adafruit learning system

Adafruit IO

Created by Justin Cooper

Last updated on 2020-02-04 04:48:05 PM UTC

Overview

This guide is outdated - visit the 'Welcome to Adafruit IO' guide for the most recent overview of Adafruit IO: https://learn.adafruit.com/welcome-to-adafruit-io

Adafruit IO is a system that makes data useful. Our focus is on ease of use, and allowing simple data connections with little programming required.

IO includes client libraries that wrap our REST and MQTT APIs. IO is built on Ruby on Rails, and Node.js.

To get started, head over to io.adafruit.com to sign up (https://adafru.it/eZ8).

https://adafru.it/plC

https://adafru.it/plC

The client libraries are a work in progress. Please check back often for updates.

Getting Started

This guide is outdated - visit the 'Welcome to Adafruit IO' guide for the most recent overview of Adafruit IO: https://learn.adafruit.com/welcome-to-adafruit-io

If you haven't already, log into your Adafruit account and then head over toio.adafruit.com (https://adafru.it/fsU).

Check out the following guides to understand the basics of creating a feed and a dashboard:

- Adafruit IO Basics: Feeds (https://adafru.it/ioA)
- Adafruit IO Basics: Dashboards (https://adafru.it/f5m)

Also check out the projects page (https://adafru.it/iQB) for a list of projects and examples to help understand the service.

Continue on reading this guide to learn about the client libraries that are available to send and receive data with Adafruit IO. In addition you can learn about the protocols that Adafruit IO uses and how to use them with your own client code.

Client Libraries

This guide is outdated - visit the 'Welcome to Adafruit IO' guide for the most recent overview of Adafruit IO: https://learn.adafruit.com/welcome-to-adafruit-io

The Adafruit IO client libraries greatly simplify interacting with the server. We have a few libraries built out already:

- Arduino (https://adafru.it/iQC)
- Ruby (https://adafru.it/iQD)
- Python (https://adafru.it/iQE)
- Node.js (https://adafru.it/iQF)

Arduino

This guide is outdated - visit the 'Welcome to Adafruit IO' guide for the most recent overview of Adafruit IO: https://learn.adafruit.com/welcome-to-adafruit-io

On an Arduino there are two different libraries you can use to access Adafruit IO. One library is based on the REST API, and the other library is based on the MQTT API. The difference between these libraries is that MQTT keeps a connection to the service open so it can quickly respond to feed changes. The REST API only connects to the service when a request is made so it's a more appropriate choice for projects that sleep for a period of time (to reduce power usage) and wake up only to send/receive data. If you aren't sure which library to use, try starting with the Adafruit MQTT library below.

Adafruit MQTT Client Library

To use Adafruit IO with the MQTT protocol on an Arduino you can use the Adafruit MQTT Arduino library (https://adafru.it/fp6). This is a general-purpose MQTT library for Arduino that's built to use as few resources as possible so that it can work with platforms like the Arduino Uno. Unfortunately platforms like the Trinket 3.3V or 5V (based on the ATtiny85) have too little program memory to use the library--stick with a Pro Trinket or better!

The Adafruit MQTT library currently supports the following networking hardware/platforms:

- Adafruit CC3000 (https://adafru.it/iRa)
- Adafruit FONA (https://adafru.it/dFz)
- ESP8266 Arduino (https://adafru.it/eSH)
- Generic Arduino Client Interface (https://adafru.it/fpb) (including Ethernet shield and similar network hardware)

To install the library you can use the Arduino library manager (https://adafru.it/flm) or download the library from GitHub (https://adafru.it/fp7) and manually install it (https://adafru.it/dNR).

On some platforms the Adafruit MQTT library uses the hardware watchdog to help ensure sketches run reliably. You'll need to install the Adafruit SleepyDog sleep and watchdog library (https://adafru.it/fp8), again using either the Arduino library manager or with a manual install.

Finally make sure you have any required libraries for your network hardware installed, such as the CC3000 library (https://adafru.it/cFn) or FONA library (https://adafru.it/dDC).

Once the library is installed open or restart the Arduino IDE and check out the example code included with the library. These examples show the basic usage of the library, like how to connect to Adafruit IO, subscribe to receive changes to a feed, and how to send values to a feed.

PubSubClient MQTT Library

Another popular MQTT library for the Arduino is the PubSubClient MQTT library (https://adafru.it/e1W) and it works great to access Adafruit IO. Note that the library only works with networking libraries that support the Arduino Client interface. This means the library will work with the Ethernet shield, CC3000 or even ESP8266 Arduino, but not the FONA platform because it does not have a Client interface.

Below is a small example that shows using the PubSubClient library with the CC3000. To use this you will need the Adafruit CC3000 library (https://adafru.it/cFn) and PubSubClient library (https://adafru.it/e1W) installed in Arduino.

Note that you'll need to change the following #define configuration lines at the top to fill in your wireless AP connection details and Adafruit IO username, key, and feed name:

```
#define WLAN_SSID       "... your WiFi SSID..."
#define WLAN_PASS       "... your WiFi password..."
#define ADAFRUIT_USERNAME  "... your Adafruit username (see accounts.adafruit.com)..."
#define AIO_KEY  "... your Adafruit IO key..."
#define FEED_PATH ADAFRUIT_USERNAME "/feeds/feed-name/"
```

The FEED_PATH is the path to publish or subscribe to for interacting with a feed. Notice that the path starts with the Adafruit account name and is followed by "/feeds/feed-name" (where "feed-name" is the name of the feed).

For example if your account name was Mosfet and you were accessing a feed called Photocell the full path would look like "Mosfet/feeds/Photocell".

Below is the full example source:

```
//Example modified from the pubsubclient library linked above

#include <Adafruit_CC3000.h>
#include <ccspi.h>
#include <SPI.h>
#include <PubSubClient.h>

#define aref_voltage 3.3

// These are the interrupt and control pins
#define ADAFRUIT_CC3000_IRQ   3  // MUST be an interrupt pin!

// These can be any two pins
#define ADAFRUIT_CC3000_VBAT  5
#define ADAFRUIT_CC3000_CS    10

// Use hardware SPI for the remaining pins
// On an UNO, SCK = 13, MISO = 12, and MOSI = 11
Adafruit_CC3000 cc3000 = Adafruit_CC3000(ADAFRUIT_CC3000_CS, ADAFRUIT_CC3000_IRQ, ADAFRUIT_CC3000_VBAT,
SPI_CLOCK_DIVIDER);

#define WLAN_SSID       "... your WiFi SSID..."
#define WLAN_PASS       "... your WiFi password..."
// Security can be WLAN_SEC_UNSEC, WLAN_SEC_WEP, WLAN_SEC_WPA or WLAN_SEC_WPA2
#define WLAN_SECURITY   WLAN_SEC_WPA2

#define ADAFRUIT_USERNAME  "... your Adafruit username (see accounts.adafruit.com)..."
#define AIO_KEY  "... your Adafruit IO key..."
#define FEED_PATH ADAFRUIT_USERNAME "/feeds/feed-name/"


Adafruit_CC3000_Client client = Adafruit_CC3000_Client();
PubSubClient mqttclient("io.adafruit.com", 1883, callback, client);

void callback (char* topic, byte* payload, unsigned int length) {
  Serial.println(topic);
  Serial.write(payload, length);
  Serial.println("");
}
```

```
void setup(void)
{
  Serial.begin(115200);
  Serial.println(F("Hello, CC3000!\n"));

  // If you want to set the aref to something other than 5v
  analogReference(EXTERNAL);

  Serial.println(F("\nInitialising the CC3000 ..."));
  if (!cc3000.begin()) {
    Serial.println(F("Unable to initialise the CC3000! Check your wiring?"));
    for(;;);
  }

  uint16_t firmware = checkFirmwareVersion();
  if (firmware < 0x113) {
    Serial.println(F("Wrong firmware version!"));
    for(;;);
  }

  displayMACAddress();

  Serial.println(F("\nDeleting old connection profiles"));
  if (!cc3000.deleteProfiles()) {
    Serial.println(F("Failed!"));
    while(1);
  }

  /* Attempt to connect to an access point */
  char *ssid = WLAN_SSID;             /* Max 32 chars */
  Serial.print(F("\nAttempting to connect to ")); Serial.println(ssid);

  /* NOTE: Secure connections are not available in 'Tiny' mode! */
  if (!cc3000.connectToAP(WLAN_SSID, WLAN_PASS, WLAN_SECURITY)) {
    Serial.println(F("Failed!"));
    while(1);
  }

  Serial.println(F("Connected!"));

  /* Wait for DHCP to complete */
  Serial.println(F("Request DHCP"));
  while (!cc3000.checkDHCP()) {
    delay(100); // ToDo: Insert a DHCP timeout!
  }

  /* Display the IP address DNS, Gateway, etc. */
  while (!displayConnectionDetails()) {
    delay(1000);
  }

   // connect to the broker, and subscribe to a path
   if (mqttclient.connect(ADAFRUIT_USERNAME, AIO_KEY, "")) {
     Serial.println(F("MQTT Connected"));
     mqttclient.subscribe(FEED_PATH);
   }
}

void loop(void) {
```

```
void loop(void) {

  // are we still connected?
  if (!mqttclient.connected()) {
 mqttclient.subscribe(FEED_PATH);
    mqttclient.publish(FEED_PATH, "11");
  } else {
    mqttclient.publish(FEED_PATH, "11");
  }

  mqttclient.loop();
  delay(250);
}


/**************************************************************************/
/*!
    @brief  Tries to read the CC3000's internal firmware patch ID
*/
/**************************************************************************/
uint16_t checkFirmwareVersion(void)
{
  uint8_t major, minor;
  uint16_t version;

#ifndef CC3000_TINY_DRIVER
  if(!cc3000.getFirmwareVersion(&major, &minor))
  {
    Serial.println(F("Unable to retrieve the firmware version!\r\n"));
    version = 0;
  }
  else
  {
    Serial.print(F("Firmware V. : "));
    Serial.print(major); Serial.print(F(".")); Serial.println(minor);
    version = major; version <<= 8; version |= minor;
  }
#endif
  return version;
}

/**************************************************************************/
/*!
    @brief  Tries to read the 6-byte MAC address of the CC3000 module
*/
/**************************************************************************/
void displayMACAddress(void)
{
  uint8_t macAddress[6];

  if(!cc3000.getMacAddress(macAddress))
  {
    Serial.println(F("Unable to retrieve MAC Address!\r\n"));
  }
  else
  {
    Serial.print(F("MAC Address : "));
    cc3000.printHex((byte*)&macAddress, 6);
  }
}
```

```
/**************************************************************************/
/*!
    @brief  Tries to read the IP address and other connection details
*/
/**************************************************************************/
bool displayConnectionDetails(void)
{
  uint32_t ipAddress, netmask, gateway, dhcpserv, dnsserv;

  if(!cc3000.getIPAddress(&ipAddress, &netmask, &gateway, &dhcpserv, &dnsserv))
  {
    Serial.println(F("Unable to retrieve the IP Address!\r\n"));
    return false;
  }
  else
  {
    Serial.print(F("\nIP Addr: ")); cc3000.printIPdotsRev(ipAddress);
    Serial.print(F("\nNetmask: ")); cc3000.printIPdotsRev(netmask);
    Serial.print(F("\nGateway: ")); cc3000.printIPdotsRev(gateway);
    Serial.print(F("\nDHCPsrv: ")); cc3000.printIPdotsRev(dhcpserv);
    Serial.print(F("\nDNSserv: ")); cc3000.printIPdotsRev(dnsserv);
    Serial.println();
    return true;
  }
}
```

Adafruit IO REST Client Library

The Adafruit IO Arduino library (https://adafru.it/fpd) is a simple library for sending and receiving the latest value for a feed. This library uses the send (https://adafru.it/iRb) and last (https://adafru.it/iRb) Adafruit IO REST API calls and takes care of all the work to use the Adafruit IO REST API.

The REST client library supports the following networking platforms/hardware:

- Adafruit CC3000 (https://adafru.it/iRa)
- Adafruit FONA (https://adafru.it/dFz)
- ESP8266 Arduino (https://adafru.it/eSH)
- Generic Arduino Client Interface (https://adafru.it/fpb) (including Ethernet shield and similar network hardware)

To install the library you can use the Arduino library manager (https://adafru.it/flm) or download the library from GitHub (https://adafru.it/fpd) and manually install it (https://adafru.it/dNR).

Finally make sure you have any required libraries for your network hardware installed, such as the CC3000 library (https://adafru.it/cFn) or FONA library (https://adafru.it/dDC).

Once the library is installed open or restart the Arduino IDE and check out the example code included with the library. These examples show the basic usage of the library, like how to connect send or receive the latest value for a feed.

Ruby

This guide is outdated - visit the 'Welcome to Adafruit IO' guide for the most recent overview of Adafruit IO: https://learn.adafruit.com/welcome-to-adafruit-io

To use Adafruit IO with a Ruby program you can install and use the Adafruit io-client-ruby code from Github (https://adafru.it/enB). This library wraps the REST API to access feeds and data on Adafruit IO.

The library readme shown on GitHub (https://adafru.it/enB) describes how to install and use the library. Be sure to also see the examples (https://adafru.it/fpf) included with the library.

Python

This guide is outdated - visit the 'Welcome to Adafruit IO' guide for the most recent overview of Adafruit IO: https://learn.adafruit.com/welcome-to-adafruit-io

To use Adafruit IO with a Python program you can install and use the Adafruit io-client-python code from Github (https://adafru.it/eli). This library can use both the REST API and MQTT API to access feeds and data on Adafruit IO.

The library readme shown on GitHub (https://adafru.it/eli) describes how to install and use the library. Be sure to also see the examples (https://adafru.it/fpg) included with the library.

Node.js

This guide is outdated - visit the 'Welcome to Adafruit IO' guide for the most recent overview of Adafruit IO: https://learn.adafruit.com/welcome-to-adafruit-io

To use Adafruit IO with a Node.js program you can install and use the Adafruit io-client-node code from Github (https://adafru.it/elj). This library can use both the REST API and MQTT API to access feeds and data on Adafruit IO.

The library readme shown on GitHub (https://adafru.it/elj) describes how to install and use the library.

Browser

This guide is outdated - visit the 'Welcome to Adafruit IO' guide for the most recent overview of Adafruit IO: https://learn.adafruit.com/welcome-to-adafruit-io

An easy way to interact with IO is just by using GET requests in your browser (or wherever).

The one drawback to this is you must pass in your unique AIO key as a query parameter which could then be cached somewhere along the way, and used to gain access to your data. That being said, if you're not doing anything mission critical, this is just another way you can interact with Adafruit IO.

The following isn't terribly secure due to the AIO Key being part of the query string.

That being said, this should work fine for something like an output only weather station.

Send Data

An example to send data (simple GET request):

```
https://io.adafruit.com/api/groups/weather/send.json?x-aio-
key=a052ecc32b2de1c80abc03bd471acd1d6b218e5c&temperature=13&humidity=12&wind=45
```

The above url would create a 'weather' group, and three feeds, 'temperature', 'humidity', and 'wind' all with corresponding stream values for you.

Let's break that down into the core components.

```
https://io.adafruit.com/api/groups/:group_name/send.json
```

group_name: alphanumeric and dashes.

The **group_name** will be created automatically, or found and used, if it already exists.

```
?x-aio-key=1234567890&feed_name=value&feed_name=value
```

x-aio-key: this is your unique AIO-key. The master key can be found on your dashboard on i.adafruit.com.

feed_name: This is the name of a new or existing feed. A new feed will be created automatically for you.

value: The new value to be stored.

Receive Data

You can receive data from a group with a get request as well.

An example (simple GET request):

```
https://io.adafruit.com/api/groups/weather/receive.json?x-aio-
key=a052ecc32b2de1c80abc03bd471acd1d6b218e5c
```

The only difference between sending and receiving is that we swapped out 'send' for 'receive', and removed the additional feed_name parameters.

REST API

This guide is outdated - visit the 'Welcome to Adafruit IO' guide for the most recent overview of Adafruit IO: https://learn.adafruit.com/welcome-to-adafruit-io

The new IO API docs can now be found at: https://io.adafruit.com/api/docs (https://adafru.it/ikf)

The IO API is over HTTPS where possible. Some devices may not support HTTPS easily, so we do offer the API over the unsecure HTTP protocol, used at your own risk.

The base URL is:

```
https://io.adafruit.com/api
```

The current version of the api is: v2.

If you'd like to keep working with the v1 API, you can use /api/v1.

HTTP Status Codes


```
HTTP 200: OK
```

Standard response, everything is OK. The response body will include the data, if applicable.

```
HTTP 400: Bad Request
```

There was a problem understanding the request sent to the service. In most cases this means the code that generated the request has a bug and generated a malformed HTTP request. For example a common problem is if the length of the request doesn't match the Content-Length header sent to the service.

```
HTTP 401: Unauthorized
```

The API Key is invalid. The response body will indicate the error condition.

```
HTTP 404: Unauthorized
```

There was a problem locating the resource you requested. Either check the spelling of the key or id given, or it's possible the resource no longer exists.

```
HTTP 503: Unavailable
```

It's possible you've bumped up against the API limits, and have been throttled. Try again in a short while.

MQTT API

This guide is outdated - visit the 'Welcome to Adafruit IO' guide for the most recent overview of Adafruit IO: https://learn.adafruit.com/welcome-to-adafruit-io

MQTT (https://adafru.it/f29), or message queue telemetry transport, is a protocol for device communication that Adafruit IO supports. Using a MQTT library or client you can publish and subscribe to a feed to send and receive feed data.

If you aren't familiar with MQTT check out this introduction from the HiveMQ blog (https://adafru.it/fpt). All of the subsequent posts in the MQTT essentials series (https://adafru.it/fpu) are great and worth reading too.

To use the MQTT API that Adafruit IO exposes you'll need a MQTT client library. For Python, Node.js, and Arduino you can use Adafruit's IO client libraries as they include support for MQTT (see the client libraries section (https://adafru.it/iRc)). For other languages or platforms look for a MQTT library that ideally supports the MQTT 3.1.1 protocol.

Connection Details

You will want to use the following details to connect a MQTT client to Adafruit IO:

- **Host:** io.adafruit.com
- **Port:** 1883 *or* 8883 (for SSL encrypted connection)
- **Username:** your Adafruit account username (see the accounts.adafruit.com (https://adafru.it/fpw) page here to find yours)
- **Password:** your Adafruit IO key (click the AIO Key button on a dashboard to find the key)

We strongly recommend using SSL (https://adafru.it/oSd) if your MQTT client allows it.

If the MQTT library requires that you set a **client ID** then use a unique value like a random GUID. Most MQTT libraries handle setting the client ID to a random value automatically though.

Topics

Adafruit IO's MQTT API exposes feed data using special topics. You can publish a new value for a feed to its topic, or you can subscribe to a feed's topic to be notified when the feed has a new value. Any one of the following topic forms is valid for a feed:

- (username)/feeds/(feed name or key)
- (username)/f/(feed name or key)

Where (username) is your Adafruit IO username (the same as specified when connecting to the MQTT server) and (feed name or key) is the feed's name or key. The smaller '/f/' path is provided as a convenience for small embedded clients that need to save memory.

Check out our guide to Feed Naming for the full details (https://adafru.it/oSe).

For example if your username is **mosfet** and you're accessing a feed called **Photocell One** (which has a Key of photocell-one) you can use any of these paths:

- mosfet/feeds/Photocell One
- mosfet/f/Photocell One
- mosfet/feeds/photocell-one
- mosfet/f/photocell-one

To append a new value to a feed perform a MQTT publish against the feed path and provide the new feed value as the payload of the request.

To be notified of a change in a feed perform a MQTT subscribe against the feed path. When a new value is added to the feed then the Adafruit IO MQTT server will send a notification with the new feed value in the payload.

You can also subscribe to the parent 'feeds' path to be notified when *any* owned feed changes using MQTT's `#` wildcard character. For example the mosfet user could subscribe to either:

- mosfet/feeds/#
- mosfet/f/#

Once subscribed to the path above any change to a feed owned by mosfet will be sent to the MQTT client. The topic will specify the feed that was updated, and the payload will have the new value.

Be aware the MQTT server sends feed updates on **all** possible paths for a specific feed. For example, subscribing to `mosfet/f/#` and publishing to `mosfet/f/photocell-one` would produce messages from: mosfet/f/photocell-one, mosfet/f/photocell-one/json, and mosfet/f/photocell-one/csv; each referring to the same updated value. To reduce noise, make sure to grab the specific topic of the feed / format you're interested in and change your subscription to that.

PLEASE NOTE: as we adjust which identifiers we use for Feeds internally, the feed updates you receive when using a wildcard will include *but may not be limited to* the ones shown above.

If you'd like to avoid the formatted feeds ("/json" and "/csv" topics) but still see all the feeds you're publishing to, you can use MQTT's `+` wildcard in place of `#`. In this case, subscribing to `mosfet/f/+` would produce output on `mosfet/f/photocell-one`, but not `mosfet/f/photocell-one/json`.

Publish QoS Levels

One feature of MQTT is the ability to specify a QoS, or quality of service, level when publishing feed data. This allows an application to confirm that its data has been sucessfully published. If you aren't familiar with MQTT QoS levels be sure to read this great blog post (https://adafru.it/fpz) explaining their meaning.

For publishing feed values the Adafruit IO MQTT API supports QoS level **0 (at most once)** and **1 (at least once)** only. QoS level 2 (exactly once) is not currently supported.

Rate Limit

Adafruit IO's MQTT server imposes a rate limit to prevent excessive load on the service. If a user performs too many publish actions in a short period of time then some of the publish requests might be rejected. The current rate limit is at most **1 request per second** (or **60 requests within 60 seconds**).

If you exceed this limit, a notice will be sent to the *(username)/throttle* topic. You can subscribe to the topic if you wish to know when the Adafruit IO rate limit has been exceeded for your user account.

This limit applies to all connections so if you have multiple devices or clients publishing data be sure to delay their

updates enough that the total rate is below 2 requests/second.

Data Format

There are a few ways to send data to our MQTT API if you're writing your own client library.

The simplest way to send values to an IO Feed topic is to just send the value. For example, a temperature sensor is going to produce numeric values like `22.587`. If you're sending to `mosfet/feeds/photocell-one` you can send using a number data type or a string data type. That means either `22.587` or `"22.587"` will be accepted and stored as a numeric value.

Adafruit IO does its best to treat data as numeric values so that we can show you your data as a chart on an Adafruit IO dashboard and through our Charting API (https://adafru.it/uff).

Data with Location

To tag your data with a location value, you'll either need to wrap it in a JSON object first or send it to the special /csv formatted MQTT topic.

Sending JSON

JSON can be sent to either the base topic or the /json topic-- for example, `mosfet/feeds/photocell-one` or `mosfet/feeds/photocell-one/json`. The proper format for location tagged JSON data is:

```
{
 "value": 22.587,
 "lat": 38.1123,
 "lon": -91.2325,
 "ele": 112
}
```

Specifically, JSON objects **must** include a "value" key, and **may** include "lat", "lon", and "ele" keys.

Sending CSV

Alternatively, you can send location tagged data to /csv topics. In this example that would be the topic `mosfet/feeds/photocell-one/csv` instead of `mosfet/feeds/photocell-one`. Both store data in the same feed. The format IO expects for location tagged CSV data is VALUE, LATITUDE, LONGITUDE, ELEVATION.

With the example data shown before, that means you could publish the string `"22.587,38.1123,-91.2325,112"` to `mosfet/feeds/photocell-one/csv`. to store the value "22.587" in the location latitude: 38.1123, longitude: -91.2325, elevation: 112.

Examples

Using a simple Ruby MQTT library and the data shown, all these examples publish the same data to the same feed:

```
# first you'll need https://github.com/njh/ruby-mqtt
require 'mqtt'

username = 'test_username'
key      = 'not-a-real-key'
url      = "mqtts://#{ username }:#{ key }@io.adafruit.com"

mqtt_client = MQTT::Client.connect(url, 8883)

# simplest thing that could possibly work
mqtt_client.publish('test_username/feeds/example', 22.587)

# sending numbers as strings is fine, IO stores all data internally
# as strings anyways
mqtt_client.publish('test_username/feeds/example', '22.587')

# CSV formatted, no location
mqtt_client.publish('test_username/feeds/example/csv', '22.587')

# CSV formatted, with location
mqtt_client.publish('test_username/feeds/example/csv',
                    '22.587,38.1123,-91.2325,112')

# JSON formatted, no location
mqtt_client.publish('test_username/feeds/example', '{"value":22.587}')
mqtt_client.publish('test_username/feeds/example/json', '{"value":22.587}')

# JSON formatted, with location
mqtt_client.publish('test_username/feeds/example',
  '{"value":22.587,"lat":38.1123,"lon":-91.2325,"ele":112}')
mqtt_client.publish('test_username/feeds/example/json',
  '{"value":22.587,"lat":38.1123,"lon":-91.2325,"ele":112}')
```

Sending JSON data through Adafruit IO

Because Adafruit IO supports additional features beyond a basic MQTT brokering service, such as location tagging for data points, the service supports data in the JSON format described above. Namely:

```
{
 "value": 22.587,
 "lat": 38.1123,
 "lon": -91.2325,
 "ele": 112
}
```

This lets us store the individual value, `22.587`, and data *about* the value: its latitude, longitude, and elevation. Metadata (https://adafru.it/Cwg)!

But what happens when the value you want to send is itself JSON? Good news! There are a few solutions available to you in that situation.

IO formatted JSON

The simplest way to send JSON data to Adafruit IO is to wrap it in the format described above. For example, if instead of `22.587`, I wanted to send something like, `{"sensor-1":22.587,"sensor-2":13.182}`, the "wrapped" version would look like this:

```
{
 "value": {"sensor-1":22.587,"sensor-2":13.182},
 "lat": 38.1123,
 "lon": -91.2325,
 "ele": 112
}
```

It's worth noting that because Adafruit IO parses the entire JSON object that you send it, any valid JSON will be parsed and when it is stored in our system and forwarded to any subscribers, it will be regenerated. The significance of that is that if you publish JSON data with whitespace, it will be stored and republished *without* whitespace, because our generator produces the most compact JSON format possible.

Double encoded JSON strings

The second way you can send JSON data as a value is to "double encode" it before sending, in which case IO will treat it as a raw string. If you're using something like javascript's `JSON.stringify` function or Ruby's `JSON.generate`, double encoding means passing the result of `JSON.stringify` through `JSON.stringify` a second time. In this node.js console example, you can see the difference:

```
> JSON.stringify({"sensor-1":22.587,"sensor-2":13.182})
'{"sensor-1":22.587,"sensor-2":13.182}'
> JSON.stringify(JSON.stringify({"sensor-1":22.587,"sensor-2":13.182}))
'"{\"sensor-1\":22.587,\"sensor-2\":13.182}"'
```

The double encoded JSON string can be sent directly through Adafruit IO without interference from our processing system, because the processing system will not interpret it as JSON. In your receiving code, because the value passed through includes surrounding double quotes, you have to call your parse function twice to restore the JSON object.

```
> var input = '"{\\\"sensor-1\\\":22.587,\\\"sensor-2\\\":13.182}"'
> JSON.parse(JSON.parse(input))
{ 'sensor-1': 22.587, 'sensor-2': 13.182 }
```

Non-IO formatted JSON

The third way you can send raw JSON data is to just send it. If Adafruit IO doesn't find a "value" key in the JSON object you send, it will treat the whole blob as plain text and store and forward the data. That means with our example JSON object, sending the string `{"sensor-1":22.587,"sensor-2":13.182}` will result in `{"sensor-1":22.587,"sensor-2":13.182}` being stored in IO and sent to MQTT subscribers.

Projects

This guide is outdated - visit the 'Welcome to Adafruit IO' guide for the most recent overview of Adafruit IO: https://learn.adafruit.com/welcome-to-adafruit-io

The following are useful guides and examples to help learn more and get started using Adafruit IO:

- Adafruit IO Basics: Feeds (https://adafru.it/ioA)
- Adafruit IO Basics: Dashboards (https://adafru.it/f5m)
- Adafruit IO Basics: Digital Output (https://adafru.it/iRe)
- Adafruit IO Basics: Button Input (https://adafru.it/m9f)
- A Sillier Mousetrap: Logging Mouse Data To Adafruit IO With The Raspberry Pi (https://adafru.it/iRA)

Check out the Adafruit IO section (https://adafru.it/iRB) of the learning system for more recent guides too.

Data Policies

This guide is outdated - visit the 'Welcome to Adafruit IO' guide for the most recent overview of Adafruit IO: https://learn.adafruit.com/welcome-to-adafruit-io

[updated as of August 2017]

We're currently locking in how much and how long data should be retained. We had to set some values for the beta, and will definitely be adjusting these as time goes on.

1. Each feed stores data for 30 days.
2. You can write data to the system, across all feeds, up to 60 times per minute. Data creating, updating, and deleting all count against the limit.
3. You may read your data an unlimited amount of time, as long as you remain within the throttle times.
4. 10k rows of "Activity" data is maintained. Activity data just tracks the last actions of your IO account on your Activities page for your information.