Profesor: Gonzalo Vera

Explicación del código Implementado paso por paso

Este código es parte de una aplicación web escrita en Flask, un microframework de Python para crear aplicaciones web. Aquí, la aplicación está configurada para recibir y procesar datos desde un dispositivo ESP32 y autenticarse mediante JWT (JSON Web Tokens).

Explicación de cada parte:

1. **Ruta POST: Recibir datos desde el ESP32**

```
@app.route('/api/recibir-datos', methods=['POST'])
@jwt_required()  # Autenticación requerida para esta ruta
```

- **@app.route('/api/recibir-datos', methods=['POST']):** Esta línea define una ruta HTTP en la aplicación, es decir, cuando el servidor recibe una solicitud POST en la URL /api/recibir-datos, Flask ejecutará la función recibir_datos(). La ruta está configurada para aceptar solo solicitudes de tipo POST, generalmente utilizadas para enviar datos (en este caso, desde un ESP32).
- **@jwt_required():** Este decorador indica que la ruta requiere autenticación mediante JWT (JSON Web Tokens). El ESP32 u otro cliente debe enviar un token válido en el encabezado de la solicitud para acceder a esta ruta. Sin el token, se negará el acceso.

2. **Función: Recibir y validar los datos**

```
def recibir_datos():
    try:
        datos = request.get_json()
```

- **def recibir_datos():** Esta es la función que maneja la solicitud POST. Recibe los datos que el ESP32 envía a través de la solicitud.
- **datos = request.get_json():** Aquí, Flask usa request.get_json() para obtener los datos JSON enviados en el cuerpo de la solicitud. Se espera que los datos estén en formato JSON.

3. **Validación inicial de los datos**

```
if 'temperatura' not in datos or 'humedad' not in datos or 'presion' not in datos or 'cal
    return jsonify({"error": "Faltan datos"}), 400
```

if 'temperatura' not in datos or 'humedad' not in datos or 'presion' not in datos or 'calidad_aire' not in datos:

return jsonify({"error": "Faltan datos"}), 400

- **Validación de campos obligatorios:** Se verifica que el JSON recibido contenga las claves 'temperatura', 'humedad', 'presion', y 'calidad_aire'. Si alguno de estos campos falta, la función responde con un código de estado HTTP 400 (mala solicitud) y un mensaje indicando que faltan datos.

4. **Validaciones específicas de cada campo**

```
temperatura = datos['temperatura']
humedad = datos['humedad']
presion = datos['presion']
calidad_aire = datos['calidad_aire']
```

- **Asignación de valores:** Se extraen los valores de cada campo enviado en el JSON.
- **Validar temperatura:**

```
if not isinstance(temperatura, (int, float)) or not (-50 <= temperatura <= 60):
    return jsonify({"error": "Temperatura fuera de rango o no válida"}), 400
```

Se valida que la temperatura sea un número (entero o decimal) y esté en un rango plausible (de -50 a 60 grados Celsius). Si no es válido, se devuelve un error.

- **Validar humedad:**

```
if not isinstance(humedad, (int, float)) or not (0 <= humedad <= 100):
    return jsonify({"error": "Humedad fuera de rango o no válida"}), 400
```

Se valida que la humedad sea un número y esté en el rango de 0 a 100 (% de humedad relativa). Si no es válido, se responde con un error.

- **Validar presión:**

```
if not isinstance(presion, (int, float)) or not (300 <= presion <= 1100):
    return jsonify({"error": "Presión fuera de rango o no válida"}), 400
```

Se valida que la presión sea un número y esté dentro del rango plausible (300 a 1100 hPa, hectopascales). Si la presión no está en ese rango, también se devuelve un error.

- **Validar calidad del aire:**

```
if not isinstance(calidad_aire, (int, float)) or calidad_aire < 0:
    return jsonify({"error": "Calidad del aire no válida"}), 400
```

Se valida que la calidad del aire sea un número positivo. Si no es un valor válido, se retorna un error.

5. **Enviar los datos a una API remota**

```
response = requests.post(api_url, json=datos)
```

- **requests.post(api_url, json=datos):** Esta línea envía los datos validados a una API remota. Aquí se usa la librería requests para realizar una solicitud POST a la dirección api_url (que debes haber definido previamente en el código). Se envía el JSON completo a la API.

6. **Manejar la respuesta de la API remota**

```
if response.status_code == 200:
    return jsonify({"msg": "Datos enviados correctamente"}), 200
else:
    return jsonify({"error": "Error al enviar datos a la API"}), response.status_code
```

- **Verificación de la respuesta:** Se verifica el estado de la respuesta de la API remota. Si la API responde con un código HTTP 200 (éxito), se informa que los datos fueron enviados correctamente. Si hay algún error en la respuesta, se devuelve un mensaje de error con el código de estado correspondiente.

7. **Manejo de excepciones**

```
except Exception as e:
    return jsonify({"error": str(e)}), 500
```

- **Manejo de errores generales:** Si algo falla en el proceso (por ejemplo, un error de conexión o un problema inesperado), se captura la excepción y se devuelve un mensaje de error al cliente, junto con un código HTTP 500 (error interno del servidor).

---

8. **Ruta GET: Verificar el estado de los dispositivos**

```
@app.route('/api/estado', methods=['GET'])
@jwt_required()
def estado_dispositivo():
    return jsonify(datos_dispositivos), 200
```

- **Ruta GET:** Esta ruta está configurada para manejar solicitudes GET en la URL /api/estado. Se utiliza para verificar el estado de los dispositivos conectados. Al igual que la ruta POST, esta ruta también requiere autenticación JWT (@jwt_required()).
- **datos_dispositivos:** El código asume que hay una variable datos_dispositivos definida en alguna parte del código (probablemente un diccionario o lista) que contiene el estado actual de los dispositivos conectados.

9. **Ejecutar la aplicación**

```
if __name__ == '__main__':
    app.run(debug=True)
```

- **if __name__ == '__main__':** Esta es una convención de Python para asegurarse de que el código en este bloque solo se ejecute si el archivo es ejecutado directamente, no cuando es importado como un módulo en otro archivo.

- **app.run(debug=True):** Esta línea inicia el servidor Flask en modo debug, lo cual es útil para desarrollo, ya que permite ver los errores y realizar reinicios automáticos cuando se cambia el código.

Resumen:

- Este código define dos rutas en una aplicación Flask:
    - ✓ Una para recibir datos (POST) desde un ESP32, validar esos datos, y enviarlos a una API remota.
    - ✓ Otra para devolver el estado de los dispositivos conectados (GET).
- Se realizan varias validaciones en los datos recibidos para asegurarse de que sean correctos antes de procesarlos.
- Se utiliza autenticación JWT para proteger ambas rutas.