Project Documentation

Varvara, Sommese, Ucchesu
2018

July 9, 2018

Contents

1 FEZ Spider II

1.1 Hardware Configuration

Our project represents a prototype for an IoT module that is able to detect fire in a close environment. At its core, there is a FEZ Spider II motherboard on which are attached the following components:

- USB DP (power and programming FEZ)
- 3 Breakout Modules (2 for analog inputs and 1 for pwm)
- Ethernet Module (Internet connectivity)
- SD Card (data storage)
- Display (debug information)
- Wifi Module (geolocation).

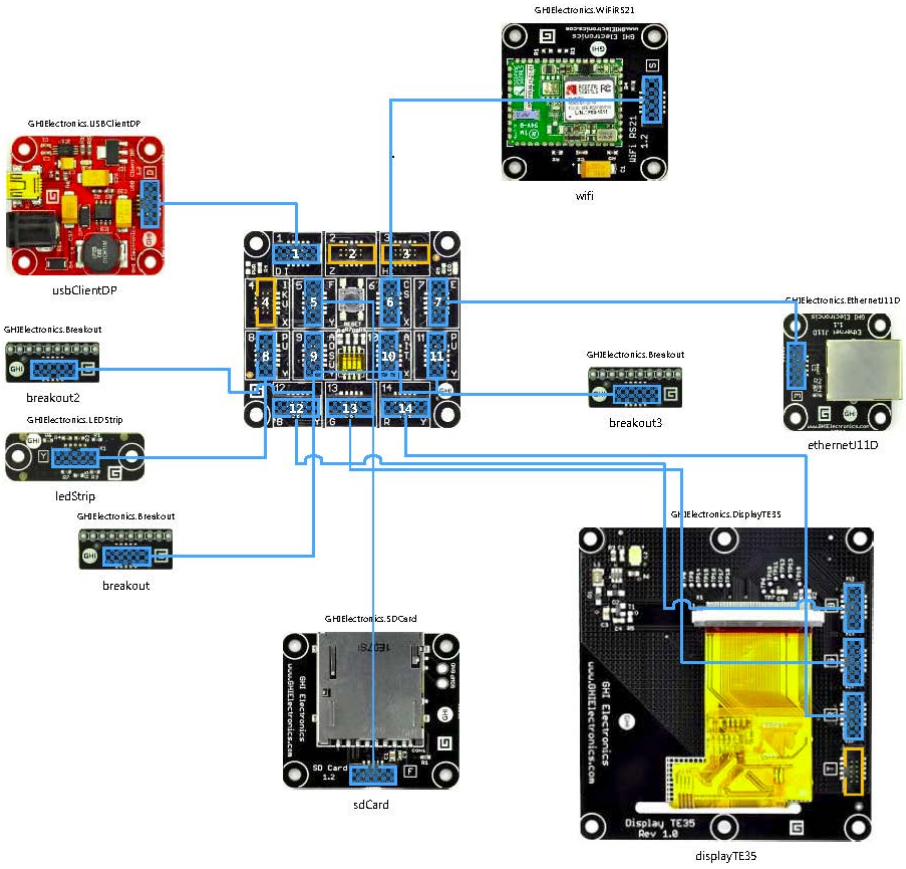

Figure 1: Fez Interconnection

The four sensors are attached to the various breakout, and cover the most important data sources for a fire detection module. In detail, the sensors are:

- the flame sensor
- the LM35 temperature sensor
- the CO sensor (MQ7)
- the smoke sensor (MQ2)

All the sensors are welded on a veroboard. Our solution also includes the TP-Link router TL-WR703N that is connected with the board through an ethernet cable. The use of this external router allowed us to have a secure and direct connection with the AWS servers (further details are provided in a later chapter). The router is also powered by the same power supply that charge the smoke and the CO sensor.

Electrical Sensor Schema

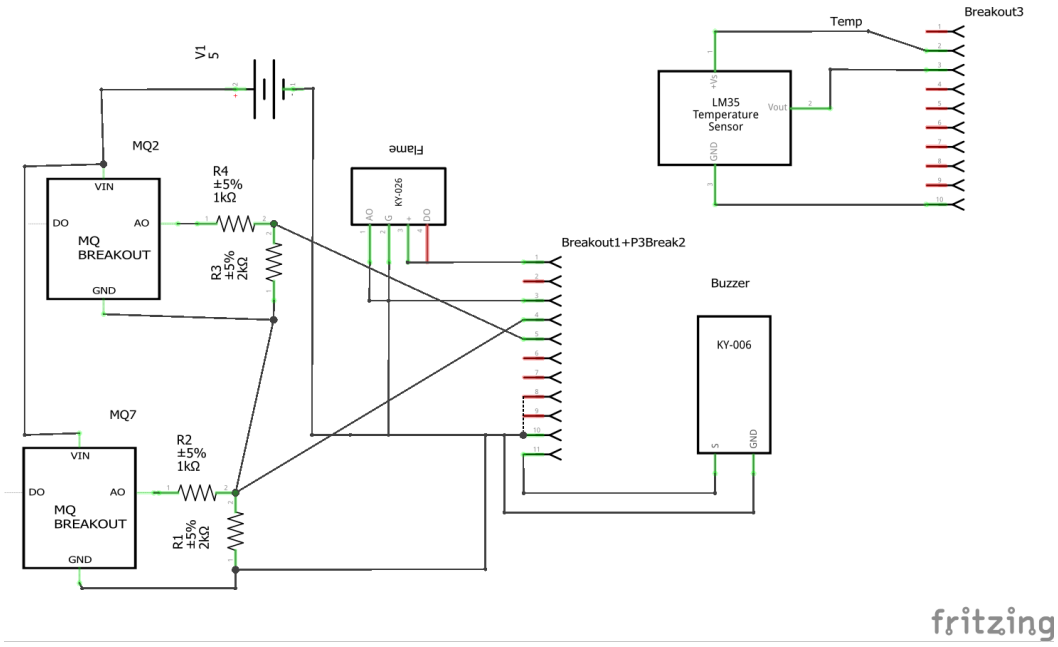

Figure 2: Electrical Schema

Figure 2 shows the electrical schema of the sensors. The first pin from the top on the central breakout is 3.3V from Fez Breakout 1. The penultimate pin is GND from Fez Breakout 1. The last pin is an interconnection with Pin 3 from Breakout 2. The Breakout 3 is directly interconnected with LM35 sensor. The **MQ2** and the **MQ7** sensors are powered by an external 5.0V source (provided by an LM2596S DC-DC converter) and are interconnected with FEZ Spider Board through a voltage divider (with 2 resistors of 1000 Ω and 2000 Ω), that brings the output voltage of the sensors from 5,0V max to 3,3V max. Since the last two sensors need some time to warm up, their initial measures could be affected by a systematic error. The buzzer and the flame sensor are directly interconnected with the board. The temperature sensor is interconnected with another breakout, and is powered with 5.0V, but since it can reach only the voltage of 1.5V, it is useless to create a voltage divider.

1.2 Embedded Software

The program on the board is loaded at startup. During the initialisation, after the Ethernet connection is checked to be in place, the program is blocked until an internet connection is established. When an internet connection is available, an MQTT client object is created and the system time is synchronised with the internet. From the FEZ point of view, the MQTT client endpoint is hard-coded to be the router itself, which will then provide to create a secure tunnel towards the AWS MQTT endpoint. The communication established in this way is secure (the router takes charge of all the TLS related operations) and ensures that the messages sent through the MQTT client are actually received by the AWS server. The time synchronisation is mandatory during the first initialisation because, otherwise, the measures would refer to past moments as the board has a fixed time at the start-up that is registered to be 01/01/2011. During this phase, the geographic position of the board is established thanks to the Wi-Fi module: through its interface, a scan of all the nearby Wi-Fi network is executed; the array containing the list of all those networks is then sent, in the appropriate JSON format, in a POST request to an Internet API which replies with the estimated position of the device in the form of longitude and latitude. With this information the board fills the appropriate field in its configuration data structure and publishes it to the MQTT topic cfg. After this, the objects that represent the sensors are instantiated and the board begins to take measure from the environment regularly. These measures are then published to the MQTT topic FEZ_24 which represents the id of our board. If the network goes down, the measures are stored in a SD and are published always to the topic previously mentioned, as soon as the Internet connection is established once again. The SD is used as a circular buffer and in case of long network outages, after that the maximum number of file is stored, the new measures overwrite the oldest one. When a measure has been sent through the MQTT client, it is deleted locally, sure that the data is safely stored on the cloud since the MQTT QoS ensures it.
A measure is taken every 3 seconds to be able to act very fast in case of fire. If the measure taken does not introduce a significant change from the last one, it is not sent: in this way the network overhead is reduced, and the database is not filled with information that are not meaningful. After 15 minutes from the last sent message, a new message is sent regardless of the change in the sensors measurements.
Every MQTT message received on the cloud is forwarded to an online AI that checks if the current state of the sensors represents a fire. In case of affirmative response, a MQTT message is published on a topic on which our board is subscribed. This message triggers the buzzer on the board to return an acoustic feedback. This aspect could be further improved if, on the same topic, various sensors or even station are subscribed. Through this feedback, they can be informed of a fire in different positions and assume an adequate level of warning, facilitating a more rapid response. Through the same MQTT topic is also received a message that announces that the current state does not represent a situation of fire anymore, thus stopping the buzzer from playing.
The live state (i.e. the last measures) of the sensors present on our board can be examined at `http://aws.r4ffy.info/ui`, while a more aggregate data set can be found at `http://aws.r4ffy.info/graph/prod`. Here it can be found the complete set of IoT modules that have registered to the MQTT topic cfg, all inserted in a map widget to observe their current

position. For each module, a query can be executed to retrieve from the AWS database all the measure for a specific sensor on that module, narrowing the search also by a specific temporal interval. The data are plotted in a line graph to better examine its evolution.

2 Esp8266 and IoT button Sensors

2.1 Esp8266

The ESP8266 is a low-cost Wi-Fi microchip with full TCP/IP stack and microcontroller capability produced by Shanghai-based Chinese manufacturer, Espressif Systems[1]. We use the NodeMCU, an open source IoT platform. It includes firmware which runs on the ESP8266 Wi-Fi SoC from Espressif Systems, and hardware which is based on the ESP-12 module[2]. We use a NodeMCU board with Mongoose OS, an Internet of Things (IoT) Firmware Development Framework that is supported by ESP8266 and has the support for MQTT and TLS built-in. We have developed an Humidity and Temperature IoT sensor using:

- NodeMCU-Esp8266
- DHT11 Sensor
- DS3231 RTC Module
- Javascript language

The entire system communicates with AWS cloud directly via Wifi and uses the deep sleep support that brings the power consumption down to 50µA during idle state.

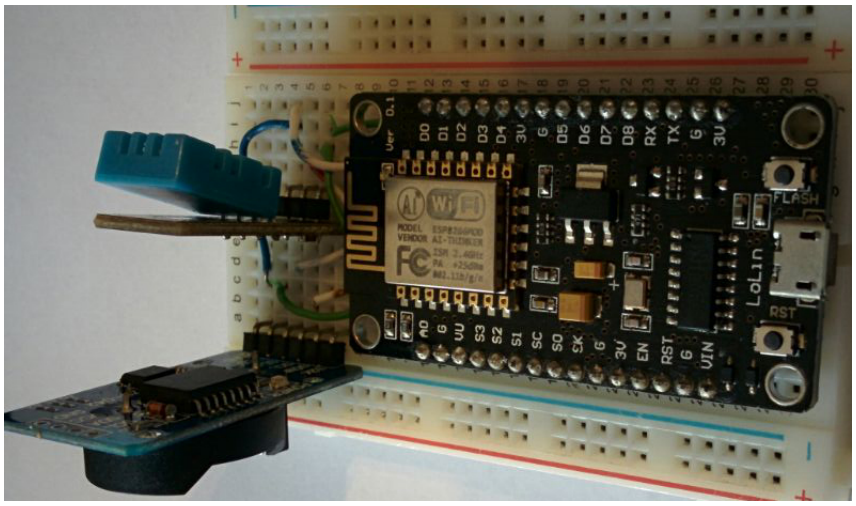

Figure 3: Esp8266

Figure 4: AWS IoT Button

2.2 Iot Button

The AWS IoT Button is a programmable button based on the Amazon Dash Button hardware. This simple Wi-Fi device is easy to configure and designed for developers to get started with AWS IoT Core, AWS Lambda, etc [3]. We use it to create a fire alarm button that is interconnected with our solution with a Lambda Function.

3 Cloud Interconnection

Due to the lack of implementation of TLS 1.2 in the .Net Framework 3.5, we use a TP-Link TL-WR703N for providing an Ethernet to Wi-Fi bridge and for terminate the TLS tunnel. In this way, we assure an End-to-End security model (it is impossible to capture the clear data in transit between FEZ and Router without making a physical man-in-the-middle) and we can use MQTT protocol with AWS Cloud. We install the last version of LEDE firmware

Figure 5: TL-WR703N

on the TL-WR703N (Figure 5) with support of overlay filesystem (on an external pendrive), due to limited flash size of the router.

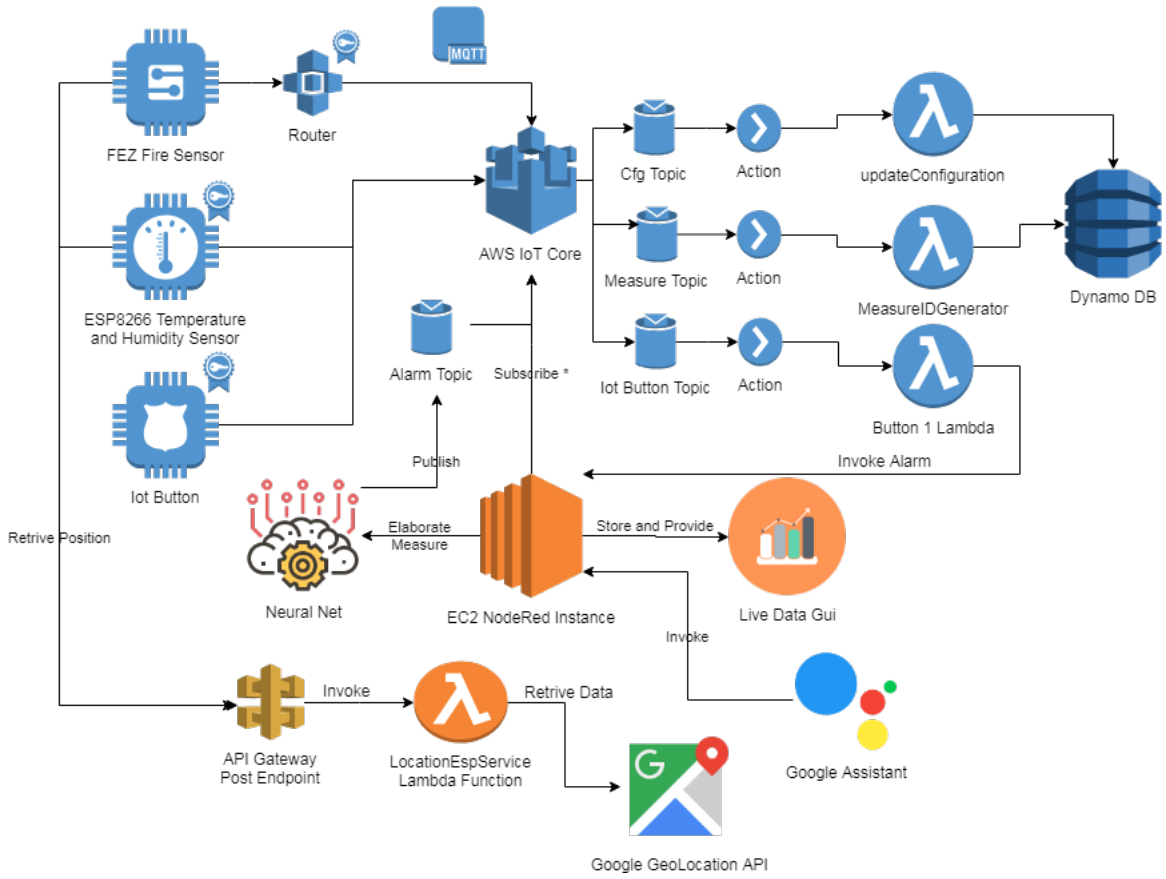

Figure 6: Cloud Architecture

3.1 Stunnel

The TLS termination is provided by Stunnel installed inside LEDE router as a package. Stunnel is a proxy designed to add TLS encryption functionality to existing clients and servers without any changes in the programs' code. This architecture is optimized for security, portability, and scalability (including load-balancing), making it suitable for large deployments. [4] We configure the TLS termination with client auth and with SSL certificate provided by AWS cloud. We expose the Plain-Text port 1883 only to the internal lan.

3.2 Lambda Functions

```
1 'use strict';
2 var AWS = require("aws-sdk");
3 var uuid = require('uuid');
4
5 var docClient = new AWS.DynamoDB.DocumentClient();
6
7 exports.handler = (event, context, callback) => {
8     for(var i=0; i<event.measurements.length; i++){
```

```
        var params = {
            TableName:'measure',
            Item:{
                    "_id": uuid.v1(),
                    "device_id": event.device_id,
                    "iso_timestamp": event.measurements[i].iso_timestamp,
                    "sensor" : event.measurements[i].sensor,
                    "status" : event.measurements[i].status,
                    "value"  : event.measurements[i].value,
                    "timestamp" : new Date(event.measurements[i].
                       iso_timestamp).getTime() / 1000,
            }
        };

        docClient.put(params, function(err, data) {
            if (err) {
                console.error("Unable to update item. Error JSON:", JSON.
                   stringify(err, null, 2));
            } else {
                console.log("PutItem succeeded:", JSON.stringify(data,
                   null, 2));
            }
        });
    }
};
```

Listing 1: Insert Measurement Lambda Function

```
'use strict';
var AWS = require("aws-sdk");
var docClient = new AWS.DynamoDB.DocumentClient();

exports.handler = (event, context, callback) => {
    var params = {
        TableName:'configuration',
        Key:{
            "id": event.id,
        },
        UpdateExpression: "set description=:description, #group=:group,
           internal=:internal, latitude=:latitude, longitude=:longitude, #
           location=:location, #name=:name, sensors=:sensors, #type=:type,
            version=:version",
        ExpressionAttributeNames:{
            "#group" : 'group',
            "#name" : 'name',
            "#type" : 'type',
            '#location' : 'location',
        },
        ExpressionAttributeValues:{
            ":description" : event.description,
            ":group" : event.group,
            ":internal" : event.internal,
            ":latitude" : event.latitude,
            ":longitude" : event.longitude,
```

```
            ":name" : event.name,
            ":sensors" : event.sensors,
            ":type" : event.type,
            ":version" : event.version,
            ":location" : event.location,
        },
        ReturnValues:"UPDATED_NEW"
    };
    docClient.update(params, function(err, data) {
    if (err) {
        console.error("Unable to update item. Error JSON:", JSON.stringify
            (err, null, 2));
    } else {
        console.log("UpdateItem succeeded:", JSON.stringify(data, null, 2)
            );
    }
    });
};
```

Listing 2: Update Configuration Lambda Function

The previous listing are the lambda functions used in our solution that deal with storing data: the first is for denormalizing the measurements that translates every message received in a set of rows to be inserted in the database, one for each measure in the message; the second lambda function updates the configuration in the database. The measurements needs to be denormalized because, otherwise, based on how it is modelled the JSON input, the database will store all the measures in one single column as a JSON object, making then difficult to execute queries efficiently. In fact, in this scenario DynamoDB will return a JSON that has as nested object the different measures, hence the queries can't exploit the database capabilities for filtering the time intervals and/or the specific sensor. The filter could be implemented in the lambda, however this would create scalability and performance issues. Our solution avoids these problems and proposes a different approach, slightly increasing the operations done when the data are stored but then leaving to DynamoDB all the data filtering.

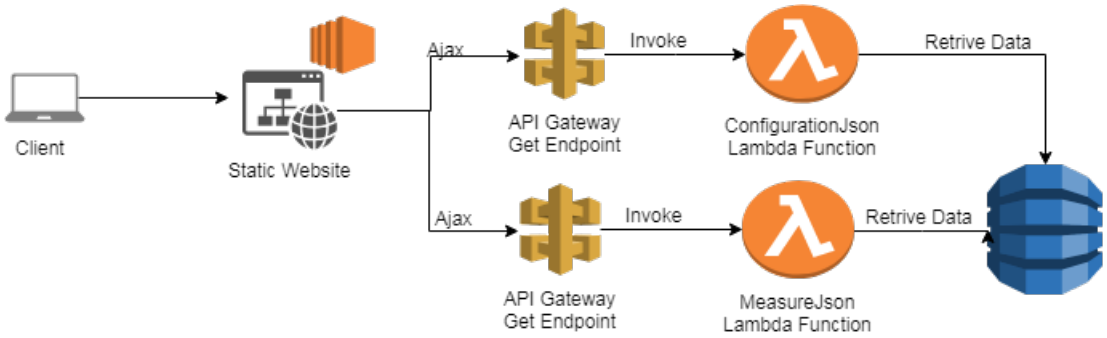

Figure 7: Cloud Architecture 2

For retrieving the data from DynamoDB the following two lambdas are used.

```
'use strict';
```

```
2
3  console.log('Loading function');
4
5  const doc = require('dynamodb-doc');
6
7  const dynamo = new doc.DynamoDB();
8
9  exports.handler = (event, context, callback) => {
10
11      const done = (err, res) =>{
12          callback(null, {
13              statusCode: err ? '400' : '200',
14              body: err ? err.message : JSON.stringify(res.Items),
15                  headers: {
16                      'Content-Type': 'application/json',
17                      'Access-Control-Allow-Origin': '*',
18                  },
19          });
20      }
21
22      switch (event.httpMethod) {
23          case 'GET':{
24              var params = {
25                  ExpressionAttributeNames: {
26                      "#device_id": "device_id",
27                      "#sensor": "sensor",
28                      "#timestamp": "timestamp",
29                      "#status" : "status"
30                  },
31                  ExpressionAttributeValues: {
32                      ":device_id": event.queryStringParameters.id,
33                      ":sensor": parseInt(event.queryStringParameters.sensor
                            ),
34                      ":sdate" : parseInt(event.queryStringParameters.start)
                            ,
35                      ":edate" : parseInt(event.queryStringParameters.end),
36                      ":status" : "OK",
37                  },
38                  FilterExpression: "#status=:status AND #device_id = :
                        device_id AND #sensor=:sensor AND #timestamp BETWEEN :
                        sdate AND :edate",
39                  TableName: "measure"
40              };
41              dynamo.scan(params, done);
42              break;
43          }
44          default:
45              done(new Error(`Unsupported method "\${event.httpMethod}"`));
46      }
47 };
```

Listing 3: Retrieve Filtered Measurement Lambda Function

```
1  'use strict';
```

```
2
3  console.log('Loading function');
4
5  const doc = require('dynamodb-doc');
6
7  const dynamo = new doc.DynamoDB();
8
9  exports.handler = (event, context, callback) => {
10     //console.log('Received event:', JSON.stringify(event, null, 2));
11
12     const done = (err, res) => callback(null, {
13         statusCode: err ? '400' : '200',
14         body: err ? err.message : JSON.stringify(res.Items),
15         headers: {
16             'Content-Type': 'application/json',
17             'Access-Control-Allow-Origin': '*',
18         },
19     });
20     dynamo.scan({ TableName: 'configuration' }, done);
21 };
```

Listing 4: Retrieve Configurations Lambda Function

The first lambda retrieves the measurements filtered by a specific time interval and by a specific sensor, received as query parameters of the HTTP request; using this information, it issues a command that performs a query in the database and returns the set of measurements. The second lambda retrieves all the configurations stored in the database, allowing to display them all in the same map in a web UI, based on their geographical position.

4 Cloud Software

4.1 Website Data Showcase

The live measures of the sensors on our board can be examined at `http://aws.r4ffy.info/ui`. Here it is possible to switch between FEZ_24 and ESP8266 24 to show the respective sensors and their values. When a fire is detected a voice alarm reports it.

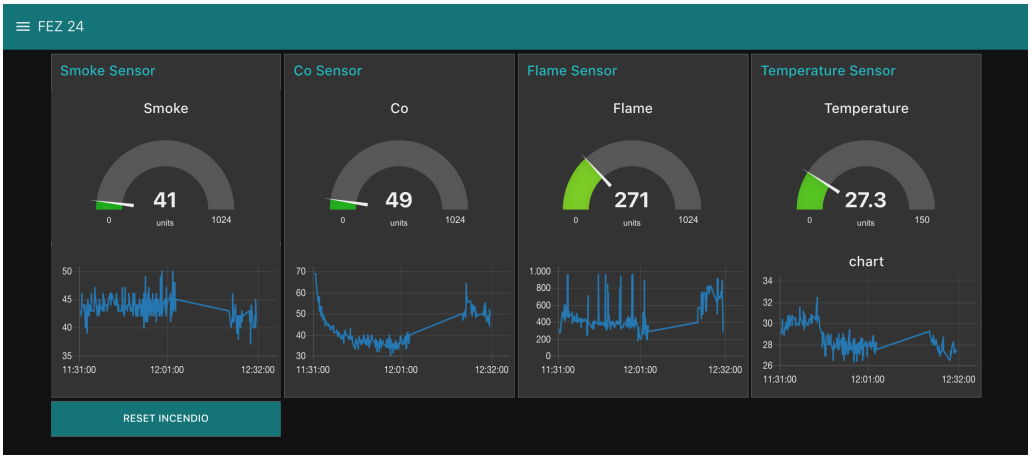

Figure 8: FEZ_24 live sensors measures

At `http://aws.r4ffy.info/graph/prod` it can be found the complete set of IoT modules that have registered to the MQTT topic "cfg". The map widget offered by Google is used to show the current position of the modules. The configuration data retrieved from the database allows to dynamically list all the sensors present on the various IoT modules, giving the possibility to access, and then plot, the history of a specific sensor on a specific module. In fact, the web UI can execute an HTTP request that sets up a query that is executed to retrieve all the measures from the AWS database, filtered by the specific sensor and by the specified temporal interval. The data are plotted in a line graph to better examine its evolution.

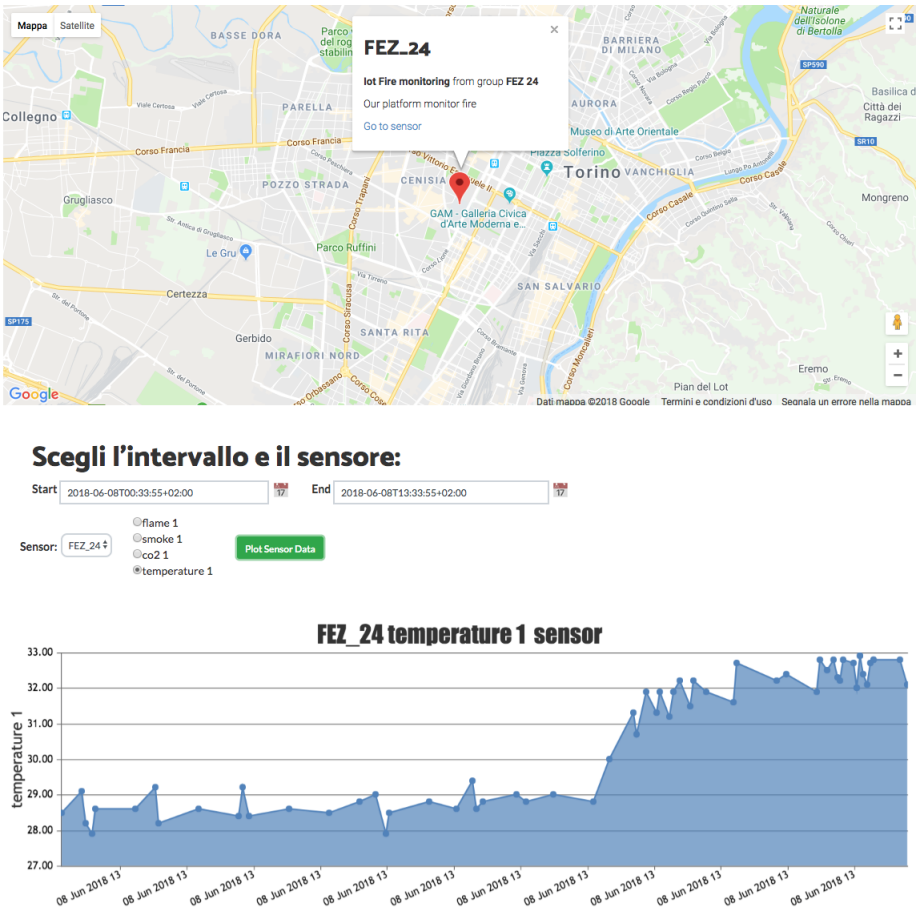

Figure 9: FEZ_24 Graph

4.2 NodeRED

Node-RED is a flow-based development tool developed originally by IBM for wiring together hardware devices, APIs and online services as part of the Internet of Things. Node-RED provides a browser-based flow editor, which can be used to create JavaScript functions. Elements of applications can be saved or shared for re-use. The runtime is built on Node.js. The flows created in Node-RED are stored using JSON. Since version 0.14 MQTT nodes can make properly configured TLS connections.

We used NodeRED as an Internet API to obtain the geolocation of the board.

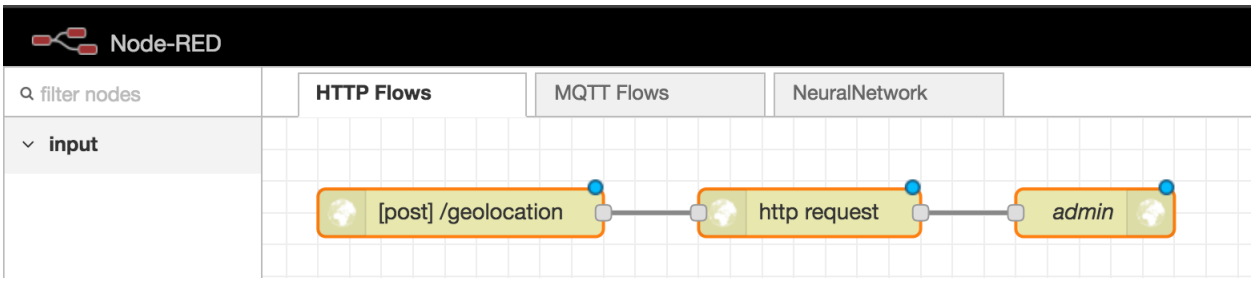

Figure 10: NodeRED geolocation

NodeRED offers also a tool to build a neural network that we used to detect the presence of a fire.

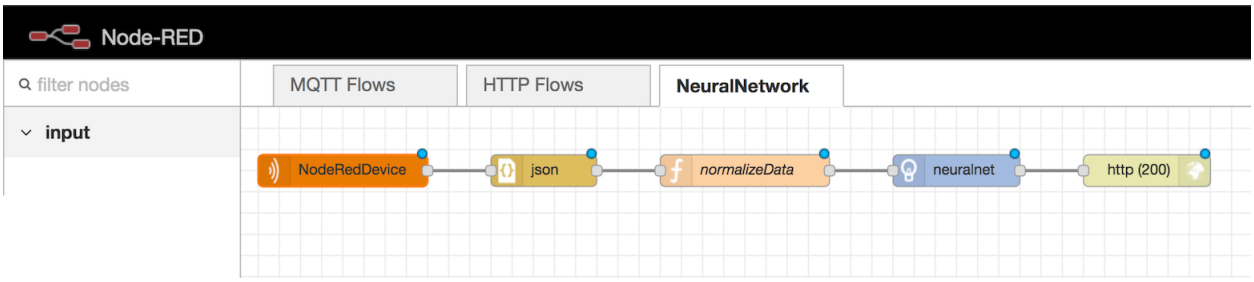

Figure 11: NodeRED Neural Network

With a similar flow, we used NodeRED to process live data and show it at `http://aws.r4ffy.info/ui`.

4.3 Neural Network

The FEZ Spider board takes the environmental measure from its sensors, it encloses them in a JSON message and it publishes it on the MQTT topic FEZ_24. Every message is then processed by an AI running on the NodeRED platform that tells if the values read by the sensors can be associated with a fire. The AI is a feedforward neural network and has 5 input neurons and 1 output neuron. In the 5 inputs, the first 4 map the values of the 4 sensors present on the FEZ Spider board, while the last one is a short-term memory neuron that is equal to the last value returned in output. The neural network has been trained with a limited train set that has been extended with every false positive and false negative encountered during the testing phase. Being a fairly simple scenario that can also be modelled with a cascade of if-then-else, even with only a dozen of input cases, the value returned by the neural network responds correctly to a controlled artificial fire scenario (e.g. a lighter).

4.4 Interconnection with Google Assistant

Google Assistant is Google's personal assistant that provides the voice recognition and natural language processing behind Google's Android and Home devices. Google Actions allow

you to build conversational agents that interact with the user using a query-response conversation style. We use a plugin for node-red to interconnect Google Assistant with our platform. It is possible to ask to Google Assistant for:

- Temperature and humidity
- Fire board Sensor
- Fire Detection

It also gives the possibility to the user to activate the fire alarm.

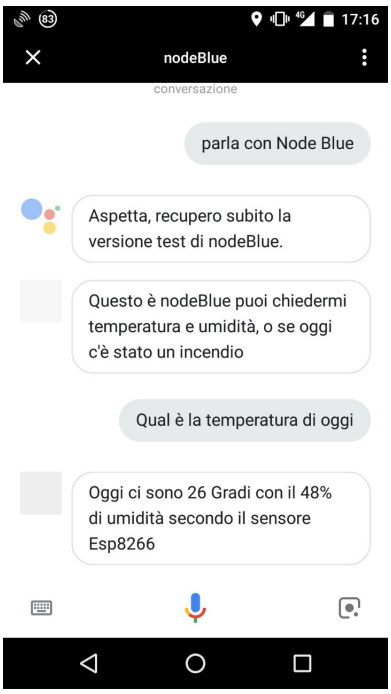

Figure 12: Google Assistant

References

[1] "Esp8266." `https://en.wikipedia.org/wiki/ESP8266`.

[2] "Nodemcu." `https://en.wikipedia.org/wiki/NodeMCU`.

[3] "Aws iot button." `https://aws.amazon.com/iotbutton/`.

[4] "Stunnel." `https://www.stunnel.org/`.