

Dakar Institute of Technology

Projet IoT pour la Pollution de l'Air

Pour le cours d'Internet des Objets

Marie-Ange Dieng
29/03/2025

Table des matières

Introduction.....	3
A. Contexte	3
B. Problématique.....	3
I. Matériel Utilisé	3
A. ESP32	3
B. Capteur de Température et d'Humidité - DHT22	4
C. Capteur de Gaz - MQ2	4
D. Capteur de Qualité de l'Air - MQ135 (Personnalisé)	4
1. Code Json du MQ135	5
2. Code C du MQ135.....	5
E. Interconnexion du mécanisme.....	6
1. Schéma de connexion	6
2. Détails des connexions	6
II. Mécanisme de fonctionnement du système	8
A. Sur Wokwi.....	8
1. Inclusion des bibliothèques	8
2. Définition des broches des capteurs	9
3. Connexion Wi-Fi	9
4. Définition des constantes pour la connection à Thingsboard	9
5. Définition des constantes pour la connection à Google Colab	9
6. Initialisation des objets pour les connexions	9
7. Définition d'un message MQTT.....	10
8. Définition de la fonction <i>void setup()</i>	10
9. Définition de la fonction <i>void reconnect (PubSubClient &client, const char* token = nullptr)</i> ..	11
10. Définition de la fonction <i>void loop()</i>	11
11. Code Wokwi Intégrale.....	13
B. Sur Colab.....	16
1. Fichier traitement IA	16
2. Fichier d'inférence externalisé	17
3. Intégralité du code d'inférence sur Google Collab.....	19
C. Sur Thingsboard	21
1. On va dans l'onglet Dispositif de notre compte ThingsBoard et on crée un nouveau dispositif ..	21
2. Définir les caractéristiques et sauvegarder l'accès Token.....	22
3. Ensuite on retrouve notre dispositif dans la liste des dispositifs	22
4. L'onglet « Dernière Télémétrie » nous donne les dernières clés reçues et heures de réception ...	23
5. La page « Tableaux de bord » nous permet de créer des dashboard pour visualiser nos données	23

6.	A l'intérieur d'un même DashBoard on peut retrouver plusieurs graphiques.....	24
III.	Améliorations Possibles.....	26
A.	Optimisation du Modèle de Détection d'Anomalies.....	26
B.	Amélioration de l'Infrastructure IoT.....	26
C.	Affinage des Seuils d'Anomalie	26
D.	Extension des Fonctionnalités.....	26
	Conclusion	27

Introduction

A. Contexte

La pollution de l'air est un enjeu environnemental et sanitaire majeur, particulièrement dans les grandes villes du Sénégal comme Dakar. La croissance démographique, l'urbanisation rapide et l'augmentation du trafic routier contribuent à la dégradation de la qualité de l'air, exposant la population à divers risques de santé tels que les maladies respiratoires et cardiovasculaires. Face à ce défi, la mise en place d'un système intelligent de surveillance en temps réel devient essentielle pour mieux comprendre les niveaux de pollution et aider à la prise de décisions informées.

B. Problématique

Les méthodes traditionnelles de mesure de la qualité de l'air sont souvent coûteuses, peu accessibles et ne permettent pas un suivi en temps réel des variations de pollution. De plus, l'absence de solutions connectées limite la capacité à analyser et anticiper les fluctuations des polluants. Comment concevoir un système IoT intelligent, capable de mesurer et d'analyser la qualité de l'air en temps réel, tout en intégrant une IA embarquée pour améliorer la précision des mesures et optimiser la gestion des données ?

Ce projet de fin de module vise ainsi à développer un dispositif basé sur un **ESP32**, des **capteurs de pollution**, l'**IA embarquée**, le protocole **MQTT** et la plateforme **ThingsBoard** pour la visualisation et l'analyse des données, afin de proposer une solution accessible et efficace pour la surveillance de la qualité de l'air au Sénégal.

I. Matériel Utilisé

Pour la mise en place de notre système de surveillance de la qualité de l'air, nous avons utilisé plusieurs composants électroniques adaptés à une architecture IoT. Étant donné que notre projet est simulé sur **Wokwi**, certains capteurs ont dû être personnalisés pour correspondre aux besoins spécifiques de notre étude.

A. ESP32

L'ESP32 est le microcontrôleur central de notre système. Il a été choisi pour ses capacités avancées en termes de connectivité Wi-Fi et MQTT, permettant ainsi d'envoyer en temps réel les données des capteurs vers les plateformes de visualisation et d'analyse. Sa faible consommation énergétique et sa puissance de calcul en font un choix idéal pour l'intégration d'une **IA embarquée** dans un futur développement du projet.



B. Capteur de Température et d'Humidité - DHT22

Le DHT22 est un capteur numérique qui mesure la température et l'humidité de l'air. Il a été intégré à notre système pour fournir des données environnementales essentielles, influençant les niveaux de pollution atmosphérique. Ces mesures sont également utilisées comme entrées dans notre modèle d'IA afin de prédire les concentrations de polluants.



C. Capteur de Gaz - MQ2

Le MQ2 est un capteur de gaz qui permet de détecter divers gaz inflammables comme le butane, propane, méthane et hydrogène. Il est utilisé dans notre système pour mesurer la présence de gaz dangereux pouvant affecter la qualité de l'air intérieur et extérieur.



D. Capteur de Qualité de l'Air - MQ135 (Personnalisé)

Le MQ135 est généralement utilisé pour mesurer la concentration de gaz polluants tels que l'ammoniac (NH_3), le dioxyde de carbone (CO_2), l'oxyde d'azote (NO_x) et le benzène. Cependant, Wokwi ne disposant pas de ce capteur, nous avons opté pour une simulation personnalisée, basée sur le code de Misbah Juwayriyyah, afin de reproduire son comportement et d'assurer la cohérence des mesures dans notre environnement simulé.



1. Code Json du MQ135

```
{
  "name": "mq135",
  "author": "Misbah Juwayriyyah",
  "pins": [
    "VCC",
    "GND",
    "A0",
    "D0", "", "", ""
  ],
  "controls": [{
    "id": "ppm",
    "label": "PPM",
    "type": "slider",
    "min": 200,
    "max": 1000
  }]
}
```

2. Code C du MQ135

```
#include "wokwi-api.h"
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  pin_t pin;
  int ppm;
} chip_data_t;

void chip_timer_callback(void *data) {
  chip_data_t *chip_data = (chip_data_t*)data;
  chip_data->ppm = attr_read(chip_data->ppm);
  printf("%d\n", chip_data->ppm);

  float volts = chip_data->ppm * 5.0 / 1000;

  printf("%d\n", chip_data->ppm);
  printf("%f\n", volts);

  pin_dac_write(chip_data->pin, volts);
}

void chip_init() {
  chip_data_t *chip_data = (chip_data_t*)malloc(sizeof(chip_data_t));
  chip_data->ppm = attr_init("ppm", 400);
  chip_data->pin = pin_init("A0", ANALOG);

  const timer_config_t config = {
    .callback = chip_timer_callback,
    .user_data = chip_data,
  };
};
```

```

timer_t timer_id = timer_init(&config);
timer_start(timer_id, 1000, true);

printf("Hello from mq-135 chip!\n");
}

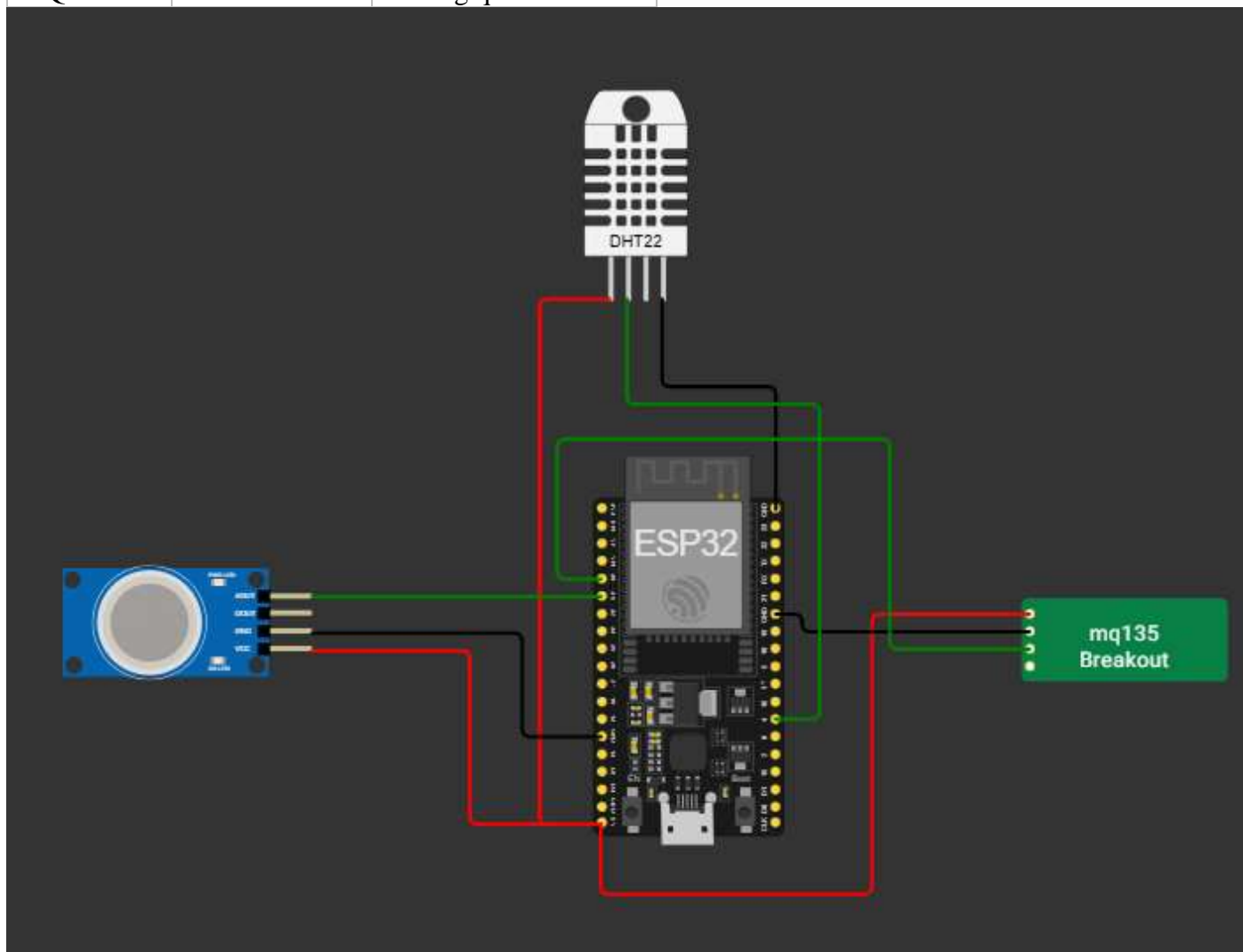
```

E. Interconnexion du mécanisme

1. Schéma de connexion

Les connexions entre l'ESP32 et les capteurs ont été réalisées comme suit :

Composant	Broche ESP32	Type de Connexion
DHT22	GPIO 4	Numérique (OneWire)
MQ2	GPIO 35	Analogique
MQ135	GPIO 34	Analogique

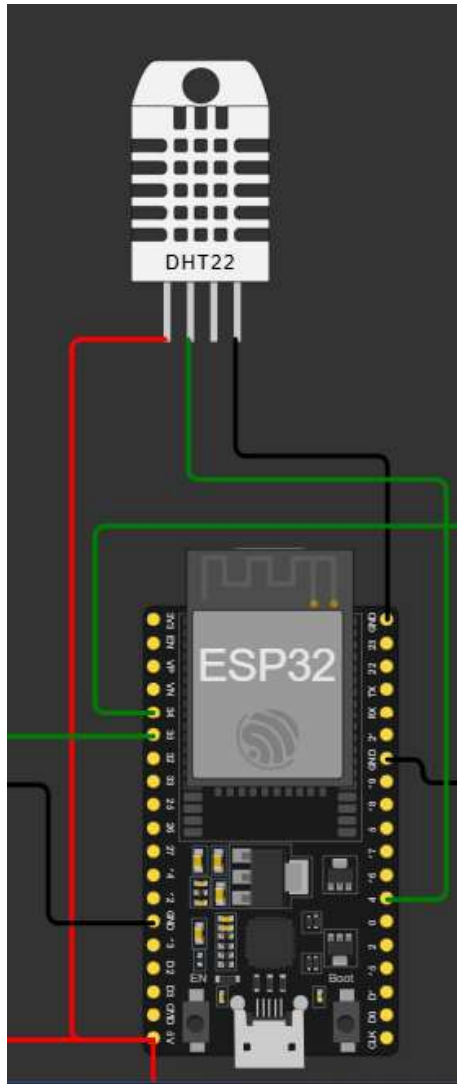


2. Détails des connexions

a. DHT22

Il nécessite une seule broche pour la transmission des données.

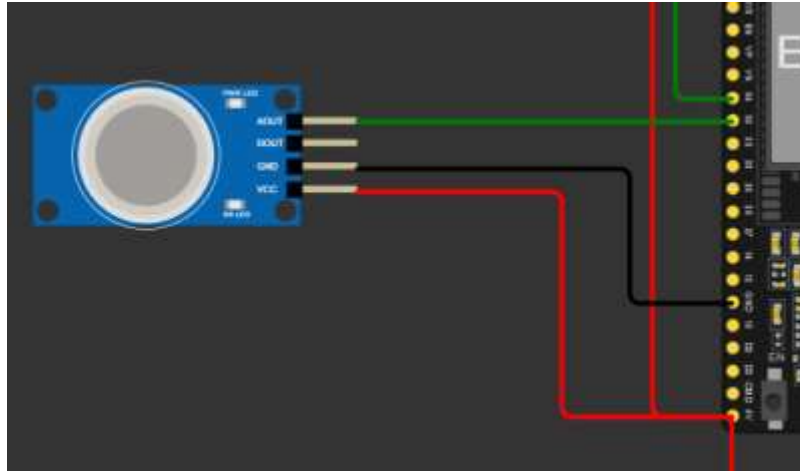
- **VCC** → 5V
- **GND** → GND
- **DATA** → GPIO 4



b. MQ2 Gas Sensor

Le capteur **MQ2** fonctionne en mode analogique et détecte la présence de gaz inflammables (butane, propane, méthane, etc.). Il délivre une tension proportionnelle à la concentration des gaz détectés.

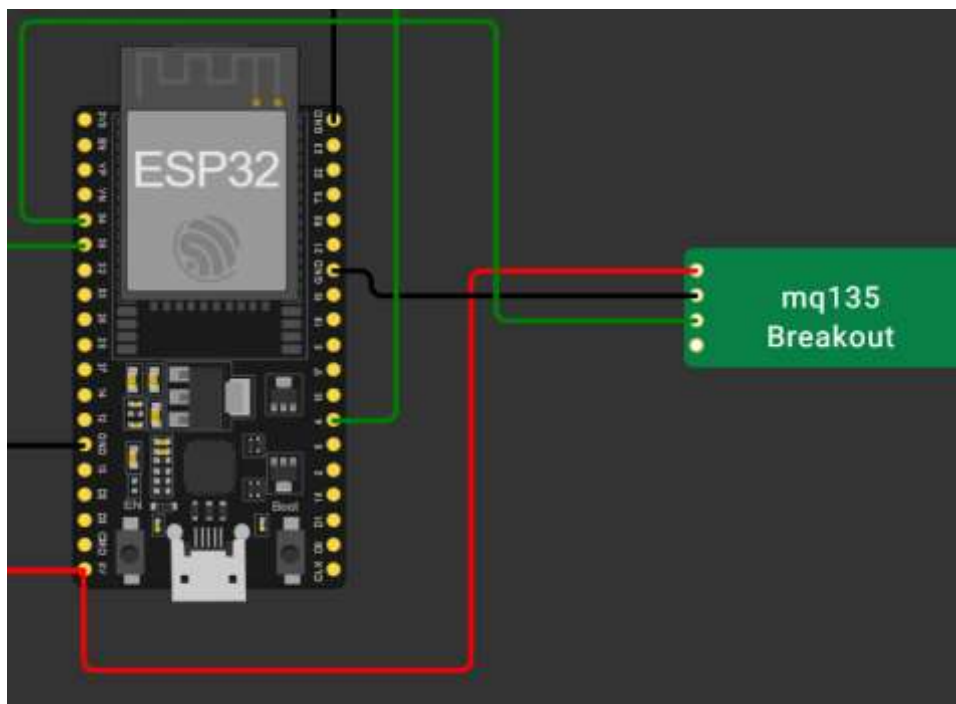
- **VCC** → 5V
- **GND** → GND
- **A0 (Sortie Analogique)** → GPIO 35



c. MQ135

Comme le **MQ2**, il fournit une sortie analogique.

- **VCC** → 3.3V
- **GND** → GND
- **A0 (Sortie Analogique)** → GPIO 34



II. Mécanisme de fonctionnement du système

A. Sur Wokwi

1. Inclusion des bibliothèques

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

- **WiFi.h** : Permet la connexion du microcontrôleur ESP32 au réseau Wi-Fi.

- **PubSubClient.h** : Bibliothèque pour utiliser le protocole MQTT, qui permet d'envoyer et de recevoir des messages via un broker.
- **DHT.h** : Bibliothèque pour lire les valeurs du capteur de température et d'humidité DHT22.

2. Définition des broches des capteurs

```
#define DHTPIN 4
#define DHTTYPE DHT22
#define MQ2PIN 35
#define MQ135PIN 34
```

- **DHTPIN 4** : Le capteur DHT22 est connecté à la broche **GPIO 4** de l'ESP32.
- **DHTTYPE DHT22** : Spécifie le modèle du capteur.
- **MQ2PIN 35** : Le capteur MQ2 (détecteur de gaz) est branché sur la broche **GPIO 35**.
- **MQ135PIN 34** : Le capteur MQ135 (qualité de l'air) est sur la broche **GPIO 34**.

3. Connexion Wi-Fi

```
const char* ssid = "Wokwi-GUEST";
const char* password = "";
```

- **ssid** : Nom du réseau Wi-Fi auquel l'ESP32 doit se connecter (ici "Wokwi-GUEST" pour la simulation).
- **password** : Mot de passe du réseau (vide ici car le Wi-Fi Wokwi n'a pas de mot de passe).

4. Définition des constantes pour la connection à Thingsboard

```
const char* thingsboard_server = "demo.thingsboard.io";
const char* thingsboard_token = "7hxlh970qiug6qnu3t4z";
const int mqtt_port = 1883;
```

- **thingsboard_server** : Adresse du broker MQTT de ThingsBoard où les données seront envoyées.
- **thingsboard_token** : Clé d'authentification pour publier des données sur ThingsBoard.
- **mqtt_port** : Port par défaut pour le protocole MQTT (1883).

5. Définition des constantes pour la connection à Google Colab

```
const char* colab_server = "broker.hivemq.com";
const int colab_port = 1883;
const char* TOPIC_PUB = "esp32/data";
const char* TOPIC_SUB = "esp32/indicator";
```

- **colab_server** : Adresse du broker MQTT **HiveMQ**, qui est utilisé ici pour la communication entre l'ESP32 et Google Colab.
- **colab_port** : Port MQTT standard utilisé pour la connexion.
- **TOPIC_PUB** : Topic MQTT sur lequel l'ESP32 **publie** les données (esp32/data).
- **TOPIC_SUB** : Topic MQTT sur lequel l'ESP32 **écoute** (esp32/indicator).

6. Initialisation des objets pour les connexions

```
WiFiClient espClient;
PubSubClient clientThingsBoard(espClient);
PubSubClient clientColab(espClient);
DHT dht(DHTPIN, DHTTYPE);
```

- **espClient**: Crée un objet Wi-Fi pour gérer la connexion réseau.

- `clientThingsBoard(espClient)`: Objet MQTT utilisé pour la connexion à ThingsBoard via Wi-Fi.
- `clientColab(espClient)`: Objet MQTT utilisé pour la connexion à Google Colab via HiveMQ.
- `DHT dht(DHTPIN, DHTTYPE)` ; : Initialise le capteur DHT22 pour mesurer la température et l'humidité.

7. Définition d'un message MQTT

```
String MSG="\indicator\":0";
```

`MSG` est une chaîne de caractères JSON qui contient un indicateur avec une valeur initiale de 0. Nous verrons plus tard à quoi elle servira.

8. Définition de la fonction `void setup()`

a. Initialisation de la communication série

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

Cette ligne initialise la communication série. Cela permet d'afficher des messages sur le moniteur série pour suivre l'exécution du programme.

b. Connexion au Wi-Fi

```
WiFi.begin(ssid, password);
Serial.print("Connexion WiFi");
while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
}
Serial.println("\nConnecté au WiFi!");
```

- `WiFi.begin(ssid, password)`; démarre la connexion au réseau Wi-Fi en utilisant les identifiants (ssid et password).
- Une boucle `while` tourne en vérifiant l'état de la connexion avec `WiFi.status() != WL_CONNECTED`.
- Tant que l'ESP32 n'est pas connecté, le programme affiche des points (".") pour montrer qu'il tente toujours de se connecter.
- Une fois connecté, un message "Connecté au WiFi!" s'affiche dans le moniteur série.

c. Configuration de la connexion MQTT avec ThingsBoard

```
clientThingsBoard.setServer(thingsboard_server, mqtt_port)
```

d. Configuration de la connexion MQTT avec Google Colab

```
clientColab.setServer(colab_server, colab_port);
clientColab.setCallback(callback); // Fonction callback pour recevoir l'indicateur
```

- `clientColab.setServer(colab_server, colab_port)`; configure une **deuxième connexion MQTT**, cette fois avec **Google Colab** (colab_server).
- `clientColab.setCallback(callback)`; définit une **fonction callback** qui sera exécutée lorsqu'un message MQTT est reçu depuis **Google Colab**. Cela permet de récupérer un indicateur (par exemple, une alerte d'anomalie).

e. Initialisation du capteur DHT

```
dht.begin();
```

9. Définition de la fonction *void reconnect (PubSubClient &client, const char* token = nullptr)*

a. Boucle de reconnexion

```
while (!client.connected()) {
```

Tant que la connexion MQTT n'est pas établie (`client.connected()` retourne `false`), on continue d'essayer.

b. Tentative de connexion à Thingsboard

```
if (token) {  
    if (client.connect("ESP32_Client", token, "")) {  
        Serial.println("Connecté à ThingsBoard !");  
    }  
}
```

- Si un token est fourni (`token != nullptr`), on tente une connexion MQTT avec un identifiant ("ESP32_Client") et un mot de passe vide ("").
- Si la connexion réussit, on affiche "Connecté à ThingsBoard !".

c. Tentative de connexion à Google Colab

```
else {  
    if (client.connect("ESP32_Client")) {  
        Serial.println("Connecté à Google Colab !");  
        client.subscribe(TOPIC_SUB);  
    }  
}
```

- Si aucun token n'est fourni (`token == nullptr`), la connexion est établie sans identifiant ni mot de passe.
- Si elle réussit, on affiche "Connecté à Google Colab !" et on s'abonne à un topic MQTT (`TOPIC_SUB`), qui permet de recevoir des messages depuis Google Colab.

d. Gestion des échecs de connexion

```
if (!client.connected()) {  
    Serial.println("Échec, tentative dans 5s...");  
    delay(5000);  
}
```

- Si la connexion échoue, un message d'échec s'affiche.
- On attend 5 secondes avant de réessayer.

10. Définition de la fonction *void loop()*

a. Lecture des capteurs et ajout de bruit simulé

```
float temperature = addNoise(dht.readTemperature(), 4.0);  
float humidity = addNoise(dht.readHumidity(), 15.0);  
int mq2_value = addNoise(analogRead(MQ2PIN), 2000);  
int mq135_value = addNoise(analogRead(MQ135PIN), 2000);
```

- `dht.readTemperature()` et `dht.readHumidity()` récupèrent les valeurs du capteur DHT22 (température et humidité).
- `analogRead(MQ2PIN)` et `analogRead(MQ135PIN)` récupèrent les valeurs des capteurs MQ2 (gaz) et MQ135 (qualité de l'air).

- `addNoise(value, noiseLevel)` ajoute une petite variation aléatoire aux valeurs pour simuler des lectures de capteurs réalistes. Il fait appel à la fonction `addNoise` définie après :

```
template<typename T>
T addNoise(T value, float noiseLevel) {
    return value + ((random(-100, 101) / 100.0) * noiseLevel);
}
```

b. Affichage des valeurs lues sur le moniteur série

```
Serial.print("Température: "); Serial.print(temperature); Serial.println(" °C");
Serial.print("Humidité: "); Serial.print(humidity); Serial.println(" %");
Serial.print("MQ2 Gaz: "); Serial.println(mq2_value);
Serial.print("MQ135 Qualité de l'air: "); Serial.println(mq135_value);
```

Affiche les valeurs lues pour aider au débogage et au suivi en temps réel.

c. Création et envoi des données à Colab via MQTT

```
payload = "{";
payload += "\"temperature\": " + String(temperature) + ",";
payload += "\"humidity\": " + String(humidity) + ",";
payload += "\"mq2_gas\": " + String(mq2_value) + ",";
payload += "\"mq135_air_quality\": " + String(mq135_value);
payload += "}";
```

Crée un message JSON contenant les données des capteurs.

```
if (!clientColab.connected()) reconnect(clientColab);
clientColab.loop();
clientColab.publish(TOPIC_PUB, payload.c_str());
```

- Vérifie la connexion avec le serveur MQTT de Google Colab.
- Si la connexion est perdue, elle est rétablie avec `reconnect(clientColab);`.
- Publie les données sous forme de JSON sur le **topic** défini par `TOPIC_PUB`.
- La fonction `callback()` est utilisée pour traiter les messages MQTT reçus du serveur.

```
void callback(char* topic, byte* payload, unsigned int length) {
    Serial.print("Message reçu sur ");
    Serial.print(topic);
    Serial.print(": ");
```

Affiche sur quel **topic** MQTT un message a été reçu.

```
String message;
for (unsigned int i = 0; i < length; i++) {
    message += (char)payload[i];
}
```

Convertit le **payload** (le message brut reçu) en une **chaîne de caractères**

```
MSG = message.substring(1, message.length() - 1);
Serial.println(message);
```

- **Stocke** le message reçu dans la variable `MSG` (sans les crochets `{ }` autour du message reçu).
- **Affiche** le message sur le moniteur série.

d. Création et envoi des données à Thingsboard via MQTT

```
String payload = "{";
```

```

payload += "\"temperature\":" + String(temperature) + ",";
payload += "\"humidity\":" + String(humidity) + ",";
payload += "\"mq2_gas\":" + String(mq2_value) + ",";
payload += "\"mq135_air_quality\":" + String(mq135_value)+ ",";
payload += MSG;
payload += "}";

```

Crée un second message JSON similaire, mais il inclut également `MSG`, une variable contenant les messages reçus du serveur MQTT depuis Colab.

```

if (!clientThingsBoard.connected()) reconnect(clientThingsBoard, thingsboard_token);

clientThingsBoard.loop();

```

- Vérifie si la connexion MQTT avec ThingsBoard est active.
- Si elle est coupée, la fonction `reconnect(clientThingsBoard, thingsboard_token);` est appelée pour rétablir la connexion.
- `clientThingsBoard.loop();` permet au client MQTT de maintenir la communication avec le serveur et de traiter les messages entrants.

```

Serial.print("Envoi MQTT: "); Serial.println(payload);
clientThingsBoard.publish("v1/devices/me/telemetry", payload.c_str())

```

- Affiche le message avant de l'envoyer.
- Publie les données sur **ThingsBoard** sur le topic `"v1/devices/me/telemetry"`.

e. Attente avant la prochaine itération

```
delay(5000);
```

Attend **5 secondes** avant de recommencer une nouvelle itération.

11. Code Wokwi Intégrale

```

#include <WiFi.h>
#include <PubSubClient.h>
#include <DHT.h>

// --- Définition des broches des capteurs ---
#define DHTPIN 4
#define DHTTYPE DHT22
#define MQ2PIN 35
#define MQ135PIN 34

// --- Connexion Wi-Fi ---
const char* ssid = "Wokwi-GUEST";
const char* password = "";

// --- Connexion MQTT ---
const char* thingsboard_server = "demo.thingsboard.io";
const char* thingsboard_token = "7hxlh970qiug6qnu3t4z";
const int mqtt_port = 1883;

// --- Connexion MQTT à Google Colab (utilise ton IP locale) ---
const char* colab_server = "broker.hivemq.com";
const int colab_port = 1883;
const char* TOPIC_PUB = "esp32/data";

```

```

const char* TOPIC_SUB = "esp32/indicator";

// --- Initialisation des objets ---
WiFiClient espClient;
PubSubClient clientThingsBoard(espClient);
PubSubClient clientColab(espClient);
DHT dht(DHTPIN, DHTTYPE);

String MSG="\indicator\":0";

void setup() {
    Serial.begin(115200);

    // Connexion Wi-Fi
    WiFi.begin(ssid, password);
    Serial.print("Connexion WiFi");
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("\nConnecté au WiFi!");

    // Configuration MQTT ThingsBoard
    clientThingsBoard.setServer(thingsboard_server, mqtt_port);

    // Configuration MQTT Google Colab
    clientColab.setServer(colab_server, colab_port);
    clientColab.setCallback(callback); // Fonction callback pour recevoir l'indicateur

    // Initialisation du capteur DHT
    dht.begin();
}

void reconnect(PubSubClient &client, const char* token = nullptr) {
    while (!client.connected()) {
        Serial.print("Connexion au serveur MQTT...");
        if (token) {
            if (client.connect("ESP32_Client", token, "")) {
                Serial.println("Connecté à ThingsBoard !");
            }
        } else {
            if (client.connect("ESP32_Client")) {
                Serial.println("Connecté à Google Colab !");
                client.subscribe(TOPIC_SUB);
            }
        }
        if (!client.connected()) {
            Serial.println("Échec, tentative dans 5s...");
            delay(5000);
        }
    }
}

```

```

// Fonction pour ajouter du bruit à une valeur (simule la variabilité en simulation)
template<typename T>
T addNoise(T value, float noiseLevel) {
    return value + ((random(-100, 101) / 100.0) * noiseLevel);
}

void loop() {

    // --- Lecture des capteurs ---
    float temperature = addNoise(dht.readTemperature(), 4.0);
    float humidity = addNoise(dht.readHumidity(), 15.0);
    int mq2_value = addNoise(analogRead(MQ2PIN), 2000);
    int mq135_value = addNoise(analogRead(MQ135PIN), 2000);

    Serial.print("Température: "); Serial.print(temperature); Serial.println(" °C");
    Serial.print("Humidité: "); Serial.print(humidity); Serial.println(" %");
    Serial.print("MQ2 Gaz: "); Serial.println(mq2_value);
    Serial.print("MQ135 Qualité de l'air: "); Serial.println(mq135_value);

    // --- Création du JSON pour Colab ---
    payload = "{";
    payload += "\"temperature\": " + String(temperature) + ",";
    payload += "\"humidity\": " + String(humidity) + ",";
    payload += "\"mq2_gas\": " + String(mq2_value) + ",";
    payload += "\"mq135_air_quality\": " + String(mq135_value);
    payload += "}";

    if (!clientColab.connected()) reconnect(clientColab);
    clientColab.loop();
    clientColab.publish(TOPIC_PUB, payload.c_str());

    // --- Création du JSON pour Thingsboard ---
    String payload = "{";
    payload += "\"temperature\": " + String(temperature) + ",";
    payload += "\"humidity\": " + String(humidity) + ",";
    payload += "\"mq2_gas\": " + String(mq2_value) + ",";
    payload += "\"mq135_air_quality\": " + String(mq135_value) + ",";
    payload += MSG;
    payload += "}";

    // Vérifier la connexion MQTT
    if (!clientThingsBoard.connected()) reconnect(clientThingsBoard, thingsboard_token);
    clientThingsBoard.loop();

    Serial.print("Envoi MQTT: "); Serial.println(payload);

    // Envoi des données aux deux serveurs
    clientThingsBoard.publish("v1/devices/me/telemetry", payload.c_str());
}

```



```

    delay(5000);
}

void callback(char* topic, byte* payload, unsigned int length) {
    Serial.print("Message reçu sur ");
    Serial.print(topic);
    Serial.print(": ");

    String message;
    for (unsigned int i = 0; i < length; i++) {
        message += (char)payload[i];
    }
    MSG = message.substring(1, message.length() - 1);
    Serial.println(message);
}

```

B. Sur Colab

Au départ, seule la connexion avec ThingsBoard était établie pour collecter les données empiriques. Une fois qu'un volume suffisant de données a été atteint, nous avons utilisé l'API REST de ThingsBoard pour les extraire au format JSON. Ces données ont ensuite été converties en CSV et utilisées pour entraîner le modèle.

1. Fichier traitement IA

```

import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow import keras
df = pd.read_csv("data.csv")
df.drop(columns=['timestamp'], inplace=True)
X = df[['temperature', 'humidity']]
y = df[['mq2_gas', 'mq135_air_quality']]
scaler = MinMaxScaler()
X_norm = scaler.fit_transform(X)
y_norm = scaler.fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(X_norm, y_norm, test_size=0.2,
random_state=42)
model = keras.Sequential([
    keras.layers.Dense(8, activation='relu', input_shape=(2,)),
    keras.layers.Dense(4, activation='relu'),
    keras.layers.Dense(8, activation='relu'),
    keras.layers.Dense(2, activation='sigmoid')
])
model.compile(optimizer='rmsprop', loss='mse', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=50, batch_size=16, validation_split=0.2)

```

```

predictions = model.predict(X_test)
mq2_residuals = abs(y_test[:, 0] - predictions[:, 0])
mq135_residuals = abs(y_test[:, 1] - predictions[:, 1])
mq2_threshold = np.percentile(mq2_residuals, 95)
mq135_threshold = np.percentile(mq135_residuals, 95)
# Conversion en modèle TensorFlow Lite
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Sauvegarde du modèle TFLite
with open("anomaly_detector.tflite", "wb") as f:
    f.write(tflite_model)

print("Modèle converti en TensorFlow Lite et sauvegardé.")

```

Le modèle d'entraînement défini ici est un réseau de neurones conçu pour prédire les valeurs des capteurs de gaz (MQ2 et MQ135) en fonction des mesures de température et d'humidité. Les données sont d'abord prétraitées : la colonne *timestamp* est supprimée, et les valeurs des capteurs sont normalisées avec *MinMaxScaler* pour s'assurer qu'elles sont dans une plage uniforme. Ensuite, les données sont divisées en ensembles d'entraînement et de test selon une répartition 80/20. Le modèle lui-même est un réseau de neurones avec une architecture simple : une couche d'entrée de taille 2 (pour la température et l'humidité), suivie de trois couches cachées de 8, 4 et 8 neurones avec des fonctions d'activation *ReLU*, et une couche de sortie de 2 neurones activés par *sigmoid*, correspondant aux prédictions des capteurs de gaz. L'entraînement se fait sur 50 époques avec une descente de gradient optimisée par *RMSprop* et une fonction de perte basée sur l'erreur quadratique moyenne (*MSE*), couramment utilisée pour les problèmes de régression.

Le principe des *thresholds* repose sur la détection des anomalies. Une fois le modèle entraîné, il est utilisé pour faire des prédictions sur l'ensemble de test. Ensuite, on calcule la différence absolue entre les valeurs réelles (*y_test*) et les valeurs prédites pour chaque capteur. Ces différences, appelées *résidus*, permettent d'évaluer l'écart entre les prédictions et la réalité. Pour déterminer un seuil d'anomalie, on utilise le 95^e percentile des résidus, ce qui signifie que toute erreur dépassant cette valeur est considérée comme une anomalie. Cela permet d'identifier les cas où les mesures des capteurs sont inhabituellement éloignées des valeurs attendues, ce qui pourrait signaler un problème dans l'environnement surveillé.

2. Fichier d'inférence externalisé

Ce code réalise l'inférence du modèle TensorFlow Lite en externalisant le processus via un serveur Python qui communique avec l'ESP32 via MQTT. Comme Wokwi a une limite de traitement trop courte pour exécuter le modèle directement sur l'ESP32, l'inférence est effectuée sur une machine externe, et seuls les résultats sont envoyés à l'ESP32.

```

import paho.mqtt.client as mqtt
import tensorflow.lite as tflite
import numpy as np
import json

```

- *paho-mqtt* est utilisé pour établir la communication MQTT entre l'ESP32 et le serveur Python.
- *tensorflow.lite* permet de charger et d'exécuter le modèle TFLite pour l'inférence.
- *numpy* est utilisé pour manipuler les données numériques, notamment pour la mise en forme des entrées du modèle.
- *json* est utilisé pour décoder les messages reçus et envoyer les résultats sous format JSON.

```

interpreter = tflite.Interpreter(model_path="/content/drive/MyDrive/Colab
Notebooks/anomaly_detector.tflite")

```

```
interpreter.allocate_tensors()
```

- *tflite.Interpreter* charge le modèle TFLite préalablement entraîné.
- `allocate_tensors()` réserve la mémoire pour les tenseurs d'entrée et de sortie, une étape obligatoire avant d'utiliser le modèle.

```
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

Ces lignes récupèrent les indices des tenseurs d'entrée et de sortie du modèle, qui seront utilisés pour passer les données en entrée et récupérer les résultats de l'inférence.

```
BROKER = "broker.hivemq.com"
PORT = 1883
TOPIC_SUB = "esp32/data"
TOPIC_PUB = "esp32/indicator"
```

- Définition du broker MQTT (HiveMQ public dans ce cas).
- `TOPIC_SUB` est le topic où l'ESP32 envoie ses données de capteur.
- `TOPIC_PUB` est le topic où le serveur Python envoie les résultats de l'inférence.

```
mq2_threshold = 0.37
mq135_threshold = 0.67
```

- Ces seuils (*thresholds*) ont été calculés lors de l'entraînement du modèle (95^e percentile des résidus).
- Si l'erreur entre la valeur réelle et la valeur prédite dépasse ces seuils, alors on détecte une anomalie.

```
def on_connect(client, userdata, flags, rc):
    print("Connecté au broker MQTT !")
    client.subscribe(TOPIC_SUB)
```

- Cette fonction est appelée lorsque le client se connecte au broker.
- Une fois connecté, il s'abonne au topic `esp32/data` pour recevoir les données envoyées par l'ESP32.

```
def on_message(client, userdata, msg):
    print(f"Message reçu : {msg.payload.decode()}")
    try:
        data = json.loads(msg.payload.decode())
```

- Cette fonction est exécutée lorsqu'un message est reçu sur `esp32/data`.
- Le message est décodé et converti en un dictionnaire Python (*json.loads*)

```
temp = (data["temperature"] - (-4.9)) / (59.6 - (-4.9))
hum = (data["humidity"] - 36) / (95 - 36)
actual_mq2 = (data["mq2_gas"] - 873) / (5588 - 873)
actual_mq135 = (data["mq135_air_quality"] - 0) / (65416 - 0)
```

- Normalisation des données pour correspondre aux échelles utilisées lors de l'entraînement du modèle (MinMaxScaler).
- Les valeurs minimales et maximales des capteurs proviennent de l'échantillon d'entraînement.

```
input_data = np.array([[temp, hum]], dtype=np.float32)
interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()
output_data = interpreter.get_tensor(output_details[0]['index'])
predicted_mq2, predicted_mq135 = output_data[0]
```

- `input_data` est transformé en un tableau NumPy de type `float32` (format requis par TFLite).
- `set_tensor()` place les données d'entrée dans le modèle.
- `invoke()` exécute le modèle et produit une prédiction.
- `get_tensor()` récupère les résultats de l'inférence (*mq2* et *mq135* prédits).

```
mq2_residual = abs(actual_mq2 - predicted_mq2)
mq135_residual = abs(actual_mq135 - predicted_mq135)
```

On calcule l'écart absolu (*résidu*) entre la valeur réelle et la valeur prédite pour chaque capteur.

```
if mq2_residual > mq2_threshold and mq135_residual > mq135_threshold:
    indicator = 3
elif mq2_residual > mq2_threshold:
    indicator = 1
elif mq135_residual > mq135_threshold:
    indicator = 2
else:
    indicator = 0
```

indicator est défini comme suit :

- 3 : anomalies détectées sur *MQ2* et *MQ135*.
- 1 : anomalie sur *MQ2* uniquement.
- 2 : anomalie sur *MQ135* uniquement.
- 0 : aucune anomalie détectée.

```
client.publish(TOPIC_PUB, json.dumps({"indicator": indicator}))
```

- L'indicateur est converti en JSON et publié sur `esp32/indicator`.
- L'ESP32 pourra récupérer cet indicateur et réagir en conséquence.

```
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect(BROKER, PORT, 60)
client.loop_forever()
```

- Création d'un client MQTT.
- Assignment des fonctions de connexion et de réception de messages.
- Connexion au broker.
- `loop_forever()` maintient l'écoute des messages en continu.

3. Intégralité du code d'inférence sur Google Collab

```
!pip install paho-mqtt tensorflow
import paho.mqtt.client as mqtt
import tensorflow.lite as tflite
```

```

import numpy as np
import json

# --- Charger le modèle TFLite ---
interpreter = tf.lite.Interpreter(model_path="/content/drive/MyDrive/Colab
Notebooks/anomaly_detector.tflite")
interpreter.allocate_tensors()

# Récupération des indices des tensors
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# --- Configuration du MQTT ---
BROKER = "broker.hivemq.com"
PORT = 1883
TOPIC_SUB = "esp32/data"
TOPIC_PUB = "esp32/indicator"

# Définition des seuils pour détecter les anomalies
mq2_threshold = 0.37
mq135_threshold = 0.67

def on_connect(client, userdata, flags, rc):
    print("Connecté au broker MQTT !")
    client.subscribe(TOPIC_SUB)

def on_message(client, userdata, msg):
    print(f"Message reçu : {msg.payload.decode()}")
    try:
        data = json.loads(msg.payload.decode())
        # Normalisation des données avec les minmax de l'échantillon d'entraînement
        temp = (data["temperature"] - (-4.9)) / (59.6 - (-4.9))
        hum = (data["humidity"] - 36) / (95 - 36)
        actual_mq2 = (data["mq2_gas"] - 873) / (5588 - 873)
        actual_mq135 = (data["mq135_air_quality"] - 0) / (65416 - 0)

        # Mise en forme pour le modèle (TensorFlow Lite attend un array numpy)
        input_data = np.array([[temp, hum]], dtype=np.float32)
        interpreter.set_tensor(input_details[0]['index'], input_data)
        interpreter.invoke()
        output_data = interpreter.get_tensor(output_details[0]['index'])

        predicted_mq2, predicted_mq135 = output_data[0]

        # Calcul des résidus
        mq2_residual = abs(actual_mq2 - predicted_mq2)
        mq135_residual = abs(actual_mq135 - predicted_mq135)

        # Détermination de l'indicateur
        if mq2_residual > mq2_threshold and mq135_residual > mq135_threshold:
            indicator = 3
    except:
        pass

```

```

elif mq2_residual > mq2_threshold:
    indicator = 1
elif mq135_residual > mq135_threshold:
    indicator = 2
else:
    indicator = 0

print(f"Résultat de l'inférence : MQ2 = {predicted_mq2:.2f}, MQ135 = {predicted_mq135:.2f}")
print(f"Résidus : MQ2 = {mq2_residual:.2f}, MQ135 = {mq135_residual:.2f}")
print(f"Indicateur : {indicator}")

# Publier l'indicateur sur MQTT
client.publish(TOPIC_PUB, json.dumps({"indicator": indicator}))

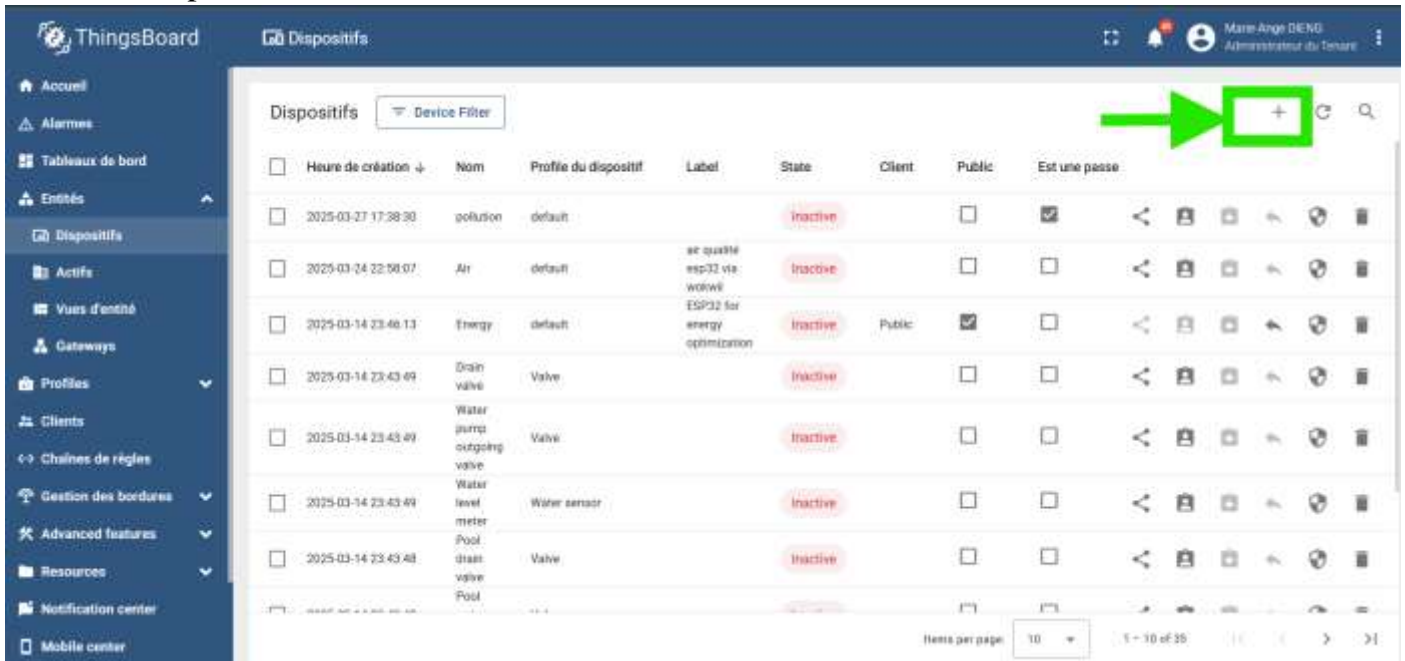
except Exception as e:
    print(f"Erreur : {e}")

# --- Gestion de la communication avec Wokwi
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect(BROKER, PORT, 60)
client.loop_forever()

```

C. Sur Thingsboard

1. On va dans l'onglet Dispositif de notre compte ThingsBoard et on crée un nouveau dispositif



The screenshot shows the ThingsBoard interface with the 'Dispositifs' (Devices) tab selected. The left sidebar contains navigation options like 'Accueil', 'Alarmes', 'Tableaux de bord', 'Entités', 'Dispositifs', 'Actifs', 'Vues d'entité', 'Gateways', 'Profilés', 'Clients', 'Chaînes de règles', 'Gestion des bordures', 'Advanced features', 'Resources', 'Notification center', and 'Mobile center'. The main area displays a table of devices with columns: 'Heure de création', 'Nom', 'Profilé du dispositif', 'Label', 'State', 'Client', 'Public', and 'Est une passe'. A green arrow points to a green '+' button in the top right corner of the device list, indicating where to click to add a new device.

Heure de création	Nom	Profilé du dispositif	Label	State	Client	Public	Est une passe
2025-03-27 17:38:30	pollution	default		inactive			<input checked="" type="checkbox"/>
2025-03-24 22:58:07	Air	default	air quality esp32 via wokwi	inactive			<input type="checkbox"/>
2025-03-14 23:46:13	Energy	default	ESP32 for energy optimization	inactive	Public	<input checked="" type="checkbox"/>	<input type="checkbox"/>
2025-03-14 23:43:49	Drain valve	Valve		inactive			<input type="checkbox"/>
2025-03-14 23:43:49	Water pump outgoing valve	Valve		inactive			<input type="checkbox"/>
2025-03-14 23:43:49	Water level meter	Water sensor		inactive			<input type="checkbox"/>
2025-03-14 23:43:48	Pool drain valve	Valve		inactive			<input type="checkbox"/>
2025-03-14 23:43:48	Pool			inactive			<input type="checkbox"/>

2. Définir les caractéristiques et sauvegarder l'accès Token

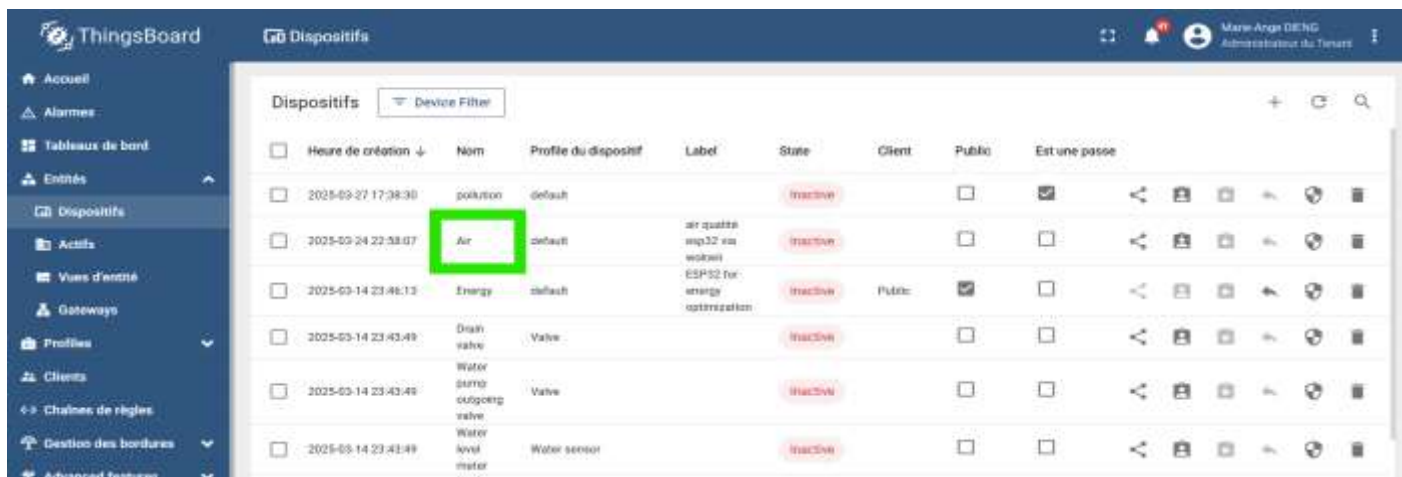
The screenshot shows the 'Ajouter un nouveau dispositif' (Add new device) dialog in the ThingsBoard interface. The dialog is divided into two tabs: '1 Détails du dispositif' (selected) and '2 Informations d'identification Optional'. In the 'Détails du dispositif' tab, the following fields are visible:

- Nom***: Air
- Label**: Verification air pour esp32
- Profil du dispositif***: default
- Est une passerelle**: ☐ (unchecked)
- Attribuer au client**: (empty field)
- Description**: (empty field)

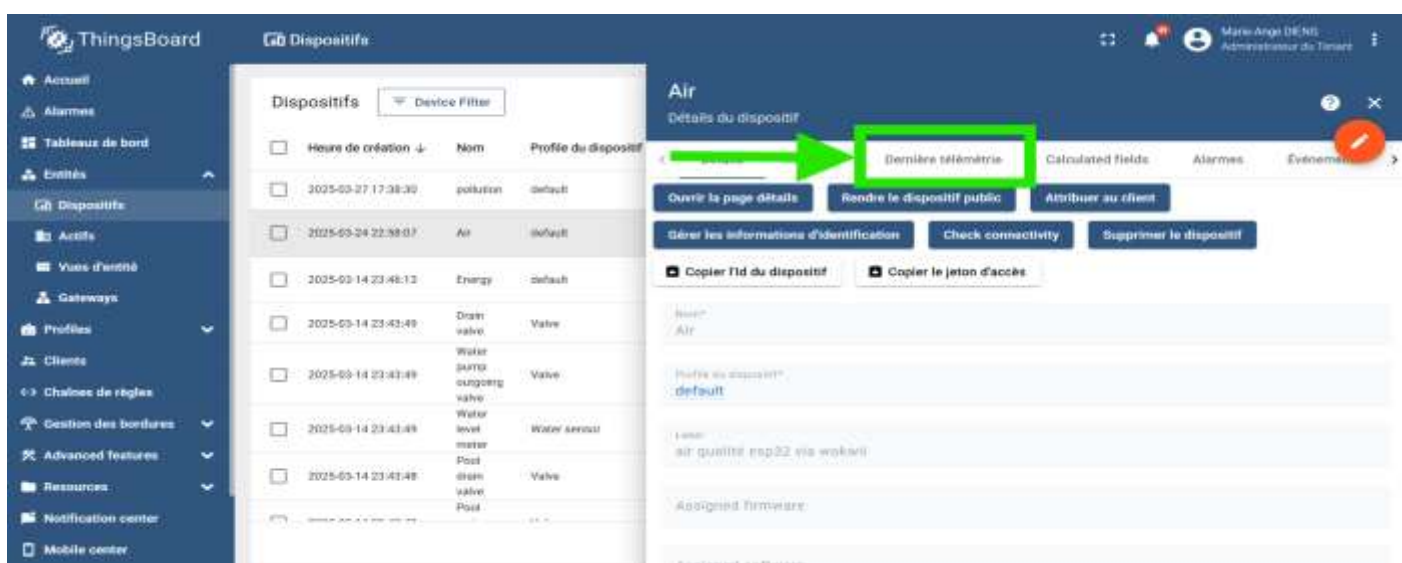
At the bottom of the dialog, there is a 'Suivant: Informations d'identification' button, an 'Annuler' (Cancel) button, and an 'Ajouter' (Add) button. The background shows the 'Dispositifs' (Devices) list in the ThingsBoard dashboard.

The screenshot shows the 'Ajouter un nouveau dispositif' dialog in the ThingsBoard interface, now on the '2 Informations d'identification Optional' tab. The 'Type d'identification' (Identification type) is set to 'Access token'. The 'Token d'accès*' (Access token) field contains a long alphanumeric string, which is partially obscured by a black bar. The 'Retour' (Back) button is now visible at the bottom left. The 'Annuler' (Cancel) and 'Ajouter' (Add) buttons remain at the bottom right. The background shows the 'Dispositifs' (Devices) list in the ThingsBoard dashboard.

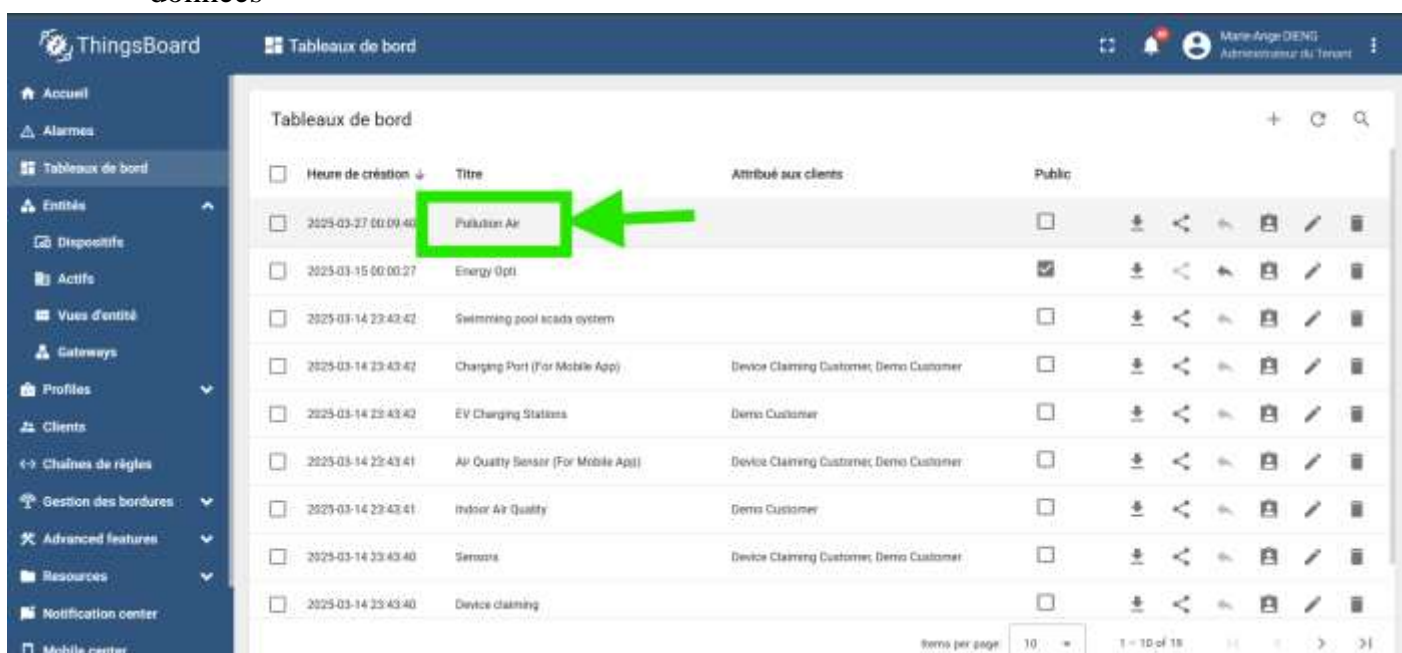
3. Ensuite on retrouve notre dispositif dans la liste des dispositifs



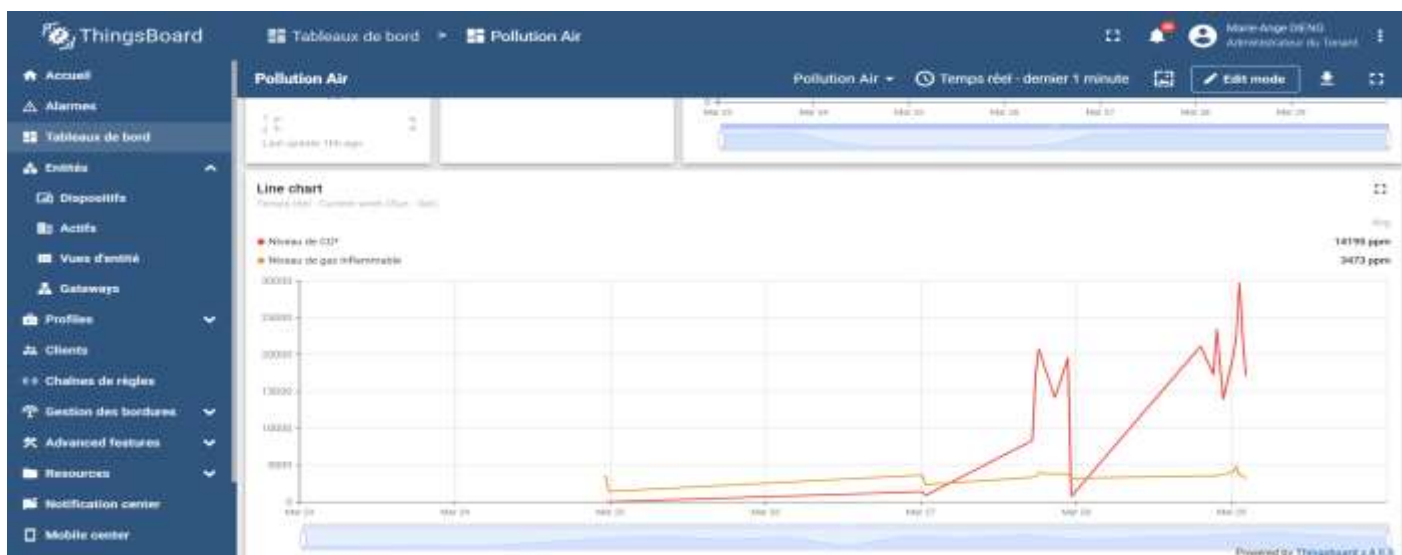
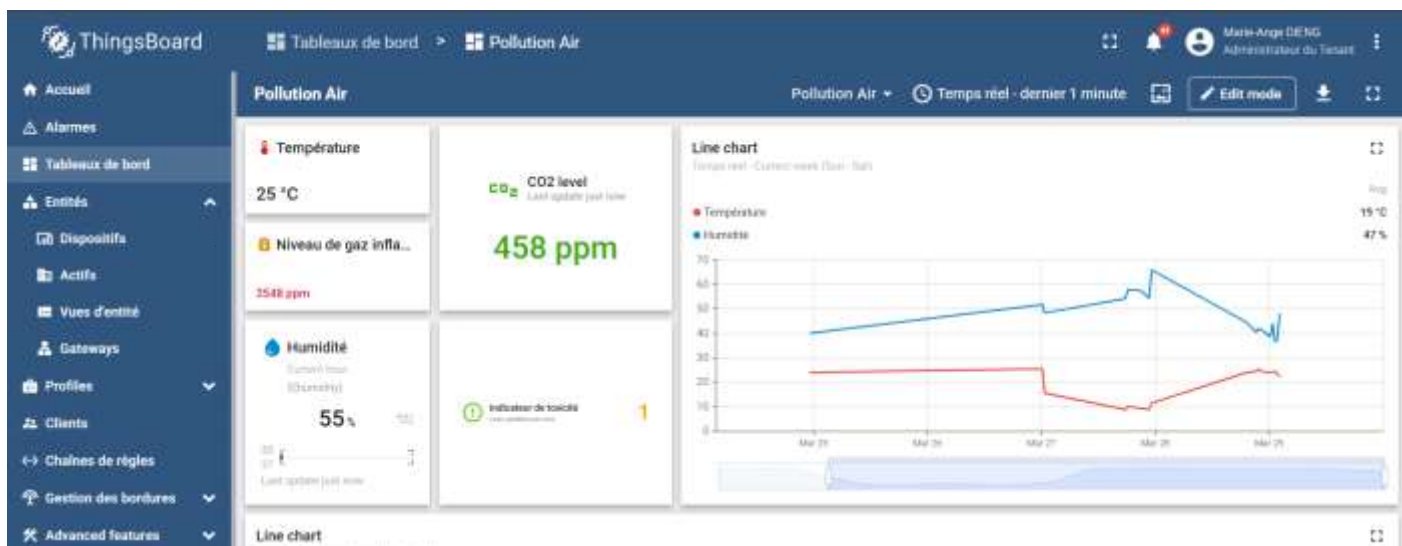
4. L'onglet « Dernière Télémétrie » nous donne les dernières clés reçues et heures de réception



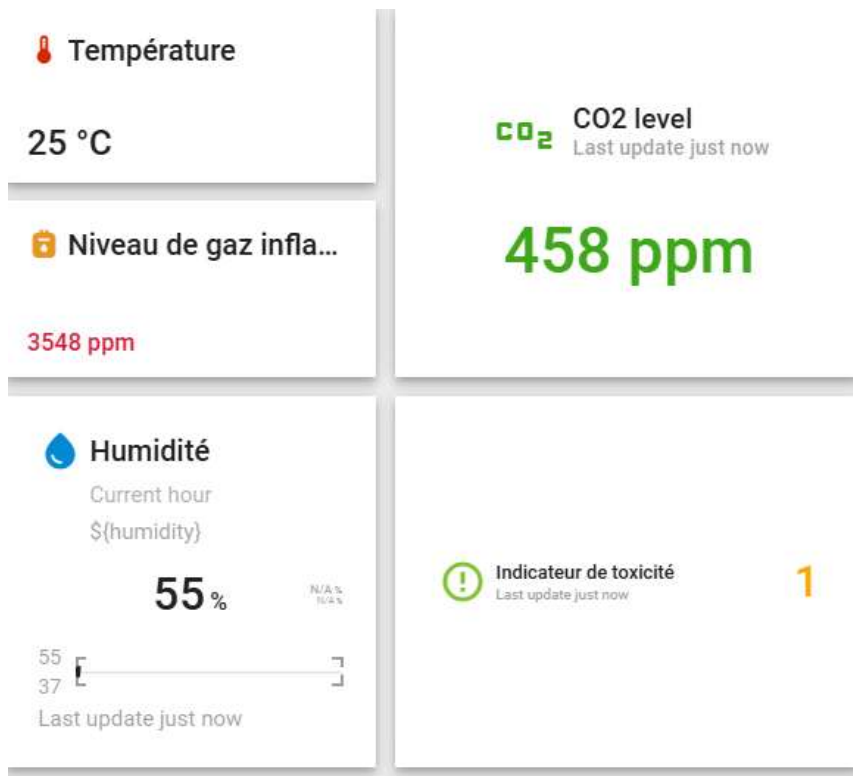
5. La page « Tableaux de bord » nous permet de créer des dashboard pour visualiser nos données



6. A l'intérieur d'un même DashBoard on peut retrouver plusieurs graphiques



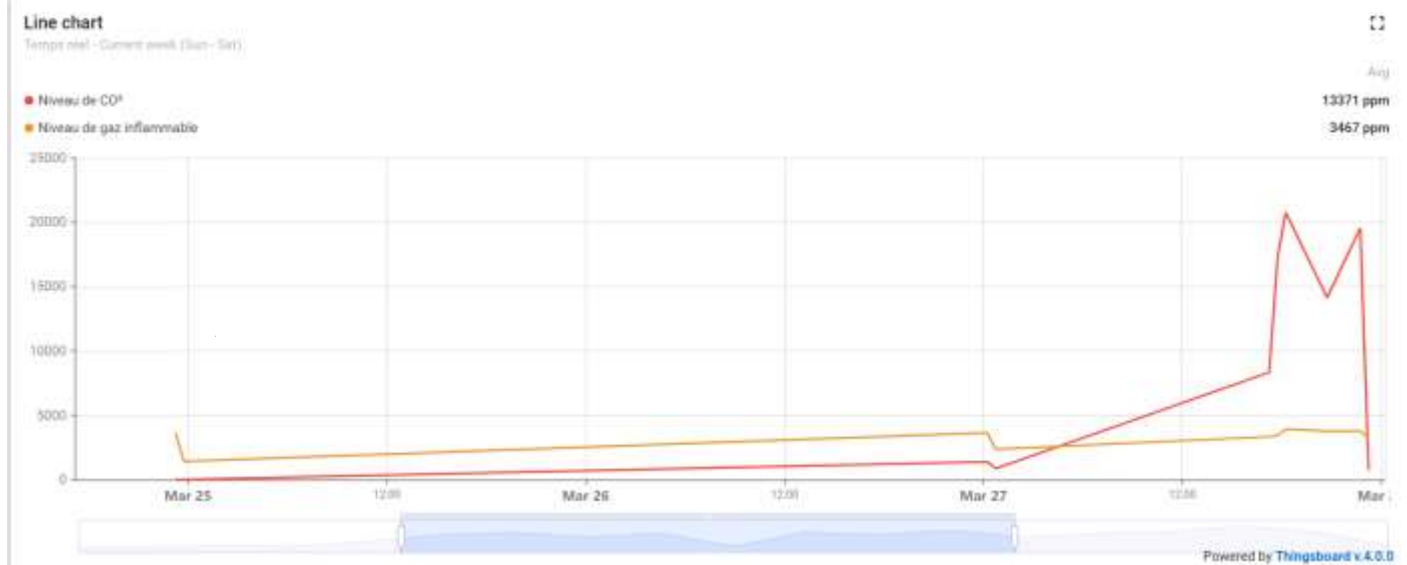
a. Visualiser les valeurs



Les premières cartes nous permettent de visualiser les valeurs. Ici on peut voir que la température est à 25°C, l'humidité à 55% la concentration en gaz inflammables de 3548 ppm, et celle en gaz polluants de 458 ppm. L'indicateur de toxicité, ou de qualité d'air est à 1, ce qui signifie que la valeur de la concentration en gaz inflammable est anormale, chose logique, vu la concentration de ce gaz.



Ce graphe permet de visualiser la variation de la température et de l'humidité au cours de la dernière semaine.



Ce graphe ci permet de visualiser la variation des concentrations en gaz polluants et inflammables au cours de la dernière semaine.

III. Améliorations Possibles

Bien que notre système fonctionne correctement et atteigne ses objectifs, plusieurs axes d'amélioration peuvent être envisagés pour optimiser ses performances et sa fiabilité :

A. Optimisation du Modèle de Détection d'Anomalies

- Tester d'autres architectures de réseaux neuronaux pour améliorer la précision.
- Tester des modèles légers comme MobileNet pour un meilleur déploiement sur ESP32.
- Entraîner le modèle avec un dataset plus large et diversifié pour mieux généraliser.

B. Amélioration de l'Infrastructure IoT

- Optimiser la communication MQTT pour réduire la latence et la consommation d'énergie.
- Tester l'ESP32 avec une carte SD pour stocker temporairement les données en cas de coupure réseau.
- Explorer d'autres plateformes cloud (AWS IoT, Azure IoT) pour comparer les performances avec ThingsBoard.

C. Affinage des Seuils d'Anomalie

- Au lieu de fixer des seuils statiques, utiliser une approche dynamique basée sur des statistiques en temps réel.
- Tester des méthodes comme Isolation Forest ou Autoencoders pour une détection plus précise.

D. Extension des Fonctionnalités

- Intégrer une application mobile pour consulter les données en temps réel.
- Mettre en place un système d'alerte avancé (notifications SMS ou email en cas d'anomalie critique).

Conclusion

Ce projet a démontré l'efficacité de l'IoT et de l'IA dans la gestion intelligente des environnements intérieurs. Grâce à l'intégration de l'ESP32, des capteurs et d'un modèle de machine learning, nous avons pu surveiller et analyser les variations de température, d'humidité et de qualité de l'air, en identifiant les anomalies potentielles.

L'utilisation de **Wokwi** a permis de simuler l'environnement sans matériel physique, tandis que **Google Colab** a servi à entraîner et externaliser l'inférence du modèle en raison des limitations de Wokwi. Enfin, **ThingsBoard** a offert un suivi visuel des données et des alertes en temps réel.

Malgré ses performances, le système peut être amélioré sur plusieurs aspects, notamment en optimisant le modèle, en réduisant la latence du traitement et en explorant d'autres approches pour la gestion des seuils d'anomalie. Ce projet ouvre ainsi la voie à de nombreuses améliorations et pourrait être étendu à des applications plus complexes, comme la gestion énergétique à grande échelle ou la surveillance environnementale avancée.