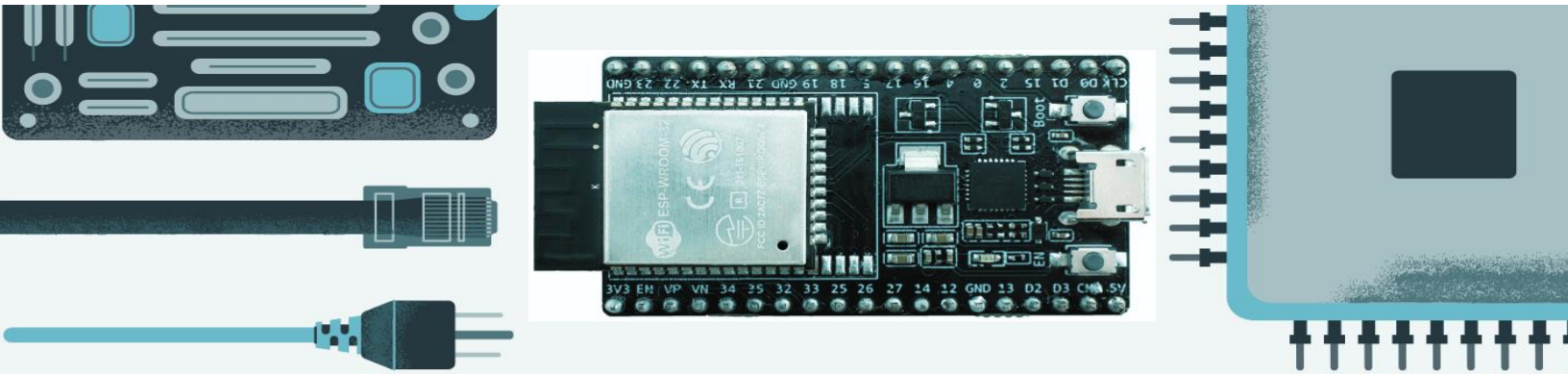
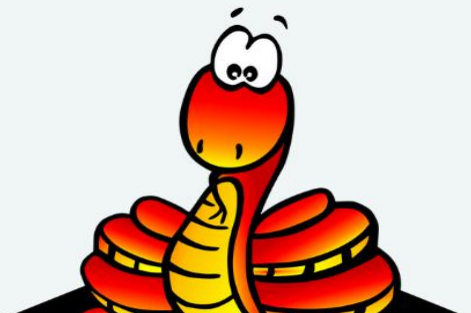
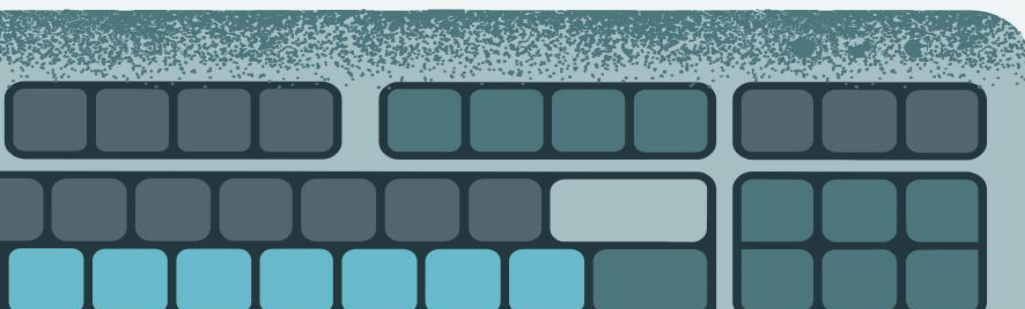


Integrantes: Juan Jose Sanchez Castellanos y Andres Villada



Proyecto Realizado con Micropython

# PROYECTO ESP32



## Tabla de contenido

Funcionalidades .....	3
1. Monitoreo de Datos Ambientales .....	3
2. Almacenamiento y Procesamiento en la Nube .....	3
3. Visualización de Datos .....	3
4. Control de Dispositivos .....	3
Funcionalidades Específicas del ESP32 .....	4
Funcionalidades Específicas del AWS.....	4
Funcionalidades Específicas de la Página Web .....	5
Funcionalidades Específicas de la App Móvil (Serial Bluetooth Terminal).....	5
Conexiones Protoboard.....	6
Componentes Necesarios.....	6
Esquema de Conexión .....	7
AWS.....	8
Aws Iot Core .....	8
Aws DynamoDB.....	9
Aws Lambda .....	10
Aws S3 Bucket.....	13
Aws Api Gateway .....	16
Código del ESP32 .....	18
Main.py.....	18
Config.py .....	20
Aws.py .....	21
Sensor.py .....	22
Wifi.py.....	23
Control_fan.py .....	24
Control_Led.py.....	25
Bluetooth.py .....	26
BLE_uart_peripheral.py .....	28
Web_server.py .....	31
Index.html.....	34
Implementación del Proyecto .....	36
Cibergrafía .....	38

## **Funcionalidades**

### **1. Monitoreo de Datos Ambientales**

- **Lectura de Datos del Sensor DHT11:**
  - Mide la temperatura y la humedad.
- **Publicación de Datos a AWS IoT Core:**
  - Envío de los datos leídos del DHT11 a AWS IoT Core usando MQTT.

### **2. Almacenamiento y Procesamiento en la Nube**

- **Función Lambda para Almacenamiento en DynamoDB:**
  - Procesa los datos recibidos desde AWS IoT Core y los guarda en una tabla DynamoDB.
- **Interfaz API Gateway:**
  - Proporciona un endpoint para interactuar con los datos.
- **Función Lambda para Transferencia a S3:**
  - Extrae los datos de DynamoDB y los guarda en un bucket S3.

### **3. Visualización de Datos**

- **Bucket S3 para Hospedaje de Página Web:**
  - Almacena y sirve una página web que grafica los datos almacenados en S3.

### **4. Control de Dispositivos**

- **Control de LED:**
  - **Desde la Página Web:**
    - Permite encender y apagar un LED conectado al ESP32 a través de una interfaz web.
  - **Por Bluetooth:**

- Permite encender y apagar el LED usando comandos enviados desde una app móvil (Serial Bluetooth Terminal).
- **Control del Ventilador:**
  - **Desde la Página Web:**
    - Permite ajustar la velocidad del ventilador a través de un control deslizante en la interfaz web.
  - **Por Bluetooth:**
    - Permite ajustar la velocidad del ventilador usando comandos enviados desde una app móvil.

### **Funcionalidades Específicas del ESP32**

1. **Conexión Wi-Fi:**
  - Conexión a una red Wi-Fi para enviar datos a AWS IoT Core y servir la página web.
2. **Servidor Web Integrado:**
  - Sirve una página web localmente, permitiendo el control del LED y el ventilador.
3. **Bluetooth:**
  - Permite el control del LED y el ventilador usando comandos enviados desde una app móvil.
4. **Publicación de Datos MQTT:**
  - Envío periódico de datos de temperatura y humedad a AWS IoT Core.

### **Funcionalidades Específicas del AWS**

1. **AWS IoT Core:**
  - Recibe los datos enviados desde el ESP32.
2. **AWS Lambda:**

- Procesa los datos y los guarda en DynamoDB.
- Transfiere los datos a S3 para visualización.

### **3. Amazon DynamoDB:**

- Almacena los datos de temperatura y humedad recibidos.

### **4. Amazon S3:**

- Almacena la página web y los datos para visualización.

### **5. Amazon API Gateway:**

- Proporciona endpoints para interactuar con los datos almacenados y transferidos.

## **Funcionalidades Específicas de la Página Web**

### **1. Gráficos de Datos Ambientales(aws):**

- Visualización de gráficos de temperatura y humedad.

### **2. Controles de Dispositivos(Servidor web):**

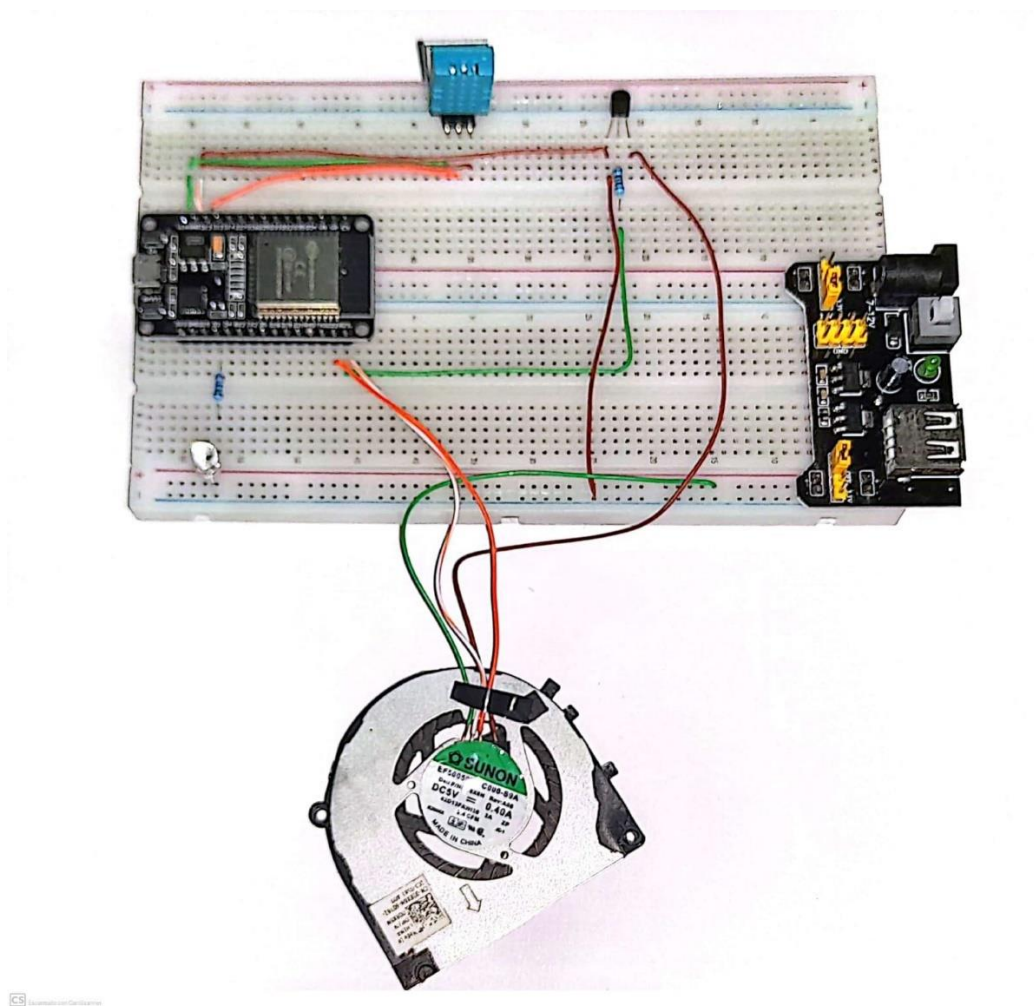
- Interfaz para encender/apagar el LED.
- Control deslizante para ajustar la velocidad del ventilador.

## **Funcionalidades Específicas de la App Móvil (Serial Bluetooth Terminal)**

### **1. Envío de Comandos:**

- Permite enviar comandos al ESP32 para controlar el LED y el ventilador.

## Conexiones Protoboard



*Ilustración 1. Conexión Protoboard*

## Componentes Necesarios

1. ESP32
2. DHT11 Sensor
3. Ventilador de 4 cables
4. Transistor 2N2222
5. Resistencia de 1k ohmio
6. LED

7. Resistencia de 220 ohmios

## **Esquema de Conexión**

### ***Sensor DHT11***

1. VCC (Pin 1 del DHT11): Conectar a 3.3V en el ESP32.
2. Data - Out (Pin 2 del DHT11): Conectar a un pin digital del ESP32 (D13).
3. GND (Pin 3 del DHT11): Conectar a GND en el ESP32.

### ***Ventilador Dell(SUNON Dc5v) de 4 cables (PWM Control)***

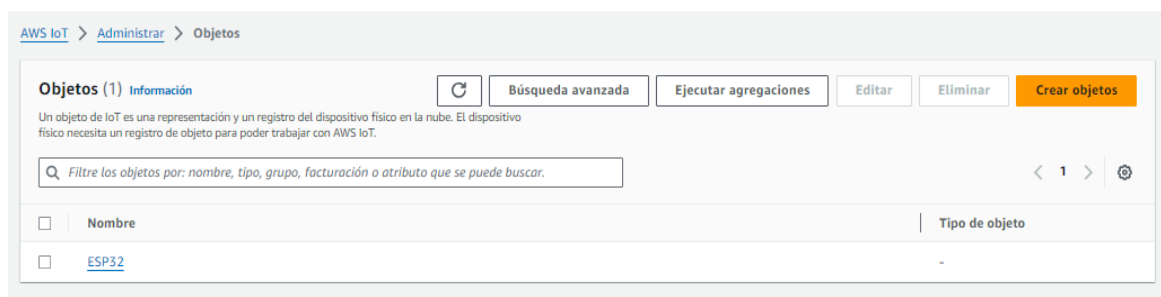
1. **VCC (Cable Rojo del Ventilador):** Conectar a **5V** en la fuente
2. **GND (Cable Negro del Ventilador):** Conectar a **colector del transistor.**
3. **Control (Cable Amarillo del Ventilador):** Conectar a un pin digital del ESP32 (D22).
4. **Tach (Cable Blanco del Ventilador):** Conectar a un pin de entrada del ESP32 si necesitas leer la velocidad del ventilador (D23).
5. **Conectar el Emisor(transistor):** Conecta la salida del emisor al GND de la fuente y del ESP32.
6. **Conectar la Base(transistor):** Conectar con una resistencia de 1kohmio la base al mismo pin del Tach.

### ***LED***

1. **Anodo del LED:** Conectar a un pin digital del ESP32 (D2) a través de una resistencia de **220 ohmios.**
2. **Cátodo del LED:** Conectar a **GND** de la fuente o del ESP32.

## AWS

### Aws Iot Core



*Ilustración 2. Objeto Esp32 Creado*

### Crear un "Objeto" en AWS IoT Core

1. Inicia sesión en la consola de AWS y navega a AWS IoT Core.
2. Ve a **Administrar > Objetos** y haz clic en **Crear Objetos**.
3. Elige **Crear un Único Objeto**.
4. Asigna un nombre a tu "objeto" y sigue las instrucciones para crearla.
5. Durante el proceso de creación, genera un certificado y una clave privada. Descarga ambos archivos junto con el archivo de Amazon Root CA (Este último solo se usa en Arduino).
6. Crea una política de IoT que permita a tu dispositivo conectar y publicar/suscribirse a los temas. Una política básica podría ser:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect",
        "iot:Publish",
        "iot:Subscribe",
        "iot:Receive"
      ]
    }
  ]
}
```

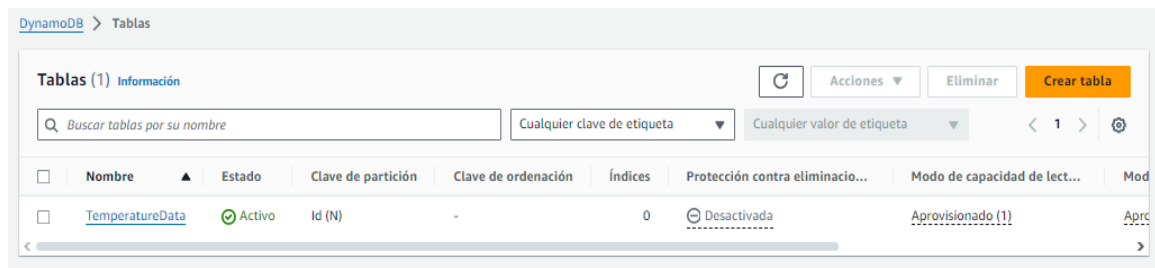


```

],
  "Resource": "*"
}
]
}

```

## Aws DynamoDB

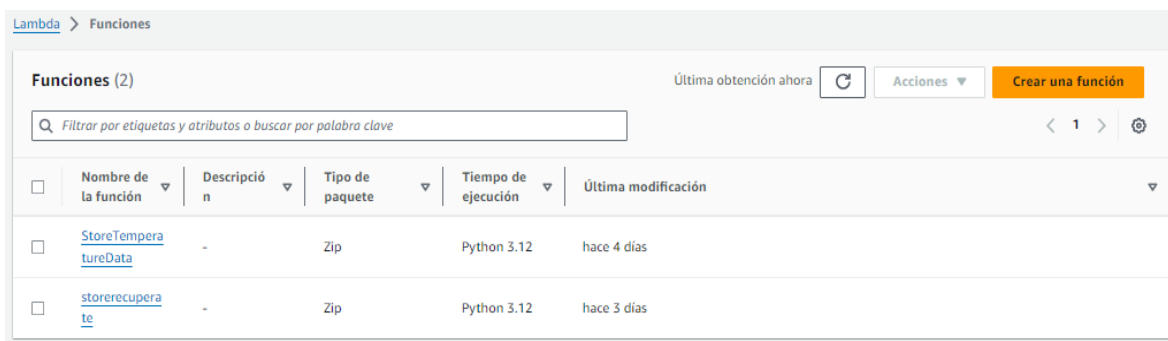


*Ilustración 3. Tabla TemperatureData*

### Crear una Tabla en DynamoDB

1. Inicia sesión en la consola de AWS.
2. Navega a **DynamoDB**.
3. Haz clic en **Crear tabla**.
4. Asigna un nombre a tu tabla, por ejemplo, **TemperatureData**.
5. Configura la clave de partición, por ejemplo, Id(numerico)) y elementos de índice, por ejemplo, **deviceId**(String), **Temperature**(Numerico), **Humidity**(Numérico) y **timestamp**(Number).
6. Crea la tabla.

## Aws Lambda



The screenshot shows the AWS Lambda console 'Funciones' page. It displays a table with two functions: 'StoreTemperatureData' and 'storetemperature'. Both functions are of type 'Zip', using 'Python 3.12', and were last modified 'hace 4 días' and 'hace 3 días' respectively. The table has columns for 'Nombre de la función', 'Descripción', 'Tipo de paquete', 'Tiempo de ejecución', and 'Última modificación'. There are also buttons for 'Crear una función' and 'Acciones'.

	Nombre de la función	Descripción	Tipo de paquete	Tiempo de ejecución	Última modificación
<input type="checkbox"/>	StoreTemperatureData	-	Zip	Python 3.12	hace 4 días
<input type="checkbox"/>	storetemperature	-	Zip	Python 3.12	hace 3 días

Ilustración 4. Funciones Lambda

## Configurar una Función Lambda para Guardar Datos en DynamoDB Crear una Función Lambda

1. Navega a **Lambda** en la consola de AWS.
2. Haz clic en **Crear Funcion**.
3. Elige **Crear desde Cero y con Python 3.12**.
4. Asigna un nombre a tu función, por ejemplo, **StoreTemperatureData**.
5. Elige un rol que permita a Lambda escribir en DynamoDB. Si no tienes uno, puedes crear uno nuevo con los permisos adecuados.

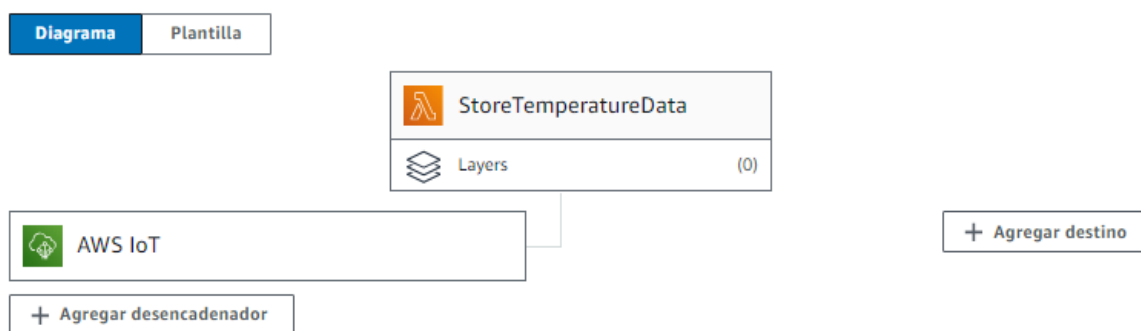


Ilustración 5. Diagrama de Funcion

## Código para la Función Lambda

Aquí tienes el código para la función Lambda que se ejecutará cuando recibas datos en AWS IoT Core y los guardará en DynamoDB:

```

1 import json
2 import boto3
3 import time
4 from decimal import Decimal
5 from botocore.exceptions import ClientError
6
7 dynamodb = boto3.resource('dynamodb')
8 table = dynamodb.Table('TemperatureData')
9
10 # Variable global para el ID
11 last_id = 0
12
13 def get_next_id():
14     global last_id
15     last_id += 1
16     return last_id
17
18 def lambda_handler(event, context):
19     print("Received event: " + json.dumps(event))
20
21     try:
22         device_id = event['device_id']
23         temperature = Decimal(str(event['temperature'])) # Convertir a Decimal
24         humidity = Decimal(str(event['humidity'])) # Convertir a Decimal
25         timestamp = int(time.time())
26
27         # Obtener el próximo ID único
28         new_id = get_next_id()
29
30         table.put_item(
31             Item={
32                 'Id': new_id, # Clave primaria única numérica
33                 'deviceId': device_id,
34                 'timestamp': timestamp,
35                 'temperature': temperature,
36                 'humidity': humidity
37             }
38         )
39
40         return {
41             'statusCode': 200,
42             'body': json.dumps('Data inserted successfully!')
43         }
44     except KeyError as e:
45         return {
46             'statusCode': 400,
47             'body': json.dumps(f'Missing key in input: {str(e)}')
48         }
49
50     except TypeError as e:
51         return {
52             'statusCode': 400,
53             'body': json.dumps(f'Type error in input: {str(e)}')
54         }
55     except ClientError as e:
56         return {
57             'statusCode': 500,
58             'body': json.dumps(f'Error: {e.response["Error"]["Message"]}')
59         }
60     except Exception as e:
61         return {
62             'statusCode': 500,
63             'body': json.dumps(f'Error: {str(e)}')
64         }
65

```

## Configurar una Función Lambda para enviar los datos de la tabla en DynamoDB y aplicarles formato json para la Api Gateway

### 1. Crear la función Lambda:

- Ve a la consola de AWS Lambda.
- Crea una nueva función que lea los datos de DynamoDB y los retorne en formato JSON.

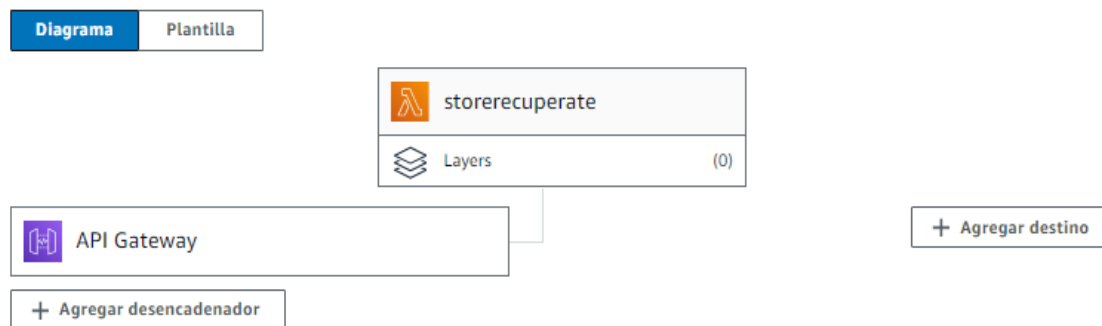


Ilustración 6. Función Recuperar Datos

## Código para la Función Lambda

Aquí tienes el código para la función Lambda que tomara los datos de la tabla en Dynamodb y los enviara en formato json:

```

1 import json
2 import boto3
3 from decimal import Decimal
4
5 dynamodb = boto3.resource('dynamodb')
6 table = dynamodb.Table('TemperatureData')
7
8 def lambda_handler(event, context):
9     response = table.scan()
10    items = response['Items']
11    return {
12        'statusCode': 200,
13        'headers': {
14            'Content-Type': 'application/json',
15            'Access-Control-Allow-Origin': '*'
16        },
17        'body': json.dumps(items, default=str)
18    }
19

```

## Crear una Regla en AWS IoT Core para Invocar la Función Lambda

1. Navega a **AWS IoT Core**.
2. Ve a **Enrutamiento de Mensajes > Reglas** y haz clic en **Crear Regla** para crear una nueva regla.

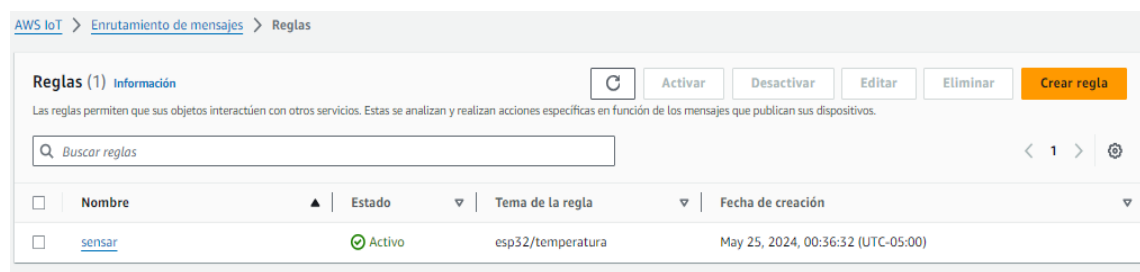


Ilustración 7. Regla Sensor

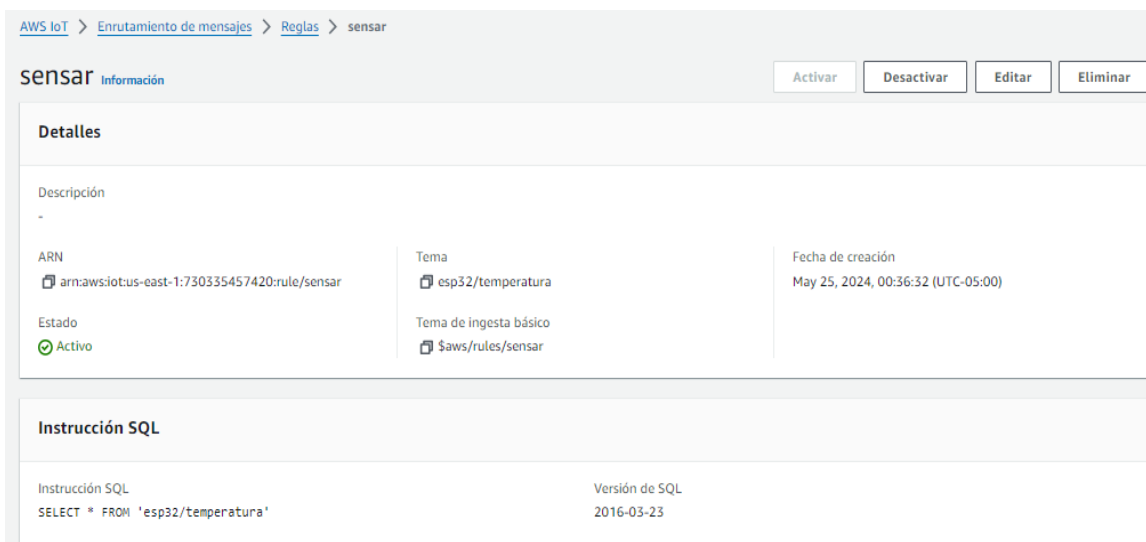


Ilustración 8. Configuración de Instrucción SQL

## Aws S3 Bucket

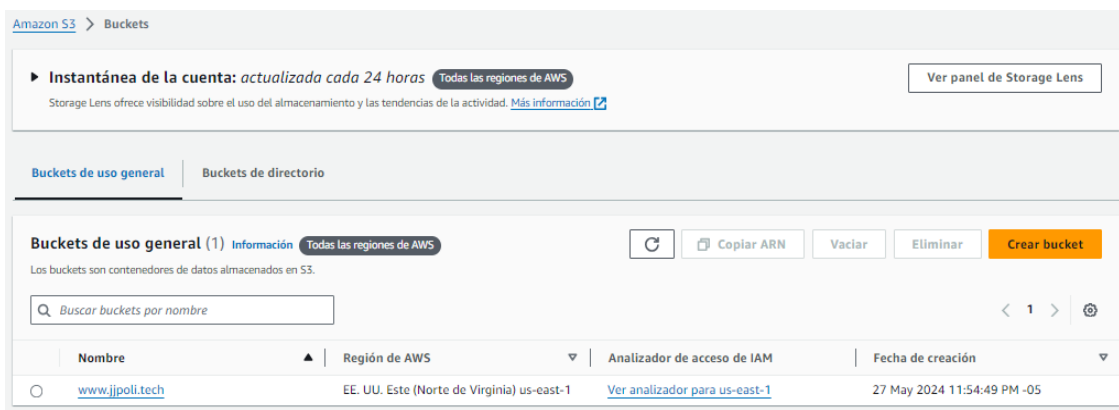


Ilustración 9. Buckets Creados

1. Debes crear el bucket como **Alojamiento de sitios web estáticos** y adjuntar el siguiente permiso:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::www.jjpoli.tech/*"
    }
  ],
}
```

```

    "Effect": "Allow",
    "Principal": "*",
    "Action": "s3:GetBucketWebsite",
    "Resource": "arn:aws:s3:::www.jjpoli.tech"
  }
]
}

```

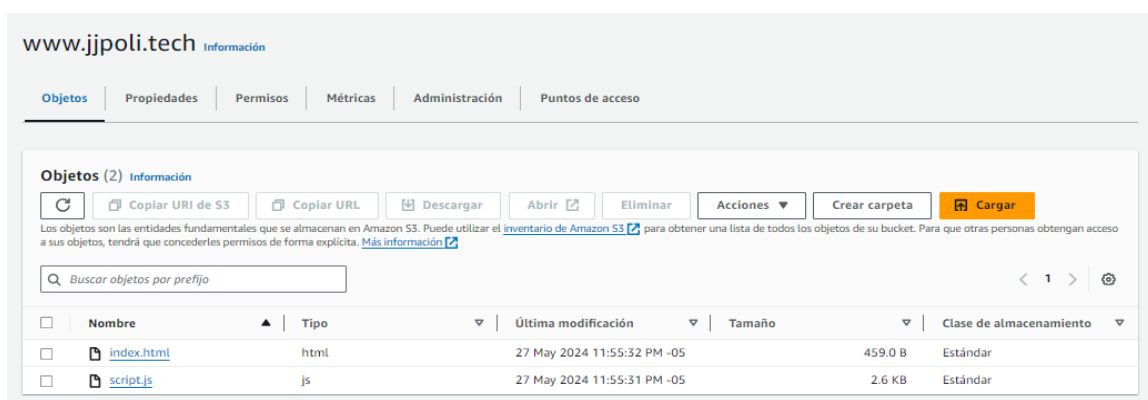


Ilustración 10. Contenido del Bucket

## Subir archivos a S3

### 1. Selecciona tu bucket en S3:

- Ve a la consola de AWS y navega a **Amazon S3**.
- Busca y selecciona el bucket que creaste anteriormente (por ejemplo, [www.jjpoli.tech](http://www.jjpoli.tech), solo si se desea usar un dominio web para visualizar, en caso contrario cualquier nombre).

### 2. Haz clic en "Cargar" y selecciona index.html y script.js:

- En la ventana de subida, haz clic en **Add files** y selecciona los archivos **index.html** y **script.js** de tu computadora.

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Gráficas de Sensores</title>
7   <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
8   <script src="script.js" defer></script>
9 </head>
10 <body>
11   <h1>Gráficas de Sensores</h1>
12   <canvas id="temperatureChart"></canvas>
13   <canvas id="humidityChart"></canvas>
14 </body>
15 </html>
16

```

Ilustración 11. Index.html

```

1  async function fetchData() {
2      ⚠ const response = await fetch('https://copw2yv17e.execute-api.us-east-1.amazonaws.com/prod/data');
3      const responseData = await response.json();
4
5      const data = JSON.parse(responseData.body);
6
7      const temperatureData = data.map(item => item.temperature);
8      const humidityData = data.map(item => item.humidity);
9      const timestamps = data.map(item => new Date(item.timestamp * 1000).toLocaleString());
10
11
12     return { temperatureData, humidityData, timestamps };
13 }
14
15 async function createCharts() {
16     const { temperatureData, humidityData, timestamps } = await fetchData();
17
18     const ctxTemperature = document.getElementById('temperatureChart').getContext('2d');
19     const ctxHumidity = document.getElementById('humidityChart').getContext('2d');
20
21     new Chart(ctxTemperature, {
22         type: 'line',
23         data: {
24             labels: timestamps,
25             datasets: [{
26                 label: 'Temperatura (°C)',
27                 data: temperatureData,
28                 borderColor: 'rgba(255, 99, 132, 1)',
29                 backgroundColor: 'rgba(255, 99, 132, 0.2)',
30             }]
31         },
32         options: {
33             responsive: true,
34             scales: {
35                 x: {
36                     display: true,
37                     title: {
38                         display: true,
39                         text: 'Tiempo'
40                     }
41                 },
42                 y: {
43                     display: true,
44                     title: {
45                         display: true,
46                         text: 'Temperatura (°C)'
47                     }
48                 }
49             }
50         }
51     });

```

```

53     new Chart(ctxHumidity, {
54         type: 'line',
55         data: {
56             labels: timestamps,
57             datasets: [{
58                 label: 'Humedad (%)',
59                 data: humidityData,
60                 borderColor: 'rgba(54, 162, 235, 1)',
61                 backgroundColor: 'rgba(54, 162, 235, 0.2)',
62             }]
63         },
64         options: {
65             responsive: true,
66             scales: {
67                 x: {
68                     display: true,
69                     title: {
70                         display: true,
71                         text: 'Tiempo'
72                     }
73                 },
74                 y: {
75                     display: true,
76                     title: {
77                         display: true,
78                         text: 'Humedad (%)'
79                     }
80                 }
81             }
82         }
83     });
84 }
85
86 createCharts();

```

Ilustración 12. Script.js

## Aws Api Gateway

Gateway de API > API

API (1/1) 🔄 Eliminar Crear API

	Nombre	Descripción	ID	Protocolo	Tipo de punto de conexión de la API	Creado
<input type="radio"/>	<a href="#">mi-primer-api</a>		copw2yvl7e	REST	Regional	2024-05-26

Ilustración 13. Api Gateway Creado

## Crear una API REST

Desarrolle una API REST en la que obtenga control total de la solicitud y la respuesta, junto con las capacidades de administración de la API.

Funciona con lo siguiente:

Lambda, HTTP, servicios de AWS



## Crear una Etapa de implementación en API Gateway

### 1. Navega a la consola de API Gateway:

- En la consola de AWS, busca y selecciona **API Gateway**.

### 2. Selecciona tu API:

- Selecciona la API que creaste anteriormente (por ejemplo, **mi-primer-api**).

### 3. Implementar API:

- En el panel de navegación de la izquierda, haz clic en **Recursos**.
- Crea un recurso y dale un nombre como data y selecciona función Get
- Haz clic en **Implementar API**.
- En el cuadro de diálogo que aparece:
  - **Implementar etapa:** Selecciona [**Nueva etapa**].
  - **Nombre Etapa:** Introduce un nombre para la etapa, como **prod** o **v1**.
  - **Descripción de Etapa** (opcional): Puedes añadir una descripción.
- Haz clic en **Implementar**.

### 4. Configurar CORS en API Gateway

- Habilita CORS para permitir las solicitudes desde tu sitio web:
- En tu API Gateway, selecciona tu API.
- Ve a Recursos y selecciona el recurso data que creaste.
- Haz clic en Habilitar CORS.
- Configura los encabezados permitidos (por ejemplo, Content-Type) y haz clic en en Get y Option, luego dale en guardar.

### 5. Obtener la URL de la API:

1. Después de desplegar la API, verás la URL de tu API en la parte superior de la pantalla bajo **URL de invocación**. Anota esta URL, ya que la necesitarás para tu archivo **script.js**.

## Paso 2: Actualizar script.js con la URL de la API

### 1. Modificar script.js:

- Abre tu archivo **script.js** y asegúrate de que la URL de la API coincide con la URL que obtuviste en el paso anterior.

## Código del ESP32

### Main.py

```

1  import time
2  from wifi import connect_wifi
3  from web_server import start_server
4  import uasyncio as asyncio
5  from bluetooth import init_bluetooth
6  from aws import connect_aws
7  from sensor import read_sensor
8  from config import TOPIC
9  import json
10
11
12  DEVICE_ID = "esp32_001"
13
14
15  async def publish_data(client):
16      while True:
17          try:
18              temp, hum = read_sensor()
19              if temp is not None and hum is not None:
20                  payload = {"humidity": hum, "device_id": DEVICE_ID, "temperature": temp}
21                  payload_json = json.dumps(payload)
22                  client.publish(TOPIC, payload_json)
23                  print("Datos publicados:", payload_json)
24              else:
25                  print("No se pudieron leer los datos del sensor.")
26                  await asyncio.sleep(20)
27          except Exception as e:
28              print("Error al publicar los datos:", e)
29              await asyncio.sleep(10)
30
31
32  async def init_services():
33      await asyncio.gather(init_bluetooth())
34      print("Bluetooth iniciado")
35
36
37  async def main():
38      connect_wifi()
39
40      client = connect_aws()
41
42      # Iniciar el Bluetooth y otros servicios
43      await init_services()
44
45      # Crear tareas para publicar datos y el servidor
46      publish_task = asyncio.create_task(publish_data(client))
47      server_task = asyncio.create_task(start_server())
48
49      # Ejecutar ambas tareas en paralelo
50      await asyncio.gather(publish_task, server_task)
51
52
53  if __name__ == "__main__":
54      asyncio.run(main())
55

```

Ilustración 14. Código main.py del esp32

### **Librerías y Módulos:**

- `time`: Proporciona varias funciones relacionadas con el tiempo.
- `connect_wifi`: Función para conectar el ESP32 a una red Wi-Fi.
- `start_server`: Función para iniciar el servidor web.
- `uasyncio`: Biblioteca de `asyncio` para MicroPython, usada para gestionar tareas asíncronas.
- `init_bluetooth`: Función para inicializar la funcionalidad Bluetooth.
- `connect_aws`: Función para conectar a AWS IoT Core.
- `read_sensor`: Función para leer datos del sensor (DHT11).
- `TOPIC`: Tópico MQTT para la publicación de datos en AWS IoT Core.
- `json`: Biblioteca para trabajar con datos en formato JSON.

### **Constante del Dispositivo:**

- `DEVICE_ID`: Identificador del dispositivo ESP32, usado en los datos enviados.

### **Función Asíncrona**

#### **Publicar Datos `publish_data(client)`:**

- **Bucle Infinito**: Corre continuamente en segundo plano.
- **Lectura del Sensor**: Llama a `read_sensor` para obtener la temperatura y la humedad.
- **Validación de Datos**: Verifica si los datos del sensor no son `None`.
- **Preparación del Payload**: Crea un diccionario con los datos y lo convierte a formato JSON.
- **Publicación en AWS IoT Core**: Publica el JSON en el tópico especificado usando el cliente MQTT (`client`).
- **Esperas Asíncronas**: Pausa la ejecución por 20 segundos entre publicaciones, y maneja errores con una pausa de 10 segundos.

### **Inicializar Servicios:**

- **Inicialización del Bluetooth**: Llama a `init_bluetooth` usando `asyncio.gather` para permitir la ejecución de otras tareas en paralelo.

- Mensaje de Confirmación: Imprime un mensaje confirmando la inicialización de Bluetooth.

### Función Principal Asíncrona main():

- Conexión Wi-Fi: Llama a connect\_wifi para conectar el ESP32 a la red Wi-Fi.
- Conexión a AWS IoT Core: Obtiene un cliente MQTT conectado a AWS IoT Core llamando a connect\_aws.
- Inicialización de Servicios: Llama a init\_services para inicializar el Bluetooth.
- Creación de Tareas Asíncronas: Crea tareas asíncronas para publish\_data y start\_server.
- Ejecución de Tareas en Paralelo: Usa asyncio.gather para ejecutar ambas tareas en paralelo.

### Ejecución del Script Condicional para Ejecutar el Script:

Verifica si el script se está ejecutando directamente (no importado como un módulo) y ejecuta main() usando asyncio.run.

### Config.py

```

1  # config.py
2  SSID = "juanjo123"
3  PASSWORD = "12345678"
4  CLIENT_ID = "ESP32juan"
5  AWS_ENDPOINT = "a1teljpf9ojn72-ats.iot.us-east-1.amazonaws.com"
6  TOPIC = "esp32/temperatura"
7  CERT_FILE = "certificate.pem.crt"
8  KEY_FILE = "private.pem.key"
9  FAN_PIN = 23
10 FAN_PWM_FREQ = 25000
11 LED_PIN = 2
12 FAN_CONTROL_PIN = 22 # Pin para el control de la velocidad del ventilador
13

```

*Ilustración 15. Código de config.py*

Este archivo se usa para establecer valores que podrían ser sensibles como parámetros necesarios para conexión wifi o conexión aws y establecer el valor de los pin de led y ventilador usados.

## Aws.py

```

1  from umqtt.simple import MQTTClient
2  from config import AWS_ENDPOINT, CLIENT_ID, CERT_FILE, KEY_FILE
3  import uio
4  import uasyncio as asyncio
5
6
7  def connect_aws():
8      try:
9          with open(KEY_FILE, "r") as f:
10             key = f.read()
11             with open(CERT_FILE, "r") as f:
12                 cert = f.read()
13             except OSError as e:
14                 print("Error al leer archivos de certificados:", e)
15                 raise
16
17             client = MQTTClient(
18                 client_id=CLIENT_ID,
19                 server=AWS_ENDPOINT,
20                 port=8883,
21                 keepalive=4000,
22                 ssl=True,
23                 ssl_params={"cert": cert, "key": key},
24             )
25
26             try:
27                 client.connect()
28                 print("Conectado a AWS IoT")
29             except Exception as e:
30                 print("Error al conectar a AWS IoT:", e)
31                 raise
32
33             return client

```

Ilustración 16. Código aws.py

## Librerías y Módulos:

- umqtt.simple.MQTTClient: Clase para manejar la conexión MQTT.
- config: Módulo que contiene configuraciones como el endpoint de AWS, el ID del cliente, y los archivos de certificado y clave privada.
- uasyncio: Biblioteca de asyncio para MicroPython, usada para gestionar tareas asíncronas, aunque no se usa directamente en esta función.

## Función para Conectar a AWS IoT Core connect\_aws():

- Lectura de Archivos de Certificados: Intenta abrir y leer el contenido del archivo de clave privada (KEY\_FILE) y el archivo de certificado (CERT\_FILE).

- Si hay algún error (por ejemplo, si los archivos no existen), se captura la excepción **OSError** y se imprime un mensaje de error, luego se vuelve a lanzar la excepción (**raise**).

### Creación del Cliente MQTT:

- `client_id`: Identificador único para el cliente MQTT, definido en `CLIENT_ID`.
- `server`: Dirección del endpoint AWS IoT, definida en `AWS_ENDPOINT`.
- `port`: Puerto usado para la conexión SSL (8883 es el puerto estándar para MQTT sobre SSL/TLS).
- `keepalive`: Intervalo de tiempo en segundos para los mensajes de keepalive (4000 segundos).
- `ssl`: Habilita SSL/TLS para la conexión.
- `ssl_params`: Parámetros SSL que incluyen el certificado y la clave privada leídos previamente.
- Llama al método `connect` del cliente MQTT para establecer la conexión con AWS IoT.
- Si la conexión es exitosa, imprime "Conectado a AWS IoT".
- Si ocurre algún error durante la conexión, se captura la excepción y se imprime un mensaje de error, luego se vuelve a lanzar la excepción (**raise**).

Retorna el objeto `client` conectado para ser usado en otras partes del código.

### Sensor.py

```

1  import machine
2  import dht
3
4
5  def read_sensor():
6      sensor = dht.DHT11(machine.Pin(13))
7      sensor.measure()
8      temp = sensor.temperature()
9      hum = sensor.humidity()
10     return temp, hum
11

```

*Ilustración 17. Código sensor.py*

### Librerías y Módulos:

- `machine`: Biblioteca de MicroPython que proporciona acceso a funcionalidades específicas del hardware, como pines GPIO, ADC, PWM, etc.

- dht: Módulo para interactuar con sensores DHT (DHT11, DHT22), que son sensores de temperatura y humedad.

### **Función para Leer Datos del Sensor DHT11 read\_sensor():**

- Crea una instancia del sensor DHT11, especificando que está conectado al pin GPIO 13 del ESP32.
- dht.DHT11(machine.Pin(13)): Indica que estamos utilizando un sensor DHT11 y que está conectado al pin 13.
- Llama al método measure() del objeto sensor para realizar una medición de temperatura y humedad.
- Este método solicita al sensor que realice una medición y actualice sus valores internos de temperatura y humedad.
- sensor.temperature(): Obtiene la última medición de temperatura del sensor.
- sensor.humidity(): Obtiene la última medición de humedad del sensor.
- Retorna una tupla con los valores de temperatura (**temp**) y humedad (**hum**) medidos por el sensor.

### **Wifi.py**

```

1  import network
2  from config import SSID, PASSWORD
3
4
5  def connect_wifi():
6      sta_if = network.WLAN(network.STA_IF)
7      sta_if.active(True)
8      sta_if.connect(SSID, PASSWORD)
9      while not sta_if.isconnected():
10         pass
11     print("Conexión Wi-Fi establecida:", sta_if.ifconfig())
12

```

*Ilustración 18. Código wifi.py*

### **Librerías y Configuración:**

- network: Biblioteca de MicroPython que proporciona acceso a la funcionalidad de red, incluyendo Wi-Fi.
- SSID y PASSWORD: Variables importadas desde un archivo de configuración (config). Contienen el nombre de la red Wi-Fi (SSID) y la contraseña necesarios para conectarse a la red.

### Función para Conectarse a la Wi-Fi `connect_wifi()`:

- Crea una instancia de la interfaz Wi-Fi en modo cliente (**STA\_IF**).
- Activa la interfaz Wi-Fi. Esto es necesario para habilitar la comunicación Wi-Fi.
- Inicia el proceso de conexión a la red Wi-Fi utilizando el SSID y la contraseña especificados.
- Un bucle while que sigue ejecutándose hasta que la conexión Wi-Fi se establece. `sta_if.isconnected()` retorna True cuando la conexión se ha realizado exitosamente.
- Una vez que la conexión Wi-Fi está establecida, se imprime un mensaje con la configuración de la conexión. `sta_if.ifconfig()` retorna una tupla con los detalles de la conexión, incluyendo la dirección IP, máscara de subred, puerta de enlace y servidor DNS.

### Control\_fan.py

```

1  from machine import Pin, PWM
2  from config import FAN_PIN, FAN_PWM_FREQ
3
4  fan = PWM(Pin(FAN_PIN), freq=FAN_PWM_FREQ)
5  fan.duty(0)
6
7
8  def set_fan_speed(speed):
9      fan.duty(speed)
10

```

*Ilustración 19. Código para ventilador*

### Importación de Bibliotecas y Configuración Inicial:

- Pin y PWM: Importados de la biblioteca machine de MicroPython, estos se utilizan para manejar los pines del ESP32 y generar señales PWM (Modulación por Ancho de Pulso).
- FAN\_PIN y FAN\_PWM\_FREQ: Variables importadas desde un archivo de config.py . FAN\_PIN indica el pin al que está conectado el ventilador, y FAN\_PWM\_FREQ establece la frecuencia de la señal PWM.
- Inicializa un objeto PWM en el pin especificado (FAN\_PIN) con la frecuencia dada (FAN\_PWM\_FREQ).
- Establece el ciclo de trabajo del PWM en 0, lo que significa que el ventilador está inicialmente apagado.

### Función para Ajustar la Velocidad del Ventilador `set_fan_speed(speed)`:



- Ajusta el ciclo de trabajo del PWM al valor speed, controlando así la velocidad del ventilador. El valor speed puede variar típicamente entre 0 (apagado) y 1023 (máxima velocidad).

### Control\_Led.py

```
1  from machine import Pin
2  from config import LED_PIN
3
4  led = Pin(LED_PIN, Pin.OUT)
5
6
7  def control_led(state):
8      led.value(state)
9
```

*Ilustración 20. Código para Led*

- Inicializa un objeto Pin en el pin especificado (LED\_PIN) y lo configura como una salida (Pin.OUT). Esto permite controlar el estado del LED (encendido o apagado).

### Función para Controlar el LED control\_led(state):

- Ajusta el estado del pin del LED al valor state. Si state es 1 (o True), el LED se enciende. Si state es 0 (o False), el LED se apaga.

## Bluetooth.py

```

1  import ubluetooth
2  from BLE_uart_peripheral import BLEUART
3  from control_fan import set_fan_speed
4  from control_led import control_led
5  import uasyncio as asyncio
6
7
8  async def init_bluetooth():
9      ble = ubluetooth.BLE()
10     uart = BLEUART(ble, "ESP32_Fan_LED_Controller")
11
12     def on_rx():
13         try:
14             data = uart.read().decode("utf-8").strip()
15             print("Received data:", data)
16             if data.startswith("FAN:"):
17                 try:
18                     speed = int(data.split(":")[1])
19                     set_fan_speed(speed)
20                     uart.write("FAN speed set to " + str(speed))
21                 except ValueError:
22                     uart.write("Invalid speed value")
23             elif data == "LED:ON":
24                 control_led(True)
25                 uart.write("LED turned ON")
26             elif data == "LED:OFF":
27                 control_led(False)
28                 uart.write("LED turned OFF")
29             else:
30                 uart.write("Unknown command")
31         except Exception as e:
32             print("Error processing data:", e)
33             uart.write("Error processing data")
34
35     uart irq(handler=on_rx)
36

```

Ilustración 21. Código bluetooth.py

### Importación de Bibliotecas y Módulos:

- ubluetooth: Biblioteca para manejar la funcionalidad Bluetooth en MicroPython.
- BLEUART: Clase personalizada importada desde BLE\_uart\_peripheral que maneja la comunicación UART sobre BLE.
- set\_fan\_speed y control\_led: Funciones importadas desde otros módulos para controlar el ventilador y el LED, respectivamente.
- uasyncio: Biblioteca para manejar programación asíncrona en MicroPython.

### Función Asíncrona para Inicializar Bluetooth:

- ubluetooth.BLE(): Inicializa el objeto BLE.

- BLEUART(ble, "ESP32\_Fan\_LED\_Controller"): Crea un objeto UART sobre BLE, con el nombre del dispositivo ESP32\_Fan\_LED\_Controller.
- Definición del Manejador para Datos Recibidos (on\_rx).
- Lee los datos recibidos, los decodifica de bytes a cadena UTF-8 y elimina espacios en blanco alrededor.

#### **Procesamiento de Comandos Recibidos:**

- **Comando para el Ventilador (FAN:):**
  - Divide el comando para obtener la velocidad (speed) y la convierte a entero.
  - Llama a set\_fan\_speed(speed) para ajustar la velocidad del ventilador.
  - Envía una respuesta a través de BLE UART confirmando el ajuste de la velocidad.
  - Si la conversión a entero falla, envía un mensaje de error.
- **Comandos para el LED (LED:ON y LED:OFF):**
  - LED:ON: Llama a control\_led(True) para encender el LED y envía una respuesta confirmando.
  - LED:OFF: Llama a control\_led(False) para apagar el LED y envía una respuesta confirmando.
- **Comando Desconocido:**
  - Envía una respuesta indicando que el comando es desconocido.
- **Captura cualquier excepción durante el procesamiento y envía un mensaje de error.**

#### **Asignación del Manejador de Interrupciones (IRQ):**

- Asigna **on\_rx** como el manejador de interrupciones para datos recibidos en UART sobre BLE. Esto asegura que **on\_rx** se ejecutará cada vez que lleguen datos.

## BLE\_uart\_peripheral.py

```

1  import ubluetooth
2  from machine import Pin
3  from ubluetooth import BLE, UUID, FLAG_NOTIFY, FLAG_WRITE, FLAG_WRITE_NO_RESPONSE
4  from micropython import const
5
6  _IRQ_CENTRAL_CONNECT = const(1)
7  _IRQ_CENTRAL_DISCONNECT = const(2)
8  _IRQ_GATTS_WRITE = const(3)
9
10
11 class BLEUART:
12     def __init__(self, ble, name="ESP32_UART"):
13         self._ble = ble
14         self._ble.active(True)
15         self._ble.irq(self._irq)
16
17         self._name = name
18         self._rx_buffer = bytearray()
19         self._conn_handle = None
20         self._handler = None
21
22         # Definir UUIDs
23         self._service_uuid = UUID(
24             "6E400001-B5A3-F393-E0A9-E50E24DCCA9E"
25         ) # UUID del servicio UART
26         self._tx_uuid = UUID(
27             "6E400003-B5A3-F393-E0A9-E50E24DCCA9E"
28         ) # UUID de la característica TX
29         self._rx_uuid = UUID(
30             "6E400002-B5A3-F393-E0A9-E50E24DCCA9E"
31         ) # UUID de la característica RX
32
33         # Configurar las características
34         self._tx = (self._tx_uuid, FLAG_NOTIFY)
35         self._rx = (self._rx_uuid, FLAG_WRITE | FLAG_WRITE_NO_RESPONSE)
36

```

### Importaciones y Configuraciones Iniciales:

- Importa las bibliotecas necesarias para manejar Bluetooth (**ubluetooth**) y pines (**Pin**).
- Define constantes para los eventos de conexión y escritura GATT, utilizando **const** de MicroPython para eficiencia.

### Clase BLEUART:

- **Constructor (`__init__`):**
  - Inicializa la interfaz Bluetooth (**self.\_ble**) y la activa.
  - Configura el manejador de interrupciones (**irq**) para manejar eventos de Bluetooth.

```

37     # Registrar el servicio UART
38     self._service = (self._service_uuid, (self._tx, self._rx))
39     ((self._tx_handle, self._rx_handle),) = self._ble.gatts_register_services(
40         (self._service,)
41     )
42
43     self._advertise()
44
45     def _irq(self, event, data):
46         if event == _IRQ_CENTRAL_CONNECT:
47             self._conn_handle, _, _ = data
48             print("Central connected")
49         elif event == _IRQ_CENTRAL_DISCONNECT:
50             self._conn_handle = None
51             print("Central disconnected")
52             self._advertise()
53         elif event == _IRQ_GATTS_WRITE:
54             conn_handle, value_handle = data
55             if value_handle == self._rx_handle:
56                 self._rx_buffer += self._ble.gatts_read(value_handle)
57                 self._on_rx()
58
59     def _advertise(self):
60         try:
61             self._ble.gap_advertise(
62                 100,
63                 b"\x02\x01\x06"
64                 + bytearray((len(self._name) + 1, 0x09))
65                 + self._name.encode("UTF-8"),
66             )
67             print("Advertising as", self._name)
68         except OSError as e:
69             print("Advertising failed with OSError:", e)
70             # Reintentar después de un breve retraso
71             time.sleep(1)
72             self._advertise()

```

```

74     def irq(self, handler):
75         self._handler = handler
76
77     def _on_rx(self):
78         if self._handler:
79             self._handler()
80
81     def read(self):
82         result = self._rx_buffer
83         self._rx_buffer = bytearray()
84         return result
85
86     def write(self, data):
87         if self._conn_handle is not None:
88             try:
89                 self._ble.gatts_notify(self._conn_handle, self._tx_handle, data)
90             except OSError as e:
91                 print("Failed to send data:", e)
92         else:
93             print("No connection handle, cannot send data")
94

```

Ilustración 22. Código Clase base de Bluetooth

- Define el nombre del dispositivo, un búfer para datos recibidos (**\_rx\_buffer**), el identificador de conexión (**\_conn\_handle**), y un manejador de eventos (**\_handler**).
- Define los UUIDs del servicio y características UART (TX y RX).
- Registra el servicio y las características UART en el stack Bluetooth.
- Llama a **\_advertise** para comenzar a anunciar el dispositivo.

#### **Manejador de Interrupciones (**\_irq**):**

- **Conexión Central (**\_IRQ\_CENTRAL\_CONNECT**):**
  - Almacena el identificador de la conexión central y notifica que una central se ha conectado.
- **Desconexión Central (**\_IRQ\_CENTRAL\_DISCONNECT**):**
  - Limpia el identificador de conexión y vuelve a anunciar el dispositivo.
- **Escritura GATT (**\_IRQ\_GATTS\_WRITE**):**
  - Lee los datos escritos en la característica RX, los almacena en el búfer y llama al manejador de recepción de datos (**\_on\_rx**).

#### **Publicidad del Dispositivo (**\_advertise**):**

- Intenta anunciar el dispositivo con un intervalo de 100 ms y los datos de nombre.
- Si falla, reintenta después de un breve retraso.

#### **Asignación del Manejador de Interrupciones Externas (**irq**):**

- Asigna un manejador de interrupciones proporcionado externamente para manejar eventos de recepción de datos.

#### **Recepción de Datos (**\_on\_rx**):**

- Llama al manejador de eventos asignado, si existe, cuando se reciben datos.

#### **Lectura de Datos (**read**):**

- Devuelve los datos recibidos y vacía el búfer.

#### **Escritura de Datos (**write**):**

- Envía datos a la central conectada a través de la característica TX.
- Maneja errores si el envío falla, e indica si no hay una conexión activa.

## Web\_server.py

```

1  import asyncio as asyncio
2  import socket
3  import network
4  from machine import Pin, PWM
5  from config import FAN_PIN, FAN_PWM_FREQ, LED_PIN
6  from control_fan import set_fan_speed
7  from control_led import control_led
8
9  # Configurar pines
10 fan = PWM(Pin(FAN_PIN), freq=FAN_PWM_FREQ)
11 fan.duty(0)
12 led = Pin(LED_PIN, Pin.OUT)
13
14 # Cargar el contenido de index.html
15 with open("index.html", "r") as file:
16     INDEX_HTML = file.read()
17
18
19 def get_local_ip():
20     sta_if = network.WLAN(network.STA_IF)
21     if sta_if.isconnected():
22         return sta_if.ifconfig()[0]
23     else:
24         return None
25
26
27 def web_page():
28     led_state = "ON" if led.value() == 1 else "OFF"
29     return INDEX_HTML.replace(
30         '<strong id="ledState">OFF</strong>',
31         f'<strong id="ledState">{led_state}</strong>',
32     )
33
34
35 async def handle_client(client_sock):
36     try:
37         request = client_sock.recv(1024).decode()
38         print("Request:", request)
39
40         if "GET /?led=on" in request:
41             print("LED ON")
42             await control_led(True)
43         elif "GET /?led=off" in request:
44             print("LED OFF")
45             await control_led(False)
46         elif "GET /fan?" in request:
47             params = request.split("GET /fan?")[1].split(" ")[0]
48             if "=" in params:
49                 speed = int(params.split("=")[1])
50                 await set_fan_speed(speed)
51             else:
52                 print("Invalid fan parameters")
53
54         response = web_page()
55         client_sock.send("HTTP/1.1 200 OK\r\n")
56         client_sock.send("Content-Type: text/html\r\n")
57         client_sock.send("Connection: close\r\n\r\n")
58         client_sock.sendall(response.encode("utf-8"))
59     except Exception as e:
60         print("Error handling client:", e)
61     finally:
62         if client_sock:
63             client_sock.close()

```

```

66 async def start_server():
67     local_ip = get_local_ip()
68     if local_ip:
69         addr = socket.getaddrinfo(local_ip, 80)[0][-1]
70         s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
71         s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
72         s.bind(addr)
73         s.listen(5)
74         s.setblocking(False)
75         print(f"Server running at http://{local_ip}/")
76
77     while True:
78         try:
79             client_sock, client_addr = s.accept()
80             print("Got a connection from", client_addr)
81             asyncio.create_task(handle_client(client_sock))
82         except OSError as e:
83             if e.args[0] == 11: # EAGAIN
84                 await asyncio.sleep(0.1)
85             else:
86                 print("Error accepting connection:", e)
87                 await asyncio.sleep(1)
88     else:
89         print("Error: No se pudo obtener la dirección IP local.")
90

```

*Ilustración 23. Código para Servidor web*

### Importación de Bibliotecas:

- **uasyncio**: Biblioteca para programación asíncrona.
- **socket**: Permite la creación de sockets para la comunicación en red.
- **network**: Proporciona funciones para la configuración y manejo de conexiones de red.
- **Pin, PWM**: Bibliotecas para controlar los pines GPIO y generar señales PWM en el ESP32.
- **config**: Módulo que contiene la configuración de los pines del ventilador y del LED.
- **control\_fan, control\_led**: Módulos que contienen funciones para controlar el ventilador y el LED, respectivamente.

### Configuración de Pines:

- Se configuran los pines GPIO para el ventilador y el LED según la configuración proporcionada en el módulo **config**.
- Se inicializa el objeto PWM para controlar la velocidad del ventilador y se establece la señal de PWM en 0 para apagar el ventilador.

### Cargar el Contenido de index.html:



- Se carga el contenido del archivo HTML **index.html** en la variable **INDEX\_HTML**. Este archivo contiene la página web que se enviará al cliente cuando se realice una solicitud HTTP.

#### **Funciones Auxiliares:**

- **get\_local\_ip()**: Devuelve la dirección IP local del ESP32 si está conectado a una red Wi-Fi, o **None** si no lo está.
- **web\_page()**: Genera la página web que se enviará al cliente, mostrando el estado actual del LED.

#### **Manejo del Cliente:**

- **handle\_client(client\_sock)**: Función asíncrona que maneja las solicitudes de los clientes. Lee la solicitud HTTP del cliente, realiza acciones según la solicitud (encender o apagar el LED, ajustar la velocidad del ventilador), genera la página web de respuesta y la envía de vuelta al cliente.

#### **Iniciar el Servidor:**

- **start\_server()**: Función asíncrona que inicia el servidor web. Escucha las conexiones entrantes en el puerto 80 y maneja cada conexión en un bucle infinito, creando una tarea asíncrona para manejar cada cliente.
- Si no se puede obtener la dirección IP local, imprime un mensaje de error.

## Index.html

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title>ESP32 Fan and LED Controller</title>
5      <meta name="viewport" content="width=device-width, initial-scale=1">
6      <link rel="icon" href="data:,">
7      <style>
8          html {
9              font-family: Helvetica;
10             display: inline-block;
11             margin: 0px auto;
12             text-align: center;
13         }
14         h1 {
15             color: #0F3376;
16             padding: 2vh;
17         }
18         p {
19             font-size: 1.5rem;
20         }
21         .button {
22             display: inline-block;
23             background-color: #e7bd3b;
24             border: none;
25             border-radius: 4px;
26             color: white;
27             padding: 16px 40px;
28             text-decoration: none;
29             font-size: 30px;
30             margin: 2px;
31             cursor: pointer;
32         }
33         .button2 {
34             background-color: #4286f4;
35         }
36     </style>
37 </head>
38 <body>
39     <h1>ESP32 Fan and LED Controller</h1>
40     <h2>Ventilador</h2>
41     <input type="range" min="0" max="1023" value="0" id="fanSlider" oninput="updateFanSpeed(this.value)" />
42     <h2>LED</h2>
43     <p>LED state: <strong id="ledState">OFF</strong></p>
44     <p><button class="button" onclick="sendRequest('/?led=on')">ON</button></p>
45     <p><button class="button button2" onclick="sendRequest('/?led=off')">OFF</button></p>
46     <script>
47         function sendRequest(url) {
48             fetch(url)
49                 .then(response => {
50                     if (!response.ok) {
51                         throw new Error('Network response was not ok');
52                     }
53                     return response.text();
54                 })
55                 .then(data => {
56                     const ledState = data.match(/<strong id="ledState">(.*?)</strong>/)[1];
57                     document.getElementById('ledState').innerText = ledState;
58                 })
59                 .catch(error => {
60                     console.error('There has been a problem with your fetch operation:', error);
61                 });
62         }
63     </script>

```

```

64     function updateFanSpeed(value) {
65         fetch('/fan?speed=' + value)
66             .then(response => {
67                 if (!response.ok) {
68                     throw new Error('Network response was not ok');
69                 }
70                 document.getElementById('fanSpeed').innerText = value;
71             })
72             .catch(error => {
73                 console.error('There has been a problem with your fetch operation:', error);
74             });
75     }
76 </script>
77 </body>
78 </html>
79

```

*Ilustración 24. Vista y funciones del Servidor web*

**Document Type Declaration y Etiqueta HTML:** `<!DOCTYPE html>` declara la versión del documento HTML que sigue, y `<html>` indica que el contenido es un documento HTML.

**Etiqueta de Cabecera (`<head>`):** Aquí se encuentra la información del encabezado del documento, como el título de la página, la configuración de la vista y el estilo CSS.

- **Título:** Se establece el título de la página como "ESP32 Fan and LED Controller".
- **Meta:** Define la configuración de visualización inicial de la página, en este caso, establece el ancho de la página al ancho del dispositivo y escala inicialmente al 100%.
- **Ícono:** Establece un ícono vacío para la pestaña del navegador.

**Estilos CSS:** Se definen estilos CSS para dar formato a la página.

- **Fuente y Alineación:** Se utiliza la fuente Helvetica y se centra el contenido de la página.
- **Título:** Se define el color y el relleno del título (`<h1>`).
- **Párrafos:** Se establece el tamaño de la fuente de los párrafos (`<p>`).
- **Botones:** Se definen los estilos para los botones (`<button>`), especificando colores de fondo, bordes, padding, etc.

**Cuerpo (`<body>`):** Aquí se encuentra el contenido principal de la página.

- **Título:** Se muestra un título principal para la página.
- **Control de Ventilador:** Se muestra un deslizador (`<input type="range">`) para controlar la velocidad del ventilador.
- **Control de LED:** Se muestra el estado actual del LED y botones para encender y apagar el LED.

- **Script JavaScript:** Se define un script JavaScript para manejar las interacciones del usuario.

### Funciones JavaScript:

- **sendRequest(url):** Esta función se llama cuando se hace clic en un botón de encendido o apagado del LED. Realiza una solicitud fetch al servidor ESP32 utilizando la URL proporcionada y actualiza el estado del LED en la página.
- **updateFanSpeed(value):** Esta función se llama cuando se cambia el valor del deslizador del ventilador. Realiza una solicitud fetch al servidor ESP32 con la velocidad seleccionada y actualiza la velocidad del ventilador en la página.

## Implementación del Proyecto

```

117: 307C F8B0B
Boot script executed.
Conexión Wi-Fi establecida: ('192.168.137.185', '255.255.255.0', '192.168.137.1', '192.168.137.1')
Conectado a AWS IoT
Advertising as ESP32_Fan_LED_Controller
Bluetooth iniciado

Server running at http://192.168.137.185/
Datos publicados: {"temperature": 26, "device_id": "esp32_001", "humidity": 75}
Got a connection from ('192.168.137.1', 49299)
Got a connection from ('192.168.137.1', 49300)
Request:
Request:
Got a connection from ('192.168.137.1', 49308)
Got a connection from ('192.168.137.1', 49309)
Request:
Request: GET / HTTP/1.1

```

Ilustración 25. Salida de Terminal

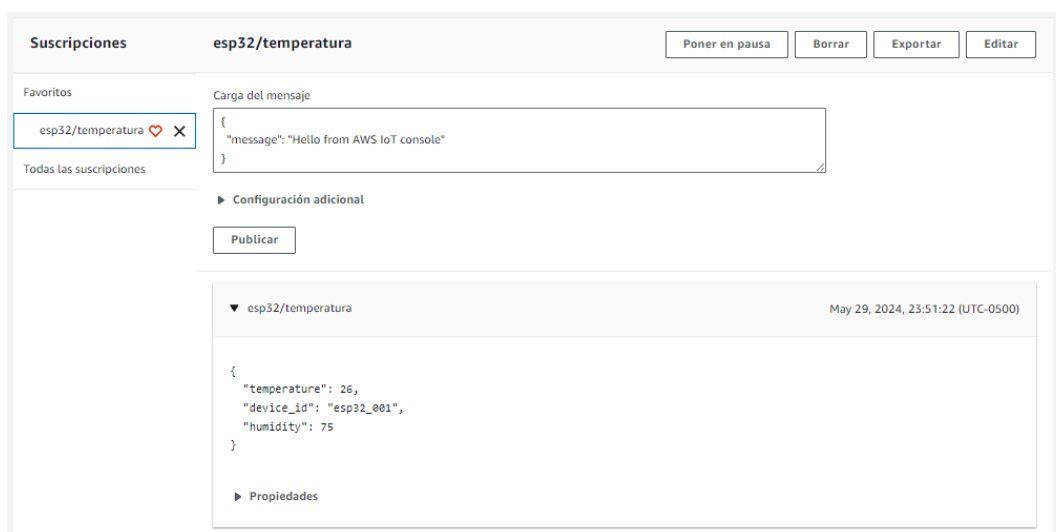


Ilustración 26. Cliente MQTT de prueba aws

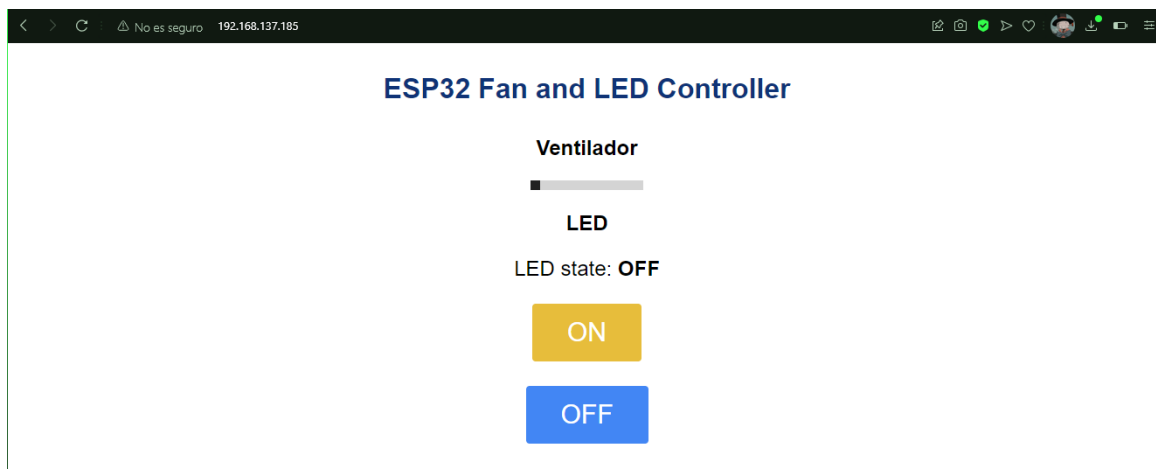


Ilustración 27. Página web Lanzada por servidor web ESP32

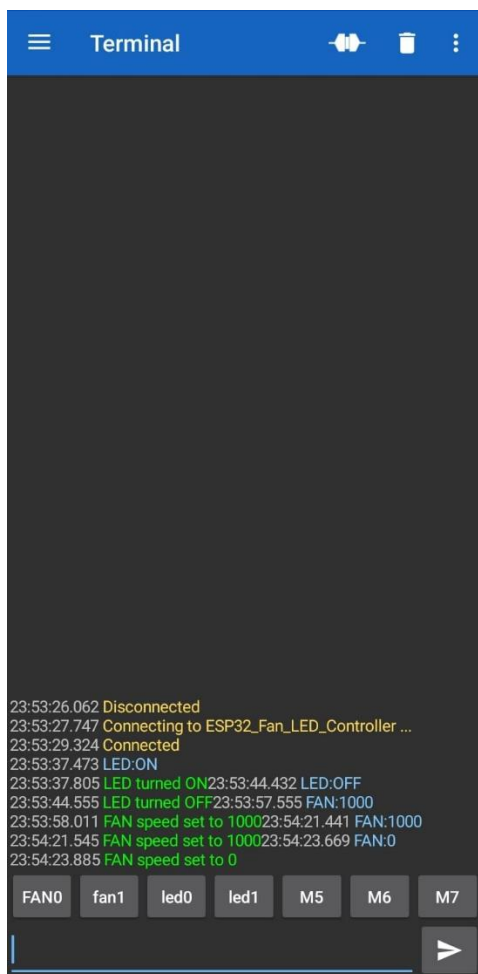


Ilustración 28. Terminal app bluetooth en Funcionamiento



*Ilustración 29. Sitio web con dominio para ver graficas temperatura vs tiempo y humedad vs tiempo*

## Cibergrafia

- <https://randomnerdtutorials.com/projects-esp32-esp8266-micropython/>
- <https://www.youtube.com/watch?v=vfAKG15olE0>
- <https://www.youtube.com/watch?v=idf-gGXvlu4>
- [https://www.youtube.com/watch?v=ynM3Y\\_dodyQ&t=503s](https://www.youtube.com/watch?v=ynM3Y_dodyQ&t=503s)
- <https://www.youtube.com/watch?v=tb-2102OzuI>
- <https://www.youtube.com/watch?v=r dyMHi0Wqpl&t=390s>
- <https://www.youtube.com/watch?v=kep8hVQMN6o>
- <https://www.youtube.com/watch?v=ZR3vgToAHbY>
- <https://www.youtube.com/watch?v=X30CJkK69so&t=368s>
- Documentación con uso de algunas inteligencias artificiales