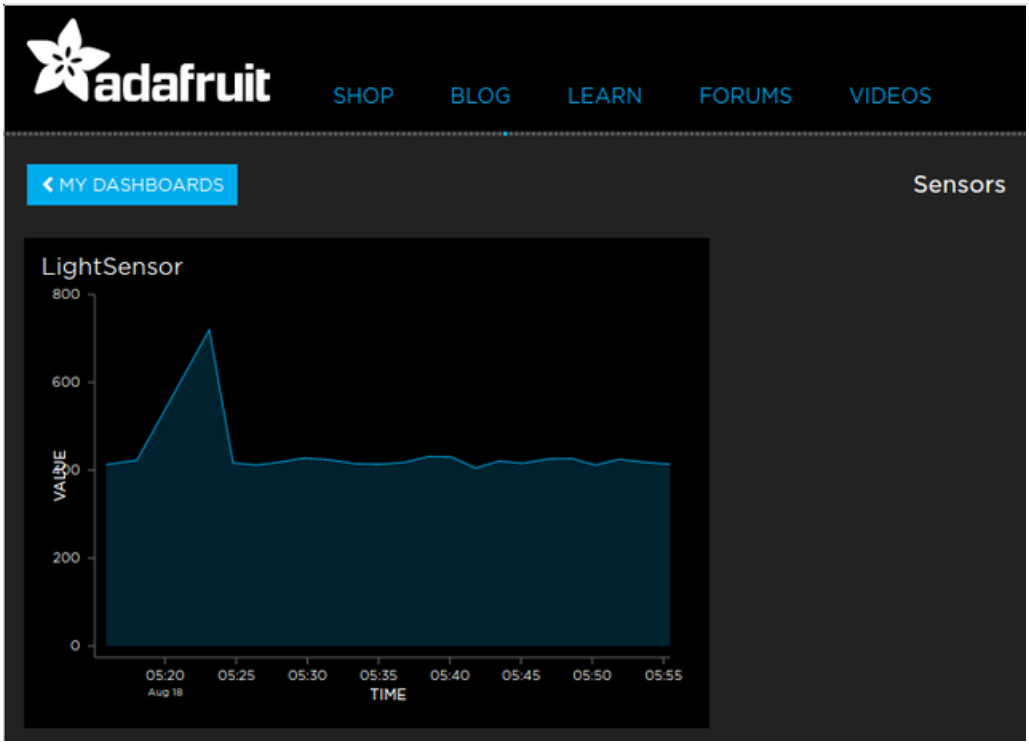
Created by Philip Moyer



Last updated on 2018-08-22 03:55:34 PM UTC

Guide Contents

Overview



I'm going to start this off by admitting something up front. I didn't build this the right way. Well, let's mellow that a little and say I didn't build this in the most optimal way. Sometimes you already have some infrastructure in place beyond simple networking and when you tinker with new sensors you just use the infrastructure you already have. Then later, when you want to extend your project from tinkering to the cloud, you have to jump through some hoops to make it work. If you want to skip what I actually did and just read about how I *should* have done it, just click on "The Right Way to Have Built This!" on the left side navigation bar.

So, on with the project! I bought one of the then-new Adafruit GA1A12S202 log-scale light sensors to tinker with (it's been around for a while now). At the time, I was teaching myself Lua so I could program the ESP8266 using NodeMCU. What a perfect project, I thought! I'll use the Feather Huzzah 8266 and the light sensor to record the light levels. At this point, I wasn't thinking about Adafruit.IO, I was just thinking of grabbing some light level data.

Skip forward to this morning when I woke up at 1:05 AM and couldn't get back to sleep. "I should hook that sensor platform up to Adafruit.IO," thought I. And I got to work. This tutorial talks about how I did what I did. Oh, and yes, it does work.

Architecture

The architecture for this project is ... roundabout. I already had a Raspberry Pi running an MQTT broker when I started out to put together the sensor and ESP8266 Feather. It made sense to me to just use my existing MQTT infrastructure to capture the data. And here is where I made my mistake: I used the topic "/sensors" and inserted the data "<sensorname> <value>" into the queue. This is great if you just want to run

```
mosquitto_sub -v -t /sensors
```

and watch the data stream by. It does, however, cause trouble later, as we'll see.

The sensor is wired to a voltage divider, which then sends the signal to the ESP8266's analog pin (A0). That pin can accept only about one volt, but the Vcc to the sensor is 3.3v. Thus, we need the voltage divider to protect the A0 pin.

I wrote a quick Lua program that used the MQTT module to connect to my local MQTT broker and insert the sensor reading, with a one second delay. So at this point, I had data in my MQTT queue that looked like **light_001 416**

This morning, I wanted to take those data and feed them into Adafruit.IO and graph the data output. There are a number of ways I could get a graph of the data, including reading it out of MQTT with R and generating a publication-quality plot. I didn't need that, though, and it's much more interesting to use Adafruit.IO - it's kind of like UNIX; the right tool for the job is already in the toolbox.

For reasons I'll get into in a bit, I wrote a Python program that runs on a Raspberry Pi, reads the data out of my local MQTT queue, and connects to Adafruit.IO and publishes the data into the Adafruit.IO feed. This Python program cold run on the MQTT broker but I have it running on a separate RPi. It's neither here nor there. It really can be done either way.

And that's it! The information flow is: **sensor->Feather ESP8266->MQTT (local)->Python->Adafruit.IO**

Full Bill of Materials

Here's the hardware I used to build this project:

- 3 x Raspberry Pi 3 (http://adafru.it/3055) (Adafruit Product ID: 3055)
- Feather Huzzah! ESP8266 (http://adafru.it/2821) (Adafruit Product ID: 2821)
- Stacking Feather headers (http://adafru.it/2830) (Adafruit Product ID: 2830)
- 4400 mAh LiPo battery (http://adafru.it/354) (Adafruit Product ID: 354)
- Half-size breadboard (http://adafru.it/64) (Adafruit Product ID: 64)
- Jumper wires, male-male and female-male (https://adafru.it/Cgs)
- USB A to micro B cable for the Feather or 5v power supply with micro b connector (http://adafru.it/1995)(Adafruit Product ID: 1995)
- GA1A12S202 Log-scale Analog Light Sensor (http://adafru.it/1384) (Adafruit Product ID: 1384)

And here are the software dependencies:

- Raspbian Jessie, up to date
- Paho MQTT library (aka Mosquitto)
- Adafruit_IO library
- Python3 (because TonyD says it's 2016 and we should all use 3 :-)
- NodeMCU Lua

MQTT on the Raspberry Pi

The first step ist o get MQTT up and running on a Raspberry Pi to handle the data queues.

I'm just going to hit the basics here, but check out mqtt.org (https://adafru.it/pYc) for tons of additional information!

There are three steps to getting MQTT running on your Raspberry Pi:

1. Install the software
2. Configure the daemon
3. Test the configuration

Step 1 - Install the software

The software is available through Raspbian's software distribution system, which makes it easy. First, be sure your system is up to date.

```
sudo apt-get update
sudo apt-get upgrade
```

Next, install the Mosquitto packages you'll need.

```
sudo apt-get install mosquitto mosquitto-clients mosquitto-dbg python-mosquitto python3-mosquitto
```

This will install all the components you will need to use a local instance of MQTT for this project.

Step 2 - Configure the daemon

The MQTT software (Mosquitto) is controlled by a configuration file: */etc/*mosquitto*/conf.d/mosquitto.conf* Please note that I am running this on a closed and encrypted network in my house, which is maintained separately from the kids' and guest networks. It is not appropriate to use this configuration on an MQTT broker that is exposed to the Internet.

> Do not use this configuration for a broker that is exposed to outside networks like the Internet! It is insecure!!!!

Here is my copy of the *mosquitto.conf* file:

```
# Config file for mosquitto
#
# See mosquitto.conf(5) for more information.

user mosquitto
max_queued_messages 200
message_size_limit 0
allow_zero_length_clientid true
allow_duplicate_messages false

listener 1883
autosave_interval 900
autosave_on_changes false
persistence true
persistence_file mosquitto.db
allow_anonymous true
password_file /etc/mosquitto/passwd
```

Once you have created the configuration file, you must stop and restart the process (called a *daemon*) that controls MQTT. You do that with the following two commands:

```
sudo systemctl stop mosquitto.service
sudo systemctl start mosquitto.service
```

Step 3 - Test the configuration

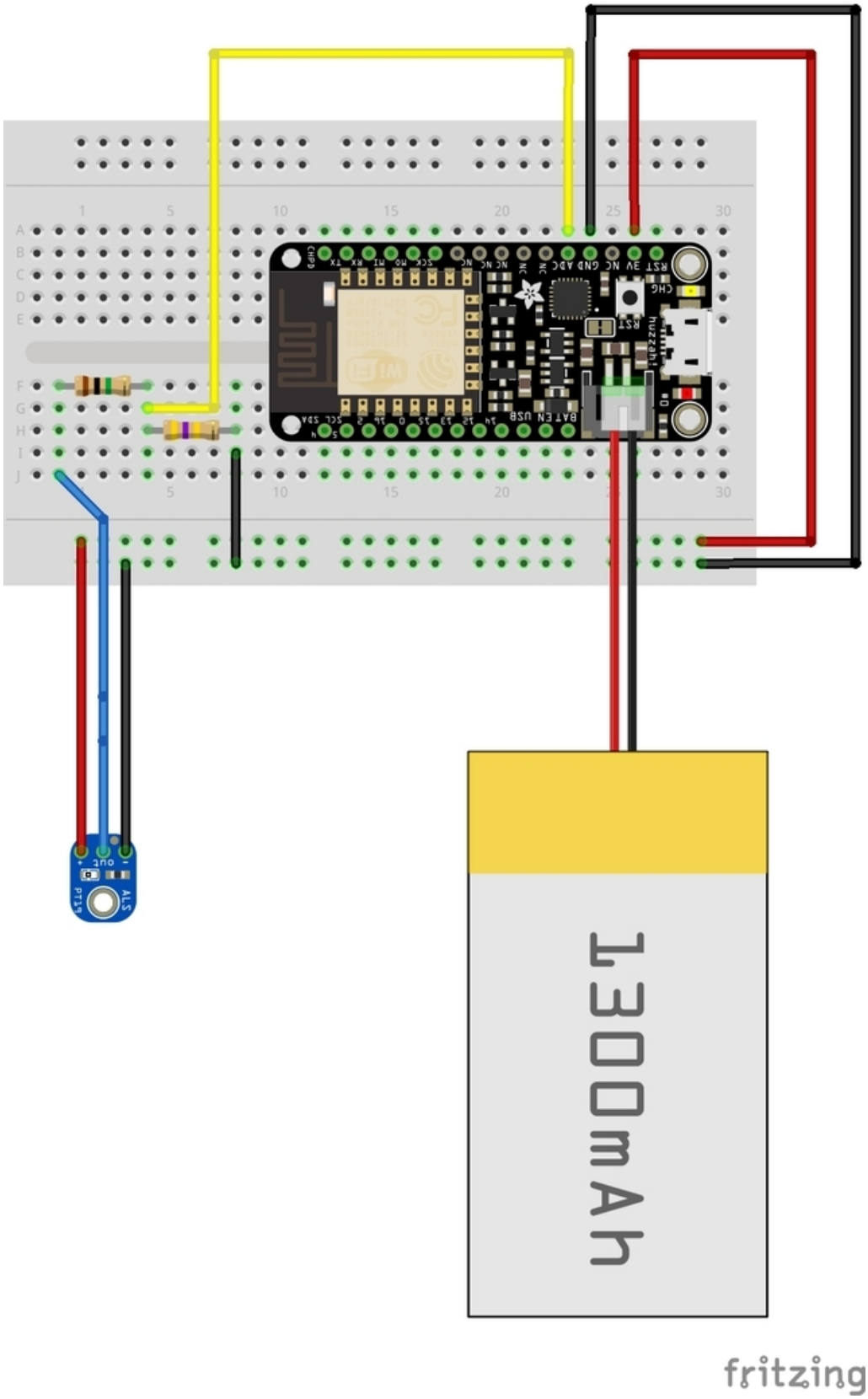
Now you're ready to test the MQTT system. On the same machine as MQTT/Mosquitto is running, execute this command:

```
mosquitto_sub -v -t '\$SYS/#'
```

This should produce a stream of diagnostic data. The contents aren't important (unless you're curious about MQTT's internals), what's important is that they show up. Once you confirm that the daemon is running, use CTRL-C to exit the program. Then we're ready to start building hardware.

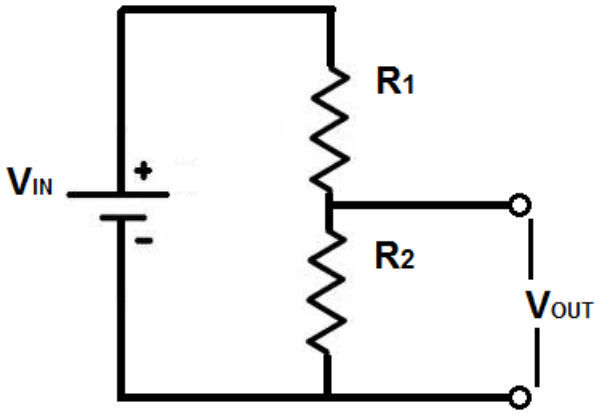
Wiring it up!

Now we'll wire up the sensor to the Feather ESP8266. I used a half-size breadboard for this, but if you're feeling bold you can solder the components together directly, or use an Adafruit perma-proto board.



Note that the sensor shown in this diagram (ALS-PT19) is different from the one specified in this tutorial; we don't have the log-scale light sensor in Fritzing yet. The wiring is identical, though, so it shouldn't matter. In fact, you can use this sensor instead of the log light sensor and it'll work fine.

First, we need to build the voltage divider out of 1M Ohm (R1) and 470K Ohm (R2) resistors. 3.3v is connected from the ESP8266 to Vcc on the light sensor. The ground pin is connected to, of course, ground. The "out" pin is wired to the voltage divider (more on the voltage divider below).



Of course, if you know the formula for the calculation, you can just use that directly. I'm not a EE, though, so I'd have had to look it up anyway. Incidentally, the voltage divider is the first "project" in Hayes and Horowitz's classic "Learning the Art of Electronics (http://adafru.it/3066)" (Adafruit Product ID 3066) - it starts on page 11.

Test the voltage divider by applying 3.3v to Vin and measuring the voltage between the two resistors with your meter. It should read about one volt. Once that's working correctly, connect 3.3v to Vcc on the sensor and connect the other side to Vin of the voltage divider. Connect one side of R2 to R1 and the other to ground. Finally, connect Vout (the connection between the two resistors) to the single Analog pin on the ESP8266. The voltage divider assures that we will not exceed the ESP8266's input voltage restriction on the analog pin, which is one volt. Remember to connect the GND pin to ground.

Now we're ready to program the ESP8266!

Programming the ESP8266

The software for this project is available on GitHub (https://adafru.it/pYd), or you can download it as a Zip file by clicking the button:

https://adafru.it/pYe

https://adafru.it/pYe

Get the software and you will see two files. First is one called *init.lua*. This is the program that runs on the ESP8266 to collect the sensor data and push it to the local MQTT broker. We'll go over the mechanism for getting that file onto the ESP8266 in a minute. The second file is the Python program that runs on a Raspberry PI; it extracts the sensor data from the local MQTT broker and sends it to Adafruit.IO.

Here are the steps necessary to flash the init.lua program onto the ESP8266.

The ESP8266 is a *wonderful* little ecosystem and I've become quite fond of it lately. The Feather HUZZAH ESP8266 comes pre-flashed with the NodeMCU (https://adafru.it/ose) Lua interpreter, as does the HUZZAH ESP8266 breakout. This means you can program the WiFi chip directly using Lua.

When you are programming the ESP8266 with Lua, you can write any Lua programs you like to the board's flash memory. If you create a program named *init.lua* it will run when the board resets.

WARNING: if you crate a tight loop (or an infinite loop) in init.lua you will effectively brick the device and have to re-flash it with a new image!

Step 1 - Edit init.lua to set your local network and MQTT broker parameters.

The init.lua file from the GitHub repository has several variables that must be set correctly to match your local configuration. These are fairly obvious because they will say things like CHANGEME. Carefully edit init.lua and set these parameters accordingly. If you break your board, don't panic! You can always flash a new version of NodeMCU to fix it. Just see the page titled "How to Re-flash Your ESP8266

Step 2 - Load Flash Memory on the ESP8266

Once you have the code for the two Lua programs saved on your local machine and edited accordingly, you need to upload them to the flash memory of the ESP8266 device. I am on a Mac and use a tool called luatool (https://adafru.it/osA) that just uploads programs into the ESP8266 flash.

Luatool is written in Python so it should run on any OS that supports Python. You can download luatool from its GitHub repository (https://adafru.it/osA).

Once you have loaded init.lua onto the board, you can press the reset button and the board should boot and run your light monitor code. You can test this by running the following command on your MQTT server:

```
mosquitto_sub -v -t /lightsensor
```

You should see a stream of sensor IDs and values representing the light leval.

The Bridging Problem (and Python)

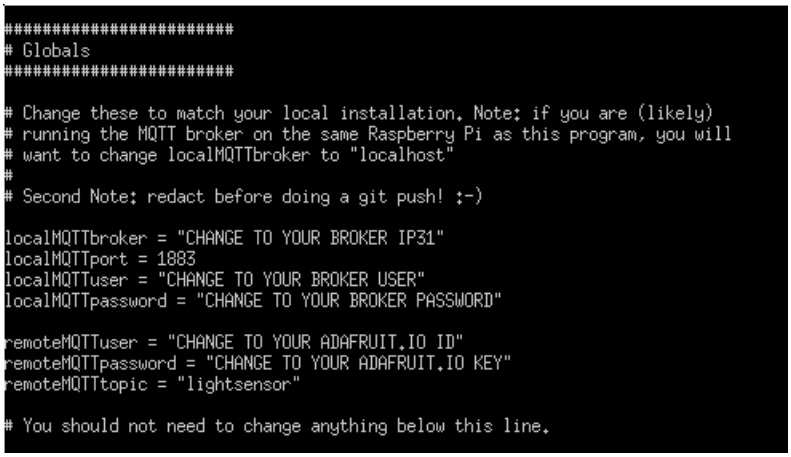
Ok, at this point you *should* have data in your MQTT broker topic queue. Assuming that's true, the next step is to get the data from your local MQTT queue into Adafruit.IO.

And here's the problem I mentioned in the introductory Overview. The data format in the local MQTT queue is **sensorname value**. There's no way (that I can figure out) to translate that automatically into a pure value stream using only MQTT broker to broker bridging. Bridging is the mechanism for a topic on one broker to be sent automatically to a topic queue on another broker. It's as if the second broker has subscribed to the topic as a client.

If I had been thinking ahead, I would have subdivided the topic queues into a tree structure, like **/sensors/light01 value**. If I had the foresight to do that, I could have configured a bridge from the local MQTT broker to Adafruit.IO. Alas, that's not what I did.

As a result, I had to write a Python program to read the data out of the local queue, connect to Adafruit.IO, and insert the values into the feed there. This is what the **ManualMQTTbridge_v01.py** program does (from the GitHub repository).

Again, you'll need to edit the global variables at the top of the Python program to match your local configuration. Don't run it yet, though. We have to configure Adafruit.IO first.

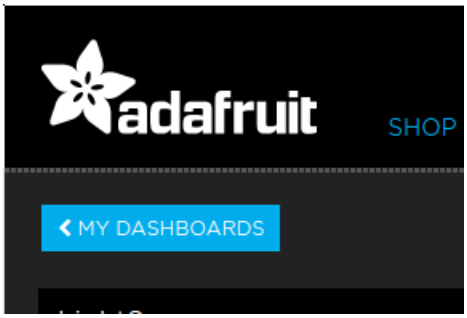


```
########################
# Globals
########################

# Change these to match your local installation. Note: if you are (likely)
# running the MQTT broker on the same Raspberry Pi as this program, you will
# want to change localMQTTbroker to "localhost"
#
# Second Note: redact before doing a git push! :-)

localMQTTbroker = "CHANGE TO YOUR BROKER IP31"
localMQTTport = 1883
localMQTTuser = "CHANGE TO YOUR BROKER USER"
localMQTTpassword = "CHANGE TO YOUR BROKER PASSWORD"

remoteMQTTuser = "CHANGE TO YOUR ADAFRUIT.IO ID"
remoteMQTTpassword = "CHANGE TO YOUR ADAFRUIT.IO KEY"
remoteMQTTtopic = "lightsensor"

# You should not need to change anything below this line.
```

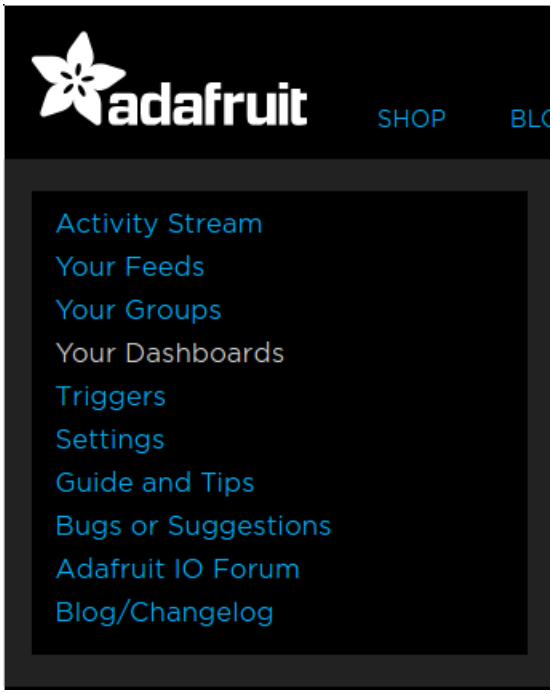
Configuring Adafruit.IO

Before we can start using Adafruit.IO to display our light level data, we need to A) have an account with Adafruit.IO and B) create the dashboard for the light sensor data. If you're keeping track of where we are, the information flow at this point is **sensor->ESP8266->MQTT->Python**. We just need to complete the last steps of configuring Adafruit.IO and we're good to go.

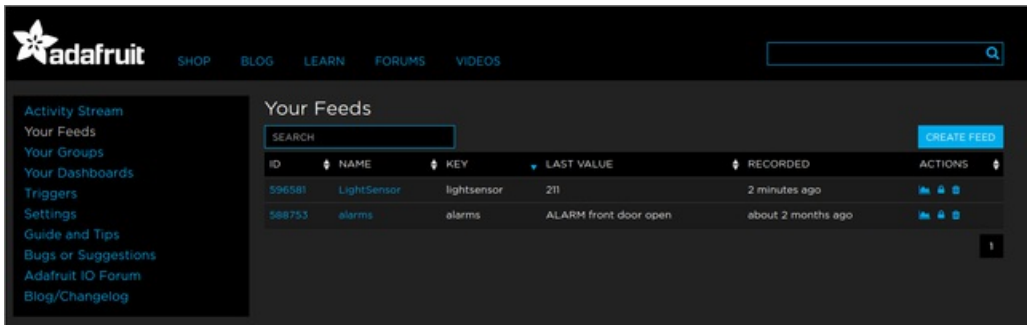
The first step, when you open io.adafruit.com, is to click the "My Dashboards" button.



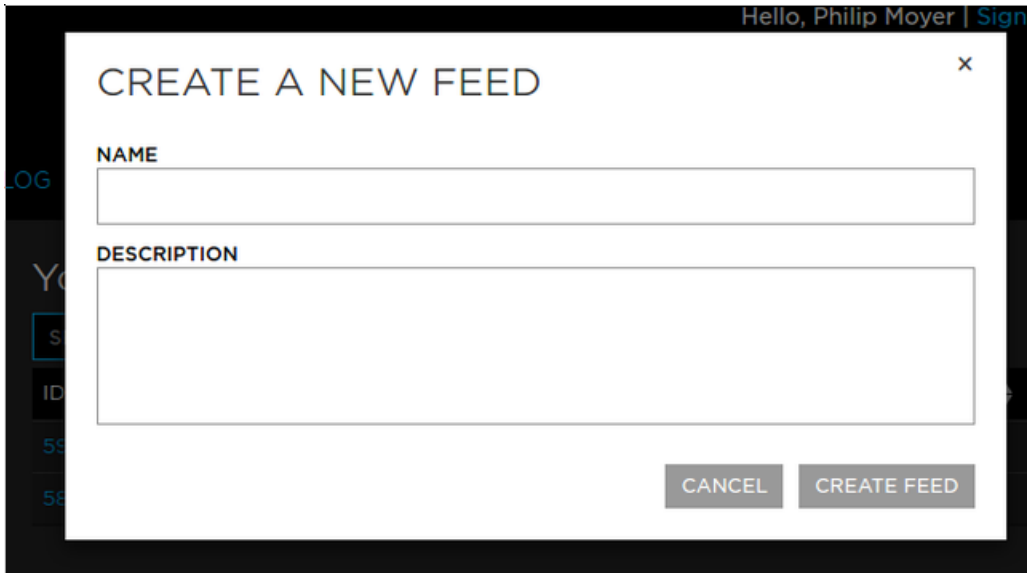
This brings up a list of any dashboards you have, but, more importantly, it brings up the navigation bar on the left side of the screen:



Click the "Your Feeds" link to open up the list of feeds you have. Remember, io.adafruit.com is in beta, so you're limited to the number of feeds you can have. The feeds list will allow you to create a new feed by clicking the blue "Create Feed" button on the right. As you can see, I already have a feed for the light sensor but we'll still walk through all the steps. Go ahead and click the "Create Feed" button.

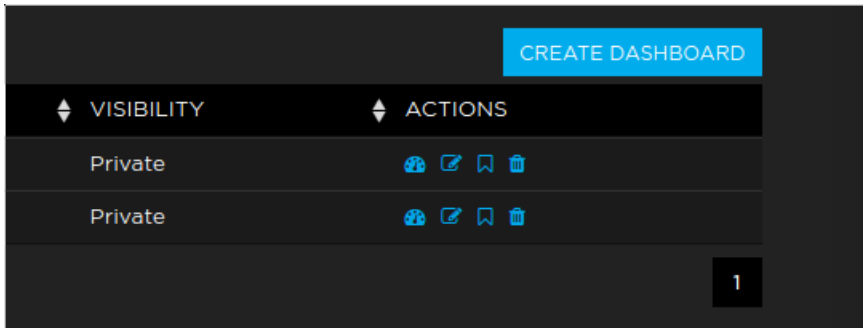


This will bring up the "New Feed" dialog.

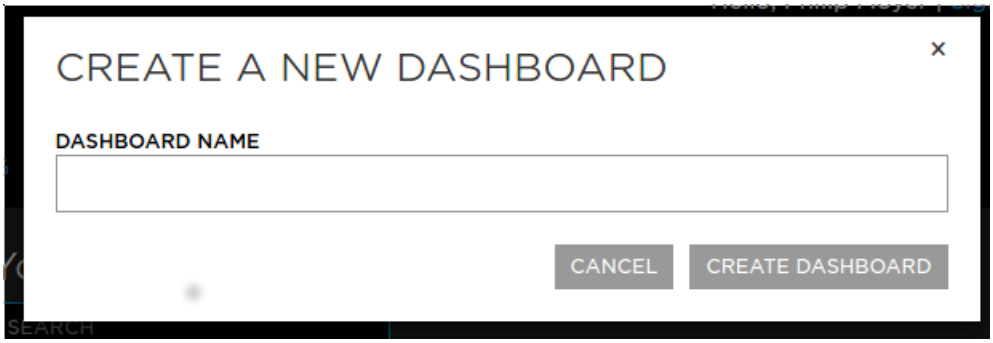


For the name, use "lightsensor" and then enter a descriptive text in the Description field. Once you're done, click on the "Create Feed" button to make the new data feed for io.adafruit.com.

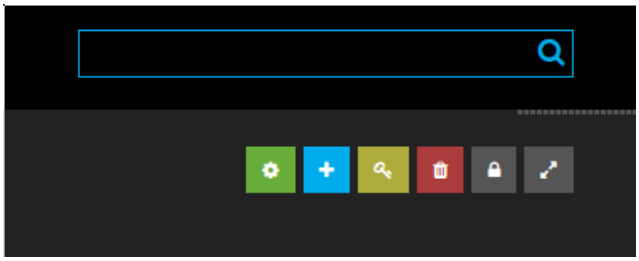
The next step is to create the Dashboard and l ink the feed to it. Go back to the navigation bar on the left side and click the "My Dashboads" link. This brings up the the list of dashboards and, more importantly, a "Create Dashboard" button on the right side of the screen. Click it.



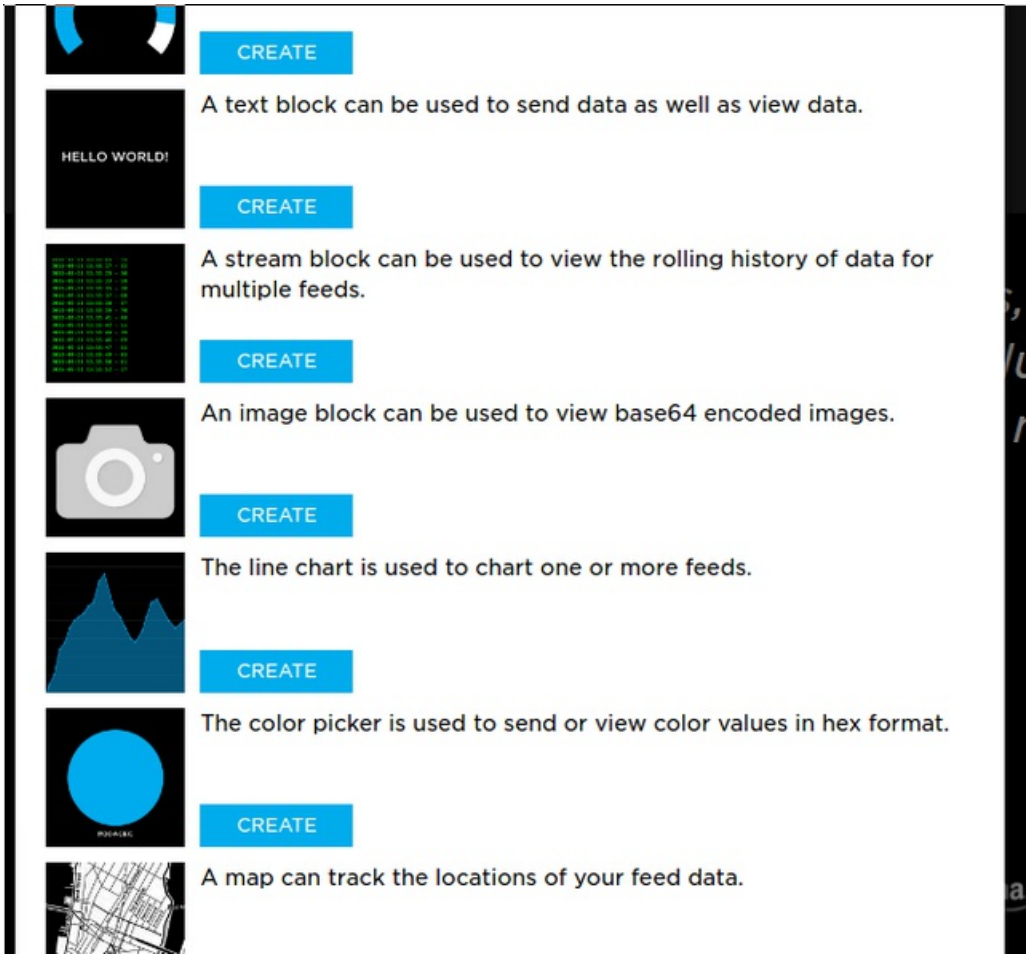
This brings up the new dashboard dialog box.



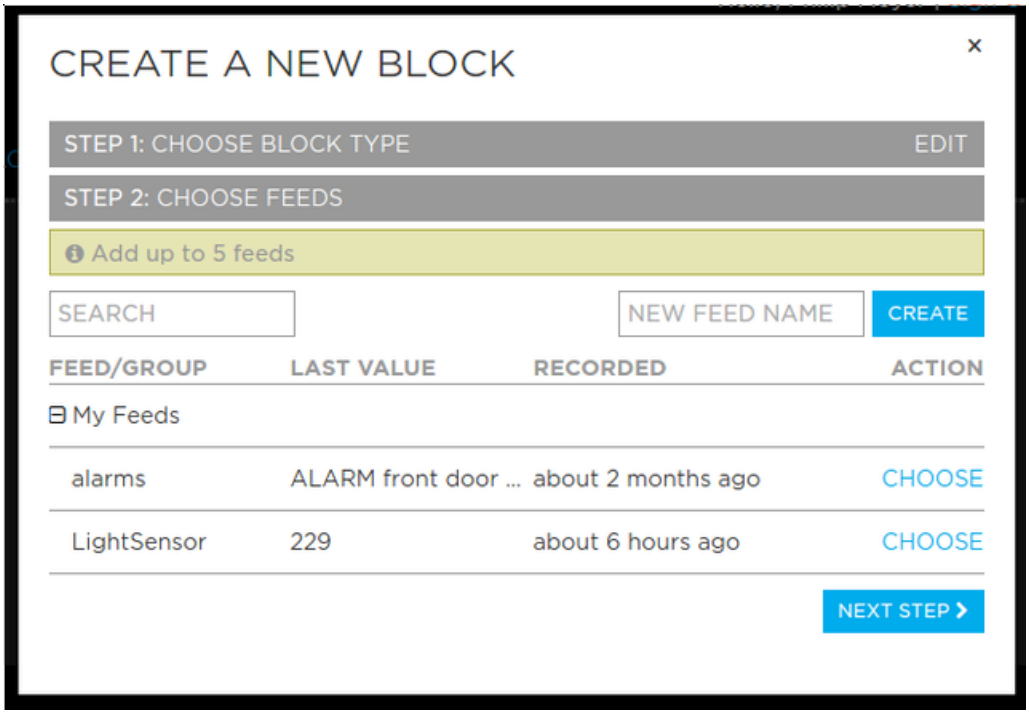
Name your dashboard something appropriate and creative, like "LightSensor" and then click the "Create Dashboard" button. This will bring up a blank dashboard. On the right side is a set of control buttons. Click the blue one with the plus sign to add a new dashboard element.



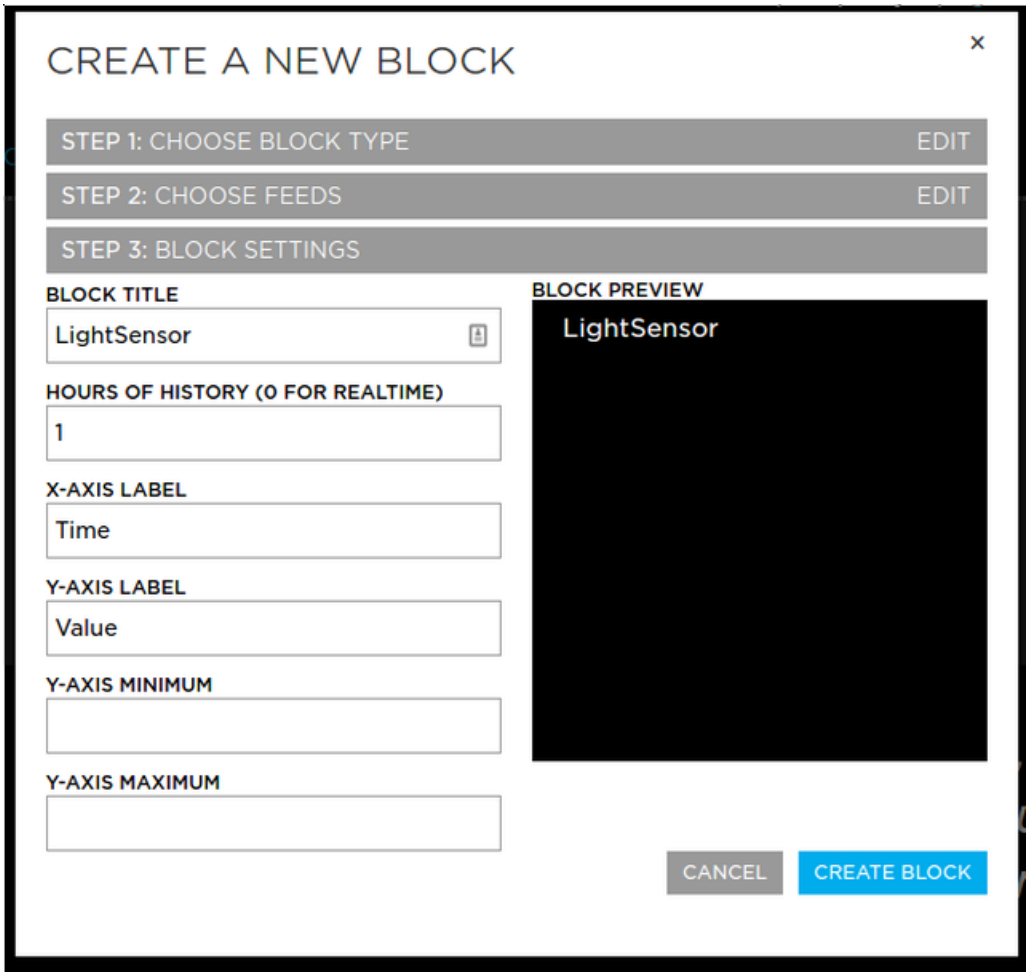
When you click that button, you'll be presented with a display of the various kinds of dashboard elements you can use. I favor the line chart, but use what feels right for you.



Once you select the dashboard element you want, you'll need to associate a topic feed with that element. Here you want to use the feed you created several steps ago The system will present a dialog box that lets you choose the feeds you want in your dashboard element.

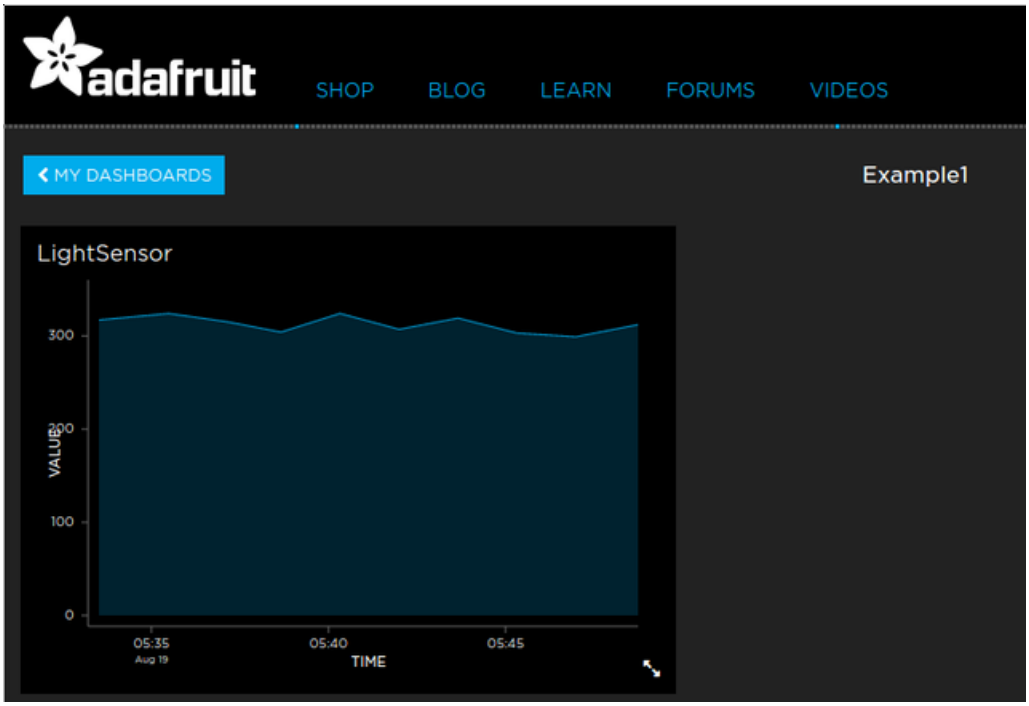


Select your light sensor feed and click "Next Step."



Now you can enter the values you want to constrain the line graph (or other element you've chosen). One of the changes I made was that I record 24 hours of data instead of just one. I'm not particular about my graph labels, but you can change whatever fields you like in order to make the dashboard as useful as possible for you.

Once you have finished editing the parameters, click the "Create Block" button in the lower right. This will create the block and give you the opportunity to change the location and size of that block.



And you're done! You now have a light sensor feeding data, via an intermediate MQTT queue, into Adafruit.IO!!!!

The Right Way to Have Built This!

I started out this tutorial by admitting I didn't build it in the best possible way. There are two ways I could have done it better.

1. Send the light sensor data directly from the Feather ESP8266 to io.adafruit.com using the MQTT library and skipped the intermediate MQTT infrastructure altogether. This is probably the simplest option.
2. Change the local MQTT topic to include subtopics, then directly bridge the local and io.adafruit.com MQTT queues.

Send directly to io.adafruit.com from the ESP8266.

I don't have the Lua code written and tested for doing this, but I know that Lua supports direct MQTT interactions. After all, that's how I get the data into the local MQTT queue. The trick is to use your Adafruit.IO key as the password when you configure the MQTT connection in Lua. Without testing this I can't guarantee it would work, but it should be ok. I'll get a couple more Feather Huzzah! ESP8266 modules and test both of these alternative configurations. I'm actually most interested in the next option

Broker-to-Broker Bridging

This is an intriguing possibility. According to the documentation, it *should* (again) work. The trick, once you have the local feed set up, is to modify your */etc/mosquitto/conf.d/mosquitto.conf* file to append this section to the example I already provided::

```
#
# Bridge to Adafruit.IO
#
connection adafruit-light-sensor
address io.adafruit.com:1883
bridge_atetempt_unsubscribe false
cleansession false
notifications false
remote_username CHANGE_TO_YOUR_USER_NAME
remote_password CHANGE_TO_YOUR_AIO_KEY
start_type automatic
topic /sensors/lightsensor  out 0 lightsensor
```

Restart mosquitto with this command:

```
sudo systemctl restart mosquitto.service
```

Once this is done the lightsensor data should be flowing automatically from your local MQTT instance into io.adafruit.com, where you can configure dashboards as before.

As I noted, I have NOT tried these two techniques yet, though I plan to do so.

How to Re-flash Your ESP8266

As you work with and learn Lua on the ESP8266, it is likely, because of some quirks in the language and processor implementation, that you'll eventually create a tight loop in your *init.lua* file. That, or an infinite loop. Either one will brick your ESP8266. Hint: if you need to do something repeatedly and quickly (like checking the sensor state rapidly), use the built-in timer function to set an interrupt with a callout function! Failing to understand that constraint is how I bricked my first ESP8266 and consequently learned how to re-flash the device.

Step 1 - Getting the firmware

Now, you *could* go and build the complete toolchain necessary to cross-compile the firmware for the ESP8266. I might do that sometime just for the experience. Also, you *could* go to the nodemcu.com (https://adafru.it/f1G) web site and try to download the current firmware. I, however, found the organization of the FTP site confusing and I couldn't find the right firmware.

Don't despair, though! A fantastic individual named Marcel Stor has created an interactive web page that allows you to build your own custom ESP8266 firmware (https://adafru.it/osD)! Just click that link and it will take you to the firmware configuration page.

Enter your e-mail address so you can receive notificatin when your bild is done and ready for download. Then scroll down to the configuration section. This is what it looks like:

Build from the Master branch unless you *absolutely* know you need dev for some reason. Also, I leave the Miscellaneous Options unchecked. In addition to the defaults, I think you'll want to add:

- ADC
- Bit
- MQTT
- Perf
- PWM

As you play more with the ESP8266 and Lua, and become ever more enamored of the platform, you'll probably want to return to this page and build specific firmware for sensors you'd like to deploy outside of the security system project.

Once you have the options configured to your satisfaction, click the blue "Start your build" link at the bottom of the page. The site will send you e-mail when your build starts, and again when it's ready to download.

There will be two firmware files: a hardware float version and an integer version. I download both, but I only use the float version.

Next you'll need to flash the firmware onto your device.

Step 2 - Getting the tool to flash an ESP8266

As I mentioned, I'm on a Mac, and the most convenient tool for me to use is called *esptool*. This is a Python program (so it shold run on any platform that supports Python) that you can clone from its GitHub repository (https://adafru.it/osE). I put my firmware builds in the same directory as *esptool.py*, just for convenience.

Step 3 - Flash the ESP8266

If you are using a Feather M0 ESP8266 or Huzzah ESP8266 Breakout, you'll need to wire Pin 0 to Ground This puts the device in bootloader mode. I use a female-female DuPont wire for this. Some boards from other vendors do not require this (in fact, many of the boards I've found arrive without any firmware at all, so the first thing you have to do is

flash them.)

Once the board is ready to flash, you need to run *esptool.py* with the correct arguments. On the Mac, the correct command is

```
python ./esptool.py --port /dev/tty.SLAB_USBtoUART write_flash 0x00000 <path to firmware>
```

Of course, replace *<path to firmware>* with your actual firmware path. It takes less than a minute to load the new firmware into flash memory on the ESP8266. Once you're done with that, you'll have a working board (or, a working board *again*) and you can get on with your project!