TUTORIEL : MARATHON CREATIF LYCEE DE SERRES 2025

1) Matériel nécessaire :

Ce projet consiste à faire une serre connectée dans le cadre du marathon créatif 2025, avec le lycée Oliver De Serres à Quetigny.

Les parties conceptions sont disponibles dans la partie SolidWorks

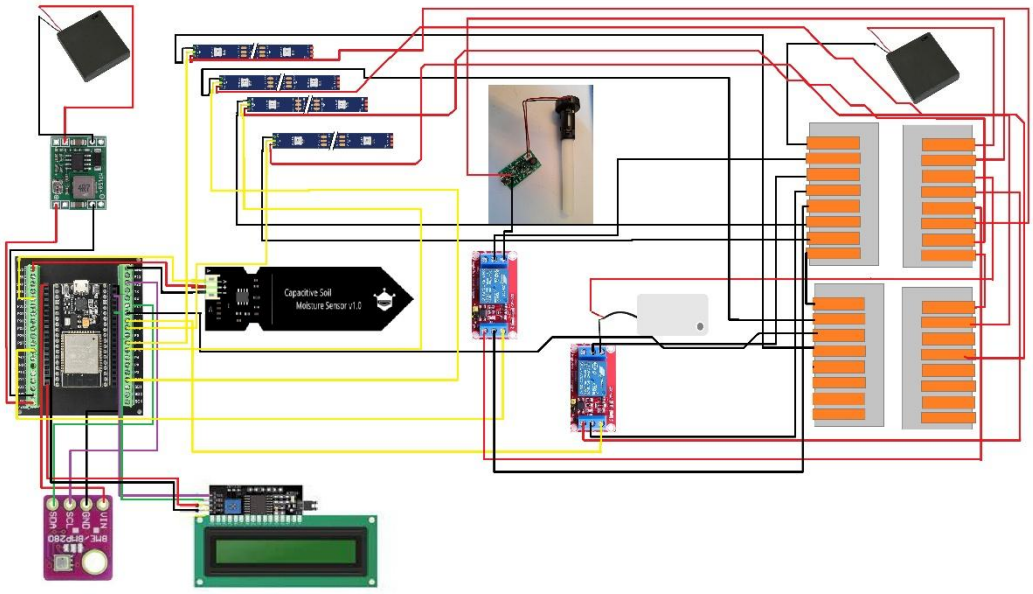
Dans le détail il s'agit de faire une serre dont on peut contrôler via IOT (ici ThingsBoard) afin de contrôler le mode d'arrosage et de pouvoir voire les données importantes d'une serre (température, humidité de l'air et du sol, pression) à distance.

Pour effectuer ce montage on va avoir besoin de :

- Au moins 4 rubans de led qui prendront des couleurs distinctes en fonction des mesures (humidité du sol, humidité de l'air, température, pression)
- Une carte Arduino (esp32 ici)
- Potentiellement un adaptateur pour votre carte
- Un brumisateur
- Un BME 280
- Un capteur d'humidité du sol (capacitif)
- Une pompe
- 2 relais 5V
- 1 à 2 alimentation extérieurs (ici des blocs de 4 piles AA)
- 8 dominos afin de faire les connections entre les éléments (sinon les soudés entre eux)
- Un convertisseur de courant (ici 4R7), si vos sources de courant ne sont pas précisément à 5V.

2) Montage à faire :

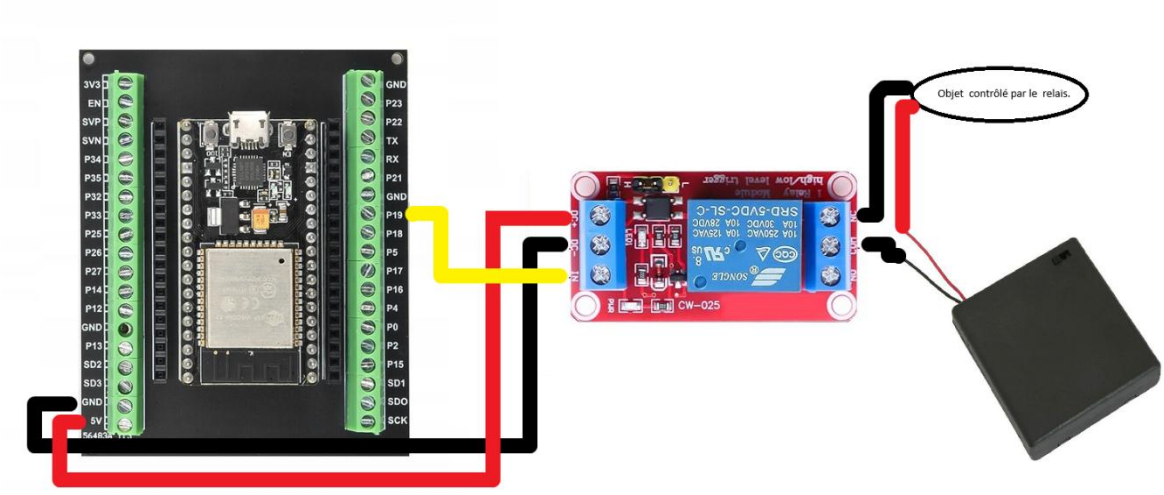
Voici le montage complet :



Ce dernier étant peu lisible on va le découper en différentes parties qui seront plus simple à reproduire :

- Les relais :

Les relais permettent de contrôler via les cartes des objets demandant trop de courant à la carte ou bien ne fonctionnant à de plus haut voltage que ce que la carte demande. Le montage général ressemblerait à cela :

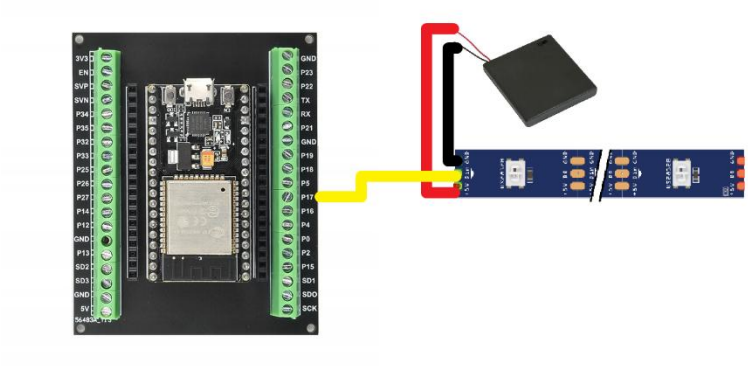


Les DC+ et - du relais vont respectivement sur 5v et GND de la carte (ou d'une autre source de courant) . Le IN vient d'un pin numéroté de la carte, en l'occurrence pour la pompe le 19 et pour le brumisateur le 14.

On relit une borne de la pile externe à l'objet, l'autre à la prise COM de l'autre côté du relais, et on relais la dernière borne de l'objet à la prise NO du relais afin qu'il soit éteint par défaut et qu'on contrôle la mise en marche (il serait possible le connecter à NC, afin qu'il soit actif par défaut et qu'on puisse contrôler son arrêt.

- Les rubans de LEDs :

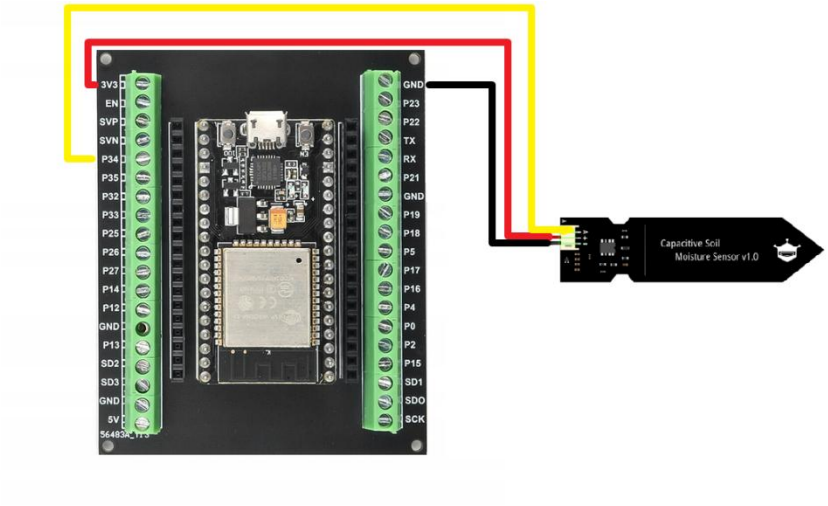
Ces derniers ont besoin de 3 prises, 5V et GND qui partent sur la même alim externe que les objets liés aux relais. Et le D0/Din qui par vers la carte et un pin numéroté. Il est possible de connecter plusieurs rubans sur le même si on souhaite qu'ils aient la même couleur.



Dans notre cas on a 4 types de led différents qui seront sur les pins de 15 à 18.

- Le capteur d'humidité sol :

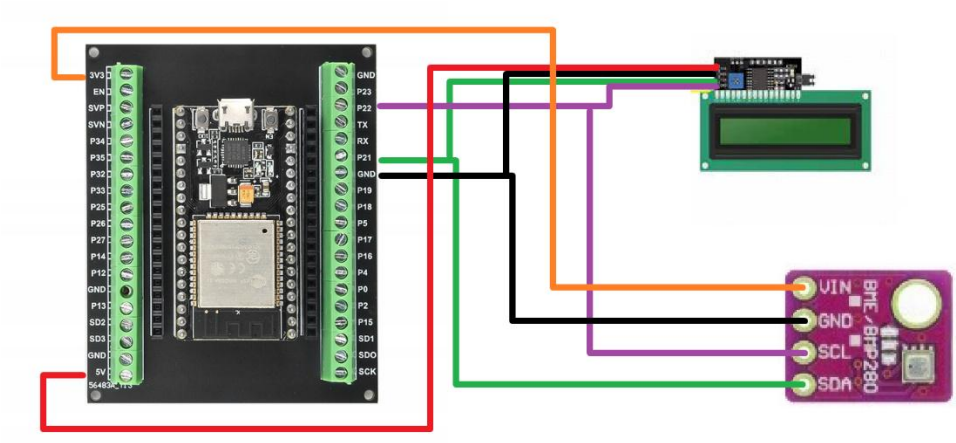
Le capteur d'humidité du sol que l'on va utiliser est capacitif, il va donc renvoyer une valeur analogique en fonction de la capacite électrique du sol. On doit donc le connecter à une prise analogique de la carte, on les voit sur le pinout de la carte avec la notation ADC



Ici nous utiliserons le port 34.

- L'écran LCD et le BME (la connexion en I2C) :

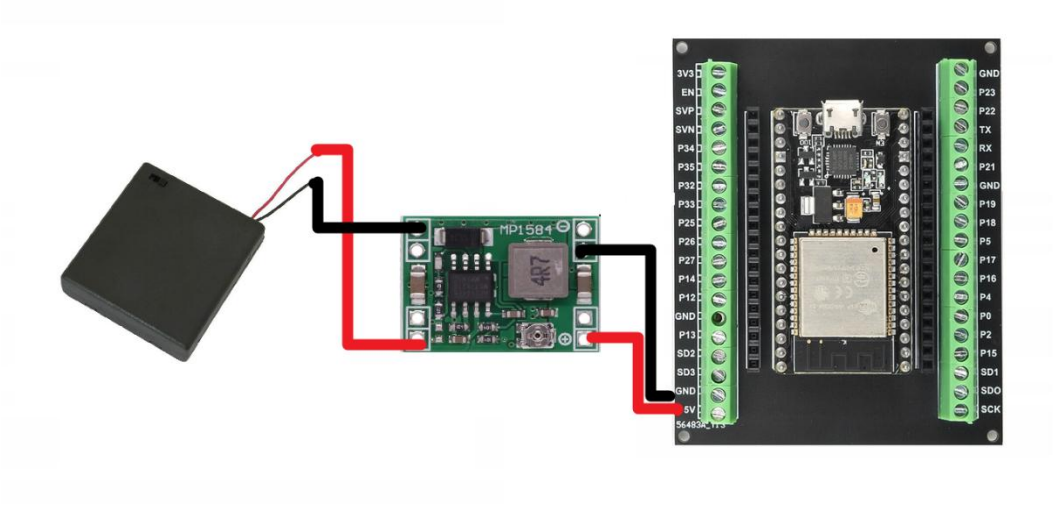
Ces 2 objets sont connectés de la même manière (à la différent du voltage : 3V pour le BME, 5V pour le LCD) à savoir GND sur GND, le VIN sur la prise 3V ou 5V correspondantes, et SDA/SCL qui permettent le transfert des informations. Chaque carte à ses propres prises correspondant à SCL et SDA . Dans le cas de l'esp 32 c'est respectivement 22 et 21.



Du fait que toute les connections en I2C se faisant par les mêmes connexions pour chaque objet, on leurs donnera une adresses propre coté code afin que la carte sache qui lui envoie les informations et à qui les envoyées.

- L’alimentation de la carte et le convertisseur de courant :

Les cartes ont besoin de 5V et non de 6V comme nos alimentations avec 4 piles AA de 1,5V donne, donc on doit mettre un convertisseur entre la batterie de la carte (différente de celle des autres composant) et la carte elle-même. On a choisi un convertisseur 4R7 dont on contrôle la tension de sortie en tournant la vise sur le convertisseur.

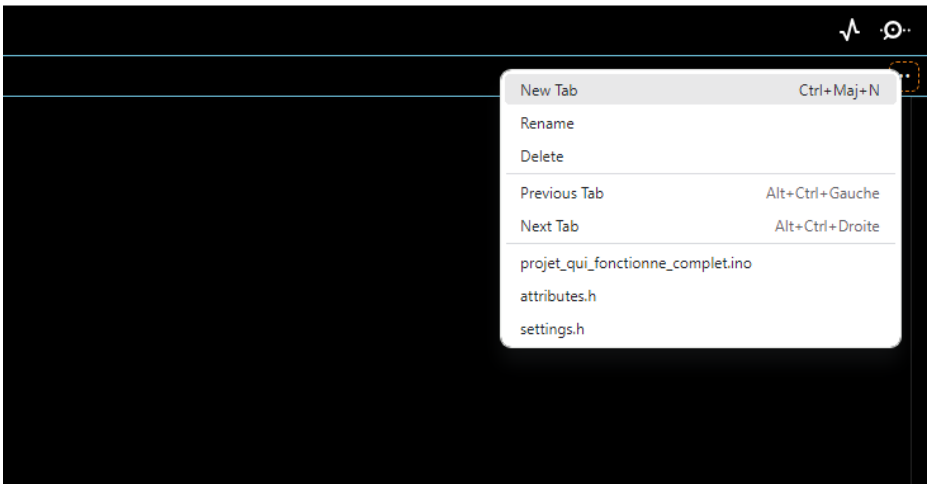


3) <u>Explication du code :</u>

Comme toujours on commence par la partie bibliothèques et initialisation.

Cette fois ci , comme le code est plus long et un peu plus complexe . On va donc créer des fichiers .h afin de séparer chaque partie du code. L’un sera appeler settings.h, l’autres attributes.h. On reviendra sur ces noms plus tard.

Tout d’abord pour les créer on va utiliser le bouton ci-dessous, situer en haut à droite de la page :



Dans settings.h vont se situer les bibliothèques et les initialisations habituelle comme :

- Pour l'écran LCD I2C :

Où on lance juste la bibliothèque pour le LCD connecté en I2C :

```
#include <LiquidCrystal_I2C.h>
```

Puis on déclare au code que cet écran existe avec son nom dans la suite (ici simplement lcd)

```
LiquidCrystal_I2C lcd(0x27, 16, 2);
```

Le code 0x27 correspond à l'adresse I2C de l'écran , permettant à la carte de le reconnaitre et lui envoyer des informations, et 16,2 correspondants respectivement aux nombres de colonnes puis de lignes de notre écran.

- Pour le bme 280 :

Le principe est le même d'abord la bibliothèque :

```
#include <Adafruit_BME280.h>
```

Et donc déclarer le bme

```
Adafruit_BME280 bme;
```

- Pour les objets reliés aux relais :

On va définir les pins de chacun des relais, le 1er pour la pompe et le 2nd pour le brumisateur :

```
#define relayPin 19
#define Brumi 14
```

- Pour le capteur d'humidité du sol :

On va définir 2 variables :

Le pin sur lequel on le connecte

```
#define moisturePin 34
```

Et la limite d'humidité qui nous servira plus tard pour déclanchement de l'arrosage automatique.

```
#define humidityRate 50
```

- Les rubans de LEDs.

On commence comme toujours par la bibliothèque dédiée :

```
#include <FastLED.h>
```

Ensuite on donne les pins de nos différents rubans de LEDs :

```
#define led_humi 17
#define led_humi_r 16
#define led_press 15
#define led_temp 18
```

On donne les données importantes à la mise en place des rubans :

- Le type de led utilisées :

```
#define LED_TYPE WS2812B // type de LED des rubans
```

- Le nombre de leds de nos rubans (on peut en déclarer plusieurs au besoin si des rubans de tailles différentes)

```
#define NUM_LED 30 // nombre de led par rubans
```

- L'ordre des couleurs, car les informations sont envoyées par défaut dans l'ordre rouge(R :red), vert (G : green) et bleu (B :blue) . Cependant certains rubans de LED demandent un autre ordre pour les couleurs, ici mes rubans inversais le rouge et le vert d'où l'ordre : GRB

```
#define COLOR_ORDER GRB
```

Enfin on déclare tous les rubans que l'on aura avec leurs noms dans le code (différents des pins car la casse est prise en compte par C, avec leurs nombres de LED.

```
CRGB led_Humi[NUM_LED];
CRGB led_Humi_R[NUM_LED];
CRGB led_Press[NUM_LED];
CRGB led_Temp[NUM_LED];
```

- Thingsboard et l'IoT.

IoT signifie Internet Of Things (en français L'internet des objets) , c'est un sujet très vaste mais dans le cadre d'Arduino c'est principalement lié aux méthodes pour contrôler à distance des cartes Arduino , soit par d'autres cartes ne soit pas des applications/site internet dédier.

Celui que nous allons utiliser ici est ThingsBoard, qui nous permettra d'avoir en direct un affichage de nos données mesurées par nos capteurs (bme et capteur d'humidité du sol) ainsi que de contrôler notre pompe avec un mode automatique et un mode manuel permettant d'activer la pompe quand on le souhaite.

Afin de faire cela on va avoir tout d'abord besoin de bibliothèques :

La bibliothèques Wi-Fi qui permettra de se connecter au Wi-Fi et donc à ThingsBoard.

```
#include <WiFi.h>
```

La bibliothèques Arduino MQTT Client, qui permettra à ThingsBoard et la carte Arduino de communique entre eux via le protocole MQTT, et donc avec un langage commun leurs permettant de se comprendre.

```
#include <Arduino_MQTT_Client.h>
```

La bibliothèque ThingsBoard qui permet d'utiliser des fonctions spécifiques au site internet ThingsBoard directement

```
#include <ThingsBoard.h>
```

Ensuite on va définir les constantes importantes :

Le nom et le mot de passe du Wi-Fi sur lequel notre carte va se connecter (hard codé donc à recompiler et envoyer sur la carte en cas de changement)

```
constexpr char WIFI_SSID[]="tplinkore";
constexpr char WIFI_PASSWORD[]="ore19944";
```

On a ensuite le nom du serveur ThingsBoard, ici thingsboard.cloud qui est le plus pratique car il est gratuit, accessible sur portable et on peut renouveler son compte tous les mois avec la même adresse en supprimant simplement son ancien compte.

```
constexpr char THINGSBOARD_SERVER[] = "thingsboard.cloud";
```

On va ensuite donner un token :

```
constexpr char TOKEN[]="QyYuVfXtXFmrEQT0ijUV";
```

Ce dernier est une suite de lettre et de chiffre qui se trouvent sur le serveur ThingsBoard dans la catégorie Entity (Entité) -> Device (Appeler dispositif si directement dans le site ou appareil) et qui permet au site d'identifié la carte Arduino de son côté et de stocker ces données à un endroit précis.

Cela permet d'éviter toute confusion si 2 cartes ont une variable ayant le même nom, grâce au token chacune de ces variables sera stockée individuellement dans la mémoire du serveur.

Une fois cela fait il faut donner au serveur le port sur lequel on se connecte (par défaut le port 1883)

```
constexpr uint16_t THINGSBOARD_PORT = 1883U;
```

La taille maximale du message échangé entre la carte et ThingsBoard.

```
constexpr uint32_t MAX_MESSAGE_SIZE = 1024U;
```

Et le nombre maximal d'attributs qui peuvent s'échanger entre la carte et le serveur (afin que le serveur donne suffisamment de place pour les traiter dans sa mémoire).

```
constexpr size_t MAX_ATTRIBUTES = 2U;
```

Un attribut se différencie d'une télémétrie (qu'on verra plus tard) dans le fait qu'une télémétrie est un envoi à sens unique de la carte vers le serveur et ne peux pas être modifié, là ou un attribut peut être modifié d'un côté comme de l'autre en fonction du besoin.

Enfin ont défini le client Wi-Fi,

```
WiFiClient wifiClient;
```

Avec ce client Wi-Fi on définit le client MQTT

```
Arduino_MQTT_Client mqttClient(wifiClient);
```

Qui sert enfin à définir le client de Thingsboard.

```
ThingsBoardSized<Default_Fields_Amount, Default_Subscriptions_Amount,
MAX_ATTRIBUTES> tb(mqttClient, MAX_MESSAGE_SIZE);
```

Auquel on doit donner les informations définies précédemment : le nombre maximum d'attributs, et la taille maximale des messages, les autres valeurs sont laissées par défaut.

On va maintenant définir le attributes.h qui sera dédier à la gestion de tous les objets utiles pour les attributs et les télémétries.

- La partie liée aux télémétries :

Elle ne se compose que 2 lignes, la 1ère défini une constante, entière, encodée sur 16 bits qui correspond au nombre de millisecondes entre les envois.

```
constexpr int16_t telemetrySendInterval = 2000U;
```

La 2nde définie une variable qui servira plus tard a stocké l'heure du dernier envoie, et donc de savoir si l'intervalle depuis le dernier envoi est suffisant pour en lancer un nouveau :

```
uint32_t previousDataSend;
```

- La partie liée aux attributs :

On commence par définir des chaines de caractères constantes qui seront les noms de nos attributs sur ThingsBoard.

```
constexpr char BTN_MANUEL_ATTR[]="btn_manuel";
constexpr char BTN_AUTO_ATTR[]= "btn_auto";
```

On leur met ensuite des valeurs par défaut (ici nos attributs serviront à des boutons ils seront donc des booléens)

```
volatile bool btn_manuel =true;
volatile bool btn_auto= true;
```

On créer une variable qui sera un booléen permettant de savoir si les attributs ont changé et donc savoir s'il faut les changer d'un côté ou de l'autre.

```
volatile bool attributesChanged = true;
```

On définit ensuite un tableau (array) de chaine de caractère (const char) de longueur 2 (2U) qui sera la liste des noms de nos attributs.

```
constexpr std::array<const char *, 2U> SHARED_ATTRIBUTES_LIST = {
  BTN_MANUEL_ATTR,
  BTN_AUTO_ATTR
};
```

Ensuite on définit la fonction processSharedAttributes qui permet de traiter les valeurs des attributs après changement sur ThingsBoard.

```
void processSharedAttributes(const JsonObjectConst &data) {
```

Cette ligne permet de passer par toutes les valeurs qui revienne du serveur ThingsBoard.

```
for (auto it = data.begin(); it != data.end(); ++it) {
```

La 1ère condition permet grâce à strcmp de comparer les chaines de caractère qui sort de ThingsBoard au nom des attributs (afin de remettre de bons attributs dans la variable avec le bon nom)

```
 if (strcmp(it->key().c_str(), BTN_MANUEL_ATTR) == 0) {
```

Et si les noms correspondent (donc que la fonction strcmp retourne 0) alors on remet la valeur de ThingsBoard dans la variable sur la carte sous la forme d'un booléen.

```
btn_manuel = it->value().as<bool>();
    }
```

La seconde condition fait exactement la même chose mais pour l'autre attributs

```
 else if (strcmp(it->key().c_str(), BTN_AUTO_ATTR) == 0) {
      btn_auto = it->value().as<bool>();
    }
  }
```

Enfin on passe le booléen qui stocke l'information du changement des attribut vers « vrai » :

```
  attributesChanged = true;
}
```

On arrive aux callbacks qui sont 2 lignes longues mais bien distincte , la 1ère est la « Shared Attributes Callbacks », elle permet de traiter les attributs partager qui aurais été reçu depuis ThingsBoard sans demande préalable et on lui donne donc la fonction à utiliser et 2 « itérateur » qui donne le début et la fin de la liste des attributs.

```
const Shared_Attribute_Callback<MAX_ATTRIBUTES>
attributes_callback(&processSharedAttributes, SHARED_ATTRIBUTES_LIST.cbegin(),
SHARED_ATTRIBUTES_LIST.cend());
```

La 2nde est la « Attributes Request Callbacks », elle permet de traiter les attributs partagés qui auraient été reçus depuis ThingsBoard avec une demande de la carte. Et on lui donne donc la fonction à utiliser et 2 itérateur qui donne le début et la fin de la liste des attributs.

```
const Attribute_Request_Callback<MAX_ATTRIBUTES>
attribute_shared_request_callback(&processSharedAttributes,
SHARED_ATTRIBUTES_LIST.cbegin(), SHARED_ATTRIBUTES_LIST.cend());
```

On passe maintenant au .ino et on commence par inclure les autres documents sur les 2 1ères lignes

```
#include "settings.h"
#include "attributes.h"
```

Avant de se lancer dans le void setup() :

On commence par lancer le moniteur série, car c'est toujours utile si besoin de débug quoi que ce soit.

```
Serial.begin(115200);
```

Ensuite on lance le bme avec l'adresse I2C nécessaire.

```
bme.begin(0x76);
```

Suivi du mode de connexion pour les connexions citées dans settings à savoir :

- Entrée (input) pour le capteur d'humidité du sol.

```
pinMode(moisturePin,INPUT);
```

- Sortie (output) pour la pompe et le brumisateur.

```
pinMode(relayPin,OUTPUT);
pinMode(Brumi,OUTPUT);
```

On ajoute ensuite les rubans de LEDs en donnant le type de LED, le pin de connexion, l'ordre de couleur entre <> et le nom et nombre de LED entre parenthèse.

```
FastLED.addLeds<LED_TYPE, led_humi, COLOR_ORDER>(led_Humi, NUM_LED);
FastLED.addLeds<LED_TYPE, led_humi_r, COLOR_ORDER>(led_Humi_R, NUM_LED);
FastLED.addLeds<LED_TYPE, led_press, COLOR_ORDER>(led_Press, NUM_LED);
FastLED.addLeds<LED_TYPE, led_temp, COLOR_ORDER>(led_Temp, NUM_LED);
```

On lance ensuite la connexion au Wi-Fi

```
WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
```

On fait une petite boucle « tant que » en attendant que le statut du Wi-Fi passe en « connected » car c'est nécessaire pour l'étape suivante.

```
 while (WiFi.status() != WL_CONNECTED) {}
```

On lance maintenant la connexion au serveur ThingsBoard en donnant le nom du serveur, le token et le port de connexion.

```
tb.connect(THINGSBOARD_SERVER, TOKEN, THINGSBOARD_PORT);
```

On s'inscrit pour les 2 callbacks défini plus tôt avec les fonctions correspondantes dans la bibliothèque.

```
tb.Shared_Attributes_Subscribe(attributes_callback);
 tb.Shared_Attributes_Request(attribute_shared_request_callback);
```

Enfin on initialise (allume et remet à 0) le LCD et allume son rétroéclairage afin qu'il soit le plus lisible possible.

```
lcd.init();
lcd.backlight();
}
```

On passe alors au void loop() :

On commence par mettre à jour tout ce qui en a besoin coté ThingsBoard.

- Tout d'abord les attributs :

Si la variable attributesChanged (est vraie, sous-entendu), donc que l'on a modifié les attributs quelque part dans une boucle précédente.

```
if (attributesChanged) {
```

Alors on envoie à ThingsBoard les nouvelles valeurs de ces attributs avec la fonction prévue pour , les 2 argument sont tout d'abord le nom de l'attributs puis la variable dans laquelle sa valeur est stockée.

```
    tb.sendAttributeData(BTN_MANUEL_ATTR,btn_manuel);
    tb.sendAttributeData(BTN_AUTO_ATTR,btn_auto);
 }
```

- Ensuite les télémétries :

Ces dernières sont ici principalement les valeurs de nos capteurs qu'on va donc devoir définir avant de les envoyées.

On commence par le capteur d'humidité du sol qui nous ressort une valeur analogique entre 0 et 4095 (donc encodée sur 12 bits) on va devoir remappé ces valeurs sur une échelle de 0 à 100 afin d'avoir une équivalence avec un pourcentage d'humidité.
Pour cela on doit mesurer la valeur qu'il sort en étant complètement sec (équivalent de 0) et plongé dans de l'eau (équivalent 100) et on dans mon cas capteur ressort respectivement 3630 pour le minimum d'humidité et 930 pour le maximum. D'où la ligne suivante :

```
int humi=map(analogRead(moisturePin),930,3630,100,0);
```

Car on donne la valeur la valeur à remappé , l'intervalle qu'il peut nous donner puis l'intervalle sur lequel on veut l'envoyer

Pour les valeurs du BME on doit juste ressortir les valeurs du capteur grâce au fonction prévu pour cela dans la bibliothèque .

```
float temp=bme.readTemperature();
 float press=bme.readPressure()/100;
 float humi_air=bme.readHumidity();
```

La seule subtilité est de devoir divisé la pression par 100 car le capteur nous la donne en Pa alors que la norme est plutôt d'utilisé hPa.

On a ensuite une condition if sur le temps depuis le dernier envoi des infos (entre millis, l'instant présent , et previousDataSend stocke le moment du dernier envoi) afin de savoir si l'intervalle voulu entre 2 télémétries est passé.

```
 if (millis() - previousDataSend > telemetrySendInterval) {
```

Si on a bien des télémétries à envoyer ont redéfini d'abord previousDataSend avec le nouveau moment de la dernière télémétrie.

```
previousDataSend = millis();
```

Ensuite on envoie toutes les télémétries avec la fonction prévue pour cela dans la bibliothèque ThingsBoard , une nouvelle fois les arguments sont d'abord le nom coté ThingsBoard puis la variable.

```
tb.sendTelemetryData("temperature",temp);
tb.sendTelemetryData("pression",press);
tb.sendTelemetryData("humidite_air",humi_air);
tb.sendTelemetryData("humidite",humi);
  }
```

On met ensuite en place les contrôle de l'arrosage avec une suite de conditions.

Si la variable btn_auto est vraie (donc le mode automatique est enclenché)

```
if (btn_auto==true){
```

Alors on va mettre en place la condition de l'arrosage automatique , c'est-à-dire si l'humidité du sol est en dessous de notre seuil fixé.

```
if (humi<=humidityRate){
```

Alors on allume la pompe pour un temps donné puis on l'éteint.

```
digitalWrite(relayPin,HIGH);
delay(1500);
digitalWrite(relayPin,LOW);
```

Enfin , comme à chaque fin de condition en cas de mode automatique on remet le mode manuel à false. Cela évite de « stocker » l'éventuel appuie sur le bouton manuel durant le mode automatique, et donc de ne pas avoir 5 arrosages manuel au changement de mode, si on a appuyé 5 fois par erreur sur le bouton manuel lorsque le mode automatique était actif. Et on passe atttributesChanged en true afin que ce changement soit pris en compte.

```
btn_manuel=false;
attributesChanged= true;

    }
```

Si le taux d'humidité est au-dessus de la limite alors on éteint la pompe et on remet le mode manuel sur « faux »

```
else {
    digitalWrite(relayPin,LOW);
```

```
    btn_manuel=false;
attributesChanged= true;

    }
  }
```

Et si le mode automatique n'est pas actif

```
  else {
```

Alors on test l'état du mode manuel, s'il est actif (true) alors on remet en place la même manœuvre d'allumer puis éteindre la pompe

```
  if (btn_manuel==true){
    digitalWrite(relayPin,HIGH);
    delay(1500);
    digitalWrite(relayPin,LOW);
```

Et on reset l'état de la variable bouton manuel afin d'éviter que ce dernier reste enfoncer en permanence.

```
    btn_manuel=false;
attributesChanged= true;
  }
```

Sinon on laisse la pompe éteinte.

```
 else {
    digitalWrite(relayPin,LOW);
  }
```

On met ensuite en place des fonctions qui feront fonctionner le reste du code :

```
  LED_Scenario();
```

Qui permet de faire fonctionner le code couleur des LED et est défini de la façon suivante :

```
void LED_Scenario(){
```

Le code couleur de nos 4 rubans de LED étant défini par les mesures de nos capteurs on doit reprendre les mesures :

```
float temp=bme.readTemperature();
  float press=bme.readPressure()/100;
  float humi_air=bme.readHumidity();
  int humi=map(analogRead(moisturePin),930,3630,100,0);
```

Et ensuite on doit mettre des conditions sur chacune d'entre elles, je ne vais pas toute les détaillés car elles sont strictement identiques d'un point de vue de la structure.

Prenons l'exemple de la température :

Si la température dépasse une certaine valeur :

```
if (temp>25){
```

Alors le ruban de LED prend une couleur donnée :

```
fill_solid(led_Temp,NUM_LED,CRGB::Yellow);
```

Qu'on lui demande d'afficher :

```
FastLED.show();
```

Sinon il prend un autre couleur :

```
else {
fill_solid(led_Temp,NUM_LED,CRGB::Orange);
  FastLED.show();
}
```

Et ce type de condition se répète pour chaque valeur que nous mesurons.

On a ensuite la fonction :

```
  brumisateur();
```

Qui est elle aussi défini en fin de code :

```
void brumisateur(){
```

Et permet sur le même principe de condition de faire fonctionner le brumisateur :

Si l'humidité de l'air est inférieure à un seuil fixé de notre part :

```
float humi_air=bme.readHumidity();
if (humi_air<40){
```

Alors on allume le brumisateur pour une durée donnée :

```
digitalWrite(Brumi,HIGH);
    delay(2500);
    digitalWrite(Brumi,LOW);
  }
```

Sinon on éteint le brumisateur.

```
else {
    digitalWrite(Brumi,LOW);
  }
}
```

On a enfin la fonction :

```
LCD_Scenario() ;
```

Qui comme les 2 précédentes est définies en fin de code :

```
void LCD_Scenario(){
```

Et sert à afficher toute nos valeurs mesurées sur l'écran LCD connectée à notre carte , valeur qu'on doit donc redéfinir une nouvelle fois :

```
float temp=bme.readTemperature();
  float press=bme.readPressure()/100;
  float humi_air=bme.readHumidity();
  int humi=map(analogRead(moisturePin),930,3630,100,0);
```

Et ensuite on a une série de bloc dont je ne vais une nouvelle fois détaillé que les 2 1ers car à partir du 2nd ils sont tous identiques dans leurs formes.

Tout d'abord avant d'afficher quoi que ce soit il faut effacer ce qu'il y avait précédemment car l'écran ne le fais pas de lui-même.

```
lcd.clear();
```

On doit aussi lui fixer ou écrire (0 ,0) pour commencer au 1er caractère de la 1ère ligne car les numérotations commencent à 0.

```
lcd.setCursor(0,0);
```

On affiche ensuite le texte que l'on souhaite

```
lcd.print("Marathon Creatif");
```

Changer le curseur de place (ici on passe en 1ère case de la 2nde ligne car les coordonnés sont notés colonnes, lignes)

```
lcd.setCursor(0,1);
```

Et affiché un autre texte (j'ai ici évité d'utiliser le « é » dans lycée car les écrans ne le comprennent pas et affiché des caractères différents si on lui demande de l'afficher.

```
lcd.print("Lycee de Serres");
```

Et enfin un délai pour permettre au texte de rester affiché un certain temps

```
delay(1000);
```

Ce principe va se répéter sur les autres choses à afficher, qui seront partir d'ici les valeurs de nos capteurs et prendront donc toujours la même forme, ici illustré avec la température :

On met le nom de la valeur sur la 1ère ligne

```
lcd.clear();
lcd.setCursor(2,0);
lcd.print("Temperature:");
```

Et la valeur avec son unité sur la 2nde ligne.

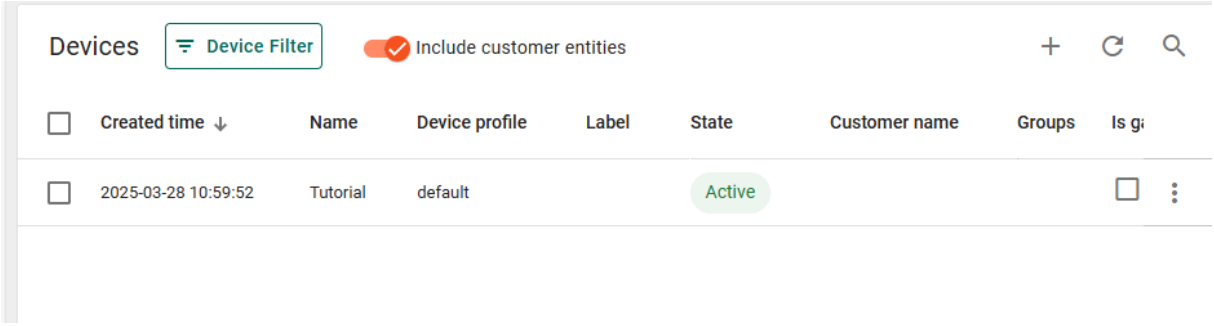
```
 lcd.setCursor(5,1);

lcd.print(temp);
lcd.print("C");
delay(1000);
```

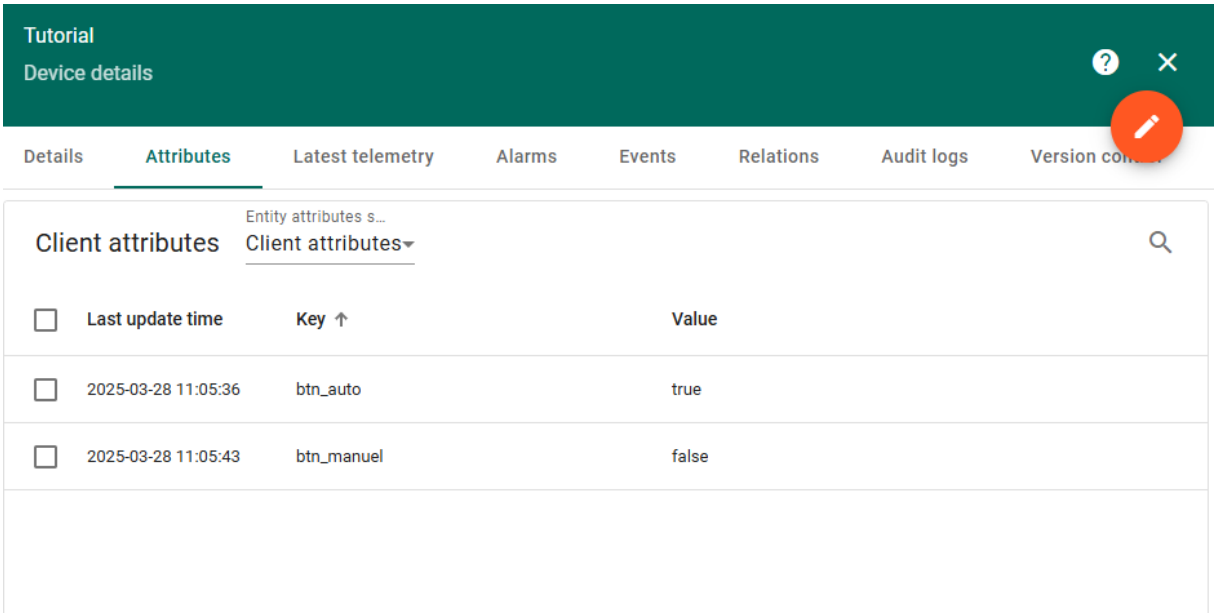
4) Mise en place de ThingsBoard :

Une fois le code fait et envoyer (voir code complet plus bas pour le faire). On va devoir mettre en place le fonctionnement coté ThingsBoard.

Une fois votre device/dispositif (je vais les appeler devise dans la suite pas soucis de clarté) créer sur ThingsBoard et le code envoyer sur votre carte avec votre token vous devriez avoir votre device qui est affiché active (inactive par défaut) .

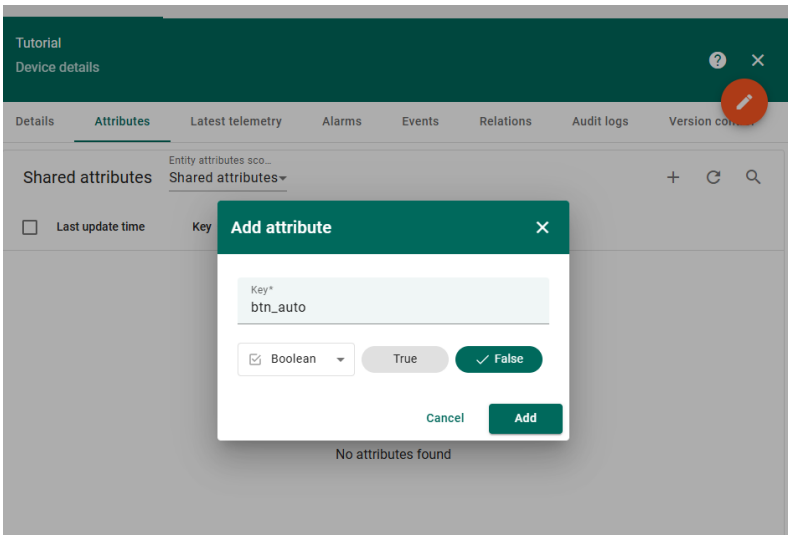


En cliquant dessus une fenêtre devrait s'ouvrir et aller dans la catégorie attributes, il ne devrait y avoir que des attributs dans la catégorie client attributes :

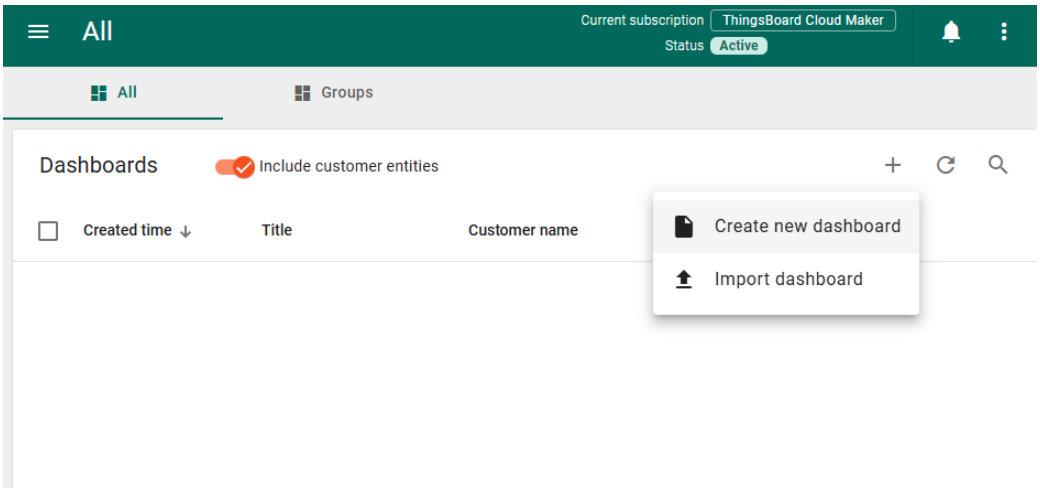


On va donc devoir ajouter les attributs partagés à la main en allant dans la catégorie shared attributes (menu défilant sur la case client attributes)

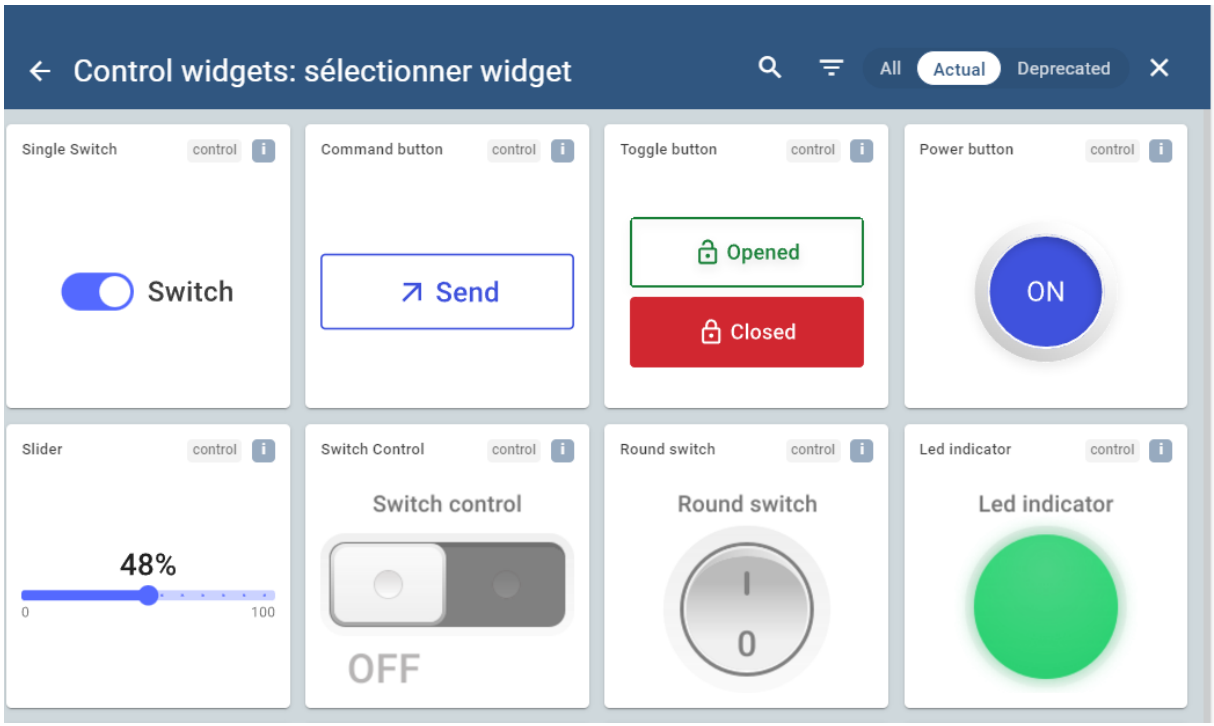
Puis en cliquant sur le « + » , mettant le type de variable sur booléen et en rentrant le nom des attributs que l'on souhaite avoir en partagés.



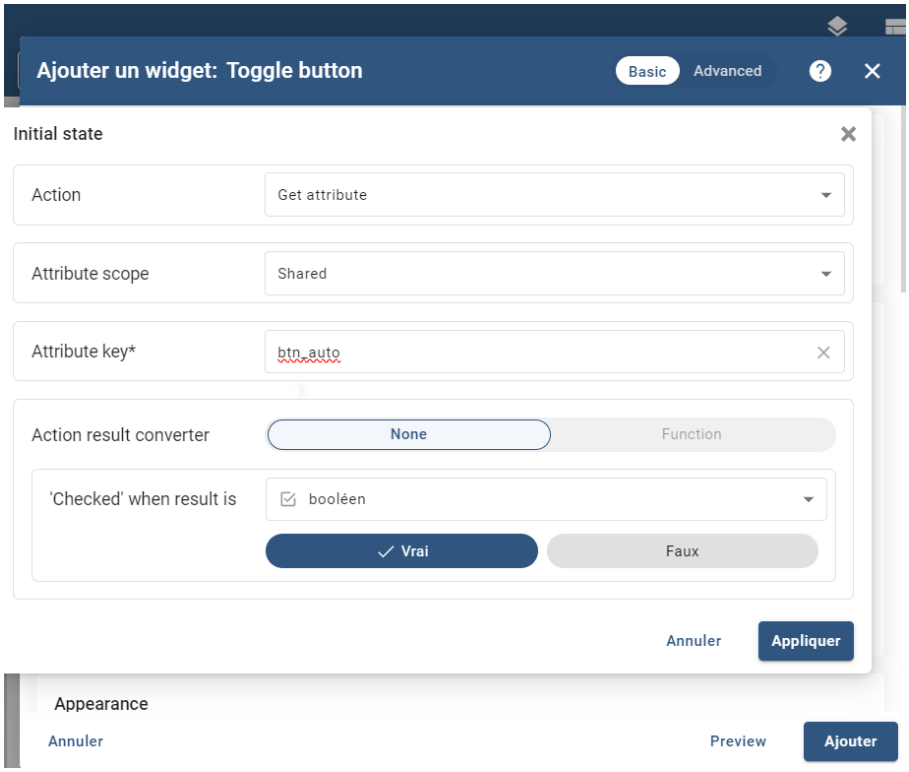
Un fois cela fait on peut passer dans la partie Dashboard. Pour ça il faut aller dans la partie dashboard/Tableaux de bord (je vais juste utiliser dashboard pour être clair) et on va en créer un nouveau.



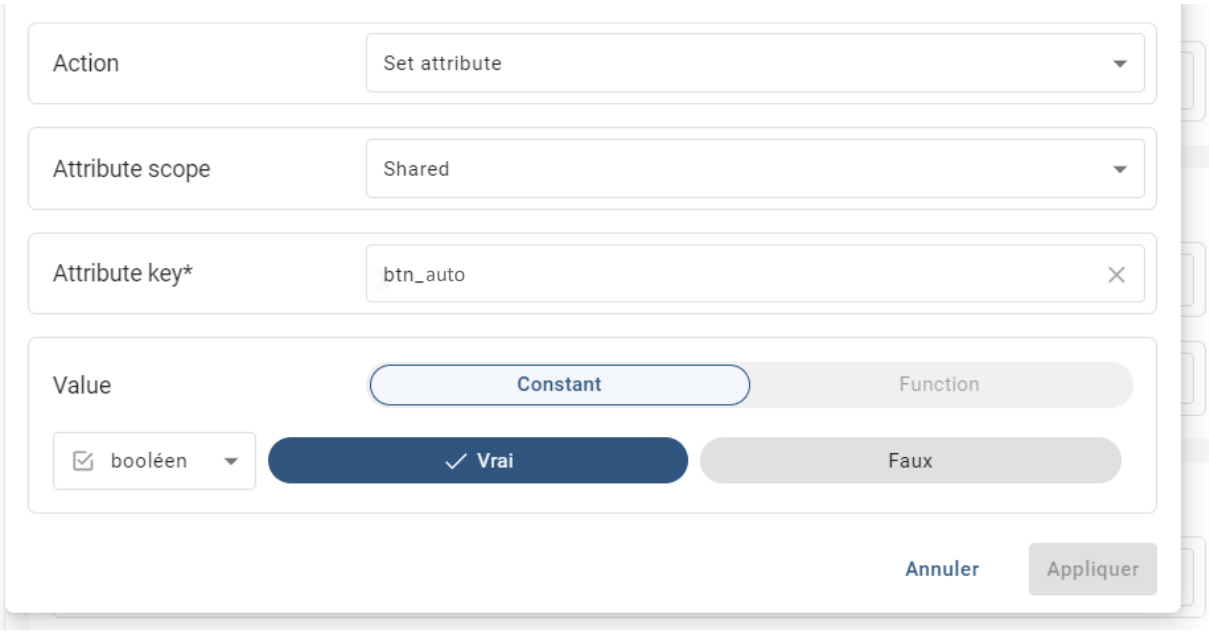
Donné lui le nom que vous souhaitez, ne vous souciez pas de la fenêtre qui apparait une fois créer, car vous pouvez juste la fermée. Vous devriez arriver sur un écran qui a « add new widget » Cliquer dessus et aller dans la catégorie « control widgets ».



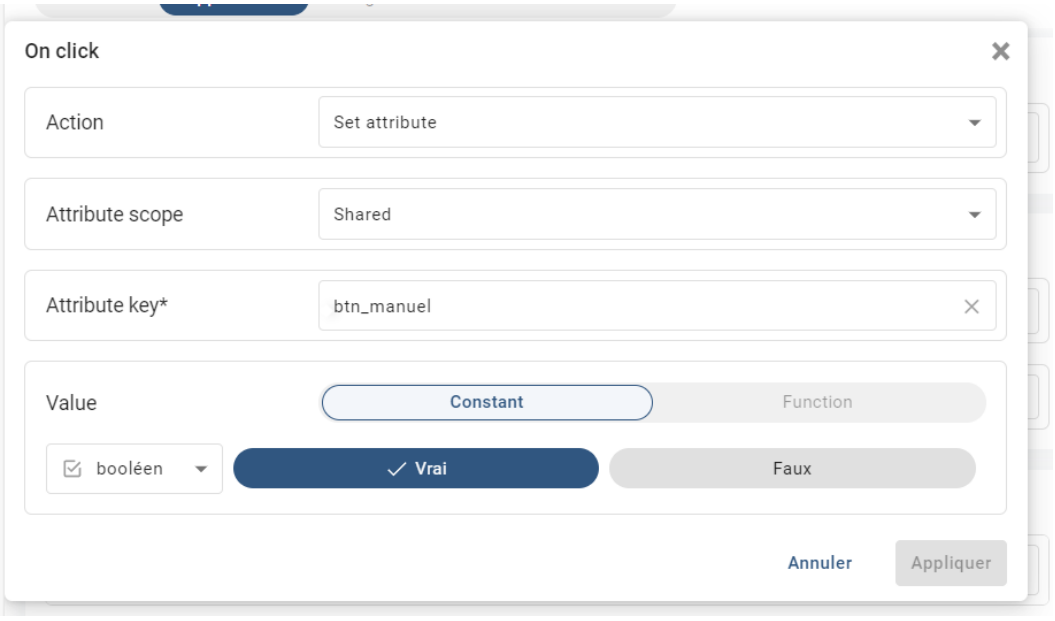
Choisissez « toggle button », choisissez le bon device dans la case en haut et mettez les paramètres suivants, qui permette de mettre le bouton par défaut à l'état de votre carte :



Et pour les options « check / uncheck » mettez les paramètres suivants en prenant soins de changer la « value » en bas pour différencier les 2 états.

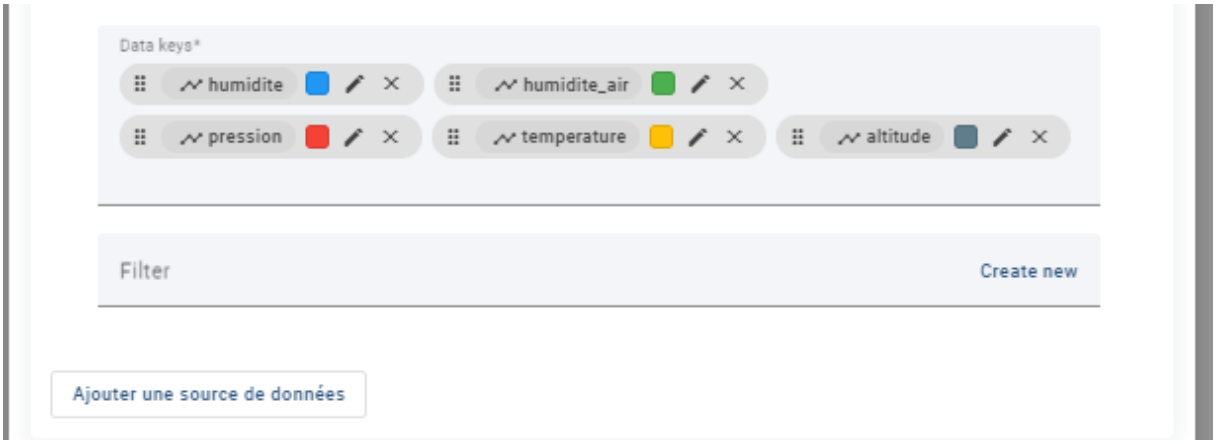


Pour ce qui concerne le bouton d'arrosage manuel on va rajouter un autre widget (bouton «+ » en haut à gauche) et choisir cette fois ci le « command button », les informations à rentrer son globalement les même que pour le bouton précédent sauf qu'il n'y a qu'un seul état (celui appuyé) à remplir.



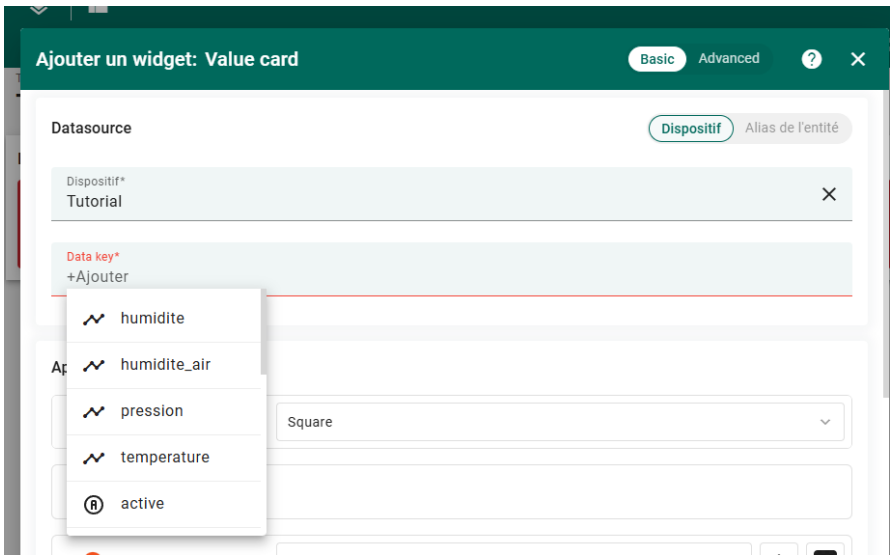
Une fois cela fait on devrait contrôler grâce à ces boutons les modes d'arrosage dans la serre.

On peut maintenant afficher les valeurs de nos capteurs, soit en les mettant dans un tableau grâce au widget de la catégorie « card », ou il faut alors donner le nom du device et choisir toutes les variables qu'on veut dans notre tableau :



soit en prenant un affichage individuel pour chaque valeur et dans ce cas il faut aller chercher « value card » dans la catégorie « card » , ou une de ses variables avec ou sans graphique etc..

Dans tous les cas il suffit alors de renseigner le device et la valeur à afficher.



En ce qui concerne la mise ne page vous pouvez modifier la taille directement en tirant sur les objets, et leur couleur dans les paramètres de chaque widget.

Il est conseiller d'enregistrer les changements régulièrement et surtout de ne pas quitter sans le faire sinon tout changement non enregistrer sera perdu.

5) Code brut à recopier :

Pour settings.h :

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27, 16, 2);
//////////////////////////////////////////////////////////////////////////////
//////////////////////
#include <Adafruit_BME280.h> // Utile pour extraire les infos du BME-BMP 280.
Adafruit_BME280 bme;
//////////////////////////////////////////////////////////////////////////////
//////////////////////
#define relayPin 19
#define Brumi 14
//////////////////////////////////////////////////////////////////////////////
//////////////////////
#define moisturePin 34
#define humidityRate 50
//////////////////////////////////////////////////////////////////////////////
//////////////////////
#include <FastLED.h>// Utile pour contrôle les rubans de LED adressable

#define led_humi 17 // pin du 1er ruban de LED pour humidité
#define led_humi_r 16 // pin du 2nd ruban de LED pour temperature.
#define led_press 15
#define led_temp 18

#define LED_TYPE WS2812B // type de LED des rubans
#define NUM_LED 30 // nombre de led par rubans
#define COLOR_ORDER GRB

CRGB led_Humi[NUM_LED];
CRGB led_Humi_R[NUM_LED];
CRGB led_Press[NUM_LED];
CRGB led_Temp[NUM_LED];

//////////////////////////////////////////////////////////////////////////////
//////////////////////
#include <WiFi.h>
#include <Arduino_MQTT_Client.h>
#include <ThingsBoard.h>

constexpr char WIFI_SSID[]="Association_ORE";
constexpr char WIFI_PASSWORD[]="Association0r3";

constexpr char THINGSBOARD_SERVER[] = "thingsboard.cloud";
constexpr char TOKEN[]="wnp3tBYNgNcBVcjMtuPi";

constexpr uint16_t THINGSBOARD_PORT = 1883U;
```

```
constexpr uint32_t MAX_MESSAGE_SIZE = 1024U;
constexpr size_t MAX_ATTRIBUTES = 2U;

WiFiClient wifiClient;
Arduino_MQTT_Client mqttClient(wifiClient);
ThingsBoardSized<Default_Fields_Amount, Default_Subscriptions_Amount,
MAX_ATTRIBUTES> tb(mqttClient, MAX_MESSAGE_SIZE);
```

Pour attributes.h :

```
constexpr int16_t telemetrySendInterval = 2000U;
uint32_t previousDataSend;
////////////////////////////////////////////////////////////////////////////////
////////////////////////
constexpr char BTN_MANUEL_ATTR[]="btn_manuel";
constexpr char BTN_AUTO_ATTR[]= "btn_auto";

volatile bool btn_manuel =true;
volatile bool btn_auto= true;

volatile bool attributesChanged = true;

constexpr std::array<const char *, 2U> SHARED_ATTRIBUTES_LIST = {
  BTN_MANUEL_ATTR,
  BTN_AUTO_ATTR
};

void processSharedAttributes(const JsonObjectConst &data) {
  for (auto it = data.begin(); it != data.end(); ++it) {
    if (strcmp(it->key().c_str(), BTN_MANUEL_ATTR) == 0) {
      btn_manuel = it->value().as<bool>();
    } a
    else if (strcmp(it->key().c_str(), BTN_AUTO_ATTR) == 0) {
      btn_auto = it->value().as<bool>();
    }
  }
  attributesChanged = true;
}

const Shared_Attribute_Callback<MAX_ATTRIBUTES>
attributes_callback(&processSharedAttributes, SHARED_ATTRIBUTES_LIST.cbegin(),
SHARED_ATTRIBUTES_LIST.cend());
const Attribute_Request_Callback<MAX_ATTRIBUTES>
attribute_shared_request_callback(&processSharedAttributes,
SHARED_ATTRIBUTES_LIST.cbegin(), SHARED_ATTRIBUTES_LIST.cend());
```

Pour le code principal en .ino :

```
#include "settings.h"
#include "attributes.h"
void setup() {
  Serial.begin(115200);
  bme.begin(0x76);
  pinMode(moisturePin,INPUT);
  pinMode(relayPin,OUTPUT);
  pinMode(Brumi,OUTPUT);
  FastLED.addLeds<LED_TYPE, led_humi, COLOR_ORDER>(led_Humi, NUM_LED);
  FastLED.addLeds<LED_TYPE, led_humi_r, COLOR_ORDER>(led_Humi_R, NUM_LED);
  FastLED.addLeds<LED_TYPE, led_press, COLOR_ORDER>(led_Press, NUM_LED);
  FastLED.addLeds<LED_TYPE, led_temp, COLOR_ORDER>(led_Temp, NUM_LED);
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
  while (WiFi.status() != WL_CONNECTED) {}
  tb.connect(THINGSBOARD_SERVER, TOKEN, THINGSBOARD_PORT);
  tb.Shared_Attributes_Subscribe(attributes_callback);
  tb.Shared_Attributes_Request(attribute_shared_request_callback);
  lcd.init();
  lcd.backlight();

}

void loop() {
  if (attributesChanged) {
    tb.sendAttributeData(BTN_MANUEL_ATTR,btn_manuel);
    tb.sendAttributeData(BTN_AUTO_ATTR,btn_auto);
  }
  int humi=map(analogRead(moisturePin),930,3630,100,0);
  float temp=bme.readTemperature();
  float press=bme.readPressure()/100;
  float humi_air=bme.readHumidity();
  if (millis() - previousDataSend > telemetrySendInterval) {
    previousDataSend = millis();
    tb.sendTelemetryData("temperature",temp);
    tb.sendTelemetryData("pression",press);
    tb.sendTelemetryData("humidite_air",humi_air);
    tb.sendTelemetryData("humidite",humi);
  }
  if (btn_auto==true){
    if (humi<=humidityRate){
      digitalWrite(relayPin,HIGH);
      delay(1500);
      digitalWrite(relayPin,LOW);
      btn_manuel=false;
    }
    else {
```

```
    digitalWrite(relayPin,LOW);
    btn_manuel=false;
    }
  }
  else {
  if (btn_manuel==true){
    digitalWrite(relayPin,HIGH);
    delay(1500);
    digitalWrite(relayPin,LOW);
    btn_manuel=false;
  }
  else {
    digitalWrite(relayPin,LOW);
  }
  }
  LED_Scenario();
  brumisateur();
  LCD_Scenario();
  Serial.println("Ca compile  en tout cas ");
  tb.loop();
}


void LED_Scenario(){
  float temp=bme.readTemperature();
  float press=bme.readPressure()/100;
  float humi_air=bme.readHumidity();
  int humi=map(analogRead(moisturePin),930,3630,100,0);
  if (temp>25){
  fill_solid(led_Temp,NUM_LED,CRGB::Yellow);
  FastLED.show();
}
else {
fill_solid(led_Temp,NUM_LED,CRGB::Orange);
  FastLED.show();
}
if (press>1000){
  fill_solid(led_Press,NUM_LED,CRGB::Green);
  FastLED.show();
}
else {
fill_solid(led_Press,NUM_LED,CRGB::White);
  FastLED.show();
}
if (humi>80) {
fill_solid(led_Humi,NUM_LED,CRGB::Blue);
  FastLED.show();
}
else {
```

```
fill_solid(led_Humi,NUM_LED,CRGB::Pink);
  FastLED.show();
}
if (humi_air>50) {
fill_solid(led_Humi_R,NUM_LED,CRGB::Red);
  FastLED.show();
}
else {
fill_solid(led_Humi_R,NUM_LED,CRGB::Purple );
  FastLED.show();
  }
}

void brumisateur(){
  float humi_air=bme.readHumidity();
  if (humi_air<40){
    digitalWrite(Brumi,HIGH);
    delay(3500);
    digitalWrite(Brumi,LOW);
  }
  else {
    digitalWrite(Brumi,LOW);
  }
}

void LCD_Scenario(){
  float temp=bme.readTemperature();
  float press=bme.readPressure()/100;
  float humi_air=bme.readHumidity();
  int humi=map(analogRead(moisturePin),930,3630,100,0);
    lcd.clear();
    lcd.setCursor(0,0);
    lcd.print("Marathon Creatif");
    lcd.setCursor(0,1);
    lcd.print("Lycee de Serres");
    delay(1000);

    lcd.clear();
    lcd.setCursor(2,0);
    lcd.print("Temperature:");
    lcd.setCursor(5,1);
    lcd.print(temp);
    lcd.print("C");
    delay(1000);

    lcd.clear();
    lcd.setCursor(1,0);
    lcd.print("Humidite sol :");
    lcd.setCursor(6,1);
```

```
    lcd.print(humi);
    lcd.print("%");
    delay(1000);

   lcd.clear();
    lcd.setCursor(1,0);
    lcd.print("Humidite air : ");
    lcd.setCursor(5,1);
    lcd.print(humi_air);
    lcd.print("%");
    delay(1000);

    lcd.clear();
    lcd.setCursor(3,0);
    lcd.print("Pression : ");
    lcd.setCursor(3,1);
    lcd.print(press);
    lcd.print("hPa");
    delay(1000);
}
```