



R A P P O R T D E P R O J E T

Sécurisation des Systèmes Embarqués :

Un Projet IoT Moderne avec MQTT, SSL/TLS et
Node-RED

Faculté des Sciences d'Agadir
Centre d'Excellence, Département Informatique
Filière : Ingénierie Informatique et Systèmes Embarqués
Module : Programmation Python Avancée

Réalisé par :

- ELMESSAOUDI Oussama

Supervisé par :

- Pr. LAFRAXO Samira
- Pr. OUKDACHE Yassine

Remerciements

Nous souhaitons remercier Madame LAFRAXO et Monsieur OUKDACH pour nous avoir permis de réaliser ce projet et nous avoir assister lorsque nous en avions besoin tout au long de notre première semestre de M1.



Table des Matières

Introduction

Présentation du Projet

Méthodologie

Implémentation

Résultats et Tests

Conclusion

Bibliographie

Dépôt GitHub



Introduction

Lors de notre M1 à la Faculté des Sciences d'Agadir, dans la filière Ingénierie Informatique et Systèmes Embarqués, nous avons eu à réaliser un projet de fin de semestre dans le cadre du module de Python Avancé, choisi parmi une liste de différents sujets proposés. Ce projet s'est déroulé pendant le premier semestre et doit être soutenu à la fin de celui-ci. Le projet a été proposé par Monsieur Oukdache et Madame Lafraxo, enseignants-chercheurs au sein du département d'Informatique, et nous avons travaillé en collaboration avec mon camarade Anass Baadi.

Ce projet consiste à développer une solution complète en lien avec la sécurisation des systèmes embarqués. Il a pour objectif de démontrer l'application pratique des connaissances acquises en systèmes embarqués, réseaux et sécurité informatique, et de les mettre en œuvre dans un projet opérationnel et fonctionnel.

Dans ce rapport de fin de semestre, nous aborderons en détail le projet, ses objectifs, les défis rencontrés ainsi que les étapes déjà réalisées. Nous détaillerons également la partie qui reste à accomplir et les solutions envisagées pour achever ce travail.



Présentation du Projet

Context

Les systèmes embarqués sont présents dans une multitude de domaines, allant des objets connectés aux véhicules autonomes, en passant par les équipements médicaux et les appareils domestiques intelligents. Ces systèmes, qui intègrent des microcontrôleurs, des capteurs et des protocoles de communication sans fil, sont essentiels pour de nombreuses applications industrielles, commerciales et domestiques.

Cependant, avec l'augmentation de l'interconnexion de ces systèmes dans l'Internet des Objets (IoT), la sécurité des systèmes embarqués devient une préoccupation majeure. Les risques de piratage, de vol de données et d'attaques à distance sont des menaces réelles qui nécessitent une attention particulière dans la conception, le développement et le déploiement de ces systèmes.

L'objectif de ce projet est de garantir la sécurité des systèmes embarqués en utilisant des techniques de cryptage, des protocoles sécurisés comme SSL/TLS, et des outils modernes tels que MQTT pour assurer une communication sécurisée entre les dispositifs IoT et les plateformes centrales.



Présentation du Projet

Objectifs

Le projet "Sécurisation des Systèmes Embarqués" a plusieurs objectifs spécifiques :

1. **Sécurisation des communications IoT** : Utilisation de SSL/TLS pour chiffrer les échanges de données entre l'ESP32 (le microcontrôleur embarqué), le serveur Django, et les autres composants du système.
2. **Implémentation de MQTT sécurisé** : Utilisation de MQTT (Message Queuing Telemetry Transport) pour la communication en temps réel entre les capteurs IoT et le serveur, avec des protocoles sécurisés pour éviter toute interception ou altération des données.
3. **Création d'une API sécurisée** : Mise en place d'une API RESTful sécurisée avec Django pour gérer les données envoyées depuis les capteurs et pour permettre la consultation de ces données via un navigateur ou une application mobile.
4. **Utilisation de Node-RED pour la visualisation en temps réel** : Configuration de Node-RED pour recevoir les données des capteurs IoT, les afficher en temps réel sur des graphiques et permettre aux utilisateurs de gérer des commandes envoyées à l'ESP32.
5. **Simulation avec Wokwi** : Utilisation de Wokwi pour simuler les capteurs et la communication MQTT, garantissant ainsi une vérification du système avant l'implémentation physique.

Présentation du Projet

Périmètre du Projet

- **Technologies principales utilisées :**

- ESP32 (microcontrôleur embarqué)
- Django (serveur backend)
- MQTT (protocole de communication)
- SSL/TLS (pour sécuriser les échanges de données)
- Node-RED (plateforme de visualisation et de contrôle)
- Wokwi (simulation d'ESP32)

- **Cibles principales du projet :**

- Systèmes IoT avec des capteurs de température et d'humidité.
- Visualisation des données et contrôle à distance via une interface web.

- **Durée du projet :**

- Prévision de développement et de tests sur un semestre (environ une semaine).



Présentation du Projet

Solutions Proposées

- **Sécurisation des Communications avec MQTT et SSL/TLS :**

- Protocole MQTT : Utilisé pour la communication entre le microcontrôleur (ESP32) et le serveur Django. MQTT est léger, adapté aux systèmes embarqués, et assure une faible consommation de bande passante, ce qui est idéal pour les environnements IoT.
- SSL/TLS : Pour sécuriser les communications via MQTT, nous utiliserons SSL/TLS pour chiffrer les messages et garantir leur intégrité. Cela empêchera les attaques de type "man-in-the-middle" et protégera les données échangées.
- Utilisation de certificats SSL pour la communication sécurisée avec un broker MQTT et le serveur Django.

- **Création d'une API RESTful avec Django :**

- Django REST Framework (DRF) : Utilisation de DRF pour créer une API qui permet au serveur de recevoir les données des capteurs IoT (température, humidité). Les données sont ensuite stockées dans une base de données relationnelle.
- Authentification et sécurité : La sécurité des API sera assurée par l'implémentation de mécanismes d'authentification (tokens JWT) pour garantir que seules les connexions autorisées peuvent envoyer ou récupérer des données.

- **Affichage en Temps Réel avec Node-RED :**

- Node-RED sera configuré pour recevoir les données en temps réel via MQTT et les afficher dans un dashboard sous forme de graphiques avec Chart.js. Cela permettra aux utilisateurs de visualiser l'évolution des données collectées par les capteurs.

Présentation du Projet

Solutions Proposées

- L'interface permettra également d'envoyer des commandes à l'ESP32, telles que des actions pour afficher un texte sur l'écran LCD.
- **Contrôle à Distance et Commandes via MQTT :**
 - Le serveur Django permettra aux utilisateurs d'envoyer des commandes via un formulaire HTML. Ces commandes seront envoyées via MQTT au microcontrôleur ESP32 pour effectuer des actions spécifiques, comme afficher un texte sur un écran LCD.
- **Simulation avec Wokwi :**
 - Wokwi sera utilisé pour simuler le microcontrôleur ESP32, permettant ainsi de tester les communications et la logique du système sans avoir besoin de matériel réel. Cela facilitera la validation des étapes avant la mise en place physique du système.



Présentation du Projet

Technologies Utilisées

- **ESP32** : Un microcontrôleur puissant pour la collecte de données à partir des capteurs et pour la gestion de la communication sans fil via Wi-Fi ou Bluetooth. L'ESP32 sera utilisé pour envoyer des données de capteurs comme la température et l'humidité vers un serveur sécurisé via MQTT.



- **Django** : Framework Python pour créer un serveur web sécurisé qui reçoit, stocke et gère les données envoyées par les capteurs IoT. Le serveur sera sécurisé par SSL/TLS et offrira une API RESTful pour l'accès aux données.

django

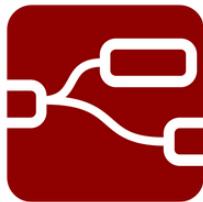
- **MQTT** : Protocole de messagerie léger utilisé pour la communication en temps réel entre l'ESP32 et le serveur Django, en utilisant des mécanismes de sécurité comme le chiffrement SSL/TLS.

 **MQTT**

Présentation du Projet

Technologies Utilisées

- **Node-RED** : Plateforme de développement basée sur des flux, utilisée pour gérer l'affichage des données en temps réel, visualiser les données des capteurs sur des graphiques interactifs, et permettre des interactions avec le serveur via des API sécurisées.



- **Wokwi** : Outil de simulation pour l'ESP32 permettant de tester et de valider le fonctionnement du système embarqué avant de passer à l'implémentation physique.



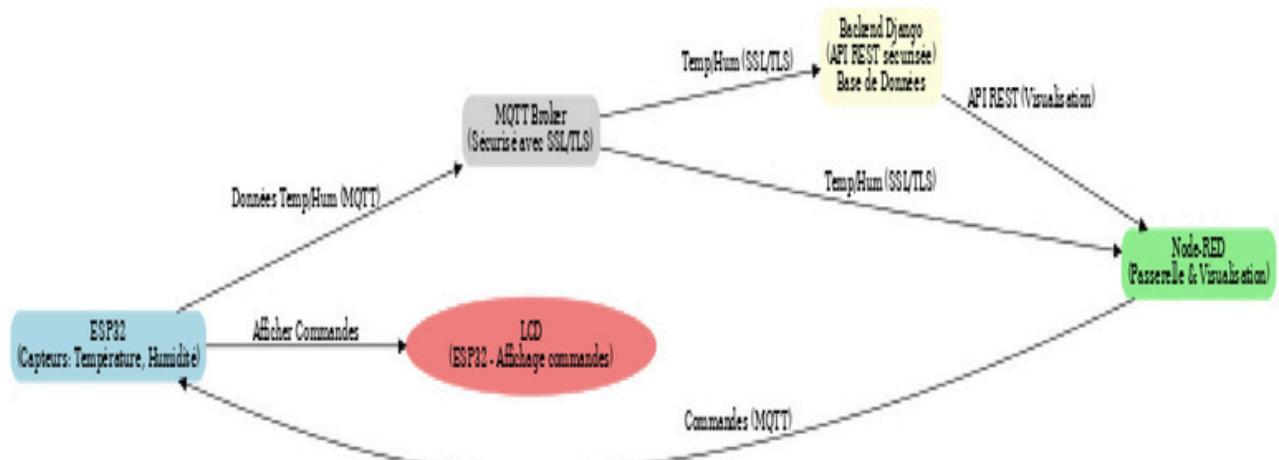
- **SSL/TLS** : SSL (Secure Sockets Layer) et TLS (Transport Layer Security) sont des protocoles utilisés pour sécuriser les communications réseau en assurant le chiffrement des données transmises, garantissant ainsi la confidentialité et l'intégrité des informations échangées entre les dispositifs.

Présentation du Projet

Architecture du Système

L'architecture du projet repose sur plusieurs composants interconnectés :

- **ESP32** : Capteurs IoT collectant des données (température, humidité) et envoyant ces informations à un serveur via MQTT écurisé.
- **Serveur Django** : API RESTful sécurisée pour la gestion et le stockage des données. Le serveur communique de manière sécurisée avec l'ESP32 via MQTT.
- **Node-RED** : Affiche les données des capteurs en temps réel et permet l'interaction avec l'ESP32.
- **Wokwi** : Simulation des capteurs et du comportement du système avant la mise en œuvre réelle.



un diagramme de l'architecture du projet, indiquant comment l'ESP32, Django, Node-RED, et Wokwi interagissent.

Méthodologie

Sécurisation des Communications

Une des parties cruciales de ce projet est la sécurisation des communications entre l'ESP32, le serveur Django, et Node-RED. Pour cela :

- **SSL/TLS** : Ces protocoles sont utilisés pour chiffrer les communications HTTP (entre le serveur et les clients) et MQTT (entre l'ESP32 et le serveur) afin d'empêcher toute interception ou modification des données échangées.
- **MQTT sécurisé** : Le serveur et l'ESP32 se connectent au broker MQTT via des connexions sécurisées sur le port 8883, avec une authentification et un chiffrement des messages.



Méthodologie

Processus de Communication des Données

Le processus de communication des données se déroule comme suit :

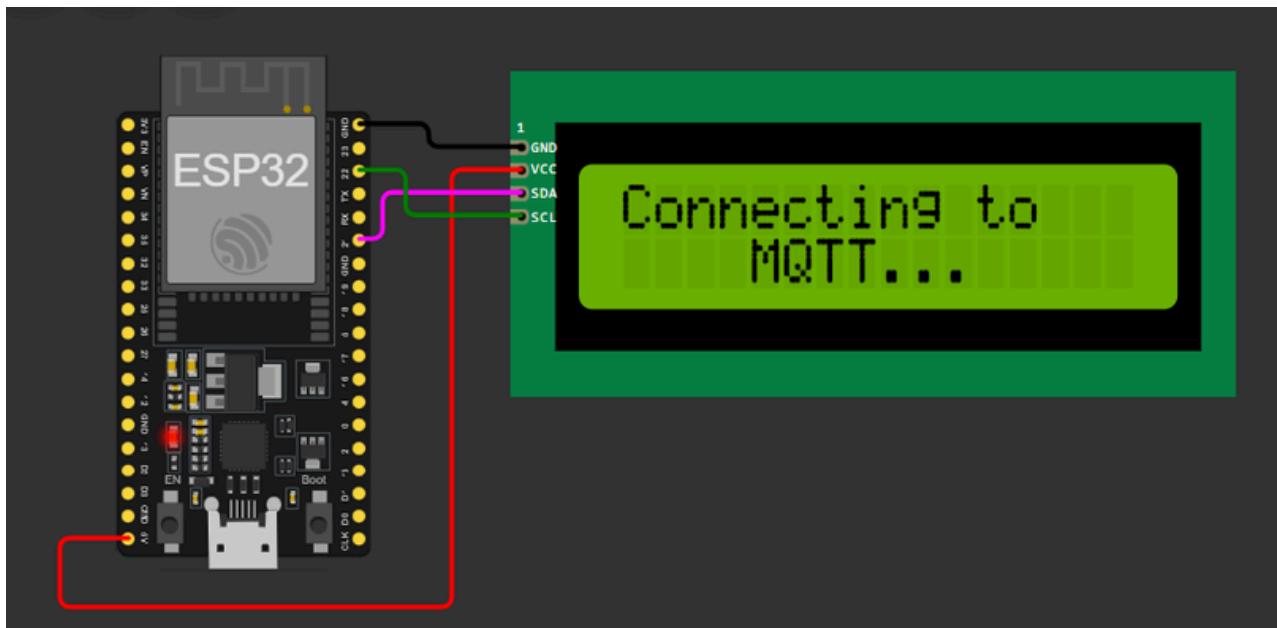
1. **Collecte de données** : Les capteurs (température, humidité) connectés à l'ESP32 collectent des informations et les envoient via MQTT au serveur Django.
2. **Transmission sécurisée** : Les données sont transmises via SSL/TLS, garantissant leur confidentialité et leur sécurité.
3. **Stockage et gestion** : Le serveur Django reçoit, stocke et gère ces données dans une base de données sécurisée.
4. **Visualisation des données** : Node-RED reçoit les données en temps réel, les affiche sous forme de graphiques interactifs via Chart.js et permet d'interagir avec le système.
5. **Contrôle à distance** : Les utilisateurs peuvent envoyer des commandes à l'ESP32 via l'interface Django, et ces commandes sont transmises via MQTT au dispositif pour effectuer des actions spécifiques (par exemple, afficher un texte sur un écran LCD).



Implémentation

ESP32

L'ESP32 est un microcontrôleur puissant utilisé pour la collecte des données à partir des capteurs (température et humidité). Il joue un rôle clé dans la communication sans fil avec le serveur via MQTT sécurisé. Dans cette partie, nous avons utilisé l'ESP32 pour envoyées données statiques et aléatoires pour avoir une variations des variables , les transmettre via MQTT avec une communication sécurisée (SSL/TLS), et recevoir des commandes de serveur Django à distance afin de les afficher dans un écran LCD.



Implémentation

Django Backend

Dans le cadre de notre projet "Sécurisation des Systèmes Embarqués", nous avons choisi d'utiliser Django comme backend serveur pour gérer la collecte, le stockage, et l'affichage en temps réel des données des capteurs IoT. Ce choix s'inscrit dans une démarche visant à garantir la sécurité des communications via des protocoles comme MQTT et SSL/TLS, et à offrir une interface utilisateur moderne et interactive via Node-RED et Chart.js pour la visualisation des données.

Le backend est constitué de plusieurs éléments principaux :

1. Django pour la gestion des données et l'implémentation des API sécurisées.
2. Django Channels pour gérer les WebSockets et assurer la transmission en temps réel des données depuis le serveur vers l'interface utilisateur.
3. Chart.js pour afficher les données en temps réel sous forme de graphiques interactifs.

Le processus d'implémentation repose sur une architecture robuste permettant de sécuriser la transmission des informations et d'assurer une gestion fluide des données en provenance des capteurs. Nous allons dans cette section détailler les étapes de l'implémentation de chaque composant clé du backend, en expliquant leur rôle, leurs interactions, et les choix technologiques qui ont été faits pour répondre aux exigences de sécurité et de performance du projet.



Implémentation

Django Backend (models.py)

Le fichier models.py définit les structures de données de votre application, comme les capteurs et les valeurs collectées.

- Le modèle principal de notre application est celui qui stocke les données des capteurs (température, humidité). Chaque donnée reçue est sauvegardée dans la base de données sous forme d'un enregistrement dans une table dédiée. Ce modèle est conçu pour être simple mais extensible, permettant d'ajouter d'autres types de capteurs à l'avenir.
- Extrait de code :

```
● ● ●                               models.py
from django.db import models

class SensorData(models.Model):
    timestamp = models.DateTimeField(auto_now_add=True)
    temperature = models.FloatField()
    humidity = models.FloatField()

    def __str__(self):
        return f"{self.timestamp}: Temp={self.temperature}, Hum={self.humidity}"
```



Implémentation

Django Backend (views.py)

Le fichier views.py gère la logique pour traiter les demandes HTTP, comme afficher les données en temps réel ou envoyer des commandes à l'ESP32.

- Les vues Django traitent les requêtes HTTP et fournissent les réponses adéquates. Nous avons une vue pour récupérer les dernières données des capteurs, qui est appelée par l'interface frontend pour mettre à jour les graphiques en temps réel. De plus, la vue envoie les données via WebSockets à l'interface Node-RED.
- Extrait de code :

```
views.py

from rest_framework.views import APIView
from rest_framework.response import Response
from .models import SensorData
from asgiref.sync import async_to_sync
from channels.layers import get_channel_layer
import json
from django.utils.timezone import localtime
from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt
import paho.mqtt.client as mqtt
import logging

class SensorDataView(APIView):
    def post(self, request):
        data = request.data
        sensor_data = SensorData.objects.create(
            temperature=data.get("temperature"),
            humidity=data.get("humidity")
        )
```

Implémentation

Django Backend (views.py)

```
# Broadcast data to WebSocket group
channel_layer = get_channel_layer()
async_to_sync(channel_layer.group_send)(
    "sensor_data",
    {
        "type": "send_sensor_data",
        "data": {
            "timestamp": sensor_data.timestamp.isoformat(),
            "temperature": sensor_data.temperature,
            "humidity": sensor_data.humidity,
        }
    }
)

return Response({"status": "success"}, status=201)

def get(self, request):
    data = SensorData.objects.all().values("timestamp", "temperature", "humidity")

    formatted_data = [
        {
            "timestamp": localtime(item["timestamp"]).isoformat(),
            "temperature": item["temperature"],
            "humidity": item["humidity"]
        }
        for item in data
    ]
    return Response(formatted_data)

class ClearSensorDataView(APIView):
    def delete(self, request):
        SensorData.objects.all().delete()
        return Response({"status": "success", "message": "All data cleared"}, status=200)
```



Implémentation

Django Backend (urls.py)

Le fichier urls.py gère le routage des différentes requêtes HTTP vers les vues appropriées.

- Le fichier urls.py de notre application Django définit les chemins pour accéder à différentes pages, telles que l'affichage des données en temps réel. Il lie les vues avec les URL appropriées pour que l'utilisateur puisse consulter les graphiques en temps réel ou accéder à d'autres pages du projet.
- Extrait de code :

```
urls.py

from django.contrib import admin
from django.urls import path, include
from sensors.views import SensorDataView, ClearSensorDataView, send_command_to_esp32

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/sensor-data/', include('sensors.urls')),
    path('api/clear-data/', ClearSensorDataView.as_view()),
    path('chart/', include('sensors.urls')),
    path('', include('sensors.urls')),
    path('api/send-command/', send_command_to_esp32, name='send-command'),
]
```



Implémentation

Django Backend (consumers.py)

Django Channels est utilisé pour gérer les WebSockets, permettant de communiquer en temps réel avec l'interface utilisateur.

- Django Channels permet d'utiliser WebSockets pour établir une communication bidirectionnelle en temps réel entre le serveur et le client. Nous avons configuré un consommateur (consumer) pour gérer les connexions WebSocket et envoyer des mises à jour en temps réel aux utilisateurs connectés chaque fois que de nouvelles données sont reçues.
- Extrait de code (Consumer) :

```
consumers.py

import json
from channels.generic.websocket import AsyncWebsocketConsumer

class SensorDataConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        await self.channel_layer.group_add(
            "sensor_data", # Group name for broadcasting
            self.channel_name
        )
        await self.accept()

    async def disconnect(self, close_code):
        await self.channel_layer.group_discard(
            "sensor_data",
            self.channel_name
        )

    async def send_sensor_data(self, event):
        # Send data to WebSocket
        data = event["data"]
        await self.send(text_data=json.dumps(data))
```

Implémentation

Django Backend (Frontend avec Chart.js)

Le frontend (Chart.html) de notre application utilise Chart.js pour afficher les données en temps réel dans des graphiques.

- Pour la visualisation des données en temps réel, nous avons utilisé la bibliothèque Chart.js. Cette bibliothèque permet de créer des graphiques interactifs qui se mettent à jour automatiquement lorsqu'un nouvel ensemble de données est reçu via WebSocket.

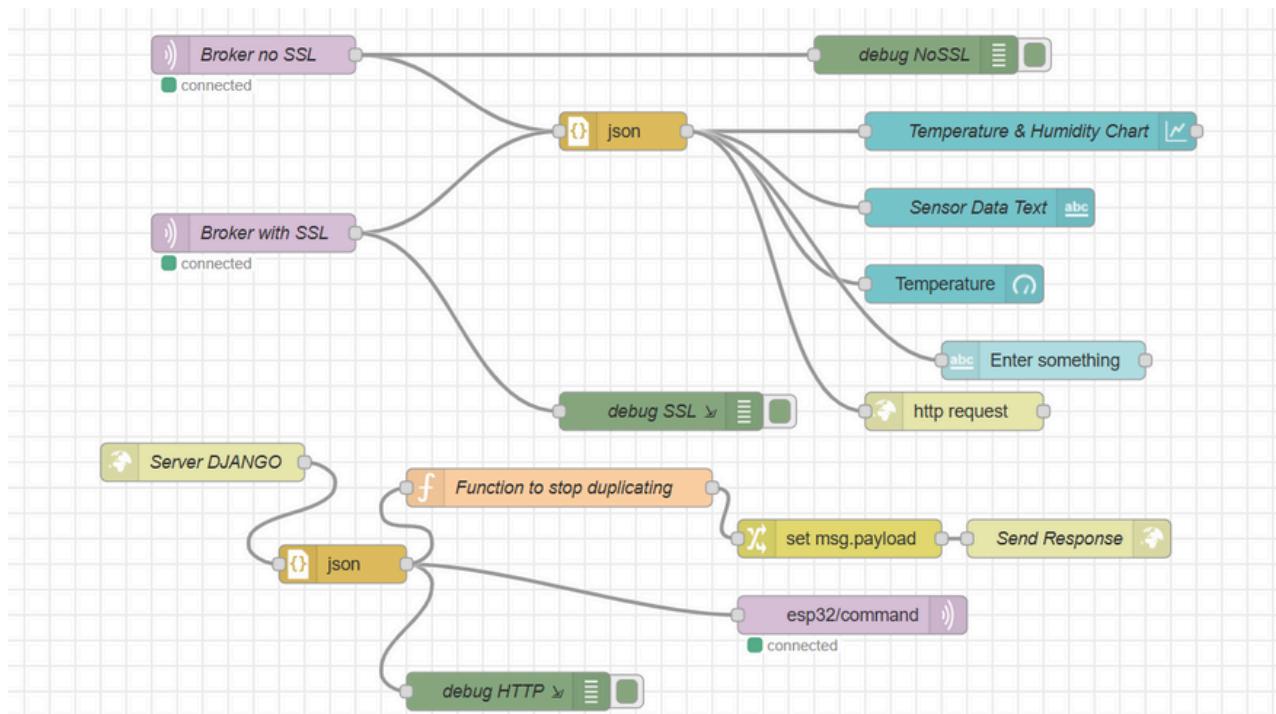


Implémentation

Node-RED

Ce flux Node-RED permet de collecter les données des capteurs, de les traiter, et de les afficher sous forme graphique en temps réel. Il permet également à l'utilisateur d'interagir avec le système, en envoyant des commandes via HTTP ou MQTT. Le traitement des données et la gestion des doublons assurent une performance et une fiabilité maximales.

En fonction de vos besoins spécifiques, vous pouvez adapter les nœuds et leurs connexions pour répondre à des exigences particulières de votre projet.



Implémentation

Node-RED (Réception et Traitement des Données)

1.1. Broker no SSL et Broker with SSL

- Rôle : Ces nœuds sont utilisés pour se connecter à des brokers MQTT (un avec SSL et l'autre sans). Le protocole MQTT est utilisé pour la communication entre le serveur (Node-RED) et le microcontrôleur (ESP32). Les nœuds sont configurés pour écouter des messages publiés par le ESP32 sur un certain topic.
- Connexions :
 - Broker no SSL : Reçoit les données via MQTT sans chiffrement SSL (par exemple mqtt://localhost:1883).
 - Broker with SSL : Reçoit les données via MQTT avec chiffrement SSL (par exemple mqtts://localhost:8883).

1.2. json

- Rôle : Ce nœud est utilisé pour convertir les données reçues en format JSON, ce qui permet de manipuler plus facilement les données (température, humidité, etc.). Le format JSON est un format de données structuré très couramment utilisé dans les systèmes IoT.
- Connexions :
 - Le nœud json reçoit les messages des brokers et convertit les données en JSON.



Implémentation

Node-RED (Réception et Traitement des Données)

1.3. debug No SSL et debug SSL

- Rôle : Ces nœuds permettent de déboguer le flux en affichant les messages reçus dans la console de Node-RED. Cela vous permet de vérifier que les données sont correctement reçues et traitées.
- Connexions :
 - Le nœud debug No SSL est connecté au nœud Broker no SSL pour afficher les messages reçus sans SSL.
 - Le nœud debug SSL est connecté au nœud Broker with SSL pour afficher les messages reçus avec SSL.



Implémentation

Node-RED (Affichage et Traitement des Données)

2.1. Temperature & Humidity Chart

- Rôle : Ce nœud permet d'afficher un graphique en temps réel des données de température et d'humidité reçues. Il permet à l'utilisateur de voir les tendances de ces mesures sur une période donnée.
- Connexions :
 - Le nœud Temperature & Humidity Chart reçoit les données JSON traitées (température et humidité) et les affiche sur un graphique.

2.2. Sensor Data Text

- Rôle : Ce nœud affiche les données textuelles de température et d'humidité. Il est utilisé pour afficher les valeurs actuelles sur l'interface utilisateur de Node-RED.
- Connexions :
 - Le nœud Sensor Data Text est alimenté par les données JSON et affiche les valeurs de température et d'humidité.

2.3. Temperature

- Rôle : Ce nœud affiche uniquement la température (si vous souhaitez afficher cette donnée séparément de l'humidité).
- Connexions :
 - Ce nœud est connecté aux données JSON traitées, mais il ne récupère que la valeur de la température pour l'afficher.



Implémentation

Node-RED (Interaction avec l'Utilisateur)

3.1. http request

- Rôle : Ce nœud permet d'envoyer une requête HTTP à un serveur externe. Il peut être utilisé pour récupérer des données supplémentaires, interagir avec d'autres services ou mettre à jour des informations.
- Connexions :
 - Ce nœud envoie des requêtes HTTP à un serveur Django ou à un autre service externe en fonction de la configuration.



Implémentation

Node-RED (Traitement et Envoi des Données)

4.1. Server DJANGO

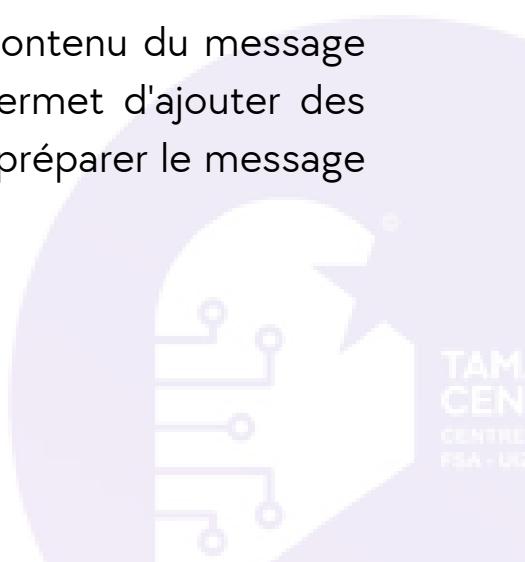
- Rôle : Ce nœud représente probablement une fonction Python définie dans un environnement Django. Il est utilisé pour effectuer des traitements complexes sur les données reçues, comme l'enregistrement en base de données ou l'exécution de requêtes spécifiques.
- Connexions :
 - Ce nœud reçoit les messages traités par les nœuds précédents (comme la température et l'humidité) et les envoie à un serveur Django pour traitement.

4.2. Function to stop duplicating

- Rôle : Cette fonction est utilisée pour éviter le traitement en double des messages. Elle vérifie si un message a déjà été traité et l'ignore si nécessaire.
- Connexions :
 - Ce nœud est connecté après la réception des données des capteurs, avant qu'elles ne soient envoyées à d'autres nœuds de traitement ou de stockage.

4.3. set msg.payload

- Rôle : Ce nœud est utilisé pour modifier le contenu du message avant de l'envoyer au nœud suivant. Cela permet d'ajouter des informations, de formater les données ou de préparer le message pour le prochain traitement.



Implémentation

Node-RED (Traitement et Envoi des Données)

- Connexions :
 - Ce nœud est généralement utilisé avant d'envoyer les messages aux nœuds de réponse ou d'affichage.

4.4. Send Response

- Rôle : Ce nœud envoie une réponse au client (généralement après une requête HTTP). Cela peut être un message confirmant la réception d'une commande ou renvoyant des résultats.
- Connexions :
 - Ce nœud est connecté au nœud http request pour envoyer une réponse au client.

4.5. esp32/command

- Rôle : Ce nœud est utilisé pour envoyer une commande au microcontrôleur ESP32 via MQTT. Cela peut être une commande pour changer un état ou pour envoyer des données à l'ESP32.
- Connexions :
 - Ce nœud est connecté à un nœud de traitement ou à un nœud de contrôle pour envoyer des commandes à l'ESP32 (par exemple, afficher un message ou effectuer une action spécifique).



Implémentation

Node-RED (Connexions entre les Nœuds)

Les nœuds sont connectés entre eux de manière à former un flux de traitement des données. Par exemple :

- Les messages reçus par les brokers sont envoyés au nœud json pour être convertis en format JSON.
- Les données JSON sont ensuite envoyées aux nœuds debug pour déboguer les messages et vérifier les informations reçues.
- Ensuite, les données sont envoyées au nœud Temperature & Humidity Chart pour être affichées sous forme de graphique.
- Les données de température et d'humidité sont aussi envoyées à d'autres nœuds comme Sensor Data Text et Temperature pour l'affichage textuel.



Implémentation

Wokwi

Dans ce projet, nous allons connecter un ESP32 à un broker MQTT (dans ce cas broker.hivemq.com), publier des données simulées de température et d'humidité, et les afficher sur un écran LCD. L'ESP32 peut également recevoir des commandes via MQTT et les afficher sur l'écran LCD.

Je vais expliquer chaque partie du code étape par étape, y compris l'intégration avec Wokwi et l'utilisation de MQTT.



Implémentation

Wokwi (Inclusion des bibliothèques nécessaires)

Le code commence par inclure les bibliothèques essentielles pour la gestion du réseau, de MQTT, du LCD et des connexions SSL (si activées).

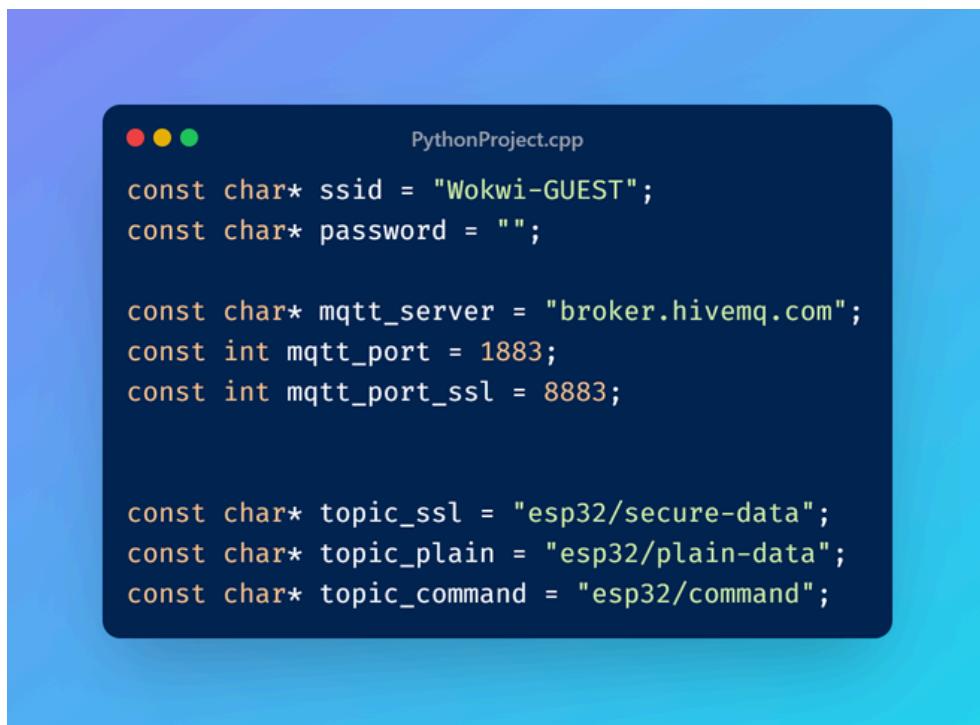


- **WiFi.h** : Bibliothèque pour gérer la connexion Wi-Fi avec l'ESP32.
- **PubSubClient.h** : Bibliothèque pour gérer la communication MQTT.
- **WiFiClientSecure.h** : Utilisée pour gérer la connexion SSL/TLS si vous utilisez une connexion sécurisée.
- **LiquidCrystal_I2C.h** : Pour gérer l'écran LCD via une interface I2C.

Implémentation

Wokwi (Définition des constantes)

Les constantes suivantes définissent les informations nécessaires pour la connexion au réseau Wi-Fi et au serveur MQTT.



The screenshot shows a code editor window titled "PythonProject.cpp". The code defines several constants used for MQTT communication:

```
● ● ● PythonProject.cpp
const char* ssid = "Wokwi-GUEST";
const char* password = "";

const char* mqtt_server = "broker.hivemq.com";
const int mqtt_port = 1883;
const int mqtt_port_ssl = 8883;

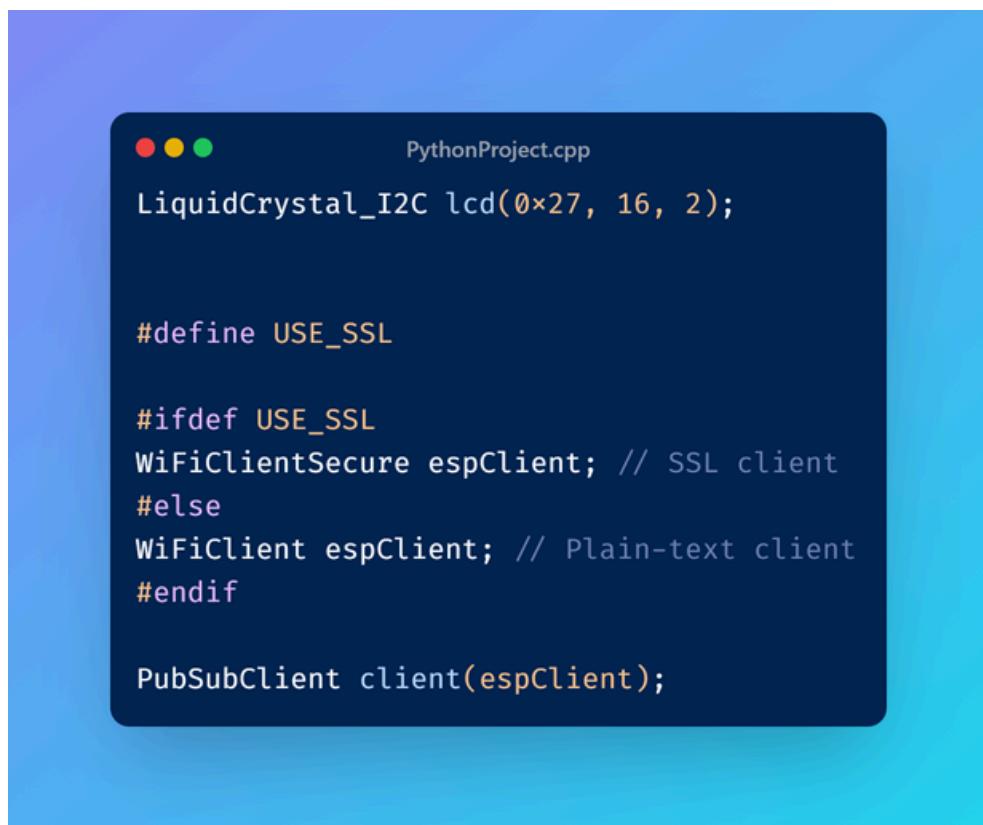
const char* topic_ssl = "esp32/secure-data";
const char* topic_plain = "esp32/plain-data";
const char* topic_command = "esp32/command";
```

- **ssid et password** : Les informations de connexion Wi-Fi. Vous pouvez utiliser "Wokwi-GUEST" comme réseau dans l'émulateur Wokwi.
- **mqtt_server** : L'adresse du broker MQTT (ici broker.hivemq.com).
- **mqtt_port et mqtt_port_ssl** : Les ports pour la connexion MQTT (non sécurisé sur 1883 et sécurisé via SSL sur 8883).
- **topic_ssl, topic_plain et topic_command** : Les différents sujets MQTT pour publier et recevoir des messages.

Implémentation

Wokwi (Définition des objets et de la connexion LCD)

Le code initialise l'écran LCD avec l'adresse I2C 0x27 (par défaut pour beaucoup d'écrans LCD) et configure le client MQTT.



```
PythonProject.cpp

LiquidCrystal_I2C lcd(0x27, 16, 2);

#define USE_SSL

#ifndef USE_SSL
WiFiClientSecure espClient; // SSL client
#else
WiFiClient espClient; // Plain-text client
#endif

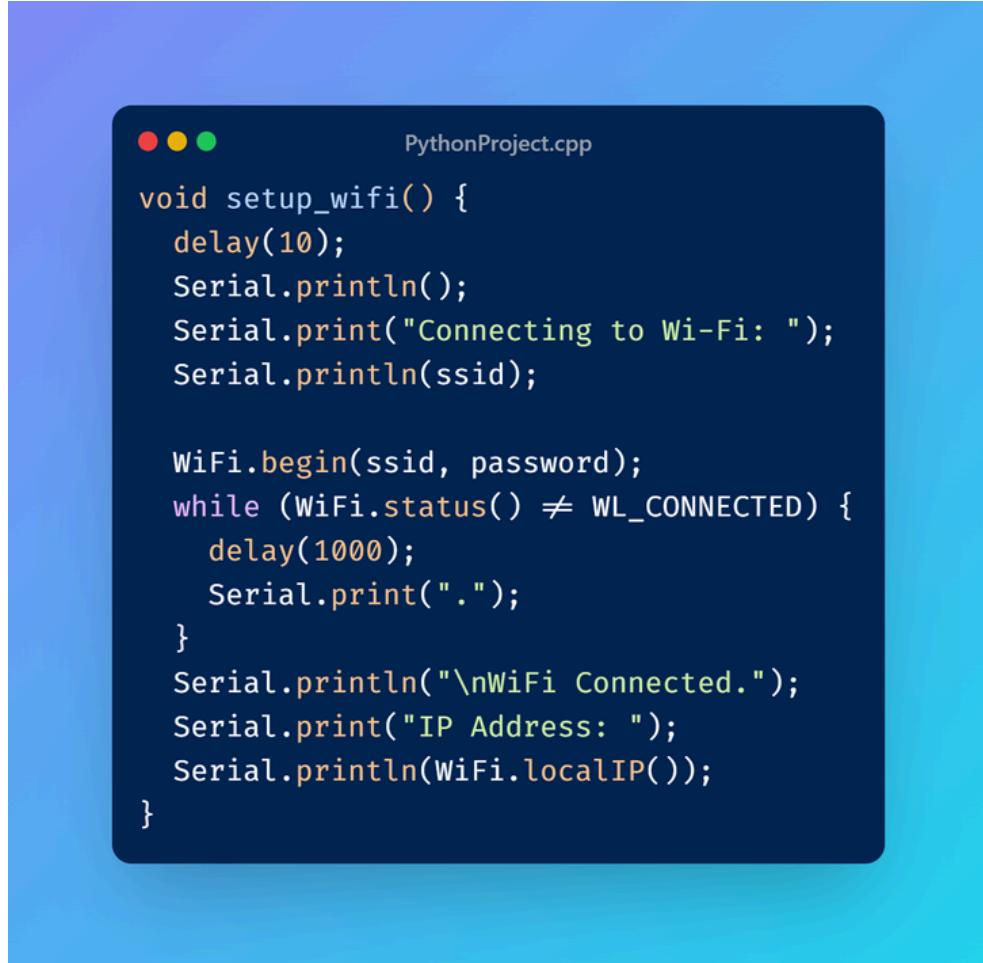
PubSubClient client(espClient);
```

- **LiquidCrystal_I2C lcd** : Définit un objet pour contrôler un écran LCD de 16 colonnes et 2 lignes via I2C.
- **WiFiClient espClient** : Un objet pour la gestion de la connexion réseau. Il est soit sécurisé (SSL) ou non, en fonction de la directive #ifdef USE_SSL.
- **PubSubClient client** : L'objet principal pour gérer la communication avec le serveur MQTT.

Implémentation

Wokwi (Configuration du Wi-Fi)

La fonction `setup_wifi()` permet de connecter l'ESP32 au réseau Wi-Fi.



```
PythonProject.cpp

void setup_wifi() {
    delay(10);
    Serial.println();
    Serial.print("Connecting to Wi-Fi: ");
    Serial.println(ssid);

    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(1000);
        Serial.print(".");
    }
    Serial.println("\nWiFi Connected.");
    Serial.print("IP Address: ");
    Serial.println(WiFi.localIP());
}
```

- Le code tente de se connecter au Wi-Fi en utilisant les informations fournies (`ssid` et `password`).
- Il vérifie la connexion Wi-Fi à chaque seconde et affiche l'adresse IP de l'ESP32 lorsque la connexion est établie.



Implémentation

Wokwi (Reconnexion MQTT)

La fonction reconnect() gère la reconnexion au serveur MQTT si l'ESP32 perd sa connexion.

```
PythonProject.cpp

void reconnect() {
    while (!client.connected()) {
        Serial.print("Connecting to MQTT Broker ... ");
        String clientId = "ESP32-" + String(random(0xffff), HEX);

        if (client.connect(clientId.c_str())) {
            Serial.println("Connected.");

            client.subscribe(topic_command);
            Serial.println("Subscribed to topic: " + String(topic_command));
        } else {
            Serial.print("Failed. Error Code: ");
            Serial.println(client.state());
            delay(5000);
        }
    }
}
```

- Le client essaie de se connecter avec un clientId unique généré aléatoirement.
- Une fois connecté, il s'abonne au sujet esp32/command pour recevoir des commandes.



Implémentation

Wokwi (Callback MQTT)

Le callback `callback()` est appelé chaque fois qu'un message est reçu sur un sujet auquel l'ESP32 est abonné.

```
PythonProject.cpp

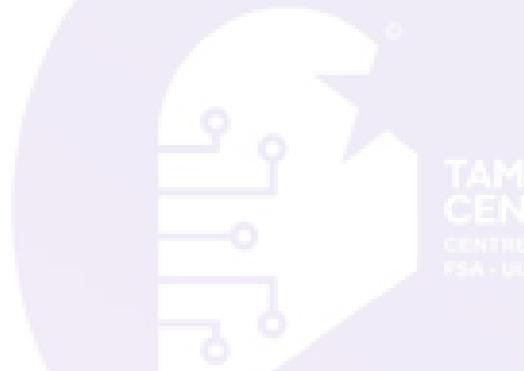
void callback(char* topic, byte* payload, unsigned int length) {
    String message = "";
    for (int i = 0; i < length; i++) {
        message += (char)payload[i];
    }
    Serial.println("Message received: " + message);

    if (String(topic) == topic_command) {

        int startIdx = message.indexOf(":") + 1;
        String trimmedMessage = message.substring(startIdx, message.length() - 1);
        Serial.println("Trimmed message: " + trimmedMessage);

        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Command:");
        lcd.setCursor(0, 1);
        lcd.print(trimmedMessage);
    }
}
```

- Lorsqu'un message est reçu sur le topic `esp32/command`, il est extrait, nettoyé, et affiché sur l'écran LCD.



Implémentation

Wokwi (Publication des données MQTT (température et humidité))

Dans la boucle `loop()`, l'ESP32 génère des valeurs aléatoires pour la température et l'humidité, puis les publie sur le serveur MQTT.

```
PythonProject.cpp

void loop() {
    if (!client.connected()) {
        reconnect();
    }
    client.loop();

    float temperature = random(0, 60);
    float humidity = random(16, 70);

    String payload = String("{\"temperature\":") + temperature + ",\"humidity\":"
                           + humidity + "}";

    #ifdef USE_SSL
    if (client.publish(topic_ssl, payload.c_str())) {
        Serial.println("Published to secure topic: " + payload);
    }
    #else
    if (client.publish(topic_plain, payload.c_str())) {
        Serial.println("Published to plain topic: " + payload);
    }
    #endif

    delay(2000);
}
```

- Les données aléatoires de température et d'humidité sont envoyées sous forme de JSON.
- Ces données sont envoyées soit sur un sujet sécurisé (`topic_ssl`) soit sur un sujet non sécurisé (`topic_plain`), selon que SSL est activé ou non.
- Les messages sont envoyés toutes les 2 secondes.

Implémentation

Wokwi (Connexion au serveur MQTT avec ou sans SSL)

Selon la définition de `#define USE_SSL`, la connexion à MQTT se fait soit avec une connexion sécurisée SSL, soit une connexion non sécurisée.

- **SSL activé** : Si `USE_SSL` est défini, un certificat CA est utilisé pour établir une connexion sécurisée sur le port 8883.
- **Non sécurisé** : Si SSL n'est pas activé, la connexion se fait via le port standard 1883.



Résultats et Tests

Cette section décrit le processus de tests et d'analyse des paquets de données, ainsi que l'implémentation de l'interface graphique et de l'affichage LCD pour les commandes MQTT envoyées et reçues par l'ESP32. Les tests sont réalisés en utilisant des outils comme Wireshark pour l'analyse des paquets, Node-RED pour l'échange des données, une interface graphique (Django) pour afficher les data envoyées par ESP32 et un écran LCD pour afficher les commandes MQTT reçues sur l'ESP32.



Résultats et Tests

Analyse des Paquets avec Wireshark

Wireshark a été utilisé pour analyser les paquets échangés entre l'ESP32 et le broker MQTT. Deux types de communication ont été étudiés : la communication non sécurisée (via le port 1883) et la communication sécurisée (via SSL sur le port 8883).



Résultats et Tests

Analyse des Paquets avec Wireshark

A. Capturer des Paquets Non Sécurisés (Port 1883)

La communication MQTT non sécurisée utilise le port 1883, permettant d'échanger des messages en clair. Pour capturer ces paquets dans Wireshark, nous avons filtré le trafic réseau sur le port 1883. Les paquets capturés incluent :

1. CONNECT : La demande de connexion au serveur MQTT.
2. PUBLISH : Les messages MQTT envoyés par l'ESP32, contenant les données telles que la température et l'humidité.
3. SUBSCRIBE : La demande d'abonnement à un sujet MQTT spécifique (par exemple, esp32/command).
4. ACK : Les confirmations des messages MQTT reçus.

Dans le cas de la communication non sécurisée, les informations échangées, telles que la température et l'humidité, sont visibles en texte clair dans les paquets de type PUBLISH. Un exemple de paquet observé pourrait être :

The screenshot shows a Wireshark capture window titled "wokwi (13).pcap". The packet list pane displays several MQTT frames. Frame 158 is selected, showing a Publish message from "10.10.0.2" to "3.77.223.206" with topic "esp32/command" and payload "temperature:25.00,humidity:45.0". The details pane shows the MQTT frame structure, and the bytes pane shows the raw hex and ASCII data.

No.	Time	Source	Destination	Protocol	Length	Info
Number	Time	Source	Destination	Protocol	Length	Info
1	1.351004844	10.10.0.2	3.77.223.206	MQTT	96	Connect Command
2	1.213910927	3.77.223.206	10.10.0.2	MQTT	76	Connect Ack
3	1.218739298	10.10.0.2	3.77.223.206	MQTT	92	Subscribe Request (id=2) [esp32/command]
4	1.316025119	10.10.0.2	3.77.223.206	MQTT	130	Publish Message [esp32/plain-data]
5	1.613913427	3.77.223.206	10.10.0.2	MQTT	77	Subscribe Ack (id=2)
158	1.227170654	10.10.0.2	3.77.223.206	MQTT	130	Publish Message [esp32/plain-data]
181	1.231590037	10.10.0.2	3.77.223.206	MQTT	130	Publish Message [esp32/plain-data]

Résultats et Tests

Analyse des Paquets avec Wireshark

Cela signifie que la donnée de température est 25.0°C et l'humidité est de 45%. Les messages sont envoyés sans aucun chiffrement et peuvent être facilement lus dans Wireshark.

B. Capturer des Paquets Sécurisés (Port 8883)

La communication MQTT sécurisée utilise le port 8883, fonctionnant avec SSL/TLS pour assurer la confidentialité des données. Dans Wireshark, nous avons capturé le trafic sur ce port. Les paquets capturés dans ce scénario sont chiffrés, et les données ne sont pas directement visibles. Voici les points importants de cette capture :

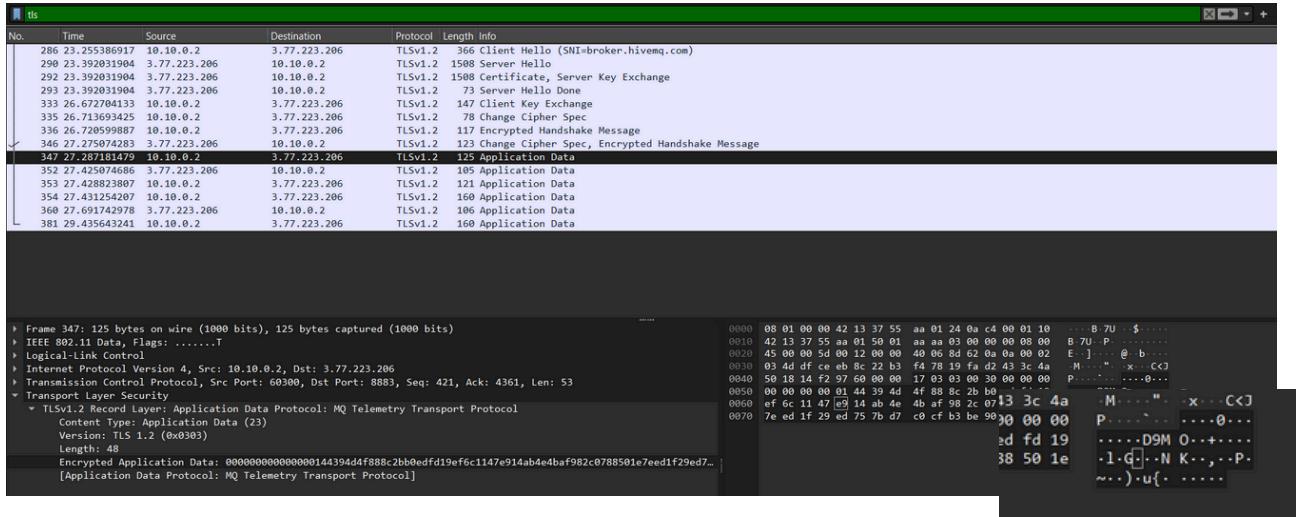
1. Handshake SSL/TLS : L'échange initial des certificats et la négociation des clés de chiffrement.
2. Données Chiffrées : Après l'établissement de la connexion sécurisée, les paquets de données sont chiffrés, et leur contenu (température, humidité, etc.) ne peut pas être directement observé sans déchiffrement.

Les paquets SSL/TLS sont visibles sous forme de données chiffrées dans Wireshark. Un exemple de paquet capturé pourrait être :



Résultats et Tests

Analyse des Paquets avec Wireshark



Pour déchiffrer ces paquets, il serait nécessaire de disposer de la clé privée du serveur ou d'utiliser un certificat CA pour déchiffrer les données dans Wireshark.

Résultats et Tests

Interface Graphique avec Django

Une interface graphique a été développée sur un serveur Django, permettant de visualiser en temps réel les données échangées via MQTT. Cette interface sert à afficher les valeurs de température et d'humidité envoyées par l'ESP32 ainsi que les commandes reçues par MQTT.

A. Développement de l'Interface avec Django

Le serveur Django a été configuré pour servir de backend et interagir avec l'ESP32 via MQTT. Voici les étapes principales pour le développement de l'interface graphique :

1. Configuration du serveur MQTT : Le serveur Django est connecté à un broker MQTT pour recevoir les données envoyées par l'ESP32.
2. Création de l'Interface Utilisateur : Un tableau de bord interactif a été développé avec Django et affiché dans le navigateur. Ce tableau de bord permet de visualiser :
 - Les graphiques de la température et de l'humidité.
 - Les commandes envoyées à l'ESP32.
3. Envoi de Commandes à l'ESP32 : L'interface permet également à l'utilisateur d'envoyer des commandes à l'ESP32, comme la mise à jour de paramètres ou l'activation de certaines fonctions.

B. Fonctionnalités de l'Interface

- Affichage des Données : L'interface Django affiche en temps réel les données de température et d'humidité reçues du périphérique ESP32.
- Visualisation des Commandes : Les commandes envoyées au périphérique ESP32 via MQTT sont également affichées dans l'interface graphique, permettant une gestion centralisée des opérations.

Résultats et Tests

Interface Graphique avec Django

Une interface graphique a été développée sur un serveur Django, permettant de visualiser en temps réel les données échangées via MQTT. Cette interface sert à afficher les valeurs de température et d'humidité envoyées par l'ESP32 ainsi que les commandes reçues par MQTT.

A. Développement de l'Interface avec Django

Le serveur Django a été configuré pour servir de backend et interagir avec l'ESP32 via MQTT. Voici les étapes principales pour le développement de l'interface graphique :

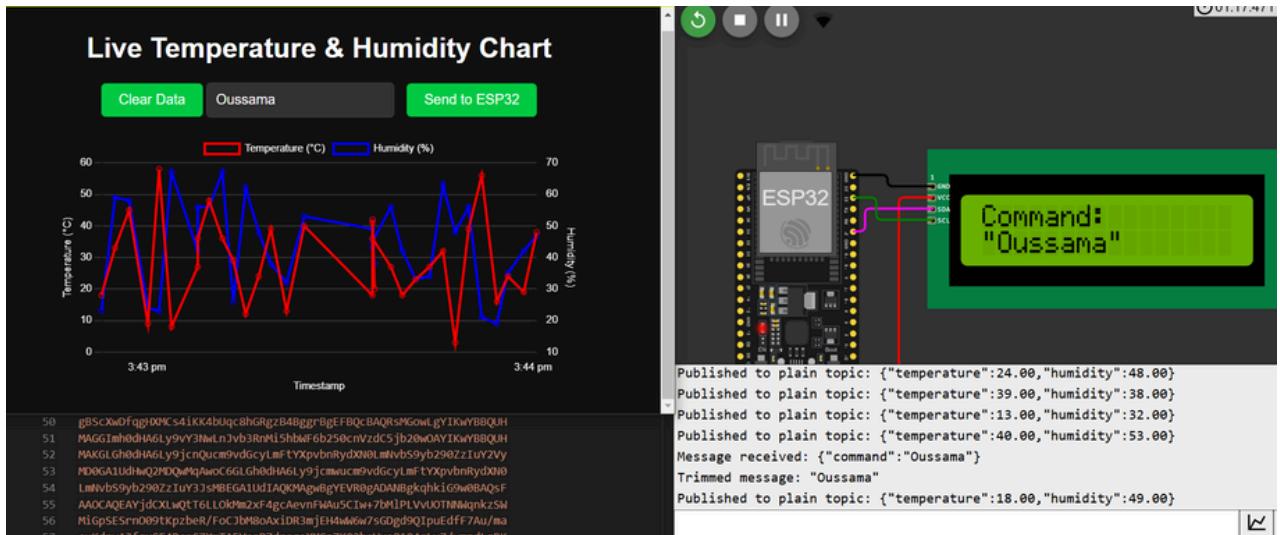
1. Configuration du serveur MQTT : Le serveur Django est connecté à un broker MQTT pour recevoir les données envoyées par l'ESP32.
2. Création de l'Interface Utilisateur : Un tableau de bord interactif a été développé avec Django et affiché dans le navigateur. Ce tableau de bord permet de visualiser :
 - Les graphiques de la température et de l'humidité.
 - Les commandes envoyées à l'ESP32.
3. Envoi de Commandes à l'ESP32 : L'interface permet également à l'utilisateur d'envoyer des commandes à l'ESP32, comme la mise à jour de paramètres ou l'activation de certaines fonctions.

B. Fonctionnalités de l'Interface

- Affichage des Données : L'interface Django affiche en temps réel les données de température et d'humidité reçues du périphérique ESP32.
- Visualisation des Commandes : Les commandes envoyées au périphérique ESP32 via MQTT sont également affichées dans l'interface graphique, permettant une gestion centralisée des opérations.

Résultats et Tests

Interface Graphique avec Django



L'interface Django récupère les informations en temps réel via des requêtes AJAX pour une mise à jour dynamique des données affichées. Ce système est alimenté par les données MQTT reçues du broker.

L'ESP32 est équipé d'un écran LCD pour afficher les commandes MQTT reçues. Lorsque l'ESP32 reçoit une commande via MQTT, celle-ci est affichée directement sur l'écran LCD, fournissant ainsi un retour immédiat à l'utilisateur.

Conclusion

Ce projet a permis d'implémenter un système de surveillance et de contrôle à distance basé sur un ESP32, un serveur Django pour l'interface graphique, et l'utilisation de MQTT pour la communication entre les différents composants du système. L'ESP32 collecte des données (température et humidité) et interagit avec un écran LCD pour afficher les commandes reçues via MQTT.

L'utilisation de Node-RED a permis de sécuriser la communication entre les différentes parties en utilisant le protocole SSL/TLS, garantissant ainsi la confidentialité et l'intégrité des données échangées. Le serveur Django fournit une interface graphique permettant à l'utilisateur de visualiser en temps réel les données de l'ESP32 et d'envoyer des commandes de manière interactive.

Les tests réalisés ont démontré que le système fonctionne efficacement avec des communications sécurisées et une interface utilisateur intuitive. La solution proposée est adaptable à divers scénarios d'IoT, offrant une plateforme sécurisée pour la gestion des capteurs et le contrôle à distance des dispositifs.

En conclusion, le système développé répond aux objectifs initiaux de communication sécurisée et d'interaction avec l'utilisateur tout en garantissant la facilité d'intégration et d'extensibilité grâce à l'utilisation de technologies modernes comme MQTT, Django et Node-RED.

Bibliographie

Pour la réalisation de ce projet, plusieurs ressources et sites web ont été consultés pour obtenir des informations techniques, des bibliothèques et des guides de mise en œuvre. Voici les principales sources utilisées :

1. Django Documentation

Documentation officielle de Django pour la création d'applications web robustes et sécurisées.

- <https://docs.djangoproject.com/en/stable/>

2. Node-RED Documentation

Site officiel de Node-RED, outil de développement pour la création de flux IoT et l'automatisation de processus.

- <https://nodered.org/docs/>

3. ESP32 Documentation

Documentation officielle pour la configuration et l'utilisation de l'ESP32, un microcontrôleur très populaire dans l'IoT.

- <https://docs.espressif.com/projects/esp-idf/en/latest/>

4. MQTT Protocol

Guide détaillant le protocole MQTT, utilisé pour les communications légères et fiables dans les projets IoT.

- <https://mqtt.org/>

6. Wireshark

Documentation et guides pour analyser les paquets de données réseau à l'aide de Wireshark.

- <https://www.wireshark.org/>



Bibliographie

1. GitHub

Références sur les projets open source liés à l'ESP32, Django, Node-RED et MQTT disponibles sur GitHub.

- <https://github.com/>

2. PubSubClient Library for Arduino

Documentation de la bibliothèque PubSubClient utilisée pour la gestion des communications MQTT avec l'ESP32.

- <https://pubsubclient.knolleary.net/>

3. LiquidCrystal I2C Library for Arduino

Documentation de la bibliothèque LiquidCrystal pour gérer les écrans LCD via l'interface I2C avec l'ESP32.

- <https://github.com/fdebrabander/Arduino-LiquidCrystal-I2C-library>

4. MQTT with SSL

Guide détaillant comment configurer MQTT avec SSL pour des connexions sécurisées.

- <https://www.hivemq.com/blog/mqtt-essentials/>

Ces ressources ont fourni des informations essentielles pour configurer les composants du projet, y compris le serveur backend Django, les flux Node-RED pour le traitement des données et la gestion des communications sécurisées via MQTT. Elles ont également aidé à l'analyse des paquets réseau pour tester la communication sécurisée entre les différents dispositifs.

Dépôt GitHub

Vous pouvez consulter le code source complet du projet sur le dépôt GitHub suivant. Scannez le QR code ci-dessous pour y accéder directement.



Le dépôt contient tout le code nécessaire pour configurer et tester le projet, y compris l'ESP32, le serveur Django, et les flux Node-RED.

