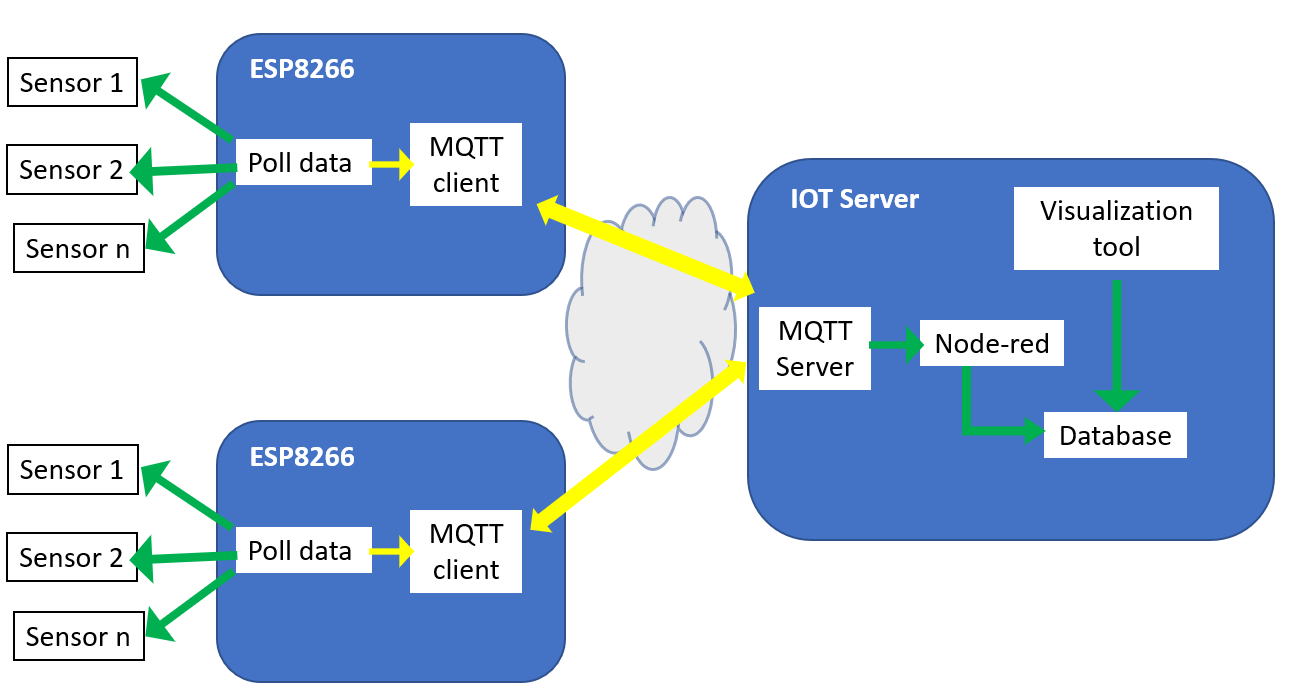
# Introduction

The goal was to set up a weather station without using IOT services from any cloud provider. Besides the learning challenge, it provides independency from internet communications and IT providers.

We will use a Raspberry as the IOT Server and ESP8266 with their correspondent gauges as clients. As you separate clients and server, you can have as many clients as you want, measuring data in different places, or polling for different data (e.g. a rain gauge only makes sense outside ;-) ) depending on where the client is installed.

MQTT is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol. It was designed as an extremely lightweight publish/subscribe messaging transport. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium. For example, it has been used in sensors communicating to a broker via satellite link, over occasional dial-up connections with healthcare providers, and in a range of home.



MQTT is a very convenient protocol to implement IOT; it's simple, reliable and light. All (or almost all) IOT cloud providers implement it and provide an MQTT server. In this case, I am going to be provider independent and connect to a local MQTT server, but the agent will be the same, with only minor changes in server address and authentication data, as userID and password.

The ESP12 (aka NodeMCU ESP8266) is a very cheap circuit that can be found for around 2€. It is an Arduino-like circuit with integrated WIFI and a USB port. There are some versions of the circuit. Even though V3 has the higher version number and hence it is supposed to be the most advanced, I would try to avoid them versions, they use a bigger footprint and have no clear advantages. I rather prefer v2; I buy them in AliExpress at a very low price and, so far, I have not had any problem. Should you want more info about these chips, there is lot of info in internet, e.g. <https://frightanic.com/iot/comparison-of-esp8266-nodemcu-development-boards/>

I program ESP8266 chips using Arduino IDE. The language used is C/C++ with some small limitations, but powerful enough. Language reference can be found at Arduino official page <https://www.arduino.cc/reference/en/> , but there is also plenty of information, forums and ideas in other popular pages.

Hence, you have a chip with multiple pins to connect sensors (Digital, analog, I2C, SPI, MISO ...), programable in C, with USB interface and WIFI in a 25mmx48mm (1"x1.9") circuit. This means that it's possible to connect multiple sensors of any kind (I have to say that I had some problems trying to use two UART interfaces simultaneously), to control them and to communicate with the server with WIFI. As there are libraries for almost every sensor and lots of them for networking and subsystems -e.g. not only TCP/IP or MQTT (see below) but also HTTP, (Web)sockets, SSL and many more- it is incredible easy to build complex systems.

This project implements sensors for a weather station, but once everything is set up, it is easy to evolve to other projects with different types of sensors.

So, let us begin.

First things first. We must install and configure Arduino IDE to edit, compile and upload programs to our ESP8266 device.

## Arduino IDE installation

### Install the Arduino IDE

Download form from <https://www.arduino.cc/en/Main/Software> , install the IDE and the drivers that come with it.

LINUX: If you are using Linux and prefer using built-in package manager (e.g. aptitude), check whether your package manager has the latest version available.

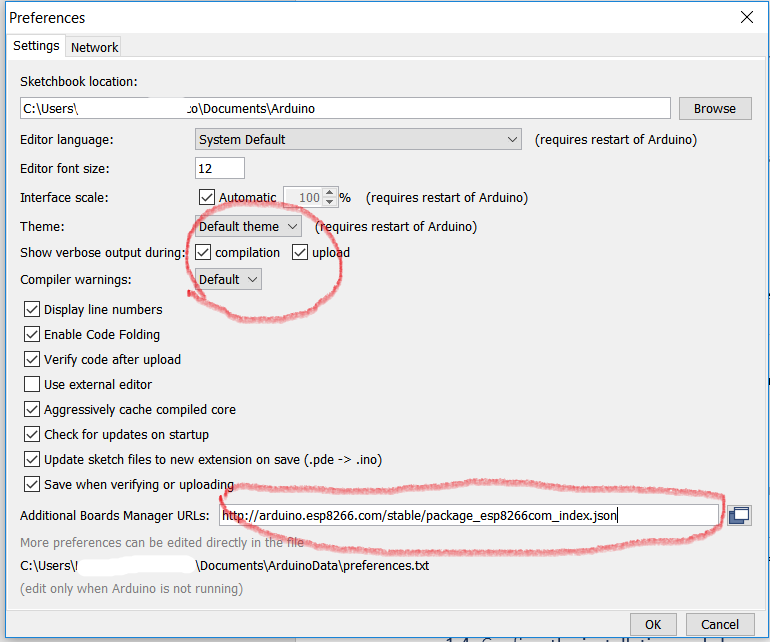
MAC: Install also these drivers <https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers>

### Add ESP8266 board to Arduino IDE, by adding the board manager link to the preferences

Start Arduino IDE and open “Preferences” menu in File -> Preferences

Check the boxes in the preferences according to the picture below for a better development interface.

Also, add the link <http://arduino.esp8266.com/stable/package_esp8266com_index.json> to “Additional Boards Manager URLs” and save settings by pressing ok.



Restart Arduino IDE so it downloads board libraries upon starting.

### Install the new board manager for ESP8266 chip

In the Arduino IDE main screen select Tools -> Board (…) -> Boards manager…

Search for “ESP8266”, then click the suggested result and press install

### Confirm the installation and choose the NodeMCU board for the current board

In Arduino main screen select Tools -> Board (…) and then select “NodeMCU 1.0 (ESP 12-E module)” from the list

### We are done!

If you are using Linux, ensure that your user have permissions to use the ports. Instructions are available at [www.arduino.cc](file:///C:\Users\LUISMorrasRuizFalco\AppData\Roaming\Microsoft\Word\www.arduino.cc).

Windows (also in MAC?): If the chip is not seen by Arduino IDE in “Tools ->port”, or there are communication problems, check that you have proper USB-to-Serial connection drivers installed in your system. We can find your chip’s Serial type from the bottom of the chip. In most cases, it should be either CP210x or CH340(G). Both drivers are found in their manufacturers page, easy to find and download.

Now, we can begin to prepare our meteo gauge.

## Sensors

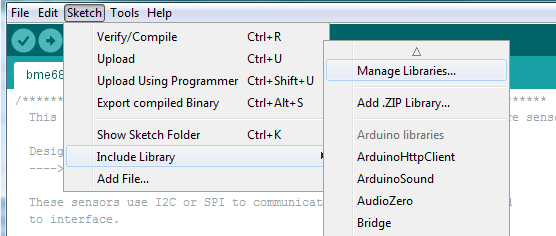
We will set up gauges for temperature, air humidity, air pressure and rain. Even though it is not a meteorological measure, we will also connect a soil moisture sensor.

### Temperature, air humidity and atmospheric pressure.

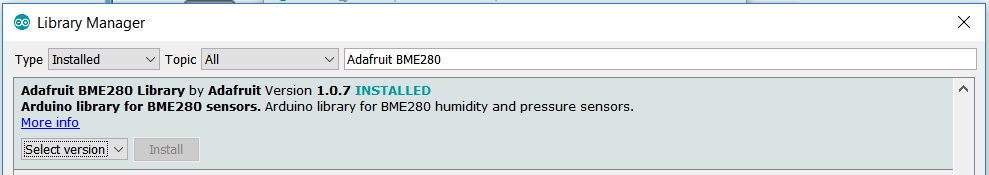
There are sensors to read each of the different magnitudes, but, suing a single sensor for the three makes everything easier: single wiring and only one library needed. That is why I prefer to use BME280.

 There are many circuits prepared for this sensor built by Adafruit, Sparkfun or Chinese manufacturers. All of them are very cheap (about 2€) and quite similar. The chip we are using here provides an I2C interface. Its address is documented as 0x77 in many places but, after some headaches and reviewing cabling, soldering ...etc., I found that in the ones I bought it is 0x76. To read sensor data, we will also need a library, I use to [the Adafruit\_BME280 library (code on github)](https://github.com/adafruit/Adafruit_BME280_Library). It is also available from the Arduino library manager and is the method I recommend, so it will notify when there are upgrades.

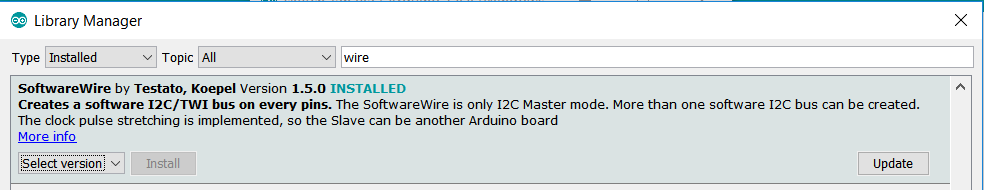
From the Arduino IDE open up the library manager...

[](https://learn.adafruit.com/assets/48076)

And type in **adafruit bme280** to locate the library. Click **Install**



If not installed, you will also need to install the wire [library](https://www.arduino.cc/en/reference/wire), that is also available in Arduino Library Manager. So look for ‘wire’ an install the library.



### Soil Moisture

Thera are also cheap soil moisture gauges but, there are not especially accurate. Cheap models (below 3€) are of two types:

* resistive where the sensor measures the electric resistance of the soil
* capacitive, where the sensor measures the electric capacity of the soil surrounding the sensor

Both are subject to corrosion, much more in the case of the resistive ones. Both type of sensors give a value depending on how moist is the soil, but there are many other variables that impact the measure (Ph, compaction, composition …). Even though the read value depends on the type of soil, it is relative to moisture, so we will need to take some measures- from completely dry to soaked – in order to know what the scale limits for that particular soil are.

To limit corrosion on the sensor, we will only power it when needed so there only will be tension when we want to read a value. What we will do is putting a transistor (a 2N222 is a suitable one) that will connect and disconnect power using the transistor base (see circuit and code in following lines).

### Rain gauge

The rain gauge I use is another sensor that does not need special libraries, only analog and interrupt counter for the rain gauge. Interrupt handling is a bit trickier (we will cover it below).

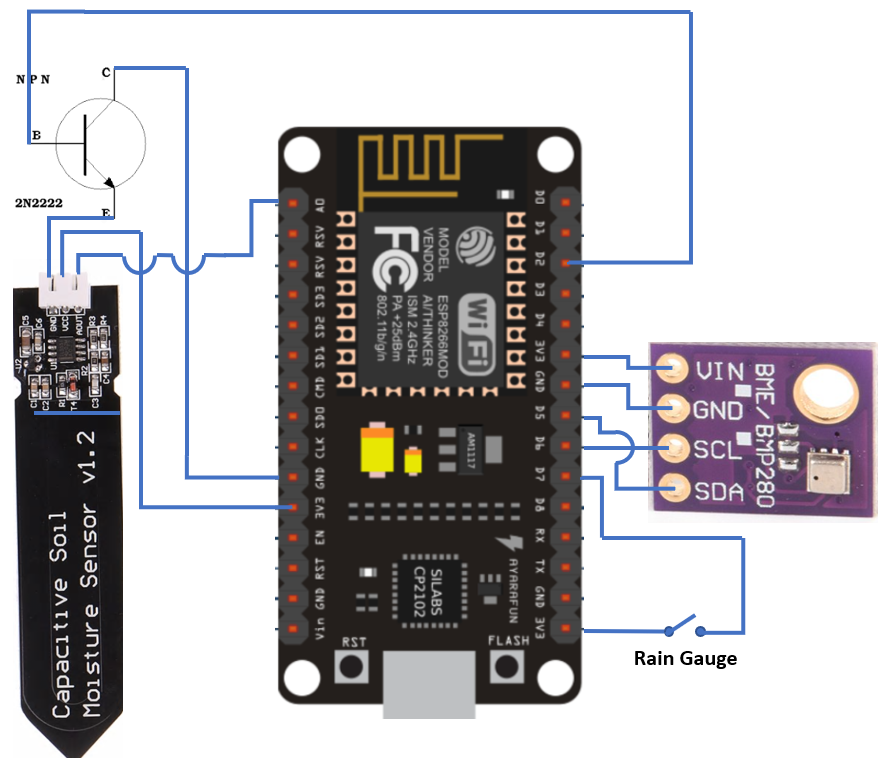
<https://www.amazon.co.uk/MISOL-Spare-weather-station-measure/dp/B00QDMBXUA>

[](https://pi.gate.ac.uk/images/environment/rain/tippingbucket.jpg)To complete the sensors used (I did not put an anemometer, because at my home is senseless, but its mechanism is very similar to this one), we will set up a rain gauge. To use an automatic rain gauge, one common solution is a [tipping bucket rain gauge](http://en.wikipedia.org/wiki/Tipping_bucket_rain_gauge#Tipping_bucket_rain_gauge). It consists in a little seesaw with two buckets. the rain fills up a bucket on one end, so it tips over (and empties) and the bucket on the other side starts to fill. Each time the bucket tips, it makes a momentary electrical connection, using a reed switch or a hall switch. The buckets are calibrated to a volume of water, which means if you count how many times the switch closes, multiply it by the bucket volume and divide by the area of the funnel that collects the water, you will know how much rainfall there has been.

In Amazon you can find rain gauges at very affordable prices. I used <https://www.amazon.co.uk/MISOL-Spare-weather-station-measure/dp/B00QDMBXUA> but surely it is sold in other Web pages. This one is calibrated so every tip represents 0.2794 mm (to translate to inches, 0.011”, obtained by dividing this number by 2.54), it has a RJ11 plug at the end of the wire, you will have to cut it to connect to your ESP8266.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Interrupciones \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### Circuit. Connecting the components



The connections of the three sensors are independent, so you can only use the ones you need. In the program, it is easy to remove those that you are not using.

## Arduino program. Sensor Module

Now, you have the environment prepared and the chip connected, so it is time to run the program that will read data from the sensors.

First, we will define a macro to expand printing in the Arduino IDE environment in debug mode, so when everything is working you can just turn a variable (CON\_DEBUG) off and make the program lighter.

# define CON\_DEBUG

#ifdef CON\_DEBUG

#define DPRINT(...) Serial.print(\_\_VA\_ARGS\_\_)

#define DPRINTLN(...) Serial.println(\_\_VA\_ARGS\_\_)

#else

#define DPRINT(...) //Blank line

#define DPRINTLN(...) // Blank line

#endif

This macro will expand to Serial.print if CON\_DEBUG is defined, and to nothing if it is not. It is quite convenient because it makes unnecessary to put #ifdef all along the code or to make a call with direct return when DEBUG is off as other coding techniques do.

### BME280

So, let us start with the program. We have to include the needed libraries Wire.h, Adafruit\_Sensor.h and Adafruit\_BME280.h for reading BME280 data; take into account that it is an I2C device and it is a bit more complicated. File PIN\_NodeMCU.h describes the reltionship between PIN number and IO number. Also, I set an altitude pressure correction: PRESSURE\_CORRECTION. My home is at 647 m altitude, to set your correction depending on your sensor altitude, there are many pages that convert barometric pressure depending on altitude (look for ‘barometric altitude’ and you will find many of them), just calculate the corrected/read factor and you have it. Other point that can be frustrating is I2C address; in some documentation it says that the I2C address for the BME280 is 0x77, but all the sensors I bought have 0x76. Try with both addresses before struggling with wires and soldering (as I did ☹). Once BME280 is correctly wired and using the provided libraries, it is quite easy to read data from it.

#include <Pin\_NodeMCU.h> // description of PINs

#define SDA D5 // for BME280 I2C

#define SCL D6 // SCL PIN

#include <Wire.h> // libraries for I2C

#include <Adafruit\_Sensor.h> // libraries for BME280

#include <Adafruit\_BME280.h>

#define PRESSURE\_CORRECTION (1.080) // HPAo/HPHh at 647m

#define BME280\_ADDRESS (0x76) //IMPORTANT, sometimes it is 0x77

Once libraries are included and constants defined, it is quite easy to read data:

float bufTemp,,bufHumedad,,bufPresion; //variables used for reading

bufHumedad= sensorBME280.readHumidity();

bufTemp= sensorBME280.readTemperature();

bufPresion=sensorBME280.readPressure()/100.0F\*PRESSURE\_CORRECTION; //HectoPascals

### Soil moisture

Also, reading the moisture sensor is quite simple, the only thing that must be remembered is that we are using a transistor to power on and off the sensor. So, we need to:

/\* activate soil sensor setting the transistor base \*/

digitalWrite(CONTROL\_HUMEDAD, HIGH); // activate moisture reading

espera(10000); //wait to stabilize (maybe less than 10 sec is enough)

humedadCrudo = analogRead(sensorPin); // and read soil moisture

digitalWrite(CONTROL\_HUMEDAD, LOW); // disconnect soil sensor

### Rain gauge

A bit trickier is reading the pluviometer. To read the number of times the tip has moved, we are going to use interrupts.

First of all, we attach and interrupt to a PIN,

#define interruptPin D7 // PIN where I'll connect the rain gauge

pinMode(interruptPin, INPUT);

Then, let’s define the function that will be called whenever an interrupt happens (i.e. whenever hall sensor detects a change). This function must be of a special type, ICACHE\_RAM\_ATTR, and as simple as possible, because the ESP8266 will only run the interrupt function above anything else.

// Interrupt counter for rain gauge

void ICACHE\_RAM\_ATTR balanceoPluviometro() {

contadorPluvi++;

}

Now, we connect the interrupt function so it is called whenever signal in interruptPIN goes from low to high;

attachInterrupt(digitalPinToInterrupt(interruptPin), balanceoPluviometro, RISING);

So, contadorPluvi is incremented by one each time interruptPIN goes from low to high, i.e. each time the rain gauge tips.

To read rain, we multiply the number of tips by the volume (mm, equal to l/m2 )

lluvia+=contadorPluvi\*L\_POR\_BALANCEO;

//to set counter to zero, we must detach the interrupt (and then, attach it back)

detachInterrupt(digitalPinToInterrupt(interruptPin));

contadorPluvi=0;

attachInterrupt(digitalPinToInterrupt(interruptPin), balanceoPluviometro, RISING);

### Reading data (Putting all together).

Now, it is time to put all together. To read data, I merged everything a a function called tomaDatos() that reads all three sensors. To make things simpler, I use some global variables to store the data read. So, when tomaDatos() is called, five global variables are set. I know that this is not a best practice, but I was lazy about setting a structure, maybe I will correct it in next version. To mitigate random deviations in the data we read, I read twice and calculate the average.

#define SDA D5 // for BME280 I2C

#define SCL D6 // SCL PIN

#define interruptPin D7 // PIN where I'll connect the rain gauge

#define sensorPin A0 // Soil humidity sensor analog PIN

#define CONTROL\_HUMEDAD D2 // Transistor base that switches on and off soil sensor

#define L\_POR\_BALANCEO 0.2794 // liter/m2 (=mm) for every rain gauge interrupt

#include <Wire.h> // libraries for I2C

#include <Adafruit\_Sensor.h> // libraries for BME280

#include <Adafruit\_BME280.h>

#include <Pin\_NodeMCU.h> // description of PINs

#define PRESSURE\_CORRECTION (1.080) // HPAo/HPHh at 647m

#define BME280\_ADDRESS (0x76) //IMPORTANT, sometimes it is 0x77

volatile int contadorPluvi = 0; // must be 'volatile', for counting interrupt. Counts rain gauge tips

// \*\*\*\*\*\*\*\*\* these are the sensor variables that will be exposed \*\*\*\*\*\*\*\*\*\*

Adafruit\_BME280 sensorBME280; // this represents the BME280 sensor

float temperatura,humedadAire,presionHPa,lluvia=0,sensacion=20;

int humedadMin,humedadMax,humedadSuelo,humedadCrudo;

int humedadCrudo1, humedadCrudo2;

// Interrupt counter for rain gauge

void ICACHE\_RAM\_ATTR balanceoPluviometro() {

contadorPluvi++;

}

/\* get data function. Read the sensors and set values in global variables \*/

boolean tomaDatos (){

float bufTemp,bufTemp1,bufHumedad,bufHumedad1,bufPresion,bufPresion1;

boolean escorrecto=true; //return value will be true unless there is a problem

/\* read and then get the mean \*/

bufHumedad= sensorBME280.readHumidity();

bufTemp= sensorBME280.readTemperature();

bufPresion=sensorBME280.readPressure()/100.0F;

/\* activate soil sensor setting the transistor base \*/

digitalWrite(CONTROL\_HUMEDAD, HIGH);

espera(10000);

humedadCrudo = analogRead(sensorPin); // and read soil moisture

humedadCrudo=constrain(humedadCrudo,humedadMin,humedadMax);

digitalWrite(CONTROL\_HUMEDAD, LOW); // disconnect soil sensor

// calculate the moving average of soil humidity of last three values

humedadCrudo=(humedadCrudo1+humedadCrudo2+humedadCrudo)/3;

humedadCrudo2=humedadCrudo1;

humedadCrudo1=humedadCrudo;

// read again from BME280 sensor

bufHumedad1= sensorBME280.readHumidity();

bufTemp1= sensorBME280.readTemperature();

bufPresion1= sensorBME280.readPressure()/100.0F;

DPRINTLN("Data read");

lluvia+=contadorPluvi\*L\_POR\_BALANCEO;

detachInterrupt(digitalPinToInterrupt(interruptPin));

contadorPluvi=0;

attachInterrupt(digitalPinToInterrupt(interruptPin), balanceoPluviometro, RISING);

if (humedadMin==humedadMax) humedadMax+=1;

humedadSuelo = map(humedadCrudo, humedadMin, humedadMax, 0, 100);

/\* if data could not be read for whatever reason, raise a message (in CONDEBUG mode)

Else calculate the mean \*/

if (isnan(bufHumedad) || isnan(bufTemp) || isnan(bufHumedad1) || isnan(bufTemp1) ) {

DPRINTLN("I could not read from BME280msensor!");

escorrecto=false; // flag that BME280 could not read

} else {

temperatura=(bufTemp+bufTemp1)/2;

humedadAire=(bufHumedad+bufHumedad1)/2;

presionHPa=(bufPresion+bufPresion1)/2\*PRESSURE\_CORRECTION;

if (temperatura>60) escorrecto=false; //if temperature out of reasonable range

if ((humedadAire>101)||(humedadAire<0)) escorrecto=false; // or humidity

DPRINT("\tTemperature: \t ") ; DPRINT(temperatura);

DPRINT("\tAir humidity: \t "); DPRINT(humedadAire);

DPRINT("\tPressure HPa : \t "); DPRINT(presionHPa);

DPRINT("\tMoisture: \t ") ; DPRINT(humedadSuelo);

DPRINT("\tRaw Moisture: \t"); DPRINTLN(humedadCrudo);

}

return escorrecto;

}

### Sending data

Once data is read, it must be sent somewhere where it can be stored, manipulated and visualized. The solution selected is to send data to an mqtt broker. This broker can be hosted in a cloud provider (e.g. IBM bluemix, or AWS). In my case, I implemented it in a Raspberry, as described in “**XXXXXXXXXX**”. To send data, we will use the function ‘bool enviaDatos(char \* topic, char \* JsonString)’, defined in another file (see ‘networking module’). This function sends a JSon string.to an MQTT server. This is done with publicaDatos()

/\* this function sends data to MQTT broker \*/

void publicaDatos() {

int k=0;

char signo;

boolean pubresult=true;

while(!tomaDatos()) { // if tomaDatos() returns false, retry 30 times

espera(1000); // waiting 1 sec between iterations

if(k++>30) { // after 30 iterations with no data, return

return;

}

}

// Data is read an stored in global var. Prepare data in JSON mode

if (temperatura<0) { // to avoid probles with sign

signo='-'; // if negative , set '-' character

temperatura\*=-1; // if temp was negative, convert it positive

} else signo=' ';

// format a string to prepare the message

sprintf(datosJson,"[{\"temp\":%c%d.%1d,\"airH\":%d,\"moisture\":%d,\"moitsRaw\":%d,\"HPa\":%d,\"mm\":%d.%1d},{\"deviceId\":\"%s\"}]",

signo,(int)temperatura, (int)(temperatura \* 10.0) % 10,\

(int)humedadAire, (int) humedadSuelo,(int)humedadCrudo,(int)presionHPa,

(int)lluvia, (int)(lluvia \* 10.0) % 10,DEVICE\_ID);

// and publish them.

pubresult = enviaDatos(publishTopic,datosJson);

if (pubresult)

{lluvia=0;} // I sent data was successful, set rain to zero

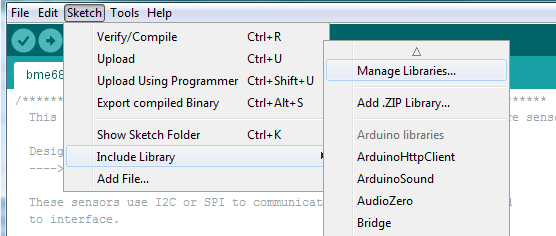
}

To wait between measurements, we will also build a function that keeps other important ESP8266 routines (e.g. those related to WiFi). I wrote espera(Ulong waitMilliseconds), that is a simple function that waits n milliseconds while maintaining internal routines:

## Arduino program. Networking module

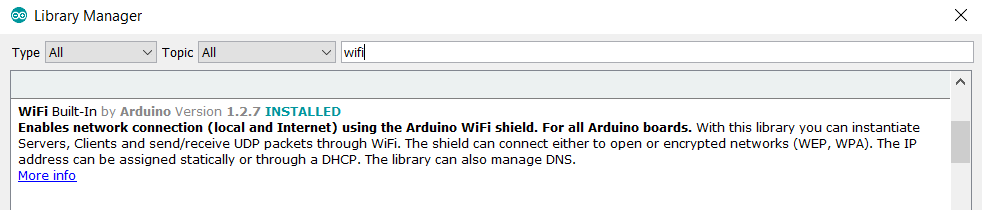
To visualize, manipulate and store data read, I send it to an mqtt broker. I use node-red to manipulate the data received, that is stored in a influxdb database and visualized with grafana. These subsystems can be hosted in a cloud provider (e.g. IBM bluemix, or AWS) or, as I preferred, implemented in a Raspberry, as described later.

We will need to include some libraries. Open, as explained before, library manager

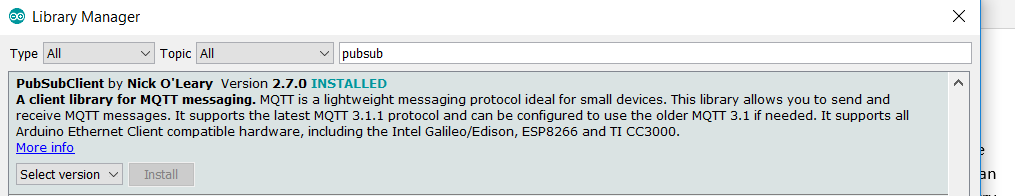
[](https://learn.adafruit.com/assets/48076)

And then look for the desired libraries (you will have to look for them individually)

Install **Wifi built-in** (look for Wifi)

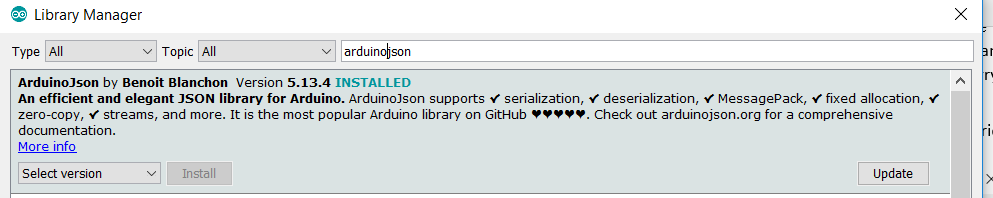


Then **Pubsubclient** (the mqtt client)



And **ArduinoJSON** (for manipulating JSON objects)

**IMPORTANT NOTE:** I am using version 5 of this JSON Library, take into account that V6 is not compatible when this code was developed and tested.



Now, let us begin with the code. First, we must connect the ESP8266 to WiFi. I developed a routine to connect to two SSIDs, one is the main one and the other a spare. When there is only one SSID, I set both with the same name and password and the program maker two iterations. Th routine is the standard one, probably you have seen something similar elsewhere. There are two functions: wifiConnect(), that tries to connect to a WiFi network and sinConectividad(), which is called when the program finds out that connectivity is lost:

For information about MQTT API and behavior, check https://github.com/knolleary/pubsubclient

#include <ESP8266WiFi.h>

#define MQTT\_MAX\_PACKET\_SIZE 455 //must be before line ‘#include pubsubclient.h’

#define MQTT\_KEEP\_ALIVE 60

#include <PubSubClient.h> // https://github.com/knolleary/pubsubclient/releases/tag/v2.3

#include <ArduinoJson.h> // https://github.com/bblanchon/ArduinoJson/releases/tag/v5.0.7

#include <Pin\_NodeMCU.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* -------- Personalized values -------------- \*\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define DEVICE\_TYPE "ESP12E " // will use to build a full qualified device name

#define ORG "Home"

char ssid1[] = "myWifiSSID1";

char password1[] = "password1";

char ssid2[] = "myWifiSSID2";

char password2[] = "password2";

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* ----- End of Personalized values ---------- \*\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

WiFiClient wifiClient;

#define ESPERA\_NOCONEX 70000 // when there is no connection, wait 70sec

boolean wifiConnect() { //This function starts WiFi and connects

int i=0,j=0;

char\* ssid=ssid1;

char\* password=password1;

DPRINT("Connecting to WiFi "); DPRINTLN(ssid);

WiFi.mode(WIFI\_STA); //The 8266 is a station, not an AP

WiFi.disconnect();

WiFi.begin(ssid,password);

while ((WiFi.status() != WL\_CONNECTED )) {

espera(500);

DPRINT(i++);

DPRINT(".");

if (i>120) {

if (ssid==ssid1){

ssid=ssid2;

password=password2;

} else {

ssid=ssid1;

password=password1;

}

i=0;

j++;

if (j>4) { return false;} /\* none of Wifi work \*/

DPRINTLN();

DPRINT("Try with other network ");DPRINTLN(j);

DPRINT("I will try to connect to "); DPRINTLN(ssid);

WiFi.disconnect();

espera(1000);

WiFi.begin(ssid,password);

}

}

DPRINTLN(ssid); DPRINT("\*\*\*\*\*\*\*Conected; ADDR= ");

DPRINTLN(WiFi.localIP()); // print local IP adress

return true;

}

/\* Function called when no connectivity; resets everything and retries\*/

void sinConectividad(){

int j=0;

clienteMQTT.disconnect();

espera(500);

while(!wifiConnect()) {

DPRINT("No connectivity Wait for secs ");DPRINTLN(int(intervaloConex/2000));

espera(ESPERA\_NOCONEX);

}

}

Once the ESP8266 is connected to WIFI, we need some functions to register it to the MQTT broker, to subscribe to different messages and to implement a function to send data. In this case, we will only subscribe to ‘reboot’, that will reboot the device (Warning, depending on the firmware, the device might not reboot correctly).

So, first we register to MQTT broker, and

#define ESPERA\_NOCONEX 70000 // when there is no connnection, wait 70sec

char server[] = "192.168.1.11"; // MQTT broker address

char \* authMethod = NULL;

char \* token = NULL;

char clientId[] = "d:" ORG ":" DEVICE\_TYPE ":" DEVICE\_ID;

char publishTopic[] = "meteo/envia"; // Device sends data to MQTT broker with this topic

char rebootTopic[] = "meteo/reboot";

void funcallback(char\* topic, byte\* payload, unsigned int payloadLength);

PubSubClient clienteMQTT(server, 1883, funcallback, wifiClient);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* connect to MQTT broker \*

\* it requires to connect to Wifi first \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void mqttConnect() {

int j=0;

if ((WiFi.status() == WL\_CONNECTED )) {

while (!clienteMQTT.connect(clientId, authMethod, token)) {

DPRINT(j);DPRINTLN(" I will retry connecting MQTT client ");

j++;

espera(2000);

if (j>20) {

sinConectividad(); //if ESP8266 is not connected

j=0;

}

}

} else {

sinConectividad(); // last try, still not connected

}

}

/\* this function is a reference, just keeping alive MQTT client and reading the queue \*/

boolean loopMQTT() {

return clienteMQTT.loop();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* initialize the device, subscribing to

\* some actions; in this case reboot (no explanation needed, I guess),

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void initManagedDevice() {

int rReboot;

rReboot= clienteMQTT.subscribe(rebootTopic,1);

DPRINT("\tSubscribe to Reboot= ");DPRINT(rReboot);

}

/\* and here are the functions invoked depending on the topic, whe MQTT broker send ¡data to the agent, this is the function that will be invoked \*/

void funcallback(char\* topic, byte\* payload, unsigned int payloadLength) {

DPRINT("funcallback invoked for topic: "); DPRINTLN(topic);

if (strcmp (rebootTopic, topic) == 0) {

DPRINTLN("Rebooting...");

ESP.restart(); // Not always works due to an issue in ESP12

}

}

There are only two additional functions needed, one to send data to the broker and one that waits to allow internal routines to be active:

/\*\*\*\*\*\*\*\*\*\* Send data to broker \*\*\*\*\*\*\*\*\*\*\*\*\*\*/

boolean enviaDatos(char \* topic, char \* datosJSON) {

int k=0;

char signo;

boolean pubresult=false;

while (!clienteMQTT.loop() & k<20 ) {

DPRINTLN("Device ws disconnected, reconnecting ");

mqttConnect();

initManagedDevice();

k++;

}

pubresult = clienteMQTT.publish(topic,datosJSON);

DPRINT("Sending ");DPRINT(datosJson);

DPRINT("to ");DPRINTLN(publishTopic);

if (pubresult)

DPRINTLN("... OK Success");

else

DPRINTLN(".....KO Failure");

return pubresult;

}

## Program files

To make things easier, I have grouped this code in four files:

ESP12\_BME280.ino : Main code -I mean, setup() and loop() functions- and sensor reading functions

1stDevice.h : Definitions for each device (there can be several devices reading sensors in different places)

Mqtt\_mosquitto.ino. networking code, setting up and connecting WiFi, initializing mqtt client and functions for sending data to the mqtt broker.

Mqtt\_mosquitto.h: definitons for WiFi (SSID, password) and mqtt broker (address, pasword).

We have an agent that sends data… now, we need a server