UNIVERSIDAD NACIONAL DE INGENIERÍA

FACULTAD DE CIENCIAS

ESCUELA DE CIENCIAS DE LA COMPUTACIÓN

SISTEMAS OPERATIVOS CC222

INFORME DE PROYECTO

Aplicación de IoT para el desarrollo de un sistema de control de un ventilador casero usando el microcontrolador ESP32

Sección: A

Apellidos y Nombres	**Código de alumno**
VEGA BENDEZU, Alex	20235519K
MORI MALCA, Jean Paul	20245016A
DELGADO ROMERO, Gustavo Iván	20235009B

Nombre de los Docentes:
– BAZÁN CABANILLAS, Carlos Alberto

Fecha de entrega del informe: 2 de Julio del 2025

2025-I

Resumen

Este proyecto presenta la implementación de un sistema IoT basado en el microcontrolador ESP32 utilizando MicroPython sobre FreeRTOS, demostrando la viabilidad de integrar lenguajes de alto nivel con sistemas operativos de tiempo real en plataformas con recursos limitados. El sistema desarrollado integra sensores de temperatura y humedad, actuadores (LEDs y ventiladores) y comunicación inalámbrica mediante WiFi y MQTT para la interacción con servicios en la nube.

La arquitectura del proyecto se fundamenta en la gestión eficiente de múltiples tareas concurrentes, incluyendo la lectura continua de sensores, el control de actuadores y la comunicación bidireccional con un servidor remoto. El uso de FreeRTOS como sistema operativo subyacente permite la programación de tareas en tiempo real, garantizando la respuesta oportuna a eventos críticos mientras se mantiene la eficiencia energética del sistema.

El sistema implementado incluye un servidor Docker desplegado en DigitalOcean que aloja tres servicios principales: una API Gateway para gestionar solicitudes HTTP y registros de dispositivos, un Event Hub para coordinar eventos y mensajes entre dispositivos, y un Dashboard para visualizar y controlar dispositivos a través de una interfaz gráfica. Esta arquitectura distribuida demuestra la escalabilidad y modularidad inherentes a los sistemas IoT modernos.

Los resultados obtenidos validan la efectividad de la integración MicroPython-FreeRTOS, mostrando un sistema capaz de procesar datos de sensores en tiempo real, transmitir información a la nube y responder a comandos remotos de control. El proyecto contribuye al estado del arte en sistemas embebidos al demostrar que es posible desarrollar aplicaciones IoT complejas utilizando herramientas de alto nivel sin comprometer el rendimiento en tiempo real, abriendo nuevas posibilidades para el desarrollo rápido de sistemas IoT robustos y escalables.

Introducción

Motivación

La creciente demanda de dispositivos IoT con capacidades de procesamiento en tiempo real ha impulsado la necesidad de comprender cómo los lenguajes de alto nivel como MicroPython pueden aprovechar las capacidades de sistemas operativos de tiempo real como FreeRTOS, especialmente en plataformas con recursos limitados como el ESP32.

Objetivos

El objetivo principal de este proyecto es analizar y documentar la implementación de un sistema IoT basado en ESP32 utilizando MicroPython sobre FreeRTOS, con énfasis en los mecanismos de concurrencia y gestión de recursos.

Descripción del Problema

Los sistemas IoT modernos requieren gestionar múltiples tareas simultáneas (lectura de sensores, control de actuadores, comunicación en red) con recursos limitados. El desafío consiste en implementar estas funcionalidades manteniendo la fiabilidad, eficiencia y capacidad de respuesta en tiempo real, utilizando herramientas que faciliten el desarrollo rápido como MicroPython.

Estructura del proyecto

El proyecto se estructura en torno a un sistema de monitoreo y control ambiental que integra sensores de temperatura y humedad, actuadores (LEDs y ventiladores) y comunicación mediante WiFi y MQTT para la interacción con servicios en la nube.

Índice

1. Estado del arte

1.1. Trabajos Relacionados

La integración de lenguajes interpretados de alto nivel con sistemas operativos de tiempo real ha sido objeto de diversos estudios. Proyectos como CircuitPython y MicroPython han demostrado la viabilidad de ejecutar código Python en microcontroladores, mientras que implementaciones como Zerynth han explorado específicamente la integración con RTOS.

El hardware utilizado en este proyecto se el microcontrolador ESP32, el cual es un componente clave en muchos proyectos de Internet de las cosas (IoT), principalmente debido a su versatilidad y capacidades inalámbricas integradas. El ESP32 ofrece procesamiento de doble núcleo, Wi-Fi integrado, Bluetooth y Bluetooth Low Energy (BLE), lo que lo convierte en un candidato ideal para diversas aplicaciones de IoT, incluido el monitoreo ambiental, la automatización del hogar inteligente, el control industrial y los sistemas de IoT descentralizados.

Figura 1: Placa Esp32 (WROOM)

1.1.1. ESP32 en aplicaciones de IoT

El ESP32 es particularmente adecuado para la transmisión y monitoreo de datos en tiempo real en proyectos de IoT. Se ha convertido en el microcontrolador preferido para los sistemas de IoT debido a su bajo consumo de energía y su capacidad para manejar la comunicación inalámbrica de manera eficiente, especialmente cuando es necesario transmitir datos a la nube para su posterior procesamiento. En el monitoreo de la calidad del aire, por ejemplo, el ESP32 ha sido fundamental para recopilar datos en tiempo real de los sensores y transmitirlos a la nube para su análisis.[1].

En el contexto de la seguridad ambiental, ESP32 desempeña un papel fundamental en el seguimiento de los niveles de temperatura y humedad en las zonas forestales, transmitiendo estos datos a un sistema central de detección de incendios, garantizando intervenciones oportunas.[2]. La capacidad de procesar datos de sensores en tiempo real y transmitirlos

[1] IV.
[2] VIII.

Figura 2: Aplicacion de IoT en casas inteligentes

para monitoreo remoto hace que ESP32 sea ideal para aplicaciones de IoT críticas para la seguridad.

1.1.2. Variantes y componentes clave del ESP32

Han surgido varias variantes del microcontrolador ESP32, como el ESP32-C3, para satisfacer necesidades específicas en aplicaciones de IoT. Por ejemplo, ESP32-C3 ofrece un excelente rendimiento inalámbrico manteniendo un bajo consumo de energía, lo que lo hace adecuado para proyectos que exigen una batería de larga duración y una conectividad confiable.[3]. Estas características son esenciales en muchos casos de uso de IoT, donde el bajo consumo de energía es una prioridad.

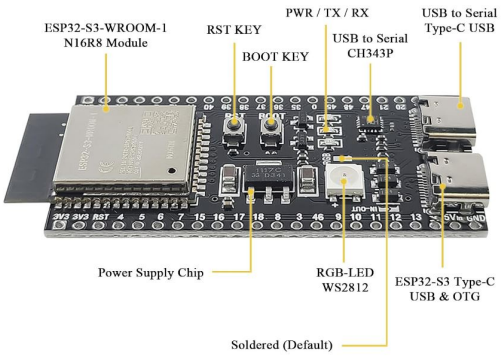


Figura 3: Componentes del ESP32

Además, el ESP32 es capaz de actuar como un punto de acceso Wi-Fi, permitiendo que los dispositivos se conecten directamente a él para comunicarse e intercambiar datos. Esta capacidad permite que el ESP32 funcione como un centro central en redes de IoT, admitiendo varios protocolos inalámbricos como Wi-Fi y Bluetooth, que son fundamentales en los sistemas de automatización industrial y de hogares inteligentes.[4].

La arquitectura de doble núcleo del ESP32 mejora aún más sus capacidades de procesamiento, permitiéndole manejar tareas complejas como la recopilación, el procesamiento y la transmisión de datos de sensores en tiempo real. Estas características hacen del

[3] IX.
[4] X.

ESP32 una poderosa plataforma para aplicaciones de IoT, particularmente en sistemas industriales y domésticos inteligentes.[5].

1.1.3. ESP32 en Casas Inteligentes y aplicaciones industriales

El ESP32 se ha utilizado ampliamente en sistemas domésticos inteligentes, donde su bajo costo y diseño energéticamente eficiente lo han convertido en un componente esencial en el monitoreo y control de dispositivos como luces, calefacción y ventilación. Permite a los propietarios realizar un seguimiento del consumo de energía y ajustar la configuración ambiental de forma remota, lo que contribuye a la automatización del hogar con eficiencia energética.[6].

El ESP32 también está integrado en aplicaciones industriales de IoT, como el control de brazos robóticos en fábricas. Su capacidad para transmitir datos en tiempo real le permite gestionar los movimientos precisos de los dispositivos robóticos, lo que permite una operación remota eficiente.[7].Además, en sistemas SCADA de bajo costo, el ESP32 se ha aprovechado para monitorear y controlar procesos industriales a través de plataformas como ThingsBoard y MQTT.[8]. Estos casos de uso resaltan la flexibilidad del ESP32 en entornos de IoT tanto domésticos como industriales.

Figura 4: Efergy Solar Kit

1.1.4. Pros y contras de ESP32 en aplicaciones de IoT

El ESP32 ofrece numerosas ventajas en aplicaciones de IoT, que incluyen:

- Bajo consumo de energía, esencial para proyectos que funcionan con baterías y entornos con limitaciones energéticas.[9].
- Procesamiento de doble núcleo, lo que le permite manejar tareas complejas en tiempo real, como el procesamiento y la transmisión de datos de sensores.[10].

[5] III.
[6] II.
[7] I.
[8] VI.
[9] IX.
[10] III.

- Wi-Fi y Bluetooth integrados, que facilitan la integración perfecta con otros dispositivos y redes[11].
- Costo-Beneficio, lo que la convierte en una opción popular tanto para proyectos de pequeña escala como para grandes implementaciones industriales.[12].

Sin embargo, algunos desafíos o limitaciones incluyen:

- Limited processing power compared to more advanced microcontrollers, which Potencia de procesamiento limitada en comparación con microcontroladores más avanzados, lo que puede restringir su uso en tareas de cálculo extremadamente intenso.
- Las limitaciones de memoria pueden ser un problema en aplicaciones que requieren grandes cantidades de procesamiento de datos, particularmente en escenarios que involucran múltiples tareas simultáneas.

A pesar de estas limitaciones, el ESP32 sigue siendo una plataforma ampliamente adoptada debido a su equilibrio entre rendimiento, costo y versatilidad, lo que lo convierte en una piedra angular en el desarrollo de sistemas de IoT.

[11] X.
[12] V.

1.2. Conceptos teóricos

Para garantizar que los lectores que no están familiarizados con la informática comprendan los elementos centrales de este proyecto, describiremos algunos de los conceptos clave relacionados con el microcontrolador ESP32 y su función en los sistemas de Internet de las cosas (IoT). Las siguientes definiciones aclararán términos y tecnologías esenciales para comprender cómo se puede utilizar ESP32 para controlar dispositivos, como un ventilador, en aplicaciones de IoT.

1.2.1. Internet of Things (IoT)

El Internet de las cosas (IoT) se refiere a una red de objetos físicos (o çosas") equipados con sensores, software y otras tecnologías que les permiten conectarse e intercambiar datos a través de Internet. Esto permite monitorear y controlar los dispositivos de forma remota, brindando soluciones más eficientes, automatizadas e inteligentes. Ejemplos de aplicaciones de IoT incluyen hogares inteligentes, dispositivos de salud portátiles y sistemas de automatización industrial.[13].

1.2.2. Microcontroladores y ESP32

Un microcontrolador es un circuito integrado compacto diseñado para gobernar una operación específica en un sistema integrado, como controlar un dispositivo como un ventilador. Incluye procesador, memoria y periféricos de entrada/salida. El ESP32, desarrollado por Espressif Systems, es un microcontrolador potente y asequible que se utiliza en muchas aplicaciones de IoT. Cuenta con Wi-Fi y Bluetooth integrados, procesamiento de doble núcleo y soporte para varios periféricos, lo que lo hace muy adecuado para conectar dispositivos a Internet y controlarlos.[14].

1.2.3. Wi-Fi y Bluetooth en IoT

Wi-Fi y Bluetooth son protocolos de comunicación inalámbrica que se utilizan para conectar dispositivos en un sistema IoT. El microcontrolador ESP32 tiene soporte integrado para ambos, lo que le permite conectarse a una red Wi-Fi para comunicarse por Internet o usar Bluetooth para la comunicación local de dispositivo a dispositivo. Esto es particularmente útil para aplicaciones como controlar un ventilador o recopilar datos de sensores y enviarlos a la nube.[15].

1.2.4. Sensores y Actuadores

En un sistema de IoT, los sensores recopilan datos del entorno, como la temperatura, la humedad o la calidad del aire, mientras que los actuadores realizan acciones físicas, como encender un ventilador o ajustar su velocidad. El ESP32 puede interactuar con una amplia variedad de sensores y actuadores, lo que le permite monitorear y controlar varios dispositivos de manera eficiente[16].

[13] II.
[14] X.
[15] IV.
[16] VIII.

1.2.5. Protocolo MQTT

El protocolo Message Queuing Telemetry Transport (MQTT) es un protocolo de mensajería liviano diseñado para dispositivos con recursos limitados, como el ESP32, en redes IoT. Sigue un modelo de publicación/suscripción, donde los dispositivos (editores) envían mensajes a un servidor (broker) y otros dispositivos (suscriptores) los reciben. Este protocolo se utiliza ampliamente en IoT debido a su eficiencia en la gestión de dispositivos intermitentes de bajo consumo. Por ejemplo, se puede utilizar para enviar comandos a un ventilador para encender/apagar o ajustar la velocidad.[17].

1.2.6. Pines GPIO e interfaces periféricas

Los pines de entrada/salida de uso general (GPIO) del ESP32 se utilizan para interactuar con componentes externos como sensores, LED o motores. Permiten que el microcontrolador lea entradas o controle actuadores. El ESP32 también admite interfaces como UART, I2C y SPI, que son esenciales para conectar periféricos adicionales en un sistema IoT.[18].

1.2.7. Consumo de energía y modos de suspensión

Una de las ventajas importantes de ESP32 es su bajo consumo de energía, que es crucial para los dispositivos IoT que a menudo funcionan con baterías. El modo de suspensión profunda, que reduce drásticamente el consumo de energía cuando el dispositivo está inactivo, permite utilizar ESP32 en proyectos a largo plazo donde la eficiencia energética es fundamental. Esta característica es ideal para aplicaciones como monitorear un ventilador, donde el ESP32 puede "dormirçuando el ventilador está apagado y reactivarse cuando el sistema necesita ajustar la configuración.[19].

1.2.8. Variantes de ESP32

Existen varias versiones del microcontrolador ESP32, cada una con características específicas que pueden ser más adecuadas para diferentes aplicaciones de IoT. Por ejemplo, ESP32-WROOM y ESP32-WROVER ofrecen diferentes configuraciones de memoria, y el ESP32-C3 proporciona características adicionales de bajo consumo de energía y factores de forma más pequeños. La elección de la variante depende de la complejidad y los requisitos del proyecto.[20].

1.2.9. RTOS y ESP-IDF

Los sistemas operativos en tiempo real (RTOS) están diseñados para aplicaciones que requieren procesamiento en tiempo real. El ESP32 puede ejecutar FreeRTOS, lo que permite una programación de tareas y una gestión de recursos más eficientes, lo que lo hace ideal para aplicaciones de IoT urgentes, como la automatización industrial. Además, ESP-IDF (Espressif IoT Development Framework) es un marco oficial para desarrollar software para ESP32, que ofrece acceso de bajo nivel a las capacidades del microcontrolador.

[17] VI.
[18] V.
[19] IX.
[20] VII.

1.2.10. MicroPython

El ESP32 fue programado utilizando MicroPython, un lenguaje de alto nivel que facilita el desarrollo en sistemas embebidos. MicroPython ofrece:

- Control directo de hardware mediante librerías específicas como machine y umqtt.simple.
- Soporte para estructuras ligeras de datos y comunicación.

1.2.11. MQTT y HTTP

Se utilizaron protocolos estándar para la comunicación:

- **MQTT**: Protocolo ligero de mensajería que garantiza la comunicación eficiente entre el ESP32 y el servidor.
- **HTTP**: Utilizado para registrar dispositivos y enviar datos al servidor.

1.2.12. Docker

El servidor fue configurado en contenedores Docker para garantizar modularidad y escalabilidad. Los servicios principales se dividieron en:

- **API Gateway**: Gestiona las solicitudes HTTP y los registros de dispositivos.
- **Event Hub**: Coordina eventos y mensajes entre dispositivos.
- **Dashboard**: Permite visualizar y controlar dispositivos a través de una interfaz gráfica.

1.2.13. Librerías y Herramientas Complementarias

- **Librería dht**: Para medir temperatura y humedad.
- **Librería urequests**: Para solicitudes HTTP.
- **Wokwi**: Simulador utilizado para pruebas iniciales del código en un entorno seguro.

1.2.14. Microcontrolador ESP32

- Características principales:
  - Procesador dual-core con conectividad WiFi y Bluetooth integrada.
  - Soporte para protocolos IoT como MQTT y HTTP. Pines GPIO configurables para conectar sensores y actuadores.
- Ventajas:
  - Alta eficiencia energética.
  - Amplia comunidad de desarrolladores y documentación.

1.2.15. Sensor DHT22

- Función: Medición de temperatura y humedad.
- Especificaciones:
- Rango de temperatura: -40 a 80 °C.
- Rango de humedad: 0 a 100

Uso en el proyecto:
Captura de datos ambientales para su visualización en el dashboard.

1.2.16. Actuadores

- **Ventiladores**: Controlados mediante pines PWM para regular su velocidad.
- **LEDs**: Utilizados para indicar estados operativos y responder a comandos del usuario.

1.2.17. Infraestructura del Servidor

DigitalOcean:

- Droplet configurado para alojar los contenedores Docker.
- Capacidad suficiente para manejar las solicitudes de múltiples dispositivos.

1.2.18. Otros Componentes

- **Resistencias**: Utilizadas para limitar la corriente en los LEDs.
- **Botones Pulsadores**: Configurados para permitir control manual de los actuadores.

1.3. Conclusiones del estado del arte

El estado del arte muestra que el ESP32 es una plataforma versátil y eficiente para aplicaciones IoT, con una comunidad activa y una amplia gama de recursos disponibles. Sin embargo, la integración de MicroPython sobre FreeRTOS presenta desafíos en términos de rendimiento y recursos, lo que sugiere la necesidad de optimizaciones adicionales para aplicaciones de tiempo real.

2. Desarrollo

2.1. Descripción del caso de estudio

2.1.1. Problemática

Los sistemas de monitoreo y control ambiental requieren respuesta en tiempo real a cambios en las condiciones ambientales, mientras mantienen comunicación constante con servicios en la nube. Esto plantea desafíos de concurrencia, gestión de recursos y sincronización.

2.1.2. Objetivo del caso de estudio

Implementar un sistema IoT que demuestre la integración efectiva entre MicroPython y FreeRTOS, permitiendo el control de actuadores, lectura de sensores y comunicación en red de manera concurrente y eficiente.

2.1.3. Evaluación de solución del caso de estudio

La solución se evalúa en términos de respuesta a eventos, eficiencia en el uso de recursos, robustez ante fallos y facilidad de mantenimiento y extensión.

2.2. Metodología de desarrollo de la solución

2.2.1. Especificación de requerimientos

Requerimientos Funcionales:

- Lectura periódica de temperatura y humedad mediante sensor DHT22
- Control de LEDs y ventiladores mediante PWM
- Conexión a red WiFi y broker MQTT
- Envío periódico de datos de telemetría a la nube
- Recepción y procesamiento de comandos remotos
- Control local mediante botones físicos

Requerimientos No Funcionales:

- Respuesta en tiempo real a cambios en condiciones ambientales
- Operación continua sin interrupciones
- Bajo consumo de recursos (memoria, CPU)
- Sincronización temporal precisa
- Recuperación ante fallos de red

2.2.2. Herramientas

MicroPython Entorno de ejecución Python para ESP32, proporcionando acceso a hardware y abstracciones de alto nivel.

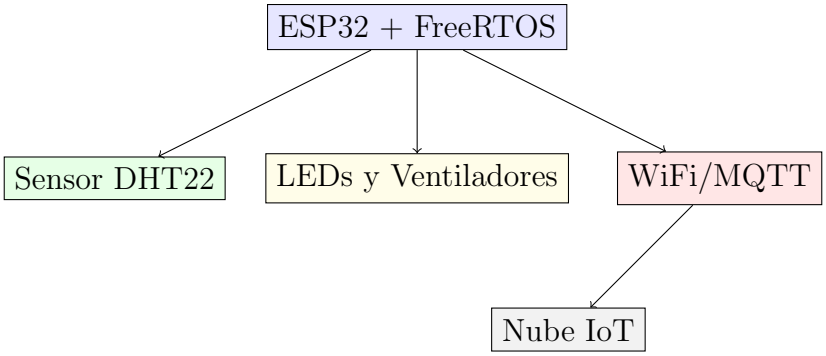
FreeRTOS Sistema operativo subyacente que gestiona la multitarea y recursos del sistema.

ESP32 Plataforma hardware con capacidades WiFi y procesamiento dual-core.

MQTT Protocolo de comunicación para la interacción con servicios en la nube.

2.3. Diseño de arquitectura de desarrollo

El sistema implementa una arquitectura en capas:



2.3.1. Integración e interfaces

La integración entre componentes se realiza mediante:

- APIs de MicroPython para acceso a hardware
- Callbacks para manejo de eventos asíncronos
- Clases de abstracción para dispositivos y comunicación
- Temporizadores para tareas periódicas

2.3.2. Análisis de datos

Los datos de sensores se procesan localmente para control de actuadores y se envían a la nube para análisis histórico y visualización.

2.4. Módulos

2.4.1. Diseño de módulos

Módulo de Control de Dispositivos: Gestiona la interacción con LEDs, ventiladores y otros actuadores mediante PWM.

```
    class LED:
        def __init__(self, pin_num, freq=1500):
            self.pin_num = pin_num
            self.freq = freq
            self.pwm = PWM(Pin(pin_num), freq=freq)
            self.is_on = False
            self.percent = 0

        def on(self, percentage = 0):
            if percentage > 0:
                self.percent = percentage
                self.duty = int(percentage / 100 * 1023)
            self.pwm.duty(self.duty)
            self.is_on = True

```

Listing 1: Clase LED para control PWM

Módulo de Comunicación: Gestiona la conexión WiFi y MQTT para interacción con servicios en la nube.

```
    class MqttConnector:
        def __init__(self, client_id, broker, user, password):
            self.client_id = client_id
            self.broker = broker
            self.user = user
            self.password = password

        def connect(self, callback):
            self.client = MQTTClient(self.client_id, self.broker)
            self.client.set_callback(callback)
            self.client.connect()

        def publish(self, topic, data):
            self.client.publish(topic, data)

```

Listing 2: Conector MQTT

Módulo de Sensores: Gestiona la lectura de sensores de temperatura y humedad.

Módulo de Gestión de Tareas: Implementa temporizadores y callbacks para simular multitarea en MicroPython.

```
    from helpers import DelayedMethod

    def push_data():
        # Envia datos de sensores por MQTT
        ...
```

```
6
7       push_device_data_delay = DelayedMethod(push_data, 10)
8
9       while True:
10          push_device_data_delay.run()
11          mqtt_connector.check_incoming_msg()
12          time.sleep(0.1)
13
```

Listing 3: Ejemplo de tarea concurrente

2.5. Estructura General del Proyecto

El sistema está compuesto por tres componentes principales:

1. **ESP32 (Microcontrolador)**:
   Ejecuta el código principal para gestionar sensores y actuadores, conectarse a WiFi y comunicarse con el servidor Docker mediante MQTT y HTTP.

2. **Servidor Docker:**
   Contiene tres repositorios:

   - docker-iot-connector-api: Proporciona una API para procesar datos del ESP32.
   - docker-iot-connector-event-hub: Gestiona eventos y datos entre el ESP32 y las aplicaciones cliente.
   - docker-iot-connector-dashboard: Permite visualizar datos y controlar dispositivos a través de un dashboard.

3. **Hardware adicional**: Sensores DHT22, LEDs y un ventilador, todos controlados por el ESP32.

2.6. Código Principal del ESP32

El código se divide en varios bloques funcionales que interactúan con los periféricos y el servidor. A continuación, se detallan los más importantes:

2.6.1. Configuración inicial

Define parámetros como el WiFi, el MQTT y los dispositivos conectados:

```
1       # Configuracion de WiFi
2       WIFI_SSID = "Wokwi-GUEST"
3       WIFI_PASSWORD = ""
4
5       # Configuracion de MQTT
6       MQTT_BROKER = "143.198.5.161"
7       MQTT_USER_NAME = ESP_32_GATEWAY_ID
8       MQTT_PASSWORD = DEVICE_SECRETE
9
10      # Definicion de dispositivos
```

```
11     RED_LED_DEVICE = {"name": "RED_LED", "type": DeviceType.LED,
   "pin": 12, "id": RED_LED_DEVICE_ID}
12     BLUE_LED_DEVICE = {"name": "BLUE_LED", "type": DeviceType.LED
   , "pin": 13, "id": BLUE_LED_DEVICE_ID}
13
```

Listing 4: Implementación en Python

2.6.2. Gestión de dispositivos

Usa clases como DevicesManager para configurar y gestionar dispositivos conectados al ESP32:

```
1     # Creacion de dispositivos
2     devices_manager = DevicesManager(gateway_name=DEVICE_NAME,
   gateway_id=DEVICE_ID)
3     devices_manager.create_device(RED_LED_DEVICE)
4     devices_manager.create_device(BLUE_LED_DEVICE)
5
```

Listing 5: Implementación en Python

2.6.3. Enlace con el servidor

Se conecta a WiFi, registra dispositivos en el servidor y suscribe tópicos MQTT:

```
1     # Conexion a WiFi
2     wifi_connector = WiFiConnector(WIFI_SSID, WIFI_PASSWORD)
3     wifi_connector.connect()
4
5     # Registro en el servidor
6     api_client = APIClient("http://{0}:{1}".format(URL_HOST,
   URL_PORT))
7     provisioning_response = api_client.post("/api/v1/devices/
   provision/group/{0}/{1}".format(GROUP_ID, API_KEY), data=
   devices_manager.get_provisioning_device_data(DEVICE_SECRETE))
8
9     # Conexion a MQTT
10    mqtt_connector = MqttConnector(MQTT_CLIENT_ID, MQTT_BROKER,
   MQTT_USER_NAME, MQTT_PASSWORD)
11    mqtt_connector.connect(did_recieve_subscription_message)
12    mqtt_connector.subscribe(MQTT_CONTROL_TOPIC)
13
```

Listing 6: Implementación en Python

2.6.4. Lógica principal

Maneja el envío de datos a la nube y la respuesta a comandos MQTT:

```
1     # Enviar datos a la nube
```

```
    def push_data():
        telemetry_data = {
            "type": ChannelTypes.TELEMETRY,
            "data": devices_manager.get_data()
        }
        mqtt_connector.publish(MQTT_TELEMETRY_TOPIC, ujson.dumps(
telemetry_data))

```

Listing 7: Implementación en Python

2.6.5. Gestión de LEDs y ventiladores

Controla el encendido, apagado y ajuste de brillo o velocidad:

```
    RED_LED.on(100)  # Encender LED rojo al 100% de brillo
    BLUE_FAN.set_brightness(50)  # Ajustar ventilador azul al 50%
    de velocidad

```

Listing 8: Implementación en Python

2.7. Instrucciones para Ejecutar el Código

- Requisitos previos:
  - ESP32 DevKit v1 con MicroPython instalado.
  - Conexión WiFi con SSID "Wokwi-GUEST".
  - Un servidor Docker configurado con los tres repositorios mencionados.
- Pasos para cargar el código:
  - Flashear el ESP32 con MicroPython usando herramientas como esptool.
  - Subir los archivos necesarios ($main.py, helpers.py, led_pwm.py$) al ESP32 usando Thonny o similares.
  - Configurar el servidor Docker usando las URLs de los repositorios.
- Librerías necesarias:
  - umqtt.simple para comunicación MQTT.
  - urequests para solicitudes HTTP.
  - dht para el sensor de temperatura y humedad.

3. Resultados

En esta sección se presentan las evidencias que demuestran el correcto funcionamiento del sistema implementado. Se incluyen capturas de pantalla y descripciones detalladas de cada etapa del proceso, desde la creación del entorno Docker hasta la interacción entre el ESP32 y el dashboard.

3.1. Creación y Configuración del Servidor Docker en DigitalOcean

Se desplegó un servidor en DigitalOcean para alojar los contenedores Docker necesarios para el funcionamiento del sistema. A continuación, se describen los pasos realizados:

- Creación del Droplet:

  Se seleccionó una imagen base de Ubuntu 20.04. Se configuraron los recursos del servidor según las necesidades del proyecto.

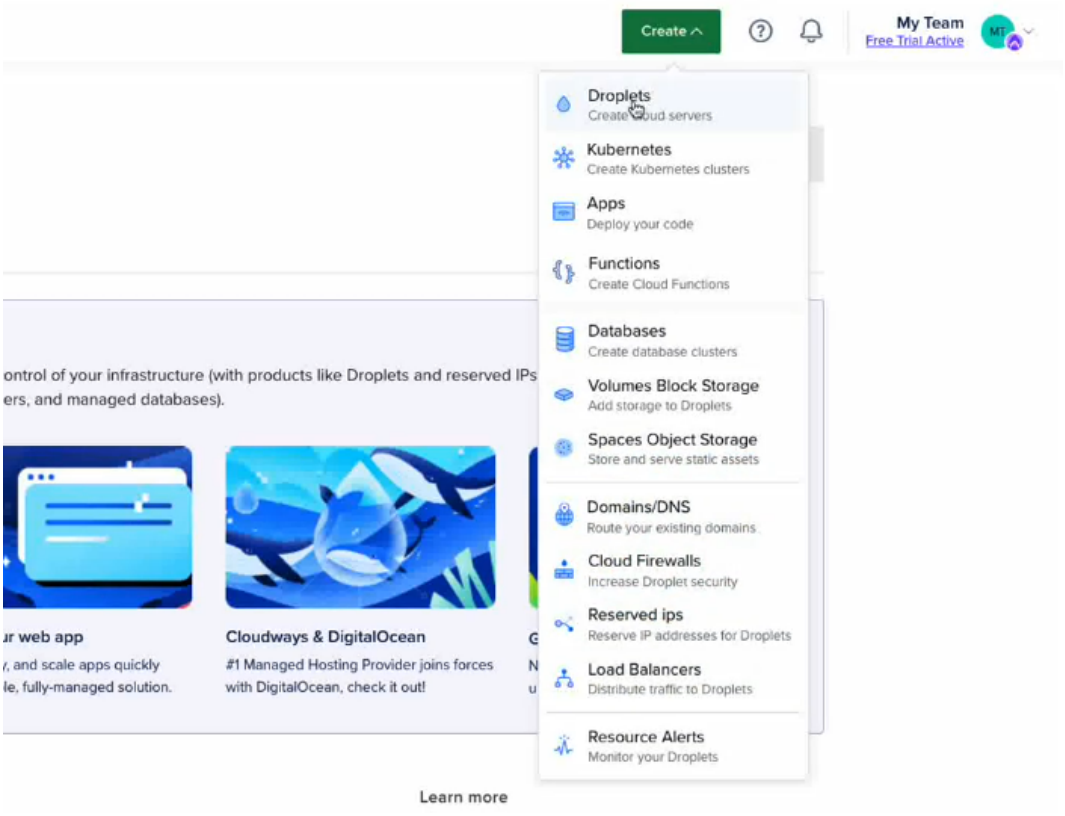


Figura 5: Creación del Droplet

- Instalación de Docker y Docker Compose:

  Se seleccionó la opción de instalación del Docker y Docker Compose en el servidor.

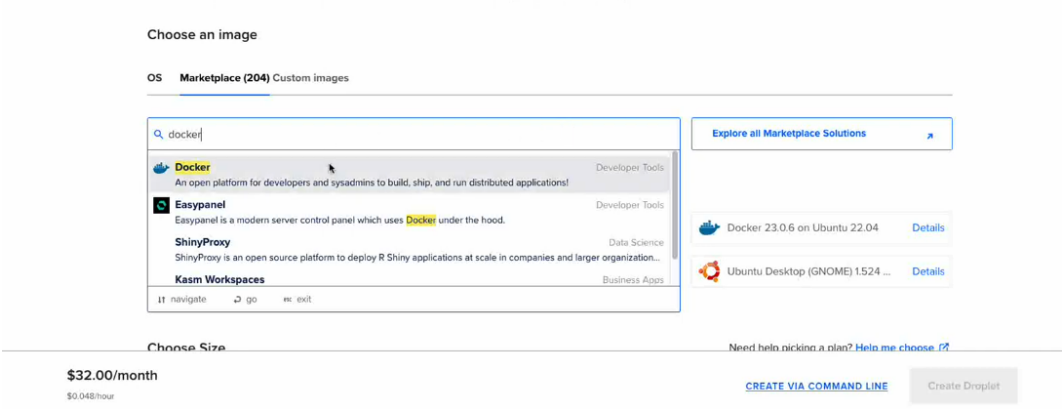


Figura 6: Utilización de Docker

- Clonación de Repositorios:

  Se clonaron los tres repositorios necesarios:

  - docker-iot-connector-api
  - docker-iot-connector-event-hub
  - docker-iot-connector-dashboard

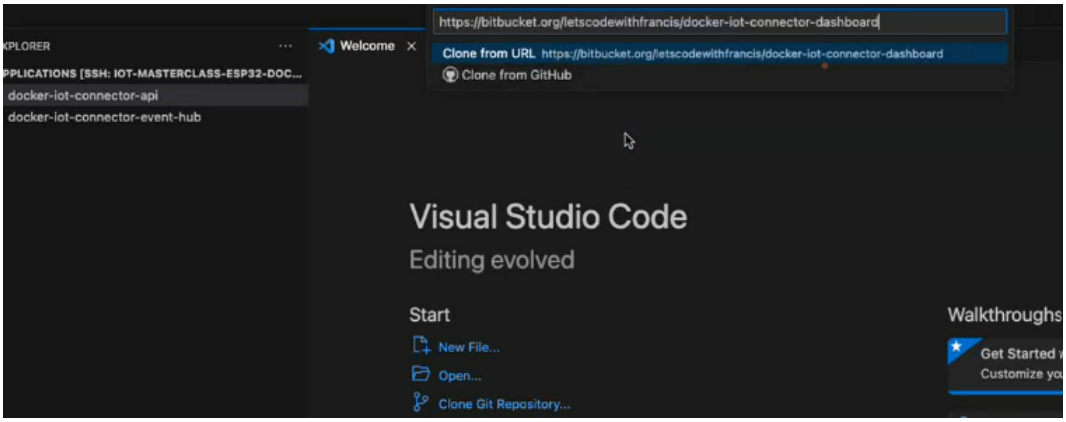


Figura 7: Clonación de repositorios

- Configuración de Contenedores:

  Se configuraron los archivos docker-compose.yml para cada repositorio. Se establecieron las variables de entorno necesarias para la comunicación entre los servicios. Despliegue de Contenedores:

  Se utilizaron los comandos docker-compose up -d para levantar los servicios en segundo plano. Se verificó el estado de los contenedores con *docker ps*.

3.2. Despliegue del Dashboard

El dashboard es la interfaz gráfica que permite visualizar y controlar los dispositivos conectados al sistema.

- Acceso al Dashboard:

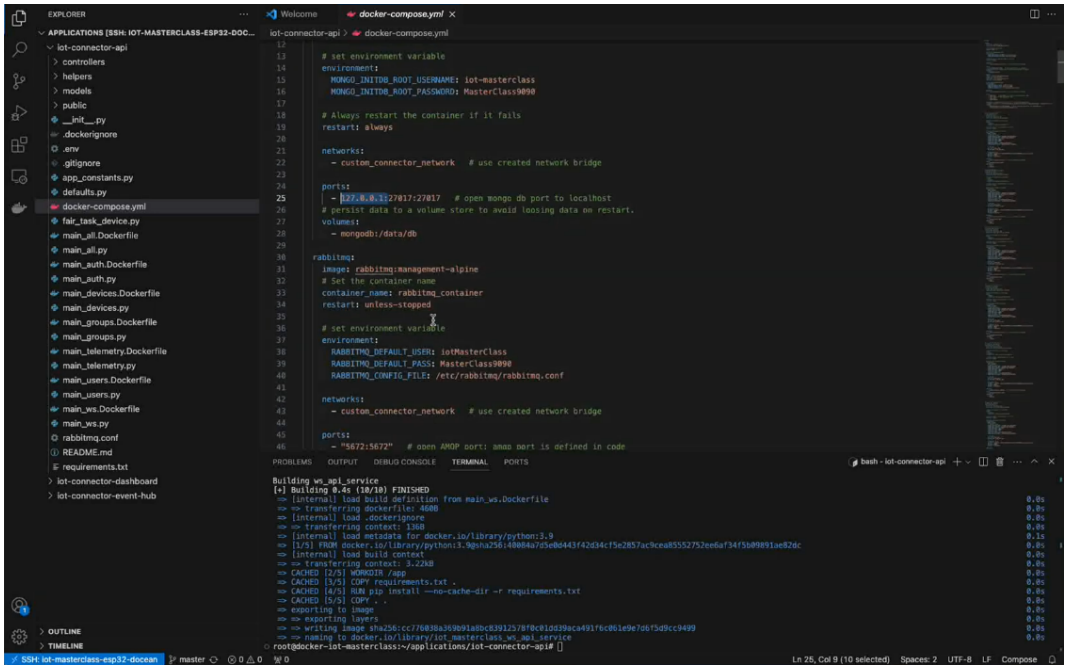


Figura 8: Configuración de contenedores

- Se accedió al dashboard mediante la dirección IP del servidor y el puerto configurado.
- Se mostró la pantalla de inicio de sesión, donde se ingresaron las credenciales establecidas.

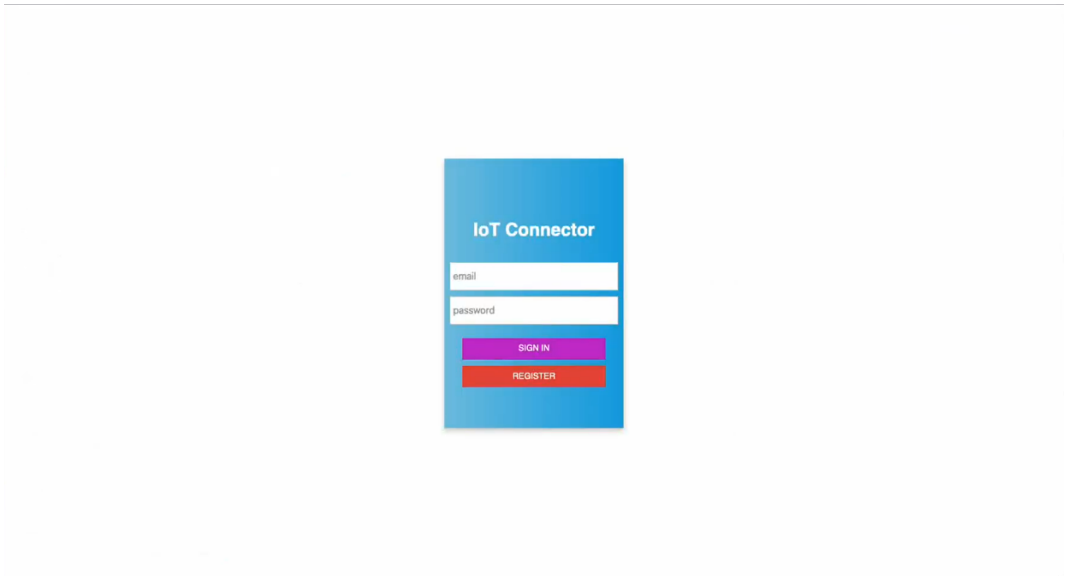


Figura 9: Acceso al dashboard

- Configuración Inicial:
  - Se agregaron los dispositivos al dashboard utilizando los identificadores únicos proporcionados.
  - Se establecieron las relaciones entre dispositivos y grupos.

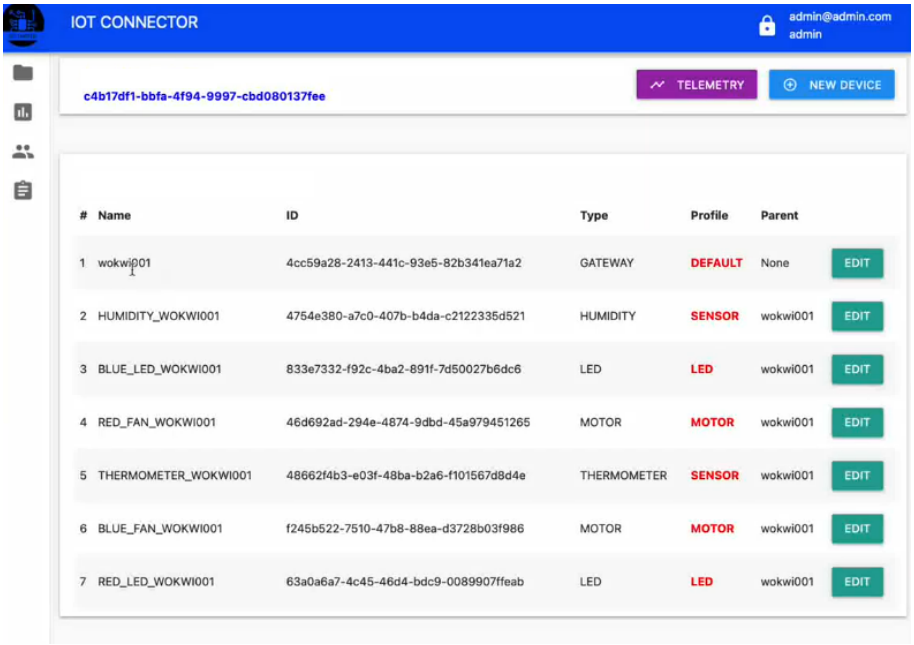


Figura 10: Configuración Inicial

- Visualización de Datos:

  El dashboard mostró en tiempo real los datos enviados por el ESP32, como temperatura, humedad y estados de los LEDs y ventiladores.

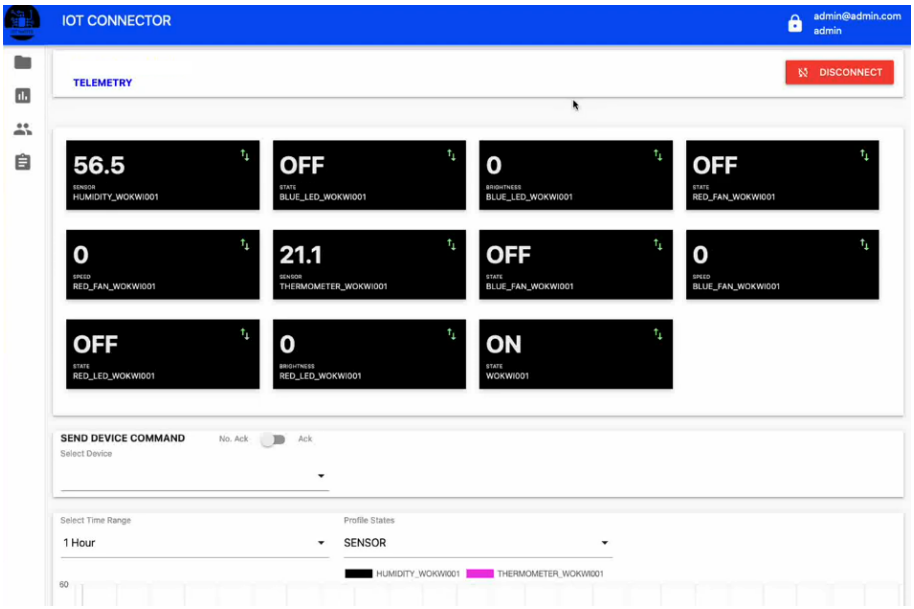


Figura 11: Visualización de datos

3.3. Configuración y Conexión con RabbitMQ

RabbitMQ se utilizó como broker MQTT para gestionar la comunicación entre el ESP32 y el servidor.

- Instalación de RabbitMQ:

  Se desplegó RabbitMQ en un contenedor Docker mediante el repositorio docker-iot-connector-event-hub. Se configuraron los puertos y las credenciales de acceso.

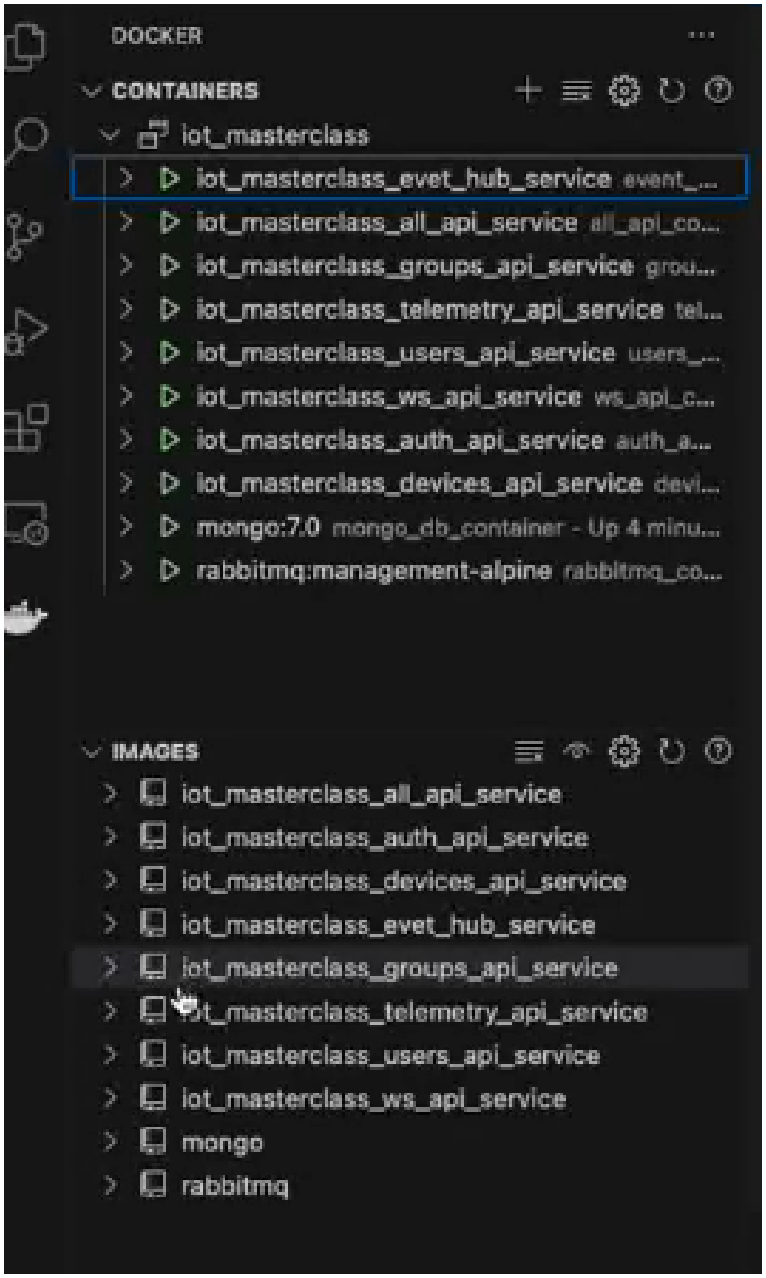


Figura 12: Instalación de RabbitMQ en docker

- Verificación de RabbitMQ:

  Se accedió a la interfaz de administración de RabbitMQ para verificar que el servicio estuviera operativo. Se observaron las colas y los exchanges configurados para el sistema.

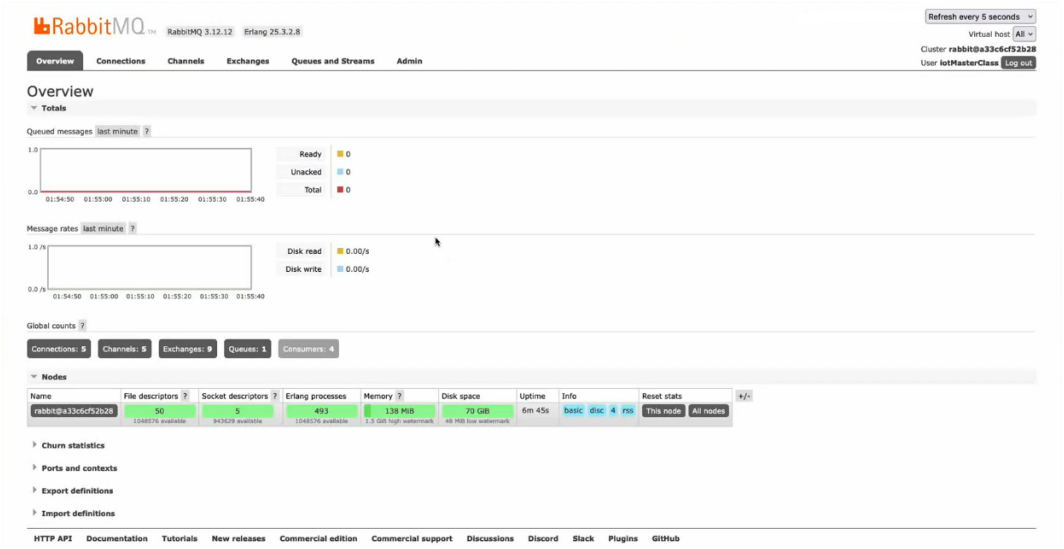


Figura 13: Acceso a RabbitMQ

3.4. Conexión del Microcontrolador ESP32 con el Dashboard

El ESP32 fue programado para conectarse al servidor y enviar datos de los sensores, así como recibir comandos para controlar los actuadores.

- Conexión a la Red WiFi:

  El ESP32 se conectó exitosamente a la red WiFi especificada. Se verificó la dirección IP asignada al dispositivo.

- Registro y Autenticación:

  El ESP32 se registró en el servidor utilizando el API Client incluido en el código. Se autenticó correctamente con el broker MQTT utilizando las credenciales configuradas.

- Intercambio de Mensajes MQTT:

  El ESP32 publicó datos de telemetría en el tópico correspondiente. Se suscribió a los tópicos de control para recibir comandos desde el dashboard.

- Control de Dispositivos:

  Desde el dashboard, se enviaron comandos para encender y apagar los LEDs y ajustar la velocidad de los ventiladores. El ESP32 recibió estos comandos y actuó en consecuencia, lo cual se reflejó físicamente en los dispositivos conectados.

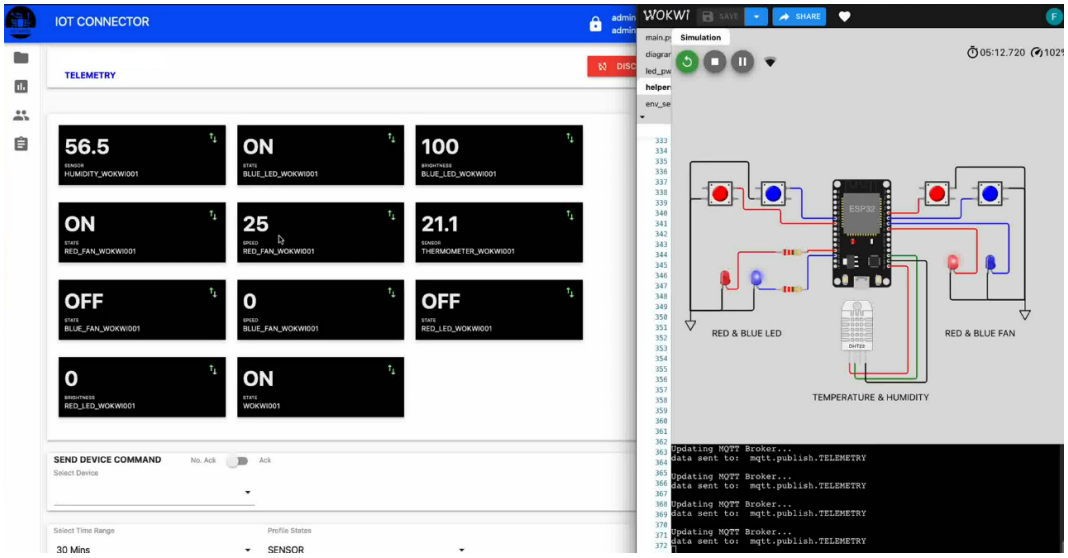


Figura 14: Conexión a red WiFi

3.5. Pruebas y Validación del Sistema

Se realizaron diversas pruebas para asegurar el correcto funcionamiento y robustez del sistema.

- Prueba de Sensores:

  Se verificó que el sensor DHT22 proporcionara lecturas precisas de temperatura y humedad. Se simularon cambios en el ambiente para observar las actualizaciones en tiempo real en el dashboard.

- Respuesta a Eventos:

  Se utilizaron los botones conectados al ESP32 para cambiar el estado de los dispositivos. El sistema respondió adecuadamente, actualizando el estado en el dashboard y actuando sobre los actuadores.

- Gestión de Errores y Reconexión:

  Se interrumpió la conexión WiFi para probar la capacidad del ESP32 de reconectarse automáticamente. El dispositivo logró restablecer la conexión y continuar enviando datos sin pérdida significativa.

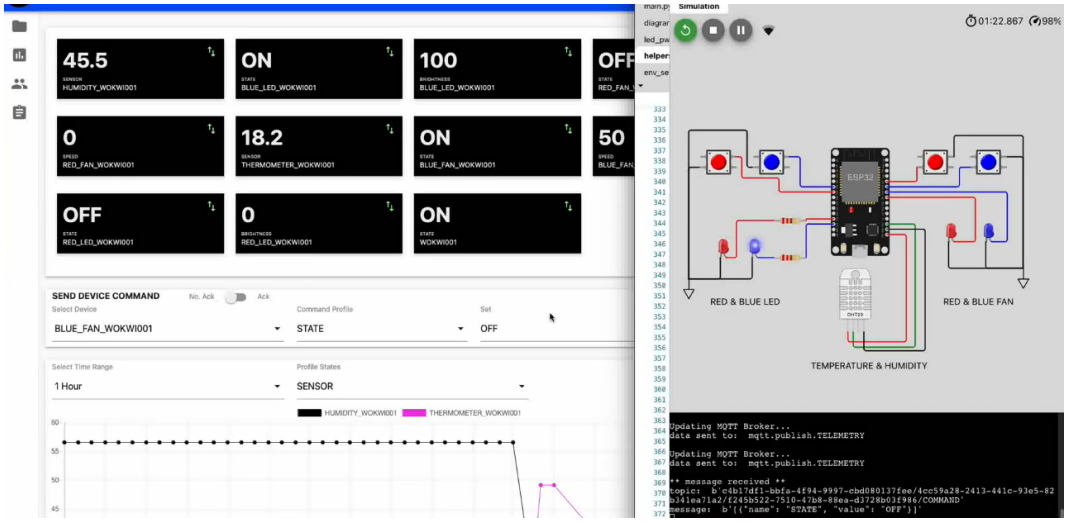


Figura 15: Pruebas diversas

3.6. Integración Completa y Funcionamiento en Tiempo Real

Finalmente, se comprobó el funcionamiento integrado del sistema en un entorno simulado.

- Operación Continua:

  El sistema se mantuvo operando durante un período prolongado, demostrando estabilidad y confiabilidad. Los datos históricos se almacenaron y visualizaron en el dashboard.

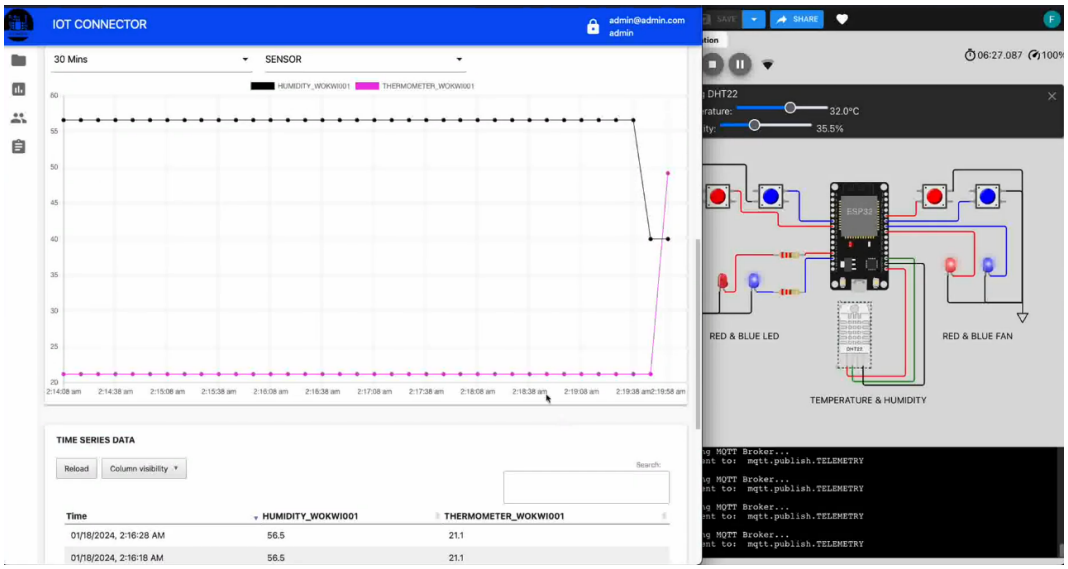


Figura 16: Funcionamiento continuo

4. Conclusiones y Trabajos Futuros

4.1. Conclusiones

- **Implementación Exitosa del Sistema IoT**
  El proyecto demostró la viabilidad de utilizar el microcontrolador ESP32 como núcleo para implementar un sistema IoT escalable. La comunicación eficiente mediante MQTT y HTTP permitió integrar sensores y actuadores con un servidor Docker en DigitalOcean, logrando un control remoto confiable y en tiempo real.

- **Versatilidad del ESP32**
  El ESP32 destacó por su capacidad de manejar múltiples dispositivos (sensores y actuadores) simultáneamente, su compatibilidad con diversas librerías y su facilidad de programación con MicroPython. Estas características lo posicionan como una solución robusta para proyectos similares.

- **Uso Eficiente de Contenedores Docker**
  La separación de funcionalidades en contenedores Docker (API, event hub y dashboard) permitió una arquitectura modular y escalable. Esto simplificó la implementación, el mantenimiento y la posibilidad de extender el sistema en el futuro.

- **Lecciones Aprendidas** Durante el desarrollo, se identificó la importancia de:

  - Diseñar flujos de trabajo claros para la integración de hardware y software.
  - Implementar mecanismos de reconexión y manejo de errores para sistemas IoT.
  - Utilizar entornos de simulación, como Wokwi, para pruebas previas al despliegue físico.

4.2. Trabajos Futuros

- **Expansión del Sistema**
  Escalar el sistema para incluir múltiples dispositivos en diferentes ubicaciones, optimizando la asignación de recursos y mejorando la comunicación entre ellos.

- **Optimización del Control de Ventiladores**
  Incorporar algoritmos de control más avanzados, como control PID, para ajustar la velocidad del ventilador de manera más precisa en función de las condiciones ambientales.

- **Análisis de Datos en Tiempo Real**
  Desarrollar una capa de análisis en el dashboard para identificar patrones de uso y generar recomendaciones automáticas basadas en datos históricos.

- **Seguridad IoT**
  Mejorar la seguridad del sistema implementando cifrado SSL/TLS para la comunicación MQTT y métodos de autenticación más robustos.

- **Automatización de Reglas**
  Configurar un sistema de automatización más avanzado para permitir que el sistema

tome decisiones de manera autónoma. Por ejemplo, encender el ventilador automáticamente cuando la temperatura supere un umbral específico o cuando se detecte alta humedad.

- **Compatibilidad con Otras Plataformas**
  Extender la compatibilidad del sistema para trabajar con otras plataformas IoT, como AWS IoT Core o Google IoT Cloud, para ampliar las opciones de integración.

Referencias Bibliográficas

[I] Anwer Sabah Ahmed, Heyam A. Marzog y Laith Ali Abdul-Rahaim. «Design and implement of robotic arm and control of moving via IoT with Arduino ESP32». En: *International Journal of Electrical and Computer Engineering (IJECE)* 11.5 (2021), págs. 3924-3933. ISSN: 2722-2578. DOI: 10.11591/ijece.v11i5.pp3924-3933. URL: https://ijece.iaescore.com/index.php/IJECE/article/view/24625.

[II] Marek Babiuch y Jiri Postulka. «Smart Home Monitoring System Using ESP32 Microcontrollers». En: *Internet of Things*. Ed. por Fausto Pedro García Márquez. Rijeka: IntechOpen, 2020. Cap. 6. DOI: 10.5772/intechopen.94589. URL: https://doi.org/10.5772/intechopen.94589.

[III] M. J. Espinosa-Gavira et al. «Characterization and Performance Evaluation of ESP32 for Real-time Synchronized Sensor Networks». En: *Procedia Computer Science* 237 (2024), págs. 261-268. DOI: 10.1016/j.procs.2024.05.104.

[IV] Omar Otoniel Flores-Cortez, Ronny Adalberto Cortez y Veronica Rosa. *Implementacion de un sistema IoT de bajo costo para el monitoreo de la calidad del aire en El Salvador*. 2022. DOI: 10.48550/arXiv.2207.09975. arXiv: 2207.09975 [eess.SY]. URL: https://arxiv.org/abs/2207.09975.

[V] Darko Hercog et al. «Design and Implementation of ESP32-Based IoT Devices». En: *Sensors* 23.15 (2023). ISSN: 1424-8220. DOI: 10.3390/s23156739. URL: https://www.mdpi.com/1424-8220/23/15/6739.

[VI] M. Tariq Iqbal Lawrence O. Aghenta. «Design and Implementation of a Low-cost, Open-source IoT-based SCADA system using ESP32 with OLED, ThingsBoard and MQTT protocol». En: *AIMS Electronics and Electrical Engineering* 4 (2020), págs. 57-86. DOI: 10.3934/ElectrEng.2020.1.57.

[VII] Ciro Edgardo Romero y Alejandro Elustondo. «Análisis de la capacidad de la placa ESP32 para integrar sistemas IoT descentralizados». En: *Revista elektron* 6.1 (2022), págs. 41-45. DOI: 10.37537/rev.elektron.6.1.142.2022.

[VIII] Subbarayudu, Yerragudipadu et al. «An Efficient IoT-Based Novel Approach for Fire Detection Through Esp 32 Microcontroller in Forest Areas». En: *MATEC Web Conf.* 392 (2024), pág. 01109. DOI: 10.1051/matecconf/202439201109. URL: https://doi.org/10.1051/matecconf/202439201109.

[IX] Espressif Systems. *ESP32-C3 Wireless Adventure: A Comprehensive Guide to IoT*. Espressif Systems, 2023.

[X] Asim Zulfiqar. *Hands-on ESP32 with Arduino IDE: Unleash the power of IoT with ESP32 and build exciting projects with this practical guide*. Packt Publishing, 2024. ISBN: 978-1-83763-803-1.

Anexos

Anexo 1:Repositorios utilizados para el docker

Los repositorios utilizados fueron:

Para el API:
https://bitbucket.org/letscodewithfrancis/docker-iot-connector-api
Para el event-hub:
https://bitbucket.org/letscodewithfrancis/docker-iot-connector-event-hub
Para el dashboard:
https://bitbucket.org/letscodewithfrancis/docker-iot-connector-dashboard

Anexo 2:Configuraciones en el Esp32

Diagrama en formato json

```
{
  "version": 1,
  "author": "Gustavo Delgado",
  "editor": "wokwi",
  "parts": [
    {
      "type": "wokwi-esp32-devkit-v1",
      "id": "esp",
      "top": 30.01,
      "left": 55.4,
      "attrs": { "env": "micropython-20220618-v1.19.1" }
    },
    {
      "type": "wokwi-led",
      "id": "led1",
      "top": 188.4,
      "left": -159.4,
      "attrs": { "color": "red" }
    },
    {
      "type": "wokwi-resistor",
      "id": "r1",
      "top": 157.55,
      "left": -48,
      "attrs": { "value": "220" }
    },
    {
      "type": "wokwi-led",
      "id": "led2",
      "top": 188.4,
      "left": -101.8,
      "attrs": { "color": "blue" }
    },
    {
      "type": "wokwi-resistor",
      "id": "r2",
      "top": 224.75,
      "left": -48,
      "attrs": { "value": "220" }
    },
    {
      "type": "wokwi-dht22",
      "id": "dht1",
      "top": 249.9,
      "left": 71.4,
      "attrs": { "humidity": "71.5", "temperature": "21.1" }
    },
    {
      "type": "wokwi-pushbutton",
      "id": "btn1",
```

```
      "top": 35,
      "left": -182.4,
      "attrs": { "color": "red" }
    },
    { "type": "wokwi-gnd", "id": "gnd1", "top": 268.8, "left": -211.8, "attrs": {} },
    {
      "type": "wokwi-pushbutton",
      "id": "btn2",
      "top": 35,
      "left": -86.4,
      "attrs": { "color": "blue" }
    },
    {
      "type": "wokwi-pushbutton",
      "id": "btn3",
      "top": 35,
      "left": 211.2,
      "attrs": { "color": "red" }
    },
    {
      "type": "wokwi-pushbutton",
      "id": "btn4",
      "top": 35,
      "left": 307.2,
      "attrs": { "color": "blue" }
    },
    {
      "type": "wokwi-led",
      "id": "led3",
      "top": 159.6,
      "left": 253.4,
      "attrs": { "color": "red" }
    },
    {
      "type": "wokwi-led",
      "id": "led4",
      "top": 159.6,
      "left": 320.6,
      "attrs": { "color": "blue" }
    },
    { "type": "wokwi-gnd", "id": "gnd2", "top": 249.6, "left": 393, "attrs": {} },
    {
      "type": "wokwi-text",
      "id": "led-text",
      "top": 297.6,
      "left": 259.2,
      "attrs": { "text": "RED & BLUE FAN" }
    },
    {
```

```
      "type": "wokwi-text",
      "id": "fan-text",
      "top": 297.6,
      "left": -163.2,
      "attrs": { "text": "RED & BLUE LED" }
    },
    {
      "type": "wokwi-text",
      "id": "temp-humidity-text",
      "top": 412.8,
      "left": 19.2,
      "attrs": { "text": "TEMPERATURE & HUMIDITY" }
    }
  ],
  "connections": [
    [ "esp:TX0", "$serialMonitor:RX", "", [] ],
    [ "esp:RX0", "$serialMonitor:TX", "", [] ],
    [ "esp:D12", "r1:2", "red", [ "h-31.6", "v-0.1", "h-3.59" ]
   ],
    [ "esp:D13", "r2:2", "blue", [ "h0" ] ],
    [ "r1:1", "led1:A", "red", [ "h-67.2", "v67.2" ] ],
    [ "led2:A", "r2:1", "blue", [ "h4.16", "v-117.2" ] ],
    [ "dht1:VCC", "esp:3V3", "red", [ "v17.86", "h105.49", "v
   -192.13" ] ],
    [ "dht1:SDA", "esp:D15", "green", [ "v27.33", "h112.09", "v
   -220.6" ] ],
    [ "dht1:GND", "esp:GND.1", "black", [ "v33.96", "h110.78", "v
   -217.73" ] ],
    [ "esp:D33", "btn1:1.r", "red", [ "h-98.8", "v-25.31", "h
   -67.2", "v115.2" ] ],
    [ "btn1:2.l", "gnd1:GND", "black", [ "h-19.2", "v201.8" ] ],
    [ "esp:D32", "btn2:1.r", "blue", [ "h-60.4", "v-54.21" ] ],
    [ "led1:C", "gnd1:GND", "black", [ "v0" ] ],
    [ "led2:C", "gnd1:GND", "black", [ "v0" ] ],
    [ "esp:D21", "btn3:2.l", "red", [ "h0" ] ],
    [ "esp:D19", "btn4:2.l", "blue", [ "h131.3", "v-35.01" ] ],
    [ "esp:D18", "led4:A", "blue", [ "h217.7", "v3.49" ] ],
    [ "led3:A", "esp:D5", "red", [ "v0", "h38.4", "v-76.8" ] ],
    [ "led3:C", "gnd2:GND", "black", [ "v0" ] ],
    [ "led4:C", "gnd2:GND", "black", [ "v0" ] ],
    [ "btn4:1.r", "gnd2:GND", "black", [ "v0", "h38.6" ] ],
    [ "btn3:1.r", "gnd2:GND", "black", [ "v-38.4", "h125" ] ],
    [ "btn2:2.l", "gnd1:GND", "black", [ "h-9.6", "v-67", "h
   -105.6" ] ]
  ],
  "dependencies": {}
}
```

Listing 9: Conexiones del esp32

Archivo Main.py

```
from machine import Pin, PWM
import ujson
import utime as time
import dht
from led_pwm import LED
from helpers import WiFiConnector, MqttConnector
from helpers import ChannelTypes, DeviceType, DevicesManager
from helpers import SwitchType, SwitchDeviceManager
from helpers import DelayedMethod, APIClient
from env_settings import *

# MQTT Setup
MQTT_CLIENT_ID = DEVICE_ID
MQTT_ENABLE_SSL = False
MQTT_SSL_PARAMS = {'server_hostname': MQTT_BROKER}

MQTT_TELEMETRY_TOPIC = "mqtt.publish.TELEMETRY" #'mqtt.publish
    .{0}'.format(ChannelTypes.TELEMETRY)
MQTT_CONTROL_TOPIC = '{0}.{1}.#'.format(GROUP_ID, DEVICE_ID)
MQTT_CONTROL_TOPIC = MQTT_CONTROL_TOPIC.replace(" ", "")

# Setup Device Params
RED_LED_DEVICE = {"name": "RED_LED_" + DEVICE_NAME.upper(), "type
    ": DeviceType.LED, "pin": 12, "id": RED_LED_DEVICE_ID}
BLUE_LED_DEVICE = {"name": "BLUE_LED_" + DEVICE_NAME.upper(), "
    type": DeviceType.LED, "pin": 13, "id": BLUE_LED_DEVICE_ID}
RED_FAN_DEVICE = {"name": "RED_FAN_" + DEVICE_NAME.upper(), "type
    ": DeviceType.MOTOR, "pin": 5, "id": RED_FAN_DEVICE_ID}
BLUE_FAN_DEVICE = {"name": "BLUE_FAN_" + DEVICE_NAME.upper(), "
    type": DeviceType.MOTOR, "pin": 18, "id": BLUE_FAN_DEVICE_ID}
THERMOMETER_DEVICE = {"name": "THERMOMETER_" + DEVICE_NAME.upper
    (), "type": DeviceType.THERMOMETER, "pin": 15, "id":
    THERMOMETER_DEVICE_ID}
HUMIDITY_DEVICE = {"name": "HUMIDITY_" + DEVICE_NAME.upper(), "
    type": DeviceType.HUMIDITY, "pin": 15, "id":
    HUMIDITY_DEVICE_ID}

# Create Devices
devices_manager = DevicesManager(gateway_name=DEVICE_NAME,
    gateway_id=DEVICE_ID)
devices_manager.create_device(RED_LED_DEVICE)
devices_manager.create_device(BLUE_LED_DEVICE)
if USE_FAN:
  devices_manager.create_device(RED_FAN_DEVICE)
  devices_manager.create_device(BLUE_FAN_DEVICE)
  RED_FAN = devices_manager.get_controller(RED_FAN_DEVICE["name"
    ])
  BLUE_FAN = devices_manager.get_controller(BLUE_FAN_DEVICE["name
    "])

```

```
39 if USE_DHT_SENSOR:
40   devices_manager.create_device(THERMOMETER_DEVICE)
41   devices_manager.create_device(HUMIDITY_DEVICE)
42
43
44 # LED/LAMP Setup
45 RED_LED = devices_manager.get_controller(RED_LED_DEVICE["name"])
46 BLUE_LED = devices_manager.get_controller(BLUE_LED_DEVICE["name"
      ])
47 FLASH_LED = Pin(2, Pin.OUT)
48
49
50 # Turn On LEDs
51 RED_LED.on(100)
52 BLUE_LED.on(100)
53 if USE_FAN:
54   RED_FAN.on(25)
55   BLUE_FAN.on(25)
56
57
58
59
60
61
62 # -------------- Application Logic --------------
63
64 # Connect to WiFi
65 wifi_connector = WiFiConnector(WIFI_SSID, WIFI_PASSWORD)
66 wifi_connector.connect()
67
68 # Register Devices to IoTConnector Cloud
69 api_client = APIClient("http://{0}:{1}".format(URL_HOST, URL_PORT
      ))
70 provision_device_data = devices_manager.
      get_provisioning_device_data(DEVICE_SECRETE)
71 url_params = "/api/v1/devices/provision/group/{0}/{1}".format(
      GROUP_ID, API_KEY)
72 provisioning_response = api_client.post(url_params, data=
      provision_device_data)
73 print("Provisioned Devices: ", provisioning_response)
74
75
76 # Connect to MQTT
77 def did_recieve_subscription_message(topic, message):
78   print("\n** message received **")
79   print("topic: ", topic)
80   print("message: ", message)
81   received_command = devices_manager.get_device_command(topic,
       message)
82   devices_manager.run_device_command(received_command)
83   # print("aaaaaa: ", ujson.dumps(received_command))
```

```
84
85
86 mqtt_connector = MqttConnector(MQTT_CLIENT_ID, MQTT_BROKER,
       MQTT_USER_NAME, MQTT_PASSWORD, MQTT_ENABLE_SSL,
       MQTT_SSL_PARAMS)
87 mqtt_connector.connect(did_recieve_subscription_message)
88 mqtt_connector.subscribe(MQTT_CONTROL_TOPIC)
89
90 # Turn Off LEDs
91 RED_LED.off()
92 BLUE_LED.off()
93 if USE_FAN:
94   RED_FAN.off()
95   BLUE_FAN.off()
96
97 # ---------
98
99 RED_LED_SWITCH = SwitchDeviceManager(33, SwitchType.TOGGLE,
       RED_LED)
100 BLUE_LED_SWITCH = SwitchDeviceManager(32, SwitchType.TOGGLE,
       BLUE_LED)
101
102 RED_FAN_SWITCH = None
103 BLUE_FAN_SWITCH = None
104 if USE_FAN:
105   RED_FAN_SWITCH = SwitchDeviceManager(21, SwitchType.INCREMENT,
       RED_FAN)
106   BLUE_FAN_SWITCH = SwitchDeviceManager(19, SwitchType.INCREMENT,
       BLUE_FAN)
107
108
109 # push sensor data to cloud via mqtt
110 def push_data():
111     telemetry_data = {
112         "type": ChannelTypes.TELEMETRY,
113         "data": devices_manager.get_data()
114     }
115
116     telemetry_data_json = ujson.dumps(telemetry_data)
117     mqtt_connector.publish(MQTT_TELEMETRY_TOPIC,
       telemetry_data_json)
118     # print("data: ", devices_manager.get_devices_list_json())
119
120
121 push_device_data_delay = DelayedMethod(push_data, 10)
122
123 # send data on connection
124 push_data()
125
126 while True:
127   push_device_data_delay.run()
```

```
  mqtt_connector.check_incoming_msg()
  time.sleep(0.1)
```

Listing 10: Archivo Main

Archivo led_pwm.py

```
from machine import Pin, PWM

class LED:
    def __init__(self, pin_num, freq=1500):
        self.pin_num = pin_num
        self.freq = freq
        self.pwm = PWM(Pin(pin_num), freq=freq)
        self.is_on = False  # Track the LED state
        self.min_percent = 0
        self.percent = 0  # Track the brightness state
        self.duty = int(self.min_percent / 100 * 1023)  # Track the duty state

    def on(self, percentage = 0):
        if percentage > 0:
            self.percent = percentage
            self.duty = int(percentage / 100 * 1023)

        self.pwm.duty(self.duty)  # Set duty cycle to 1023 (maximum value) to fully turn on the LED
        self.is_on = True  # Update LED state

    def off(self):
        self.pwm.duty(0)  # Set duty cycle to 0 to fully turn off the LED
        self.is_on = False  # Update LED state

    def set_brightness(self, percentage):
        if percentage > 0:
            # Convert percentage to a duty cycle value between 0 and 1023 (100%)
            pvalue = percentage if percentage > self.min_percent else self.min_percent
            self.percent = percentage
            self.duty = int(pvalue / 100 * 1023)
            if self.is_on == True:
                self.pwm.duty(self.duty)  # Set duty cycle to the calculated value

        else:
            self.off()


```

```
38      def get_brightness(self):
39          # when off, return brightness value as 0
40          if self.is_on == False:
41              return 0
42
43          return self.percent
44
45      def set_value(self, state):
46          if state == 0:
47              self.off()
48          elif state == 1:
49              self.on()
50          else:
51              raise ValueError("Invalid state value. Use 0 for off
    and 1 for on.")
52
53      def value(self):
54          return 1 if self.is_on == True else 0  # Return current
    LED state (1 if on, 0 if off)
55
56      def deinit(self):
57          self.pwm.deinit()  # Deinitialize the PWM to clean up
```

Listing 11: Archivo de control del LED

Archivo Helper.py

```
1  from machine import Pin, PWM, Timer
2  import network
3  import utime as time
4  from umqtt.simple import MQTTClient
5  from led_pwm import LED
6  import dht
7  import ntptime
8  import ujson
9  import urandom
10 import urequests as requests
11
12 # ------------ HELPER METHODS ----------------- #
13 # Synchronize with NTP server
14 def synchronize_ntp_time():
15     # Specify a different NTP server (e.g., Google's NTP server)
16     ntptime.host = "time.google.com"
17     # Synchronize with the NTP server
18     ntptime.settime()
19
20 # Get UTC Date String
21 def get_formatted_time_string():
22     # Get the current time in UTC
23     current_time_utc = time.gmtime()
```

```
24
25      # Format the time
26      formatted_time = "{:04d}-{:02d}-{:02d}T{:02d}:{:02d}:{:02d
    }.{:06d}Z".format(
27          current_time_utc[0], current_time_utc[1],
    current_time_utc[2],
28          current_time_utc[3], current_time_utc[4],
    current_time_utc[5],
29          current_time_utc[6]
30      )
31
32      return formatted_time
33
34  # Generate random UUID
35  def generate_uuid():
36      uuid_format = "xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx"
37      uuid = ""
38      for char in uuid_format:
39          if char == "x":
40              uuid += "{:x}".format(urandom.getrandbits(4))
41          elif char == "y":
42              uuid += "{:x}".format(8 | (urandom.getrandbits(3)))
43          else:
44              uuid += char
45      return uuid
46
47
48  # ------------ HELPER CLASSES ----------------- #
49  # WiFi Connection Manager
50  class WiFiConnector:
51      def __init__(self, ssid, password):
52          self.ssid = ssid
53          self.password = password
54          self.wifi_client = None
55
56      def connect(self):
57          # Connect to WiFi
58          self.wifi_client = network.WLAN(network.STA_IF)
59          self.wifi_client.active(True)
60          print("Connecting device to WiFi")
61          self.wifi_client.connect(self.ssid, self.password)
62
63          # Wait until WiFi is Connected
64          while not self.wifi_client.isconnected():
65              print("Connecting...")
66              time.sleep(0.1)
67          print("WiFi Connected!")
68          print(self.wifi_client.ifconfig())
69          synchronize_ntp_time()
70
71          # Print the current time in UTC
```

```
72          print("Current Time (UTC): ", get_formatted_time_string()
    )
73
74
75  # MQTT Connection Manager
76  class MqttConnector:
77      def __init__(self, client_id, broker, user, password, ssl=
    False, ssl_params=None):
78          self.client_id = client_id
79          self.broker = broker
80          self.user = user
81          self.password = password
82          self.ssl = ssl
83          self.ssl_params = ssl_params
84          self.client = None
85
86
87      def connect(self, did_recieve_callback):
88          print("Connecting to MQTT broker ...")
89          if self.ssl:
90              self.client = MQTTClient(self.client_id, self.broker,
     user=self.user,
91                                       password=self.password, ssl=self.
    ssl_params, ssl_params=self.ssl_params)
92          else:
93              self.client = MQTTClient(self.client_id, self.broker,
     user=self.user, password=self.password)
94
95          self.client.set_callback(did_recieve_callback)
96          self.client.connect()
97          print("MQTT broker is Connected.")
98
99          return self.client
100
101     def subscribe(self, topic):
102         self.client.subscribe(topic)
103         print("subscribed to topic = " + topic)
104
105     def publish(self, topic, data):
106         try:
107             print("\nUpdating MQTT Broker...")
108             self.client.publish(topic, data)
109             print("data sent to: ", topic)
110         except:
111             print("ERROR: MQTT client may not be initialized.")
112
113     def check_incoming_msg(self):
114         self.client.check_msg()
115
116 # Device Types
117 class DeviceType:
```

```
    LED = "LED"
    THERMOMETER = "THERMOMETER"
    HUMIDITY = "HUMIDITY"
    GATEWAY = "GATEWAY"
    MOTOR = "MOTOR"

# Device Profiles
class DefaultProfileName:
    SENSOR = "SENSOR"
    LED = "LED"
    LAMP = "LAMP"
    HUMIDITY = "HUMIDITY"
    TEMPERATURE = "TEMPERATURE"
    MOTOR = "MOTOR"
    DEFAULT = "DEFAULT"

# Device States
class StateName:
    STATE = "STATE"
    BRIGHTNESS = "BRIGHTNESS"
    TEMPERATURE = "TEMPERATURE"
    HUMIDITY = "HUMIDITY"
    SENSOR = "SENSOR"
    SPEED = "SPEED"

# Device Channel Types
class ChannelTypes:
    COMMAND = "COMMAND"
    TELEMETRY = "TELEMETRY"
    ALERTS = "ALERTS"

# Devices Manager
class DevicesManager:
    def __init__(self, gateway_name, gateway_id):
        self.gateway_name = gateway_name
        self.gateway_id = gateway_id
        self.devices = {}
        device = {
            "device_id": self.gateway_id,
            "device_name": self.gateway_name,
            "device_type": DeviceType.GATEWAY,
            "parent": "0",   # Default Gateway Parent = 0
            "pin": -1,
            "controller": None
        }
        self.devices[self.gateway_name] = device

    def add_device(self, device_id, device_name, device_type, pin
    ):
        if device_name not in self.devices:
            controller = self.create_controller(device_type, pin)
```

```
168             device = {
169                 "device_id": device_id,
170                 "device_name": device_name,
171                 "device_type": device_type,
172                 "pin": pin,
173                 "parent": self.gateway_id,
174                 "controller": controller
175             }
176             self.devices[device_name] = device
177 
178             return device
179         else:
180             raise ValueError("A device with name '{}' already
    exists. Device Names must be unique.".format(device_name))
181 
182     def create_device(self, device_data):
183         return self.add_device(device_data["id"], device_data["
    name"], device_data["type"], device_data["pin"])
184 
185     def create_controller(self, device_type, pin):
186         if device_type == DeviceType.LED or device_type ==
    DeviceType.MOTOR:
187             return LED(pin)
188         elif device_type == DeviceType.THERMOMETER or device_type
     == DeviceType.HUMIDITY:
189             DHT_PIN = Pin(pin)
190             dht_sensor = dht.DHT22(DHT_PIN)
191             return dht_sensor
192 
193         return None
194 
195     def get_controller(self, device_name):
196         return self.devices[device_name]["controller"]
197 
198     def get_devices_list_json(self):
199         return ujson.dumps(self.get_data())
200 
201     def get_device_command(self, topic, message):
202         command_message = ujson.loads(message.decode())
203         command_topic = topic.decode().split('/')
204         command_data = {}
205 
206         if command_topic[-1] == ChannelTypes.COMMAND:
207             command_data = {
208                 "type": ChannelTypes.COMMAND,
209                 "group_id": command_topic[0],
210                 "parent_id": command_topic[1] if not
    command_topic[1] == command_topic[-2] else "0",
211                 "device_id": command_topic[-2],
212                 "commands": command_message,
213             }
```

```
214         else:
215             print("** command is not supported **")
216 
217         return command_data
218 
219     def run_device_command(self, device_command):
220         # if device is not a gatweay
221         if not device_command["parent_id"] == "0":
222             s_device = {}
223             for key, device in self.devices.items():
224                 if device["device_id"] == device_command["
    device_id"]:
225                     s_device = device
226                     self._set_command_for_device(device,
    device_command)
227                     break
228         else:
229             print("gateway commands are not yet supported")
230 
231     def _set_command_for_device(self, device, device_command):
232         states = []
233         for command in device_command["commands"]:
234             command_name = command["name"]
235             command_value = command["value"]
236 
237             if device["device_type"] == DeviceType.GATEWAY:
238                 print("gateway commands are not yet supported")
239             elif device["device_type"] == DeviceType.LED:
240                 if command_name == StateName.STATE:
241                     device["controller"].set_value(1 if
    command_value == "ON" else 0)
242                 elif command_name == StateName.BRIGHTNESS:
243                     command_value_int = int(command_value)
244                     device["controller"].set_value(1 if
    command_value_int > 0 else 0)
245                     device["controller"].set_brightness(
    command_value_int)
246             if device["device_type"] == DeviceType.MOTOR:
247                 if command_name == StateName.STATE:
248                     device["controller"].set_value(1 if
    command_value == "ON" else 0)
249                 elif command_name == StateName.SPEED:
250                     command_value_int = int(command_value)
251                     device["controller"].set_value(1 if
    command_value_int > 0 else 0)
252                     device["controller"].set_brightness(
    command_value_int)
253             elif device["device_type"] == DeviceType.THERMOMETER
    or device["device_type"] == DeviceType.HUMIDITY:
254                 print("** humidity and temperature sensor
    commands are not yet supported")
```

```
255
256     def get_data(self):
257         devices_data_list = []
258         uuid_str = generate_uuid()
259         for key, device in self.devices.items():
260             device_data = {}
261             device_data["device_id"] = device["device_id"]
262             device_data["type"] = device["device_type"]
263             device_data["name"] = device["device_name"]
264             device_data["parent"] = device["parent"]
265             device_data["log_id"] = uuid_str
266             device_data["log_time"] = get_formatted_time_string()
267             device_data["states"] = self.get_device_states(device
    )
268             devices_data_list.append(device_data)
269
270         return devices_data_list
271
272     def get_device_states(self, device):
273         states = []
274         if device["device_type"] == DeviceType.GATEWAY:
275             states.append({"name": StateName.STATE, "value": "ON"
    })
276         elif device["device_type"] == DeviceType.LED:
277             states.append({"name": StateName.STATE, "value": "ON"
     if device["controller"].value() == 1 else "OFF"})
278             states.append({"name" : StateName.BRIGHTNESS , "value
    ": device["controller"].get_brightness()})
279         if device["device_type"] == DeviceType.MOTOR:
280             states.append({"name": StateName.STATE, "value": "ON"
     if device["controller"].value() == 1 else "OFF"})
281             states.append({"name" : StateName.SPEED , "value":
    device["controller"].get_brightness()})
282         elif device["device_type"] == DeviceType.THERMOMETER or
    device["device_type"] == DeviceType.HUMIDITY:
283             dht_sensor = device["controller"]
284             dht_sensor.measure()
285             time.sleep(0.1)
286             states.append({"name" : StateName.SENSOR , "value":
    dht_sensor.temperature() if device["device_type"] ==
    DeviceType.THERMOMETER else dht_sensor.humidity()})
287
288         return states
289
290     def get_provisioning_device_data(self, secrete_key):
291         all_devices = self.get_data()
292         parent_device = {}
293         sub_devices = []
294
295         for device_data in all_devices:
```

```
296             p_device_data = self.map_provision_device_data(
    device_data)
297             if device_data["type"] == DeviceType.GATEWAY:
298                 parent_device = p_device_data
299             else:
300                 sub_devices.append(p_device_data)
301 
302         response = {
303             "parent_device": parent_device,
304             "sub_devices": sub_devices,
305             "provision_parent": True,
306             "secrete_key": secrete_key
307         }
308 
309         return response
310 
311     def map_provision_device_data(self, device_data):
312 
313         profile_name = DefaultProfileName.DEFAULT
314         if device_data["type"] == DeviceType.HUMIDITY or
    device_data["type"] == DeviceType.THERMOMETER:
315             profile_name = DefaultProfileName.SENSOR
316         elif device_data["type"] == DeviceType.LED:
317             profile_name = DefaultProfileName.LED
318         elif device_data["type"] == DeviceType.MOTOR:
319             profile_name = DefaultProfileName.MOTOR
320 
321 
322         provision_device_data = {
323             "name": device_data["name"],
324             "type": device_data["type"],
325             "parent_id": device_data["parent"],
326             "location": "",
327             "zone": "",
328             "profile_name": profile_name,
329             "id": device_data["device_id"],
330             "is_new": True
331         }
332 
333         return provision_device_data
334 
335 
336 
337 # Switch Types
338 class SwitchType:
339     TOGGLE = "TOGGLE"
340     INCREMENT = "INCREMENT"
341 
342 # Switch Device Manager
343 class SwitchDeviceManager:
```

```
344     def __init__(self, switch_pin, switch_type, pwm_device,
    increment=25):
345         self.switch = Pin(switch_pin, Pin.IN, Pin.PULL_UP)
346         self.pwm_device = pwm_device
347         self.debounce_timer = Timer(0)
348         self.timer_running = False
349 
350         self.switch.irq(trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING,
     handler=self.handle_switch_interrupt)
351         self.switch_type = switch_type
352         self.increment = increment
353 
354         self.switch_state = 1  # Initial switch state (1 for pull
    -up)
355 
356     def handle_switch_interrupt(self, pin):
357         self.debounce_timer.init(mode=Timer.ONE_SHOT, period=50,
    callback=self.debounce_callback)
358 
359         # if not self.timer_running:
360         #     print("bbbbb")
361         #     self.debounce_timer.init(mode=Timer.ONE_SHOT,
    period=50, callback=self.debounce_callback)
362         #     self.timer_running = True
363 
364     def debounce_callback(self, timer):
365         self.timer_running = False
366         current_state = self.switch.value()
367 
368         if current_state != self.switch_state:  # Check if switch
     state has changed
369             self.switch_state = current_state
370             switch_is_pressed_down = 0 # Switch pressed (falling
    edge)
371             if not current_state == switch_is_pressed_down:  #
    switch is released
372                 if self.switch_type == SwitchType.TOGGLE:
373                     self.toggle_pwm_device()
374                 if self.switch_type == SwitchType.INCREMENT:
375                     self.increment_pwm_device()
376 
377     def toggle_pwm_device(self):
378         if self.pwm_device.value() == 1:
379             self.pwm_device.off()
380         else:
381             self.pwm_device.on()
382 
383     def increment_pwm_device(self):
384         current_percent = self.pwm_device.get_brightness()
385         set_percent = current_percent + self.increment
386 
```

```
387         if set_percent > 100:
388             set_percent = 0
389 
390         if set_percent == 0:
391             self.pwm_device.set_brightness(set_percent)
392             self.pwm_device.off()
393         else:
394             self.pwm_device.on(set_percent)
395             self.pwm_device.set_brightness(set_percent)
396 
397 
398 # DelayedMethod: Run a method after a delay
399 class DelayedMethod:
400     def __init__(self, callback, delay_seconds=10):
401         # Keep track of the start time
402         self.start_time = time.time()
403         self.callback = callback
404         self.delay_seconds = delay_seconds
405         self.current_time = time.time()
406 
407     def run(self):
408         # Check if delayed seconds have passed
409         self.current_time = time.time()
410         if self.current_time - self.start_time >= self.
    delay_seconds:
411             # Call the delayed method
412             self.callback()
413             # Reset the start time for the next interval
414             self.start_time = self.current_time
415 
416 
417 
418 
419 class APIClient:
420     def __init__(self, base_url):
421         self.base_url = base_url
422 
423     def _make_request(self, method, endpoint, data=None, params="
    ", headers=None, bearerToken=""):
424         url = self.base_url + endpoint
425         headers = headers or {
426             "Content-Type": "application/json",
427             "Authorization": "Bearer " + bearerToken,  # Include
    any other headers as needed
428         }
429         response = None
430 
431         print("url = ", url)
432 
433         try:
434             if method == "GET" and params:
```

```
435                 response = requests.get(url, params=params,
    headers=headers)
436             elif method == "POST":
437                 response = requests.post(url, json=data, headers=
    headers)
438             else:
439                 raise ValueError("Unsupported HTTP method")
440 
441             # Check if the status code is in the 2xx range (
    success)
442             if 200 <= response.status_code < 300:
443                 return ujson.loads(response.text)
444             else:
445                 raise Exception("HTTP Error - Status Code:",
    response.status_code, "Error Message:", response.text)
446         except Exception as e:
447             print("An error occurred...")
448             raise e
449         finally:
450             if not response == None:
451                 response.close()
452 
453     def get(self, endpoint, params=None, headers=None):
454         return self._make_request("GET", endpoint, params=params,
     headers=headers)
455 
456     def post(self, endpoint, data=None, headers=None):
457         return self._make_request("POST", endpoint, data=data,
    headers=headers)
```

Listing 12: Archivo Helper

Archivo env_settings.py

```
# Env Setup
USE_FAN = True
USE_DHT_SENSOR = True

# HTTP HOST
URL_HOST = "xxx.xxx.xxx.xxx" # "Please Enter your HOST IP"
URL_PORT = 7000

# Device Setup
GROUP_ID = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx" # "Please Enter your
   GROUP ID"
ESP_32_GATEWAY_ID = "4cc59a28-2413-441c-93e5-82b341ea71a2"  # "
   Please Enter your DEVICE ID"
DEVICE_NAME = "wokwi001"    # "Please Enter your Device Name"
DEVICE_SECRETE = DEVICE_NAME + "_ekMh2ZfG2vLgGg0Q"

# WiFi Setup
WIFI_SSID = "Wokwi-GUEST"
WIFI_PASSWORD = ""

# MQTT Setup
MQTT_BROKER = URL_HOST  # "Please Enter your MQTT BROKER ID"
MQTT_USER_NAME = ESP_32_GATEWAY_ID
MQTT_PASSWORD = DEVICE_SECRETE

# Setup Device ID Params
DEVICE_ID = ESP_32_GATEWAY_ID
RED_LED_DEVICE_ID = "63a0a6a7-4c45-46d4-bdc9-0089907ffeab"
BLUE_LED_DEVICE_ID = "833e7332-f92c-4ba2-891f-7d50027b6dc6"
RED_FAN_DEVICE_ID = "46d692ad-294e-4874-9dbd-45a979451265"
BLUE_FAN_DEVICE_ID = "f245b522-7510-47b8-88ea-d3728b03f986"
THERMOMETER_DEVICE_ID = "48662f4b3-e03f-48ba-b2a6-f101567d8d4e"
HUMIDITY_DEVICE_ID = "4754e380-a7c0-407b-b4da-c2122335d521"

# Keys
API_KEY = "
   sPpqDue2OWqliMZGxmMRW4TK8R0uPUjlKe3i7S1oiJ5DfqgkQ9BvDRPT9sxOjN3y
   "
```

Listing 13: Archivo de conexión