Fonctionnalité de Configuration à Distance des Nœuds Capteurs ESP32 via Grafana

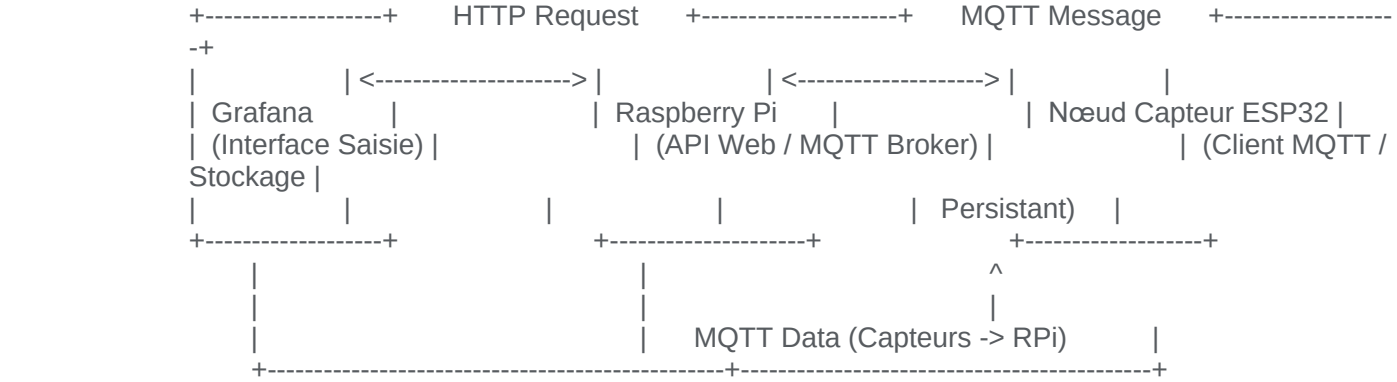
Date : 13 juin 2025

1. Introduction

Ce document décrit en détail la mise en place d'une fonctionnalité clé pour le système de surveillance d'hangar avicole : la capacité à modifier des paramètres de configuration des nœuds capteurs ESP32 (tels que les intervalles d'envoi de données ou les seuils d'alerte locaux) directement depuis l'interface utilisateur de Grafana. Cette approche permet une gestion centralisée et flexible du comportement des capteurs sans nécessiter d'intervention physique sur chaque boîtier.

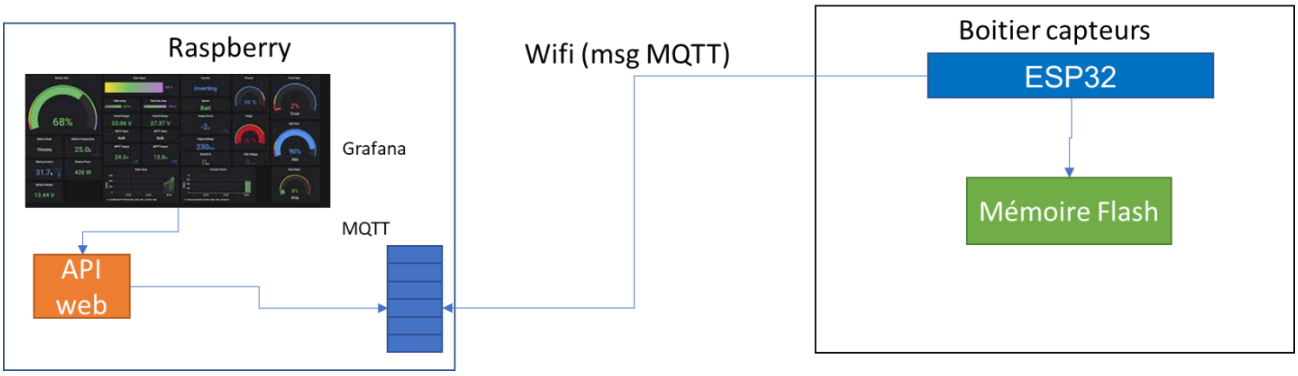
2. Architecture Fonctionnelle

La fonctionnalité repose sur une chaîne de communication bidirectionnelle utilisant **MQTT** comme protocole principal de communication entre le Raspberry Pi et les ESP32, et une **API Web (HTTP)** comme interface entre Grafana et la couche MQTT. Les données de configuration sont stockées de manière persistante sur les ESP32.



Composants impliqués :

- **Grafana :** Interface utilisateur pour la saisie et le déclenchement des commandes.
- **API Web (Python Flask) :** S'exécute sur le Raspberry Pi. Reçoit les requêtes HTTP de Grafana et les convertit en messages MQTT.
- **Mosquitto (MQTT Broker) :** S'exécute sur le Raspberry Pi. Relais central des messages MQTT.
- **Nœuds Capteurs ESP32 :** S'abonnent à des sujets MQTT spécifiques, reçoivent les messages de configuration, les analysent et les sauvegardent.
- **Preferences (ESP32) :** Bibliothèque pour le stockage persistant des données de configuration sur la mémoire flash de l'ESP32.



3. Implémentation Détaillée par Composant

3.1. Grafana : Interface de Saisie et de Déclenchement

Rôle : Permettre à l'utilisateur de saisir ou de choisir des paramètres et de déclencher l'envoi de commandes via des "boutons" virtuels.

Technologie : Panneaux de Texte Grafana avec Markdown (pour la flexibilité et la personnalisation visuelle).

Logiciels à installer : Grafana (déjà installé).

Détails de configuration sur Grafana :

1. Création de Variables de Tableau de Bord (pour la Saisie Utilisateur) :

   Pour permettre à l'utilisateur de spécifier des valeurs (comme un intervalle de temps ou un seuil), utilisez les variables de tableau de bord de type "Text box".

   - Dans les paramètres du tableau de bord (icône engrenage) -> "Variables" -> "Add variable".
   - **Exemple pour l'intervalle :**
     - Name: intervalle_envoi
     - Type: Text box
     - Label: Intervalle d'envoi (min)
     - Default value: 1 (valeur par défaut suggérée)
   - **Exemple pour le seuil de température :**
     - Name: seuil_temp_haut
     - Type: Text box
     - Label: Seuil Temp. Haute (°C)
     - Default value: 30.0
2. Création des "Boutons" avec des Panneaux de Texte :

   Ces panneaux enverront des requêtes HTTP à l'API du Raspberry Pi.

   - Ajouter un nouveau panneau (Add panel -> Add new panel).
   - Sélectionner Text comme type de visualisation.
   - Dans l'éditeur, choisir le mode Markdown.
   - Utiliser la syntaxe Markdown avec un lien HTML stylisé pour créer le bouton.

Exemple de code Markdown pour un bouton "Définir Intervalle" pour la Zone 1 :

```
Markdown
<a href="http://localhost:5000/set_interval?zone=1&value=${intervalle_envoi}"
  target="_blank"
  style="display: block; width: 100%; padding: 15px; background-color: #007bff; color: white; text-align: center; text-decoration: none; font-size: 20px; border-radius: 5px;">
   Définir Intervalle Zone 1
</a>
```

- **http://localhost:5000**: Adresse et port de l'API Flask sur le Raspberry Pi. Si Grafana tourne sur le même Pi, localhost est correct.
- /set_interval: Point d'accès (endpoint) de l'API Flask.
- ?zone=1&value=${intervalle_envoi}: Paramètres passés à l'API. zone identifie le nœud ESP32, et value est la valeur de la variable Grafana.
- target="_blank": Ouvre le lien dans un nouvel onglet. C'est une limitation des panneaux de texte Grafana. Pour une expérience utilisateur plus fluide (requêtes en arrière-plan), il faudrait un plugin Grafana dédié ou une personnalisation JavaScript avancée.

Exemple de code Markdown pour un bouton "Définir Seuil Température" pour la Zone 2 :

```
Markdown
<a href="http://localhost:5000/set_temp_threshold?zone=2&value=${seuil_temp_haut}"
  target="_blank"
  style="display: block; width: 100%; padding: 15px; background-color: #28a745; color: white; text-align: center; text-decoration: none; font-size: 20px; border-radius: 5px;">
   Définir Seuil Temp. Zone 2
</a>
```

- Ajuster zone et value selon les besoins pour d'autres capteurs/paramètres.

3.2. Raspberry Pi : API Web (Flask) & Broker MQTT (Mosquitto)

Rôle :

- **API Web :** Reçoit les requêtes HTTP de Grafana, extrait les paramètres et publie des messages MQTT.
- **Broker MQTT :** Gère la distribution des messages MQTT entre l'API et les nœuds ESP32. **Technologies :** Python 3 (Flask, paho-mqtt), Mosquitto.

Logiciels à installer :

- Mosquitto (broker MQTT) :

```
Bash
sudo apt install mosquitto mosquitto-clients
sudo systemctl enable mosquitto.service
sudo systemctl start mosquitto.service
```

- Python 3 (généralement préinstallé).
- pip (gestionnaire de paquets Python) :

```
Bash
sudo apt install python3-pip
```

- Bibliothèques Python :

```
Bash
pip install Flask paho-mqtt
```

Détails d'implémentation de l'API Flask :

1. **Fichier /home/pi/config_api/config_api.py :**

```
Python
from flask import Flask, request, jsonify
```

```
import paho.mqtt.client as mqtt
import time
import threading
import os

app = Flask(__name__)

# Configuration MQTT
MQTT_BROKER_IP = os.getenv('MQTT_BROKER_IP', 'localhost') # Utilise localhost par
defaut
MQTT_PORT = int(os.getenv('MQTT_PORT', 1883))
MQTT_CLIENT_ID = "RPi_Config_API"
mqtt_client = None

# Initialisation et gestion de la connexion MQTT
def init_mqtt_client():
    global mqtt_client
    mqtt_client = mqtt.Client(client_id=MQTT_CLIENT_ID)
    mqtt_client.on_connect = on_connect_mqtt
    mqtt_client.on_disconnect = on_disconnect_mqtt
    try:
        mqtt_client.connect(MQTT_BROKER_IP, MQTT_PORT, 60)
        mqtt_client.loop_start() # Démarrer le thread de boucle MQTT pour gérer les
connexions
        print(f"Tentative de connexion au broker MQTT à
{MQTT_BROKER_IP}:{MQTT_PORT}...")
    except Exception as e:
        print(f"Erreur d'initialisation MQTT: {e}")

def on_connect_mqtt(client, userdata, flags, rc):
    if rc == 0:
        print("Connecté au broker MQTT.")
    else:
        print(f"Échec de la connexion MQTT, code {rc}. Veuillez vérifier le broker.")

def on_disconnect_mqtt(client, userdata, rc):
    print(f"Déconnecté du broker MQTT (code: {rc}). Tentative de reconnexion
automatique...")
    # La boucle start() gère la reconnexion automatiquement, pas besoin de reconnecter
ici manuellement.

# Assurer que le client MQTT est initialisé au démarrage de l'application
init_mqtt_client()

# --- Routes de l'API Flask ---

@app.route('/')
def home():
    return "API de configuration ESP32 - Fonctionnelle."

@app.route('/set_interval', methods=['GET'])
def set_interval():
    zone = request.args.get('zone')
    interval = request.args.get('value')

    if not zone or not interval:
```

```
        return jsonify({"status": "error", "message": "Paramètres 'zone' et 'value' requis."}),
400

    try:
        interval_val = int(interval)
        if interval_val <= 0:
            raise ValueError("L'intervalle doit être positif.")
    except ValueError:
        return jsonify({"status": "error", "message": "La valeur d'intervalle doit être un nombre
entier positif."}), 400

    mqtt_topic = f"hangar/zone{zone}/config"
    mqtt_message = f"interval={interval_val}"

    if mqtt_client and mqtt_client.is_connected():
        mqtt_client.publish(mqtt_topic, mqtt_message)
        print(f"API: Envoyé '{mqtt_message}' à '{mqtt_topic}'")
        return jsonify({"status": "success", "message": f"Intervalle de zone {zone} défini à
{interval} minutes."})
    else:
        return jsonify({"status": "error", "message": "Client MQTT non connecté. Veuillez
vérifier le broker."}), 500

@app.route('/set_temp_threshold', methods=['GET'])
def set_temp_threshold():
    zone = request.args.get('zone')
    threshold = request.args.get('value')

    if not zone or not threshold:
        return jsonify({"status": "error", "message": "Paramètres 'zone' et 'value' requis."}),
400

    try:
        threshold_val = float(threshold)
    except ValueError:
        return jsonify({"status": "error", "message": "La valeur de seuil doit être un
nombre."}), 400

    mqtt_topic = f"hangar/zone{zone}/config"
    mqtt_message = f"temp_high_thresh={threshold_val}"

    if mqtt_client and mqtt_client.is_connected():
        mqtt_client.publish(mqtt_topic, mqtt_message)
        print(f"API: Envoyé '{mqtt_message}' à '{mqtt_topic}'")
        return jsonify({"status": "success", "message": f"Seuil de température de zone {zone}
défini à {threshold}."})
    else:
        return jsonify({"status": "error", "message": "Client MQTT non connecté. Veuillez
vérifier le broker."}), 500

# D'autres routes peuvent être ajoutées ici pour d'autres paramètres
# @app.route('/set_nh3_threshold', methods=['GET']): ...

if __name__ == '__main__':
    # L'API écoute sur toutes les interfaces (0.0.0.0) sur le port 5000
    # Pour le déploiement, il est recommandé d'utiliser un serveur WSGI comme Gunicorn
ou uWSGI
```

```
    # derrière un proxy inverse comme Nginx, mais pour un petit projet, Flask's dev server est suffisant.
    app.run(host='0.0.0.0', port=5000)
```

2. Service Systemd pour l'API Flask :

   Pour assurer le démarrage automatique de l'API au boot du Raspberry Pi.

   - Créez le fichier de service : sudo nano /etc/systemd/system/config_api.service
   - Collez le contenu :
   - [Unit]
   - Description=API de configuration pour ESP32
   - After=network.target mosquitto.service # S'assure que Mosquitto est démarré
   -
   - [Service]
   - User=pi # Exécuter en tant qu'utilisateur 'pi' (ou autre utilisateur non-root)
   - WorkingDirectory=/home/pi/config_api/ # Le dossier où se trouve config_api.py
   - ExecStart=/usr/bin/python3 config_api.py # La commande pour lancer l'API
   - Restart=always # Redémarre en cas de crash
   - StandardOutput=syslog # Redirige la sortie standard vers les logs système
   - StandardError=syslog # Redirige les erreurs vers les logs système
   - SyslogIdentifier=config-api # Identifiant pour les logs (facilite la recherche)
   -
   - [Install]
   - WantedBy=multi-user.target # Démarrer une fois que le système est en mode multi-utilisateur
   - Activez et démarrez le service :

     Bash
     ```
     sudo systemctl daemon-reload # Recharger la configuration systemd
     sudo systemctl enable config_api.service # Activer le démarrage automatique au boot
     sudo systemctl start config_api.service # Démarrer le service maintenant
     sudo systemctl status config_api.service # Vérifier le statut du service
     ```
   - Pour voir les logs : sudo journalctl -f -u config_api.service

3.3. Nœud Capteur ESP32 : Réception et Enregistrement de la Configuration

Rôle : S'abonner aux sujets de configuration, analyser les messages reçus, sauvegarder les paramètres dans la mémoire flash et les appliquer.

Technologies : Arduino (C++), bibliothèques PubSubClient et Preferences.

Logiciels à installer (sur l'ordinateur de développement) :

- Arduino IDE
- Support des cartes ESP32 pour Arduino IDE.
- Bibliothèques PubSubClient et Preferences (via le gestionnaire de bibliothèques de l'IDE Arduino).

Détails d'implémentation du firmware ESP32 :
Le code de l'ESP32 doit être modifié pour inclure la logique de réception et de persistance des configurations.

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

```
#include <PubSubClient.h>
#include <Preferences.h> // Bibliothèque pour le stockage persistant sur ESP32

// --- Paramètres WiFi et MQTT ---
const char* ssid = "Nom_De_Votre_Reseau_Hangar"; // SSID du reseau Wi-Fi du Raspberry Pi
const char* password = "VotreMotDePasseSecret"; // Mot de passe du reseau Wi-Fi du
Raspberry Pi

const char* mqtt_server = "192.168.4.1"; // Adresse IP statique du Raspberry Pi
const char* mqtt_client_id = "ESP32_Zone1"; // ID unique pour ce noeud ESP32
const char* data_topic = "hangar/zone1/data"; // Sujet MQTT pour envoyer les donnees
const char* config_topic = "hangar/zone1/config"; // Sujet MQTT pour recevoir les configurations

// --- Variables de configuration (avec valeurs par défaut) ---
long sendIntervalMinutes = 1;      // Intervalle d'envoi des données en minutes
float tempThresholdHigh = 30.0;    // Seuil d'alerte de température (°C)
// Ajoutez d'autres variables de configuration ici (ex: float nh3Threshold, float co2Offset, etc.)

Preferences preferences; // Instance de la classe Preferences pour NVS

WiFiClient espClient;
PubSubClient client(espClient);

unsigned long lastSendMillis = 0; // Pour gérer l'intervalle d'envoi

// --- Fonctions de Connexion WiFi et MQTT ---
void setup_wifi() {
  delay(10);
  Serial.print("Connecting to WiFi: ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("\nWiFi connected.");
  Serial.print("IP Address: ");
  Serial.println(WiFi.localIP());
}

void reconnectMqtt() {
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    if (client.connect(mqtt_client_id)) {
      Serial.println("connected");
      // S'abonner au sujet de configuration apres une reconnexion
      client.subscribe(config_topic);
      Serial.print("Subscribed to config topic: ");
      Serial.println(config_topic);
      // Optionnel: Envoyer un message de "boot" ou de "statut"
      client.publish("hangar/zone1/status", "ESP32 started and connected.");
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" trying again in 5 seconds");
      delay(5000);
    }
```

```
  }
}

// --- Fonction de Callback MQTT (Gestion des messages reçus) ---
void callback(char* topic, byte* payload, unsigned int length) {
    Serial.print("Message received on topic: ");
    Serial.println(topic);

    String message = "";
    for (int i = 0; i < length; i++) {
        message += (char)payload[i];
    }
    Serial.print("Message: ");
    Serial.println(message);

    // Vérifier si le message provient du sujet de configuration
    if (String(topic) == config_topic) {
        int eq_pos = message.indexOf('=');
        if (eq_pos != -1) {
            String paramName = message.substring(0, eq_pos);
            String paramValue = message.substring(eq_pos + 1);

            // Gérer le paramètre "interval"
            if (paramName == "interval") {
                long newInterval = paramValue.toInt();
                if (newInterval > 0) {
                    sendIntervalMinutes = newInterval;
                    preferences.putLong("sendInterval", sendIntervalMinutes); // Sauvegarde persistante
                    Serial.print("New send interval: ");
                    Serial.print(sendIntervalMinutes);
                    Serial.println(" minutes");
                    client.publish("hangar/zone1/status", "Interval updated.");
                } else {
                    Serial.println("Invalid interval received.");
                    client.publish("hangar/zone1/status", "Invalid interval value.");
                }
            }
            // Gérer le paramètre "temp_high_thresh"
            else if (paramName == "temp_high_thresh") {
                float newThreshold = paramValue.toFloat();
                tempThresholdHigh = newThreshold;
                preferences.putFloat("tempThreshH", tempThresholdHigh); // Sauvegarde persistante
                Serial.print("New high temp threshold: ");
                Serial.println(tempThresholdHigh);
                client.publish("hangar/zone1/status", "Temp threshold updated.");
            }
            // Ajoutez ici la gestion d'autres paramètres de configuration (ex: nh3_thresh, co2_offset,
etc.)
            // Exemple pour un paramètre nommé "nh3_thresh"
            // else if (paramName == "nh3_thresh") {
            //     float newNh3Threshold = paramValue.toFloat();
            //     preferences.putFloat("nh3Thresh", newNh3Threshold);
            //     Serial.print("New NH3 threshold: "); Serial.println(newNh3Threshold);
            // }
        }
        // Pour les configurations JSON plus complexes, utilisez la bibliothèque ArduinoJson
        // et deserializeJson(doc, message); pour analyser le payload.
```

```
    }
}

// --- Fonction de Lecture et Publication de Données (Exemple) ---
void publishSensorData() {
  // Ici, vous liriez vos vrais capteurs (SHT31, MH-Z19B, MQ, Anémomètre)
  float currentTemp = 25.5; // Exemple
  float currentHum = 60.2;  // Exemple
  int currentCO2 = 500;     // Exemple

  String payload = "temp=" + String(currentTemp) + "&hum=" + String(currentHum) + "&co2=" +
String(currentCO2);
  Serial.print("Publishing data: ");
  Serial.println(payload);
  client.publish(data_topic, payload.c_str());

  // Logique d'alerte locale basée sur les seuils configurés
  if (currentTemp > tempThresholdHigh) {
    Serial.println("ALERT: High Temperature!");
    // Ici, l'ESP32 pourrait allumer une LED, un buzzer, etc.
  }
}

// --- Setup et Loop de l'ESP32 ---
void setup() {
  Serial.begin(115200);
  // Initialiser les préférences NVS
  preferences.begin("my-app", false); // "my-app" est le nom de l'espace NVS pour votre
application

  // Charger les configurations précédentes, avec des valeurs par défaut si non trouvées
  sendIntervalMinutes = preferences.getLong("sendInterval", 1);
  tempThresholdHigh = preferences.getFloat("tempThreshH", 30.0);
  // Charger d'autres paramètres ici

  Serial.print("Loaded send interval: "); Serial.print(sendIntervalMinutes); Serial.println(" minutes");
  Serial.print("Loaded high temp threshold: "); Serial.print(tempThresholdHigh); Serial.println(" C");

  setup_wifi(); // Connecter l'ESP32 au Wi-Fi du Raspberry Pi
  client.setServer(mqtt_server, 1883); // Configurer l'adresse du broker MQTT
  client.setCallback(callback); // Définir la fonction de callback pour les messages MQTT reçus
}

void loop() {
  if (!client.connected()) {
    reconnectMqtt(); // Assurer que le client MQTT est connecté
  }
  client.loop(); // Traiter les messages MQTT entrants et maintenir la connexion

  // Logique d'envoi des données basée sur l'intervalle configurable
  if (millis() - lastSendMillis > (sendIntervalMinutes * 60 * 1000)) {
    publishSensorData(); // Lire et publier les données des capteurs
    lastSendMillis = millis();
  }
}
```

4. Flux de Fonctionnement Complet

1. **Démarrage du Raspberry Pi :**

    - Le service mosquitto.service démarre.
    - Le service config_api.service démarre, l'API Flask s'initialise et se connecte au broker Mosquitto.
    - Le service grafana-server.service démarre.
    - Votre script Python de collecte de données démarre et se connecte à Mosquitto.
    - Le navigateur en mode kiosque démarre et affiche le tableau de bord Grafana.
2. **Démarrage de l'ESP32 :**
    - L'ESP32 se connecte au réseau Wi-Fi du Raspberry Pi (en mode AP).
    - Il charge ses dernières configurations depuis la mémoire flash (Preferences).
    - Il se connecte au broker Mosquitto et s'abonne à son sujet de configuration (hangar/zoneX/config).
    - Il commence à lire et envoyer les données des capteurs selon l'intervalle configuré.
3. **Interaction Utilisateur via Grafana :**
    - L'utilisateur accède au tableau de bord Grafana sur l'écran tactile.
    - Il saisit une nouvelle valeur dans le champ de la variable Grafana (ex: intervalle_envoi = 5).
    - Il touche le "bouton" Grafana (Text Panel).
    - Le navigateur de Grafana envoie une requête HTTP à l'API Flask sur le Raspberry Pi (ex: GET http://localhost:5000/set_interval?zone=1&value=5).
4. **Traitement par l'API Flask :**
    - L'API Flask reçoit la requête HTTP.
    - Elle extrait les paramètres zone=1 et value=5.
    - Elle utilise le client paho-mqtt pour publier le message interval=5 sur le sujet MQTT hangar/zone1/config.
5. **Réception par l'ESP32 :**
    - Le broker Mosquitto transmet le message interval=5 à l'ESP32 abonné au sujet hangar/zone1/config.
    - La fonction callback() de l'ESP32 est déclenchée.
    - Elle analyse le message, extrait interval=5.
    - La variable sendIntervalMinutes de l'ESP32 est mise à jour à 5.
    - La nouvelle valeur est sauvegardée dans la mémoire flash (preferences.putLong("sendInterval", 5)).
    - L'ESP32 commence immédiatement à envoyer les données toutes les 5 minutes.

5. Considérations Techniques et Bonnes Pratiques

- **Sécurité de l'API Flask :**
    - Actuellement, l'API est accessible sans authentification sur le port 5000. Pour un environnement de production, il est crucial d'ajouter une couche de sécurité (authentification par jeton API ou clé partagée) et/ou de la protéger derrière un proxy inverse (comme Nginx) configuré pour SSL/TLS et un contrôle d'accès strict.
- **Gestion des Erreurs et Retour Visuel :**

    - L'API Flask renvoie des statuts JSON (`success`/`error`). Grafana ne peut pas facilement afficher ces réponses directement avec des liens simples. Pour un meilleur retour utilisateur, il faudrait une intégration plus poussée (plugins Grafana, ou un script JS personnalisé qui ferait l'appel Fetch/AJAX et afficherait un message).
- **Validation des Données :**
    - Effectuez une validation des données robustes à la fois côté API Flask (comme la conversion en `int`/`float` et la vérification des plages) et côté ESP32.
- **Scalabilité :**
    - Pour un grand nombre de zones, la gestion des sujets MQTT dynamiquement (ex: `hangar/zoneX/config` où X est une variable) est cruciale.
- **Gestion des Identifiants (Secrets) :**
    - Les identifiants Wi-Fi et les mots de passe MQTT (si vous en utilisez) ne devraient pas être en clair dans le code déployé. Pour l'ESP32, utilisez le `Preferences` pour les stocker après une première configuration. Pour le Raspberry Pi, utilisez des variables d'environnement ou un fichier de configuration sécurisé.
- **Robustesse du Stockage Persistant :**
    - Bien que `Preferences` soit robuste, il est toujours bon de prévoir des valeurs par défaut dans le code de l'ESP32 au cas où la mémoire flash serait corrompue ou effacée.
- **Gestion de la Déconnexion MQTT :**
    - L'ESP32 et l'API Flask doivent gérer les déconnexions et reconnexions automatiques au broker MQTT. `paho-mqtt.Client.loop_start()` et `PubSubClient` (avec `reconnectMqtt`) s'en occupent en grande partie.

6. Résumé des Logiciels et Commandes d'Installation

Ce tableau récapitule toutes les installations nécessaires sur le Raspberry Pi. Les outils de développement pour l'ESP32 sont installés sur le PC du développeur.

Composant	Système	Logiciels / Paquets	Commandes d'Installation (sur RPi)
Raspberry Pi OS	OS	Raspberry Pi OS Desktop (64-bit recommandé)	Télécharger l'image et flasher sur la carte SD.
MQTT Broker	RPi	Mosquitto	sudo apt update && sudo apt install mosquitto mosquitto-clients
Base de Données	RPi	InfluxDB	(Voir documentation InfluxDB pour l'installation sur ARM - généralement via `apt`)

Serveur Visualisation	RPi	Grafana	(Voir documentation Grafana pour l'installation sur ARM - généralement via apt)
API Web (Python)	RPi	Python 3, pip, Flask, paho-mqtt	sudo apt install python3-pip; pip install Flask paho-mqtt
Services Système	RPi (systemd)	Fichiers .service personnalisés	Création des fichiers (/etc/systemd/system/*.service), sudo systemctl daemon-reload, enable, start
Nœuds Capteurs	PC de Développement	Arduino IDE, Support ESP32, PubSubClient, Preferences	Installation de l'IDE, gestionnaire de cartes, gestionnaire de bibliothèques.

Ce rapport fournit une base solide pour le développement et le déploiement de votre fonctionnalité de configuration à distance. N'hésitez pas à poser d'autres questions si des points nécessitent des éclaircissements supplémentaires.