

# IoT Engineering

## 9: Dashboards and Apps for Sensor Data

CC BY-SA, Thomas Amberg, FHNW  
(unless noted otherwise)

# Today

$\frac{1}{3}$  slides,

$\frac{2}{3}$  hands-on.

Slides, code & hands-on: [tmb.gr/iot-9](https://tmb.gr/iot-9)



# Prerequisites

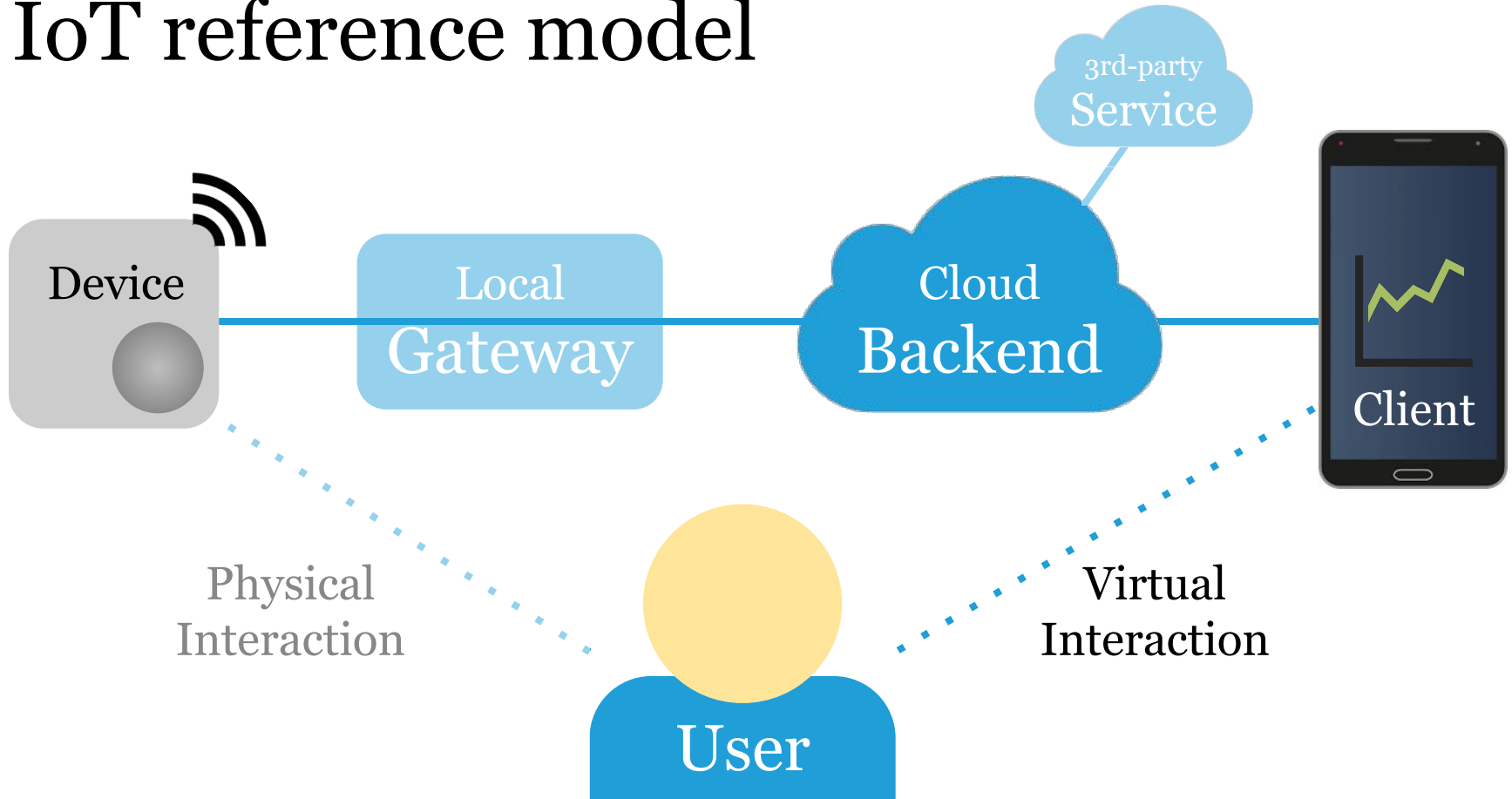
We use [curl](#) and the [mqtt](#) CLI tool to emulate devices.

The [Raspberry Pi](#) with [Node.js](#) will be our "backend".

Some examples require [Docker](#) on your computer.

Note: Docker part is still in beta, will be updated.

# IoT reference model



# Dashboards

Dashboard as a service — easy set up, but: dependency.

Self-hosted dashboard — keep control, but: operations.

Graph libraries — re-use, flexible, but: dev & ops work.

Build your own — max. control, but also: max. work.

# Dashboard as a service

Backend, defining data formats & information model.

Device-side backend API to get data in (HTTP/MQTT).

Data storage or caching functionality (sliding window).

Client-side API to get data out (HTTP or Websocket).

Private or public dashboard Web UI or client app.

# Information model

The information model defines how data is structured.

It's the "common denominator" of all involved parties.

Data formats (on the wire) define how it's transported.

The information model is more about data semantics.

E.g. what is a device, what is a sensor measurement? 7

# ThingSpeak

ThingSpeak timestamps, stores and displays data.

It supports per device *channels* with 1-N *fields* each.

Graph controls can be embedded in HTML Web UIs.

ThingSpeak provides HTTP and MQTT endpoints.



# ThingSpeak HTTP API

ThingSpeak has a **device-** and **client-side HTTP API**.

Host: `api.thingspeak.com`

Port: 80 or 443

POST `/update?key=WRITE_API_KEY&field1=42`

GET `/channels/CHANNEL_ID/feed.json?  
key=READ_API_KEY`

# ThingSpeak MQTT API

ThingSpeak has a [device-](#) and [client-side MQTT API](#).

Host: `mqtt.thingspeak.com`

Port: 1883 or 8883 (or Websocket: 80, 443)

```
PUB -t 'channels/CHANNEL_ID/publish/\nWRITE_API_KEY' -m 'field1=42&field2=23'
```

```
SUB -t 'channels/CHANNEL_ID/subscribe/\nFORMAT/READ_API_KEY'
```

# Uniontown Weather Data

Channel ID: 3

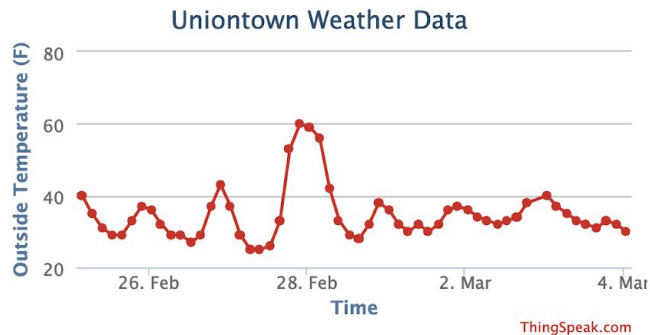
Author: [iothans](#)

Access: Public

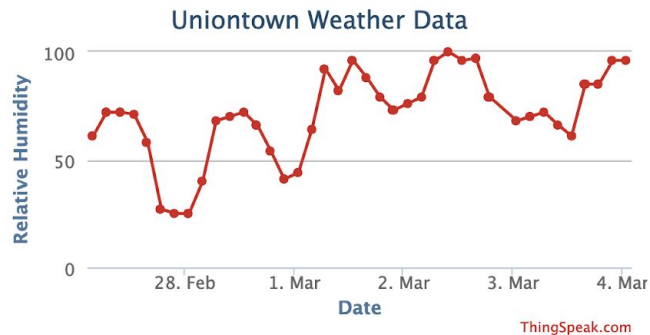
Weather data from Uniontown, PA

🔑 [temperature](#), [humidity](#), [weather station](#), [dew point](#), [channel\\_3](#)

Field 1 Chart



Field 2 Chart



# Cayenne

**Cayenne** apps display data from any MQTT broker.

Per *thing* (device), multiple data *channels* are supported, with *type*, *unit* and *value* fields.

SDKs for **ESP8266**, **Node.js**, etc., simplify sending values, e.g. encoded in the **CayenneLPP** data format.

Cayenne also provides an MQTT broker endpoint.

# Cayenne MQTT API

Cayenne specifies a device-side [MQTT API](#) and SDKs.

Host: `mqtt.mydevices.com`

Port: 1883 or 8883

```
PUB -t 'v1/MQTT_USER/things/DEVICE_ID/\n      data/CHANNEL_ID' -m 'TYPE,UNIT=VALUE'
```

For details, see the Cayenne [payload documentation](#), which is wrapped in device specific integrations.

# Cayenne HTTP API

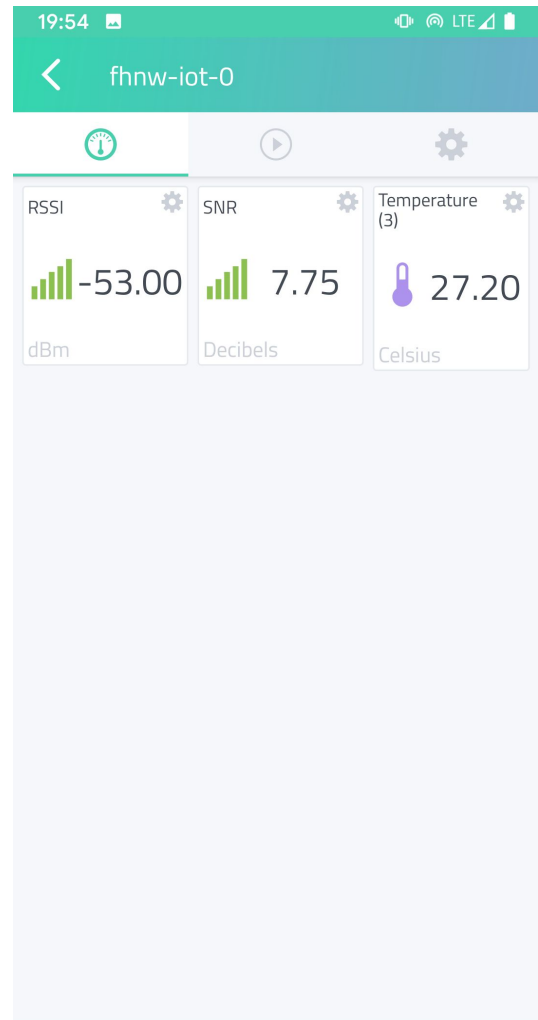
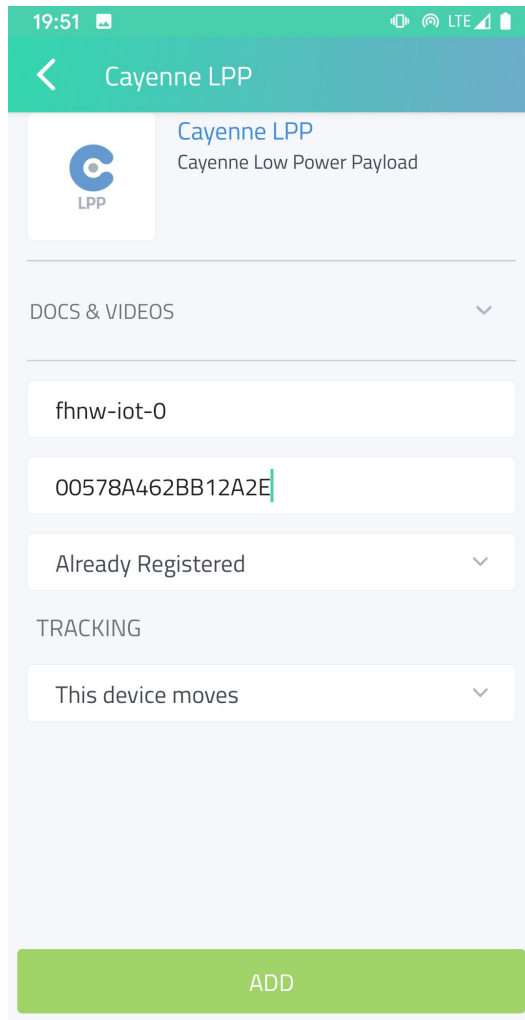
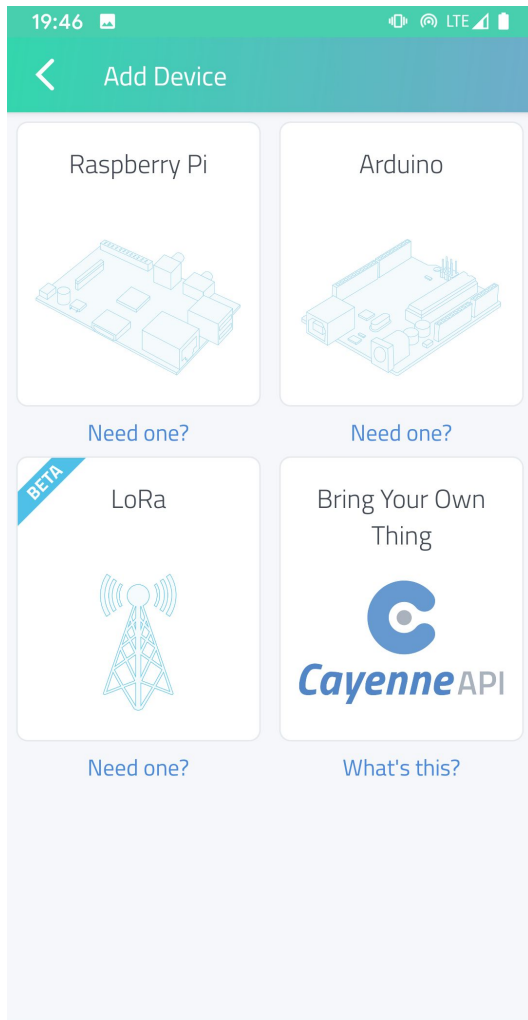
The Cayenne **iOS** and **Android** apps use an **HTTP API**.

It enables device management, to set up devices/keys.

It is client-only, devices connect to the MQTT API.

As soon as data comes in, graphs are generated\*.

\*) The data format includes types, units and values. 14



# Integrations

Integrations allow backend services to work together.


E.g. TTN backend integration w/ Cayenne MyDevices.

Integration adapters can be provided by either party.

The adapter code can be hosted on either backend.

Usually API keys or tokens are all that is needed.



Applications >  fhnw-iot > Integrations

## ADD INTEGRATION



### MyDevices (v2.6.0)

myDevices

Quickly design, prototype and commercialize IoT solutions with myDevices Cayenne

[documentation](#)

#### Process ID

The unique identifier of the new integration process

fhnw-iot-cayenne



#### Access Key

The access key used for downlink

default key devices messages



# Hands-on, 15': Dashboard as a service

Choose a dashboard service\* and a transport protocol.

Check the API docs to understand the payload format.

Send data "as a device" with [curl](#) or with the [mqtt](#) CLI.

The CLI runs on the Raspberry Pi or on your laptop.

\*) Try ThingSpeak, Cayenne or [ThingsBoard.io](#)

# Glue code

**Glue code** is a simple way to integrate service APIs.

A custom adapter acts as a client of both services, e.g. to get data from a LoRaWAN backend to a dashboard.

It converts payload formats, can be hosted anywhere.



# Node.js glue code

`.js`<sup>MQTT</sup>, `.js`<sup>HTTP</sup>

...

```
var client = new ttn.Client('eu', appId, key);
client.on('message', (devId, msg) => {
  var bytes = msg.payload_raw;
  var x = ((bytes[0] << 8) | bytes[1]) / 100.0;
  http.post('http://api.thingspeak.com/update?'
    + 'api_key=' + writeApiKeys[msg.dev_id]
    + '&field1=' + x);
});
```

# Serverless lambda functions

**Serverless** lambda functions are cloud hosted handlers.

Conceptually, a server is started for each Web request.

No resources are consumed between Web requests.

This execution model works well for glue code.

E.g. **AWS Lambda**, **Azure Functions** or **Zeit Now**.

# Serverless Node.js with Zeit Now

**Zeit Now** provides hosting for serverless functions.

On you MacOS, Windows or Linux computer:

- Install *now* with `$ sudo npm install -g now`
- Get examples `$ git clone https://github.com/zeit/now-examples`
- Open example `$ cd /now-examples/nodejs`
- Deploy with `$ now`

# Serverless Node.js glue code

.js

Creating a Web service in Node.js ...

```
let server = http.createServer((req, res) => {  
  res.end("200 OK");  
});
```

... becomes exporting a handler in Zeit Now Node.js:

```
module.exports = (req, res) => {  
  res.end("200 OK");  
};
```

# Hands-on, 15': Glue code

Configure the [TTN to ThingSpeak adapter](#) glue code.

Create a free account and host the code on [Zeit Now](#).

Use [curl](#) to simulate calls from the TTN backend:

```
$ curl -v http://127.0.0.1:8080/ --data  
'{"app_id":"fhnw-iot","dev_id":"fhnw-iot-arduino-1","payload_raw":"FwAqAA=="}' # Base64
```

Replace 127.0.0.1:8080 with your Zeit Now URL.



# Self-hosted dashboard

A self-hosted dashboard backend includes:

- A way to run services (and keep them running)
- A service with an API to store or cache data
- A service serving dashboard resources

The backend can be hosted locally or in the cloud, storage or cache can be a database, broker or both.

# Docker

**Docker** provides OS-level virtualisation/containers.

Use it to run services on **Windows**, **MacOS** and **Linux**.

On the **Raspberry Pi**\*, try installing Docker with

```
$ sudo curl -sSL https://get.docker.com | sh
```

```
$ sudo apt-get install docker-ce=18.06.1
```

```
~ce~3-0~raspbian # see this issue
```

\*) Many containers don't work on a Zero / ARMv6. 26

# InfluxDB

**InfluxDB** is an open source time-series database.

- To run **InfluxDB on Docker**, type:  

```
$ docker run --name influxdb -p 8086:8086  
quay.io/influxdb/influxdb:2.0.0-alpha
```
- To set it up, open **the Web UI** or a new terminal:  

```
$ docker exec -it influxdb /bin/bash  
$ influx setup
```

# Getting data into InfluxDB

InfluxDB has a number of mechanisms to get data in:

- **Telegraf**, a data collection agent, supports **MQTT**.
- **Data formats** include **InfluxDB**, **CSV** and **JSON**.
- **InfluxDB scrapers** can collect data from any HTTP endpoint using the **Prometheus data format**
- Additional InfluxDB **integrations** include **Kafka**.

It was built for operations monitoring/metrics.

# Telegraf

**Telegraf** is a data collection agent with many plugins.

- To run **Telegraf on Docker**, type:  

```
$ docker run --net=container:influxdb  
telegraf
```
- To enable the MQTT plugin, use a *telegraf.conf*:  

```
$ docker exec -it telegraf /bin/bash  
$ nano telegraf.conf #see next page
```

# Telegraf MQTT input

.conf

The *telegraf.conf* for an MQTT to InfluxDB adapter:

```
[[inputs.mqtt_consumer]]
    servers = ["ssl://MQTT_HOST_OR_IP:8883"]
    topics = ["TOPIC/SUBTOPIC"]
    data_format = "json" ...

[[outputs.influxdb]]
    urls = ["http://INFLUXDB_HOST_OR_IP:8086"]
    database = "telegraf"
```

# Grafana

**Grafana** is an open source Web dashboard backend.

It integrates nicely with InfluxDB and other sources.

- To run **Grafana on Docker**, type:  
`$ docker run -d -p 3000:3000 grafana/grafana`
- Then configure it to connect to InfluxDB:  
`http://127.0.0.1:3000/`
- And create some graph views to display data.

LoRa WAN demo project by



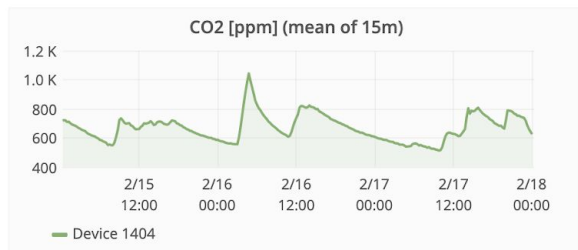
[CO2, temperature, and humidity sensor](#)



Node 1404 (CO2)



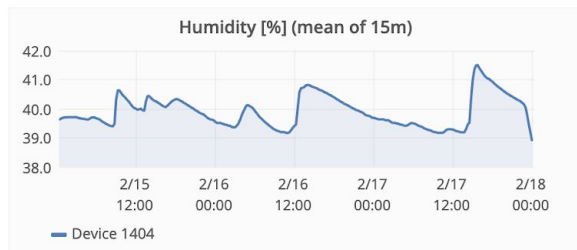
LoRa infrastructure provided by



Current

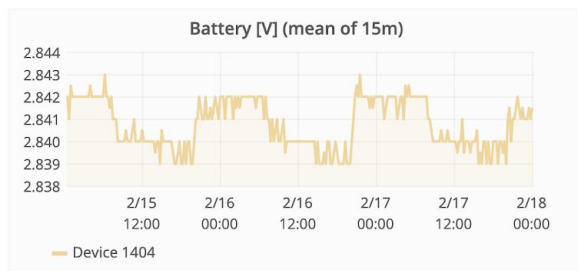
625.5

ppm



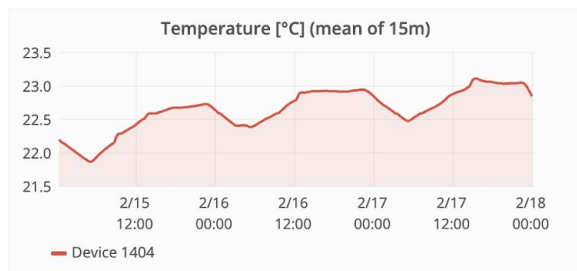
Current

38.9 %



Current

2.84 V



Current

22.84 °C



# Hands-on, 15': Docker hosted dashboard

Install Docker on your computer (not Raspberry Pi).

Run InfluxDB and run/create a Grafana dashboard.

Run Telegraf to get data from [test.mosquitto.org](https://test.mosquitto.org).

Send data "as a device" with [mqtt](#), to Mosquitto.

How does the reference model of this setup look?

# Graph libraries

For custom dashboards, graph libraries are available:

- [Plotly.js](#) is quite easy to [get started](#) in Node.js
- [CanvasJS](#) has a big collection of [React Charts](#)
- [Google Charts](#) has been around for a while

There are many other libraries, make sure to check (long term) availability and source code license.

# Summary

We created Web dashboards & apps for sensor data.

We saw how data gets to a hosted dashboard service.

We wrote adapter glue code for backend integration.

We set up a self hosted dashboard with Docker.

Next: Rule Based Integration of IoT Devices.

# Feedback?

Find me on <https://fhnw-iot.slack.com/>

Or email [thomas.amberg@fhnw.ch](mailto:thomas.amberg@fhnw.ch)

Slides, code & hands-on: [tmb.gr/iot-9](http://tmb.gr/iot-9)

