

Sistema de monitoreo y gestión remota del clima en invernaderos

Lic. Martín Anibal Lacheski

Carrera de Especialización en Internet de las Cosas

Director: Mg. Lic. Leopoldo Alfredo Zimperz (FIUBA)

Jurados:

Jurado 1 (pertenencia)
Jurado 2 (pertenencia)
Jurado 3 (pertenencia)

Ciudad de Montecarlo, Misiones, junio de 2025

Resumen

La presente memoria describe el desarrollo de un prototipo para monitorear y controlar de manera remota las condiciones climáticas en los invernaderos de la Facultad de Ciencias Forestales de la Universidad Nacional de Misiones. La solución propuesta integra sensores y actuadores conectados a un servidor remoto a través de una red inalámbrica y un protocolo de mensajería ligero, así como una aplicación web que permite la supervisión y el control a distancia del sistema.

Este trabajo permite optimizar el uso de recursos, mejorar la productividad y reducir costos operativos. Su implementación requirió conocimientos en sistemas embebidos, sensores, protocolos de comunicación, desarrollo de software, técnicas de seguridad y la implementación de soluciones cloud.

Agradecimientos

Esta sección es para agradecimientos personales y es totalmente **OPCIONAL**.

Índice general

Resumen	I
1. Introducción general	1
1.1. Problemática actual	1
1.2. Motivación	2
1.3. Estado del arte	2
1.4. Objetivos y alcance	3
1.4.1. Objetivo principal	3
1.4.2. Objetivos específicos	3
1.4.3. Alcance del trabajo	3
1.5. Requerimientos	4
2. Introducción específica	7
2.1. Protocolos de comunicación	7
2.1.1. Wi-Fi	7
2.1.2. MQTT	7
2.1.3. TLS	8
2.2. Componentes de hardware	8
2.2.1. Microcontrolador	8
2.2.2. Sensor de temperatura ambiente, humedad relativa y presión atmosférica	8
2.2.3. Sensor de luz digital	9
2.2.4. Sensor de dióxido de carbono	9
2.2.5. Sensor de detección de pH	9
2.2.6. Sensor de conductividad eléctrica	10
2.2.7. Sensor de sólidos disueltos totales	10
2.2.8. Sensor de temperatura digital sumergible	10
2.2.9. Sensor ultrasónico	11
2.2.10. Sensor de medición de consumo eléctrico	11
2.2.11. Módulo Relay	11
2.3. Desarrollo de firmware	12
2.3.1. MicroPython	12
2.4. Desarrollo backend y API	12
2.4.1. FastAPI	12
2.4.2. MongoDB	12
2.5. Desarrollo frontend	13
2.5.1. React	13
2.6. Infraestructura y despliegue	13
2.6.1. Docker	13
2.6.2. AWS IoT Core	13
2.6.3. AWS EC2	13
2.7. Herramientas de desarrollo	14

2.7.1. Visual Studio Code	14
2.7.2. Postman	14
2.7.3. GitHub	14
3. Diseño e implementación	15
3.1. Arquitectura del sistema	15
3.1.1. Capa de percepción	15
3.1.2. Capa de red	16
3.1.3. Capa de aplicación	16
3.2. Modelo de datos	16
3.2.1. Parametrización del sistema	17
3.2.2. Gestión de usuarios y roles	17
3.2.3. Sensores y actuadores	18
3.2.4. Auditoría y seguimiento	18
3.3. Servidor IoT	18
3.3.1. Arquitectura del servidor	18
3.4. Desarrollo del backend	19
3.4.1. Diseño de la API	19
3.4.2. Autenticación y autorización	20
3.4.3. Persistencia de datos	20
3.4.4. Comunicación con el broker MQTT	22
Pasos en AWS IoT Core	22
Implementación de MQTT en FastAPI	22
Comunicación con MQTT en FastAPI	23
3.4.5. Implementación de WebSockets	24
3.5. Desarrollo del frontend	25
3.6. Desarrollo de nodos sensores y actuadores	25
3.7. Despliegue del sistema	25
A. Modelo de datos implementado en el trabajo	27
B. Resumen de endpoints de la API	29
C. Autenticación con JWT	33
D. Conexión de la aplicación FastAPI con la base de datos MongoDB	35
E. Conexión al broker MQTT con FastAPI y AWS IoT SDK para Python	37
F. Conexión por WebSocket en FastAPI	43
F.1. Introducción	43
F.2. Autorización para WebSocket	43
F.3. Gestión de conexiones WebSocket	44
F.4. Integración con FastAPI	45
Bibliografía	47

Índice de figuras

1.1. Diagrama en bloques del sistema.	4
2.1. Microcontrolador ESP-WROOM-32 ¹	8
2.2. Sensor BME280 ²	9
2.3. Sensor BH1750 ³	9
2.4. Sensor MHZ19C ⁴	9
2.5. Sensor PH-4502C ⁵	10
2.6. Sensor CE ⁶	10
2.7. Sensor TDS ⁷	10
2.8. Sensor de temperatura DS18B20 ⁸	11
2.9. Sensor HC-SR04 ⁹	11
2.10. Sensor de medición de consumo eléctrico ¹⁰	11
2.11. Relay de 2 Canales 5 V 10 A ¹¹	12
3.1. Arquitectura de la solución propuesta.	15
3.2. Modelo de datos implementado.	17
3.3. Arquitectura del servidor del sistema IoT.	18
3.4. Esquema de autenticación y autorización.	20
3.5. Diagrama de clases de los modelos implementados.	21
3.6. Pasos para la conexión de FastAPI y MongoDB.	21
3.7. Pasos para verificar la conexión con el broker MQTT.	23
3.8. Pasos para publicar un mensaje en un tópico específico.	23
3.9. Pasos para la conexión del cliente MQTT.	24
3.10. Pasos para establecer la conexión WebSocket.	24
A.1. Modelo de datos implementado en el trabajo.	28

Índice de tablas

1.1. Características de la competencia.	2
3.1. Resumen de principales endpoints de la API	19
B.1. Resumen de principales endpoints de la API	29
B.2. Resumen de principales endpoints de la API	30
B.3. Resumen de principales endpoints de la API.	31

Dedicado a... [OPCIONAL]

Capítulo 1

Introducción general

Este capítulo presenta una visión general de los sistemas de gestión y monitoreo en invernaderos, se abordan los desafíos actuales y las oportunidades de mejora en el ámbito de la agricultura. Se describe la problemática relacionada con la falta de optimización en los sistemas de cultivo tradicionales. Además, se describen la motivación, los objetivos, el alcance y los requerimientos asociados a los diferentes componentes del sistema.

1.1. Problemática actual

La agricultura enfrenta desafíos crecientes en la optimización de la productividad y la eficiencia, especialmente en regiones con condiciones climáticas adversas y variables. Según la FAO (del inglés, *Food and Agriculture Organization of the United Nations*) [1], para el año 2050, se estima que la población superará los 9 mil millones de personas, lo que demandará un aumento del 60 % en la producción de alimentos. Para abordar este desafío, es fundamental optimizar el uso del agua, mejorar la productividad agrícola y fomentar prácticas que contribuyan a la sostenibilidad ambiental.

Ante estos retos, los cultivos hidropónicos han surgido como una solución prometedora debido a su capacidad para utilizar los recursos de manera más eficiente. Entre sus principales ventajas se destacan la reducción en el consumo de agua [2], la posibilidad de cultivar durante todo el año en entornos controlados y un aumento significativo en la productividad, gracias a la mayor velocidad de crecimiento y rendimiento de los cultivos.

En la provincia de Misiones, la producción hidropónica ha experimentado un crecimiento notable en los últimos años [3], [4]. No obstante, persisten desafíos en la gestión eficiente de los recursos esenciales. Actualmente, la mayoría de los productores emplean sistemas de control basados en temporizadores programables, los cuales no consideran las variaciones ambientales. Esto implica la necesidad de intervenciones manuales frecuentes y mediciones directas, limitando la eficiencia del proceso.

La ausencia de un monitoreo en tiempo real impacta negativamente en la calidad y el rendimiento de los cultivos, aumentando los costos operativos y afectando la sostenibilidad ambiental debido a la implementación de prácticas poco optimizadas.

1.2. Motivación

La motivación de este trabajo radica en el desarrollo e implementación de un sistema basado en IoT (del inglés, *Internet of Things*) y de bajo costo, que permite monitorear en tiempo real y controlar de manera remota los invernaderos de la Facultad de Ciencias Forestales (FCF) de la Universidad Nacional de Misiones (UNaM).

Este sistema posibilita el registro continuo de diversas variables de interés, como temperatura ambiente, humedad relativa, dióxido de carbono (CO_2), niveles de nutrientes, y consumo de agua y energía, entre otros. Los datos generados están disponibles para docentes, estudiantes e investigadores, para su uso en la realización de tesis, investigaciones y trabajos académicos.

Así, el trabajo no solo tiene un impacto directo en la producción, sino también en la formación académica y el avance científico. Proporciona una plataforma de datos para el análisis y el desarrollo de nuevas soluciones tecnológicas, alineadas con las demandas actuales de sostenibilidad ambiental y seguridad alimentaria [5].

1.3. Estado del arte

En el mercado actual, existen diversas empresas que ofrecen soluciones comerciales para optimizar la gestión de invernaderos. Estas herramientas permiten el control automatizado de variables clave como temperatura, humedad, ventilación y circulación de nutrientes o riego. La tabla 1.1 presenta una comparación de algunas de las soluciones disponibles y sus características más relevantes.

TABLA 1.1. Características de la competencia.

Empresa	Características
Hidroponía FIL [6]	Ofrece servicios en comodato de sensores y actuadores para monitorear y controlar en tiempo real variables críticas como temperatura ambiente, humedad relativa, conductividad eléctrica, pH, riego e iluminación.
Hidrosense [7]	Ofrece productos para automatizar la inyección de nutrientes en el sistema de riego a través del control del nivel de la conductividad eléctrica, la temperatura y el nivel de pH. Ofrece una plataforma para la visualización del estado, reportes y el envío de alertas.
iPONIA [8]	Ofrece productos y una plataforma para monitorear y controlar el invernadero hidropónico. Integra sensores para medir el nivel de pH, conductividad eléctrica, temperatura de la solución, temperatura ambiente y humedad relativa del aire. También ofrece dosificadores para inyectar los fertilizantes a la solución nutritiva.
Growcast [9]	Ofrece productos y una plataforma para controlar cultivos a través de sensores y actuadores que procesan y reportan datos en tiempo real. Integra sensores para medir temperatura ambiente, humedad relativa y CO_2 . Realiza el control del riego, la iluminación y la ventilación.

1.4. Objetivos y alcance

1.4.1. Objetivo principal

Diseñar y desarrollar un prototipo de sistema para el monitoreo y control remoto de las condiciones climáticas en invernaderos, mediante sensores y actuadores conectados a través de Wi-Fi, un servidor IoT en la nube y una aplicación web, con el fin de optimizar el uso de los recursos, reducir costos operativos y mejorar la sostenibilidad ambiental, además de servir como plataforma de datos para la investigación académica y científica.

1.4.2. Objetivos específicos

- Implementar una arquitectura IoT basada en Wi-Fi para monitorear sensores y actuadores en tiempo real.
- Desarrollar un servidor IoT en la nube para la recolección, almacenamiento y procesamiento de los datos obtenidos.
- Diseñar una aplicación web que permita la visualización en tiempo real y el control remoto de las condiciones del invernadero.
- Facilitar el acceso a los datos generados para su uso en investigaciones académicas, trabajos finales y estudios específicos.

1.4.3. Alcance del trabajo

El alcance del trabajo incluyó las siguientes tareas:

- Diseño e implementación de nodos IoT.
 - Selección de sensores, actuadores y microcontroladores.
 - Configuración de conexión Wi-Fi en nodos sensores y actuadores.
 - Desarrollo de firmware para la adquisición de datos de los sensores y el control de los actuadores.
- Comunicación y protocolos.
 - Configuración de un servidor IoT para gestión de mensajes entre nodos y aplicaciones.
 - Transmisión de datos al servidor IoT mediante MQTT (del inglés, *Message Queue Telemetry Transport*).
 - Cifrado de comunicaciones mediante TLS (del inglés, *Transport Layer Security*).
- Desarrollo de software.
 - Diseño e implementación de una base de datos para almacenar los datos recolectados por los sensores y permitir su consulta y análisis.
 - Diseño y desarrollo de una API (del inglés, *Application Programming Interface*) REST (del inglés, *Representational State Transfer*) que permita la comunicación con el sistema utilizando HTTP (del inglés, *Hypertext Transfer Protocol*), MQTT y WebSockets.

- Desarrollo de una aplicación web responsive para la visualización de datos en tiempo real y el control remoto de actuadores.
- Entregables.
 - Código fuente completo del sistema (sensores, actuadores, servidor IoT, API y aplicación web).
 - Guías de instalación, configuración y operación.

El trabajo no incluyó:

- Armado de PCB.
- Desarrollo de una aplicación móvil compatible con iOS y Android.

La figura 1.1 muestra el diagrama en bloques del sistema, que evidencia la integración de hardware, software y servicios en la nube.

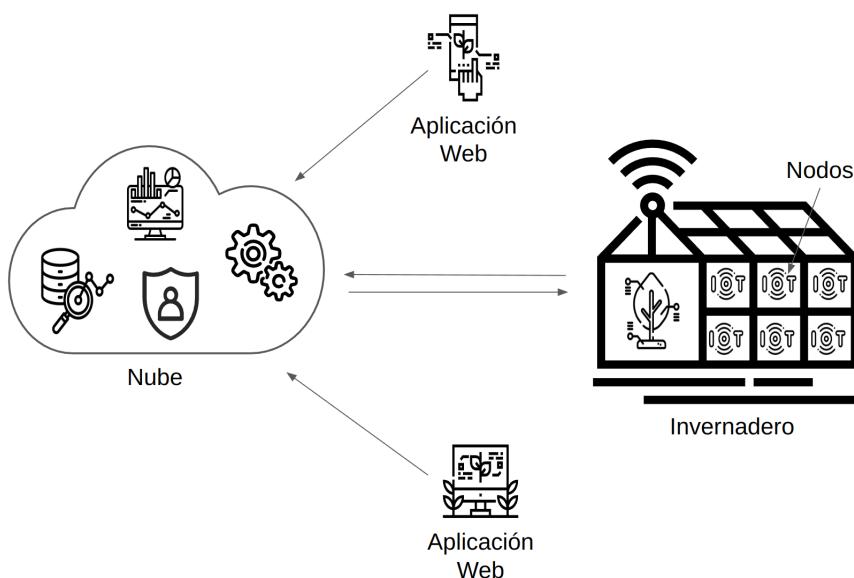


FIGURA 1.1. Diagrama en bloques del sistema.

1.5. Requerimientos

A continuación, se detallan los requerimientos técnicos asociados a los diferentes componentes del sistema.

1. Requerimientos de los nodos:
 - a) Utilizar microcontroladores basados en ESP32.
 - b) Implementar certificados TLS para seguridad en las comunicaciones.
 - c) Permitir conexión Wi-Fi.
 - d) Identificador único por nodo dentro del sistema.
 - e) Configuración remota del intervalo de envío de datos.
 - f) Los nodos sensores deben transmitir al servidor IoT;

- 1) Nodos ambientales: temperatura ambiente, humedad relativa, presión atmosférica, nivel de luminosidad y nivel de CO_2 .
- 2) Nodos de solución nutritiva: valores de pH (potencial de Hidrógeno), conductividad eléctrica (CE) y TDS (del inglés, *Total Dissolved Solids*); nivel y temperatura de la solución.
- 3) Nodos de consumos: agua, nutrientes y energía eléctrica.
- g) Los nodos actuadores deben transmitir al servidor IoT:
 - 1) Configuración remota de parámetros por cada canal.
 - 2) Reporte del estado de cada canal.
- h) Los nodos actuadores deben recibir desde el servidor IoT:
 - 1) Comandos de activación/desactivación remota de canales.
2. Broker MQTT:
 - a) Soportar conexiones cifradas mediante TLS.
 - b) Poseer comunicación bidireccional (publicación/suscripción).
 - c) Implementar QoS (del inglés, *Quality of Service*) para garantizar entrega de mensajes.
3. Frontend (aplicación web)
 - a) Interfaz intuitiva y responsive (accesible desde móviles y escritorio).
 - b) Autenticación de usuarios mediante credenciales.
 - c) Realización de las operaciones CRUD (Crear, Leer, Actualizar, Eliminar).
 - d) Visualización en tiempo real de datos de sensores y actuadores.
 - e) Envío remoto de comandos y configuraciones.
 - f) Acceso a datos históricos mediante gráficos y tablas.
 - g) Tablero interactivo para monitoreo y control centralizado.
4. Backend:
 - a) Tener conexiones seguras mediante TLS.
 - b) Implementar JWT (del inglés, *JSON Web Token*).
 - c) Realizar la persistencia de los datos.
 - d) Soportar métodos HTTP (CRUD y reportes), WebSockets (datos en tiempo real) y MQTT (interacción con dispositivos).
5. Requerimientos de documentación:
 - a) Se entregará el código del sistema, que incluye todos los componentes desarrollados (sensores, actuadores, broker MQTT, frontend, backend y API).
 - b) Se entregará las guías y diagramas de instalación, configuración y operación.

Capítulo 2

Introducción específica

En este capítulo se presentan los protocolos de comunicación, componentes de hardware y herramientas de software utilizados en el desarrollo del trabajo. Se detallan las características y sus especificaciones técnicas.

2.1. Protocolos de comunicación

En esta sección se describen los diferentes protocolos de comunicación utilizados en el desarrollo del trabajo.

2.1.1. Wi-Fi

Wi-Fi es el nombre comercial propiedad de la Wi-Fi Alliance para designar a su familia de protocolos de comunicación inalámbrica basados en el estándar IEEE 802.11 para redes de área local sin cables [10].

El estandar identifica dos modos principales de topología de red: infraestructura y ad-hoc.

- Modo infraestructura: los dispositivos se conectan a una red inalámbrica a través de un router o AP (del inglés, *Access Point*) inalámbrico, como en las WLAN. Los AP se conectan a la infraestructura de la red mediante el sistema de distribución conectado por cable o de manera inalámbrica.
- Modo ad-hoc: los dispositivos se conectan directamente entre sí sin necesidad de un punto de acceso.

2.1.2. MQTT

MQTT es un protocolo de mensajería estándar internacional OASIS [11] para Internet de las Cosas (IoT). Está diseñado como un transporte de mensajería de publicación/suscripción extremadamente ligero, ideal para conectar dispositivos remotos con un consumo de código reducido y un ancho de banda de red mínimo.

MQTT es un protocolo ligero basado en TCP/IP [12] que sigue un modelo de publicación/suscripción, donde:

- Broker: funciona como un servidor central que recibe los mensajes de los clientes y los distribuye a los suscriptores correspondientes, actúa como intermediario en la comunicación.

- Cliente: puede ser un dispositivo que publica mensajes en un tópico o que recibe mensajes al estar suscrito a un tópico.
- Tópico: es la dirección a la que se envían los mensajes en MQTT. El broker se encarga de distribuirlos a los clientes suscritos. Los temas se organizan en una estructura jerárquica de tópicos.

2.1.3. TLS

TLS es un protocolo de seguridad criptográfica diseñado para garantizar la privacidad y la integridad de los datos en comunicaciones sobre redes, como Internet [13]. Opera sobre la capa de transporte y permite autenticación, cifrado de datos y protección contra manipulación.

TLS se utiliza para garantizar la confidencialidad de los protocolos de aplicación (MQTT [11], HTTP [14] y WebSocket [15]) [16].

2.2. Componentes de hardware

En esta sección se describen los diferentes elementos de hardware utilizados en el desarrollo del trabajo.

2.2.1. Microcontrolador

El microcontrolador ESP-WROOM-32 (figura 2.1), es un chip de tipo SoC (del inglés, *System on Chip*) de bajo costo y bajo consumo de energía que integra WiFi, Bluetooth y Bluetooth LE en un solo paquete. El ESP-WROOM-32 [17] es un microcontrolador de 32 bits con una arquitectura Xtensa LX6 de doble núcleo, lo que le permite ejecutar dos hilos de ejecución simultáneos. Además, cuenta con una amplia gama de periféricos, como UART, I2C, SPI y ADC, que lo hace ideal para aplicaciones de IoT.



FIGURA 2.1. Microcontrolador ESP-WROOM-32¹.

2.2.2. Sensor de temperatura ambiente, humedad relativa y presión atmosférica

El BME280 (figura 2.2) es un sensor digital de alta precisión para la medición de temperatura ambiente, humedad relativa y presión atmosférica. Se comunica a través de las interfaces I2C y SPI y ofrece una precisión de $\pm 1^{\circ}\text{C}$ para la temperatura ambiente, $\pm 3\%$ para la humedad relativa y $\pm 1 \text{ hPa}$ para la presión atmosférica [18].

¹Imagen tomada de [Nodemcu Esp32 Wifi HobbyTronica](#).

FIGURA 2.2. Sensor BME280².

2.2.3. Sensor de luz digital

El BH1750 (figura 2.3) es un sensor digital de intensidad luminosa que mide la iluminación ambiental en lux. Utiliza la interfaz I2C para la comunicación y puede medir niveles de luz en un rango de 1 a 65.535 lux, con una precisión de 1 lux [19].

FIGURA 2.3. Sensor BH1750³.

2.2.4. Sensor de dióxido de carbono

El sensor MH-Z19C (figura 2.4) es un detector de CO_2 por NDIR (del inglés, *Non Dispersive Infrared Detector*). Se comunica a través de la interfaz UART y es capaz de medir la concentración de CO_2 en un rango de 0 a 5000 ppm con una precisión de 50 ppm [20].

FIGURA 2.4. Sensor MH-Z19C⁴.

2.2.5. Sensor de detección de pH

El sensor PH-4502C (figura 2.5) mide la acidez o alcalinidad del líquido mediante un electrodo de vidrio. Se comunica a través de la interfaz analógica y es capaz de medir el pH en un rango de 0 a 14 [21].

²Imagen tomada de [Sensor BME280 Naylamp Mechatronics](#).

³Imagen tomada de [Sensor BH1750 HobbyTronica](#).

⁴Imagen tomada de [MH-Z19C PartsCabi.net](#).



FIGURA 2.5. Sensor PH-4502C⁵.

2.2.6. Sensor de conductividad eléctrica

El sensor de CE (figura 2.6) mide la capacidad de una solución para conducir electricidad, lo que depende de la presencia de iones. A mayor concentración de iones, mayor es la conductividad [22]. Este sensor se comunica a través de una interfaz analógica y puede medir la conductividad en un rango de 0 a 20 mS/cm [23].



FIGURA 2.6. Sensor CE⁶.

2.2.7. Sensor de sólidos disueltos totales

El sensor TDS (figura 2.7) mide la cantidad de sales, minerales y metales que se encuentran disueltos en la solución [24]. Se comunica a través de la interfaz analógica y es capaz de medir la concentración de TDS en un rango de 0 a 1000 ppm [25].



FIGURA 2.7. Sensor TDS⁷.

2.2.8. Sensor de temperatura digital sumergible

El DS18B20 (figura 2.8) es un sensor digital de temperatura sumergible. Se comunica a través de la interfaz OneWire y puede medir la temperatura en un rango de -55 °C a 125 °C con una precisión de ±0.5 °C [26].

⁵Imagen tomada de [Sensor PH-4502C Mercado Libre Static](#).

⁶Imagen tomada de [Sensor CE Amazon](#).

⁷Imagen tomada de [Sensor TDS Aliexpress](#).



FIGURA 2.8. Sensor de temperatura DS18B20⁸.

2.2.9. Sensor ultrasónico

El sensor HC-SR04 (figura 2.9) mide distancias por ultrasonido en un rango de 2 cm a 400 cm con una precisión de 3 mm. Se comunica a través de la interfaz GPIO [27].



FIGURA 2.9. Sensor HC-SR04⁹.

2.2.10. Sensor de medición de consumo eléctrico

El sensor PZEM-004T (figura 2.10) es un módulo de medición de parámetros eléctricos que mide la tensión, corriente, potencia activa y energía consumida. Se comunica a través de la interfaz UART y es capaz de medir la tensión en un rango de 80 a 260 V, la corriente en un rango de 0 a 100 A, y la potencia en un rango de 0 a 22 kW [28].

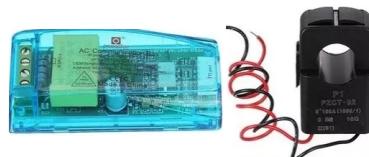


FIGURA 2.10. Sensor de medición de consumo eléctrico¹⁰.

2.2.11. Módulo Relay

El módulo Relay (figura 2.11) es un actuador eléctrico de dos canales optocoplados que permite el control de encendido y apagado de dispositivos eléctricos. Se comunica a través de la interfaz GPIO y es capaz de controlar dispositivos de hasta 10 A y 250 VAC [29].

⁸Imagen tomada de [Sensor DS18B20 Mercado Libre](#).

⁹Imagen tomada de [Sensor HC-SR04 Aliexpress](#).

¹⁰Imagen adaptada de [Sensor PZEM-004T Mercado Libre](#).



FIGURA 2.11. Relay de 2 Canales 5 V 10 A¹¹.

2.3. Desarrollo de firmware

En esta sección se describe la herramienta de software utilizada para la programación de los microcontroladores ESP32.

2.3.1. MicroPython

MicroPython es una implementación optimizada de Python 3 para microcontroladores y sistemas embebidos. Está diseñado para ejecutarse en dispositivos con recursos limitados, como el ESP32, y proporciona una forma sencilla de programar microcontroladores con un lenguaje de alto nivel como Python [30].

Su facilidad de uso, la amplia disponibilidad de bibliotecas y la reducción del tiempo de desarrollo lo convierten en una opción eficiente. Además, al ser un lenguaje interpretado, posibilita la ejecución interactiva de pruebas y depuración, facilitando la identificación y corrección de errores en el código [31].

2.4. Desarrollo backend y API

En esta sección se presentan las herramientas de software utilizadas en el desarrollo del backend y la API REST.

2.4.1. FastAPI

FastAPI es un framework moderno para la construcción de APIs REST rápidas y escalables en Python. Está diseñado para ser fácil de usar, rápido de desarrollar y altamente eficiente en términos de rendimiento. FastAPI utiliza Python 3.6+ y aprovecha las características de tipado estático de Python para proporcionar una API autodocumentada y con validación de tipos integrada [32].

2.4.2. MongoDB

MongoDB es una base de datos NoSQL (del inglés, *Not Only SQL*) de código abierto y orientada a documentos que proporciona una forma flexible y escalable de almacenar y recuperar datos. Utiliza un modelo de datos basado en documentos que almacena datos en un formato similar a JSON (del inglés, *JavaScript Object Notation*) llamado BSON (del inglés, *Binary JSON*) que permite almacenar datos de forma anidada y sin esquema fijo, lo que facilita la manipulación y consulta de datos no estructurados [33].

¹¹Imagen tomada de [Relay de 2 Canales Amazon](#).

2.5. Desarrollo frontend

En esta sección se presentan las herramientas de software utilizadas en el desarrollo del frontend.

2.5.1. React

React es una biblioteca de JavaScript de código abierto para construir interfaces de usuario interactivas y reutilizables. Desarrollada por Facebook, React permite crear componentes de interfaz de usuario que se actualizan de forma eficiente cuando cambian los datos, lo que facilita la creación de aplicaciones web rápidas y dinámicas [34].

2.6. Infraestructura y despliegue

En esta sección se presentan las herramientas de software utilizadas en la infraestructura y despliegue del sistema.

2.6.1. Docker

Docker es una plataforma de código abierto que permite a los desarrolladores y a los equipos de operaciones construir, empaquetar y desplegar aplicaciones en contenedores. Los contenedores son unidades de software ligeros y portátiles que incluyen todo lo necesario para ejecutar una aplicación, incluidas las bibliotecas, las dependencias y el código [35].

Docker facilita la creación de entornos de desarrollo y despliegue consistentes y reproducibles, lo que garantiza que las aplicaciones se ejecuten de la misma manera en cualquier entorno.

2.6.2. AWS IoT Core

AWS IoT Core es un servicio de AWS (del inglés, *Amazon Web Services*) que permite a los dispositivos conectarse de forma segura a la nube y comunicarse entre sí a través de protocolos de comunicación estándar como MQTT y HTTP. Proporciona una infraestructura escalable y segura para la gestión de dispositivos, la recopilación de datos y la integración con otros servicios de AWS [36]. Utiliza TLS para cifrar la comunicación entre los dispositivos y la nube, para garantizar la confidencialidad y la integridad de los datos.

2.6.3. AWS EC2

Amazon EC2 (del inglés, *Elastic Compute Cloud*) es un servicio de AWS que proporciona capacidad informática escalable en la nube. Permite a los usuarios lanzar instancias virtuales en la nube con diferentes configuraciones de CPU, memoria, almacenamiento y red, lo que facilita la implementación de aplicaciones escalables y de alta disponibilidad [37].

2.7. Herramientas de desarrollo

En esta sección se presentan las herramientas de software utilizadas en el desarrollo del sistema.

2.7.1. Visual Studio Code

Visual Studio Code, comúnmente abreviado como VS Code, es un entorno de desarrollo integrado (IDE, del inglés, *Integrated Development Environment*) de código abierto, altamente extensible y multiplataforma compatible con Windows, macOS y Linux [38].

VS Code es un editor de código ligero y rápido con soporte para muchos lenguajes de programación y extensiones que permiten personalizar y mejorar la funcionalidad del editor. Además, cuenta con herramientas de depuración integradas, control de versiones y terminal integrada.

2.7.2. Postman

Postman es una plataforma de colaboración para el desarrollo de APIs que permite a los desarrolladores diseñar, probar y documentar de forma rápida. Proporciona una interfaz gráfica intuitiva para enviar solicitudes HTTP a un servidor y visualizar las respuestas, lo que facilita la depuración y el desarrollo de APIs [39].

2.7.3. GitHub

GitHub es una plataforma de alojamiento de repositorios Git [40] que permite a los desarrolladores colaborar en proyectos de software de forma distribuida. Proporciona herramientas para gestionar el código fuente, realizar seguimiento de los cambios, revisar el código, realizar integración continua y despliegue automático [41].

Capítulo 3

Diseño e implementación

En este capítulo se describe el diseño y la implementación del sistema de monitoreo y control de invernaderos. Se detallan los componentes principales del sistema, las decisiones de diseño tomadas y los pasos seguidos para su implementación.

3.1. Arquitectura del sistema

La figura 3.1 ilustra la arquitectura general del sistema y la interacción entre los diferentes componentes.

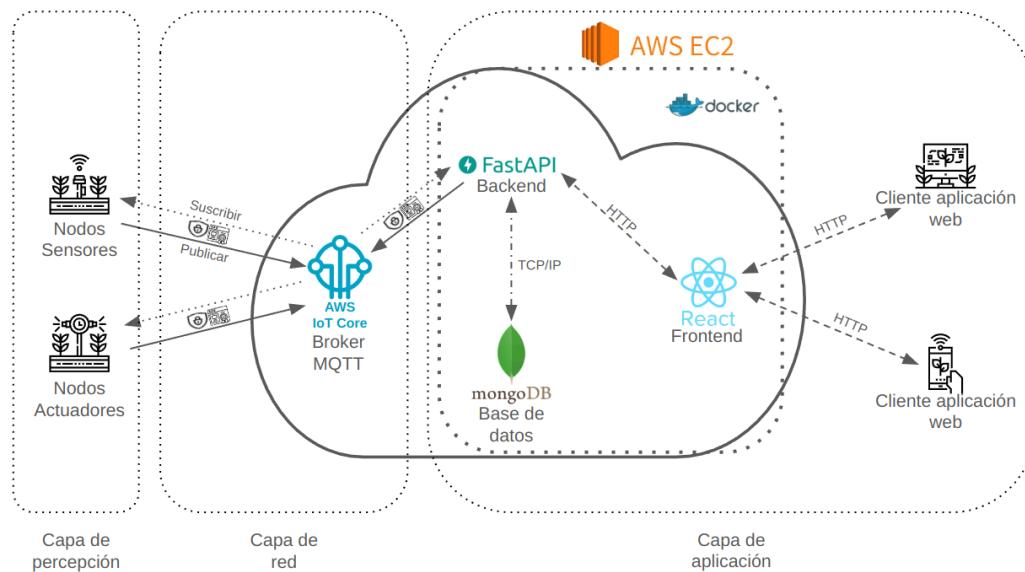


FIGURA 3.1. Arquitectura de la solución propuesta.

La arquitectura planteada para el desarrollo del trabajo sigue el modelo de tres capas típico de un sistema IoT: percepción, red y aplicación.

3.1.1. Capa de percepción

La capa de percepción está constituida por los nodos sensores y actuadores, que se encargan de recopilar datos del entorno y ejecutar acciones específicas en función de los parámetros configurados.

Cada nodo sensor incluye un microcontrolador ESP-WROOM-32, este se conecta a diversos sensores que miden parámetros como temperatura ambiente, humedad relativa, presión atmosférica, luminosidad, concentración de CO_2 , pH, conductividad eléctrica, temperatura de la solución nutritiva, nivel de líquidos, consumo eléctrico, entre otros. Los nodos actuadores, por su parte, cuentan con relés para controlar dispositivos como ventiladores, iluminación y sistemas de recirculación de nutrientes.

Los nodos están conectados a una red Wi-Fi local, lo que les permite establecer comunicación con la red y transmitir los datos de los sensores hacia el servidor IoT. La transmisión de datos se realiza con el protocolo MQTT.

3.1.2. Capa de red

La capa de red está compuesta por la infraestructura que gestiona la comunicación entre los nodos sensores y actuadores y la plataforma de backend. Los nodos sensores y actuadores se conectan a la red Wi-Fi local, lo que les permite acceder a internet y a la infraestructura de la nube. Una vez conectados, los dispositivos transmiten los datos a través del protocolo MQTT.

La comunicación entre los nodos y el broker MQTT se asegura mediante el uso de certificados de seguridad, los que garantizan la autenticación de los dispositivos y el cifrado de los datos.

El broker MQTT utilizado en este trabajo es AWS IoT Core, un servicio completamente gestionado que permite establecer una conexión segura y escalable entre los dispositivos IoT y la nube. Este broker actúa como intermediario para la transmisión de datos entre los nodos y la capa de aplicación.

3.1.3. Capa de aplicación

La capa de aplicación es responsable del procesamiento, almacenamiento y visualización de los datos recopilados por los nodos. Para esta capa, se implementó el servidor IoT en la nube utilizando el servicio AWS EC2, que permite ejecutar aplicaciones y servicios en instancias virtuales.

El procesamiento y la gestión de datos se realiza a través de un backend desarrollado con FastAPI, mientras que la base de datos MongoDB se utiliza para el almacenamiento de la información. Además, se implementó una interfaz gráfica de usuario en React para la visualización y gestión de los datos. Todos estos servicios fueron desplegados a través de contenedores Docker.

3.2. Modelo de datos

En esta sección se presenta el modelo de datos implementado en el sistema.

La figura 3.2 permite visualizar las principales colecciones y sus relaciones dentro de la base de datos. El diseño del modelo de datos se desarrolló en base a los tipos de datos proporcionados por los sensores, así como los requerimientos técnicos establecidos para el sistema.

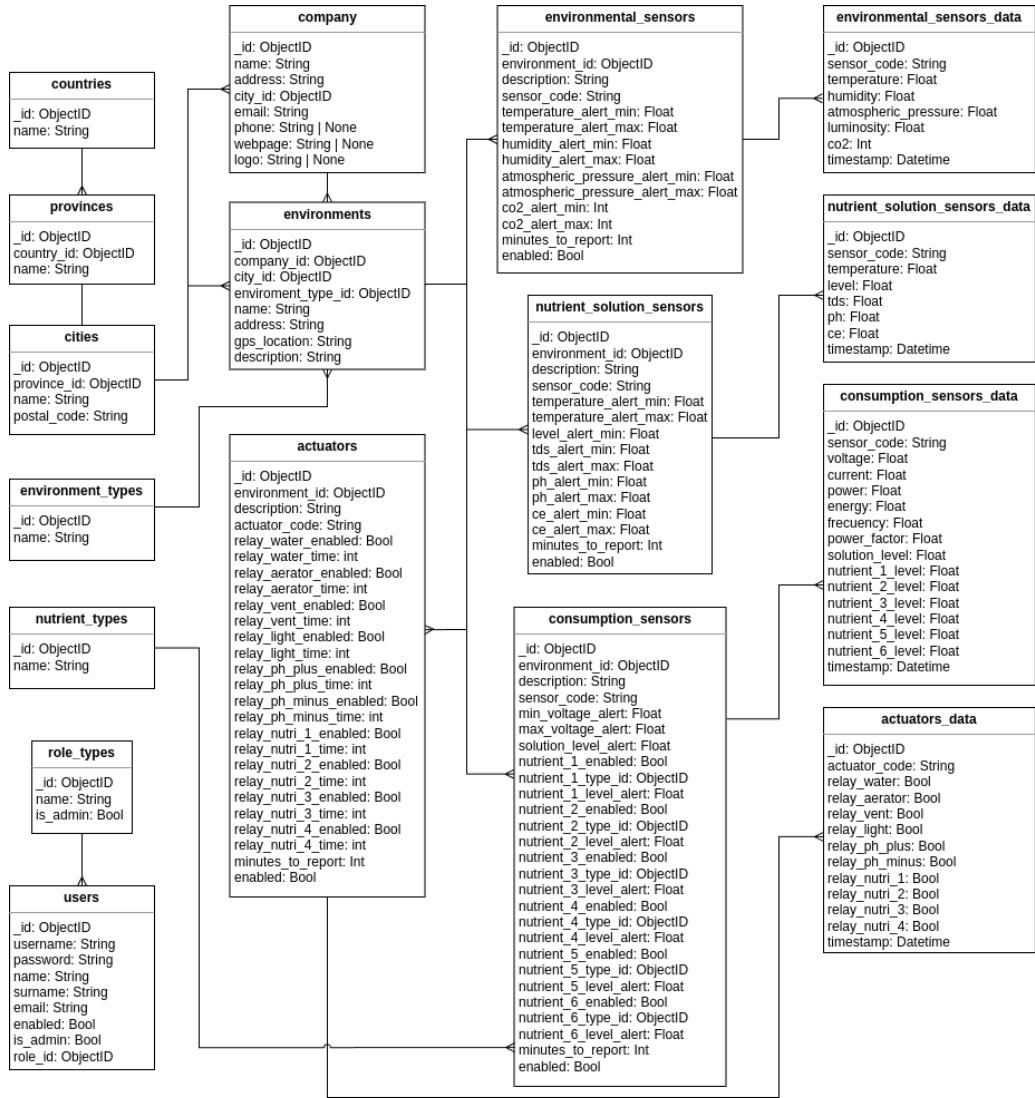


FIGURA 3.2. Modelo de datos implementado.

El modelo de datos se estructuró en colecciones dentro de MongoDB, organizadas en las siguientes categorías principales:

3.2.1. Parametrización del sistema

- Colecciones de configuración básica: países, provincias, ciudades, empresa.
- Colecciones para gestión de espacios: ambientes y tipos de ambientes.
- Colecciones específicas del dominio: tipos de nutrientes.

3.2.2. Gestión de usuarios y roles

- Colección de usuarios y roles: almacena información de los usuarios y sus credenciales.
- Colección de roles: se plantea como funcionalidad futura, para poder parametrizar permisos para cada tipo de rol.

3.2.3. Sensores y actuadores

- Colecciones de sensores y actuadores: almacena la información de cada tipo de dispositivo, los parámetros de alerta y la frecuencia de muestreo.
- Colecciones de datos históricos: registran las mediciones vinculadas a cada dispositivo mediante identificadores únicos.

3.2.4. Auditoría y seguimiento

- Colecciones de logs: permiten registrar los cambios en las configuraciones de sensores y actuadores realizados por los usuarios.

El modelo de datos completo del sistema, puede consultarse en el apéndice A.

3.3. Servidor IoT

En esta sección se presenta la arquitectura del sistema y se detallan las tecnologías utilizadas y la arquitectura del servidor.

3.3.1. Arquitectura del servidor

La arquitectura del servidor está compuesta por tres componentes principales: backend, capa de datos y frontend, como se muestra en la figura 3.3.

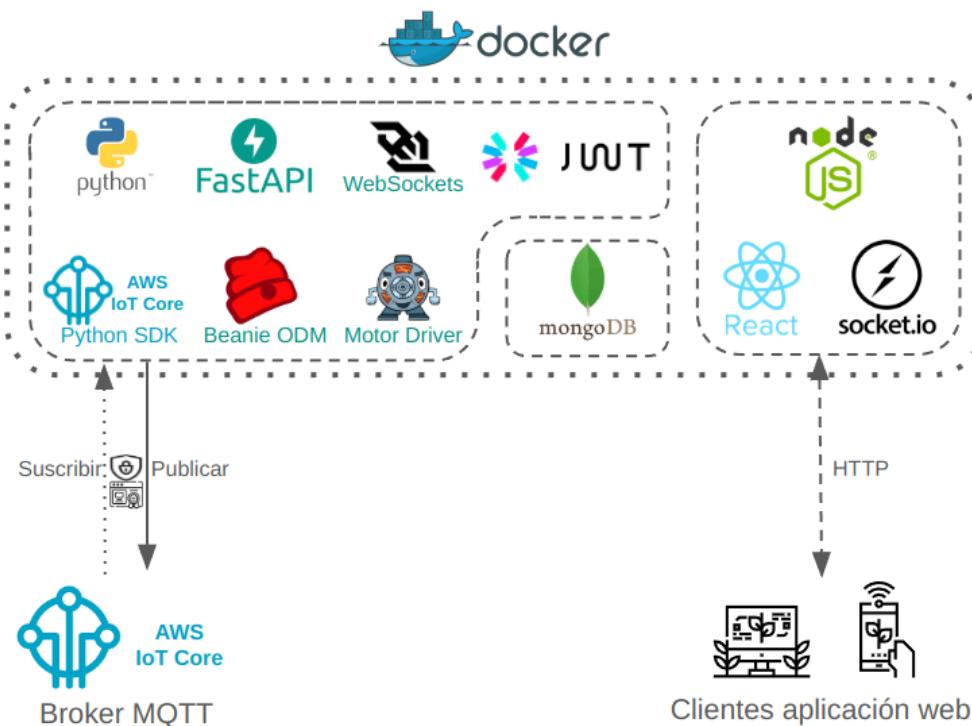


FIGURA 3.3. Arquitectura del servidor del sistema IoT.

A continuación, se describe brevemente cada uno de estos componentes:

- Backend: implementado con FastAPI, expone una API REST que permite gestionar sensores, actuadores, ambientes y usuarios. Incluye autenticación

y autorización basada en JWT, integración con MongoDB a través de Beanie [42] y Motor [43], conexión con el broker MQTT para la comunicación con los dispositivos IoT implementada con el SDK (del inglés, *Software Development Kit*) de AWS IoT para Python [44] y soporte para comunicaciones en tiempo real con clientes mediante WebSocket [45].

- Capa de datos: utiliza MongoDB como base de datos NoSQL. La información se organiza en colecciones que almacenan datos de sensores, actuadores, usuarios, ambientes y registros de configuración.
- Frontend: desarrollado en React, permite a los usuarios visualizar datos en tiempo real y configurar el sistema. Se conecta con la API REST del backend y utiliza WebSocket [46] para actualizaciones en tiempo real.

3.4. Desarrollo del backend

En esta sección se detallan los aspectos clave en el diseño y desarrollo del servidor backend, así como la lógica de negocio implementada.

3.4.1. Diseño de la API

El diseño se estructuró en base a las necesidades del sistema y los requerimientos funcionales y no funcionales establecidos. Se organizaron los archivos en carpetas de acuerdo a su funcionalidad.

La tabla 3.1 presenta un resumen de los principales endpoints de la API, junto con una breve descripción de la acción y el método HTTP utilizado.

TABLA 3.1. Resumen de principales endpoints de la API.

Método	Endpoint	Acción
GET	/mqtt/test	Test conexión cliente MQTT
POST	/mqtt/publish	Publicar en tópico MQTT
POST	/login	Login de usuarios
GET	/renew-token	Renovar token
GET	/users/	Obtener usuarios
GET	/environments/	Obtener ambientes
GET	/actuators/	Obtener actuadores
GET	/sensors/environmental/	Obtener sensores
GET	/sensors/nutrients/solution/	Obtener sensores
GET	/sensors/consumption/	Obtener sensores
GET	/actuators/data/	Obtener datos históricos
GET	/sensors/environmental/data/	Obtener datos históricos
GET	/sensors/consumption/data/	Obtener datos históricos
GET	/sensors/nutrients/solution/data/	Obtener datos históricos

El listado completo de endpoints de la API se puede consultar en el apéndice B.

3.4.2. Autenticación y autorización

Se implementó un sistema de autenticación basado en JWT, que permite a los usuarios acceder a la API de manera segura. La autenticación se realiza mediante el envío de las credenciales del usuario en el cuerpo de la solicitud, y el servidor responde con un token JWT que se utiliza para autenticar las solicitudes posteriores.

El token JWT contiene la información del usuario, este token se envía en el encabezado de las solicitudes a la API. El servidor verifica la validez del token y permite o deniega el acceso a los recursos solicitados. El token se diseña para que tuviera vencimiento, por lo que se implementó un sistema de renovación que permite a los usuarios mantener su sesión activa sin necesidad de autenticarse nuevamente con sus credenciales.

El código de la implementación de la autenticación y autorización se puede consultar en el apéndice C.

La figura 3.4 muestra el esquema de autenticación, autorización y renovación de tokens implementado en el sistema.

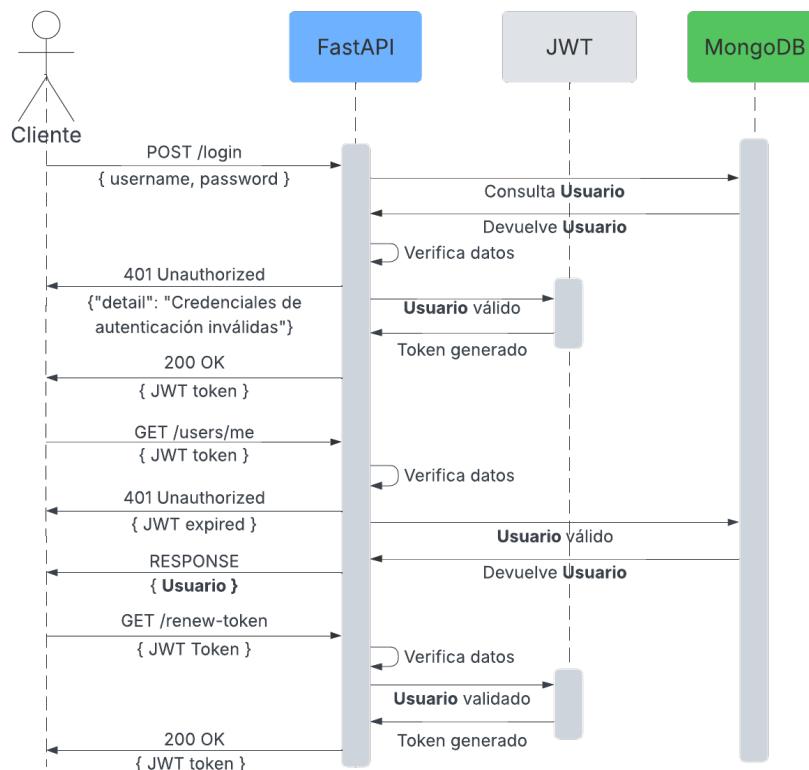


FIGURA 3.4. Esquema de autenticación y autorización.

3.4.3. Persistencia de datos

En FastAPI, cada modelo representa una colección en la base de datos e incluye los campos necesarios para almacenar la información requerida. La comunicación entre el backend y la base de datos se realizó a través de la biblioteca Motor, que proporciona una interfaz asíncrona para interactuar con MongoDB. Además se utilizó el ODM (del inglés, *Object Document Mapper*), a través de la biblioteca

Beanie, que permite definir modelos y realizar consultas y operaciones sobre la base de datos de manera sencilla.

La figura 3.5 muestra un ejemplo de la relación entre los modelos implementados en el sistema y los métodos HTTP de la colección EnvironmentalSensor.

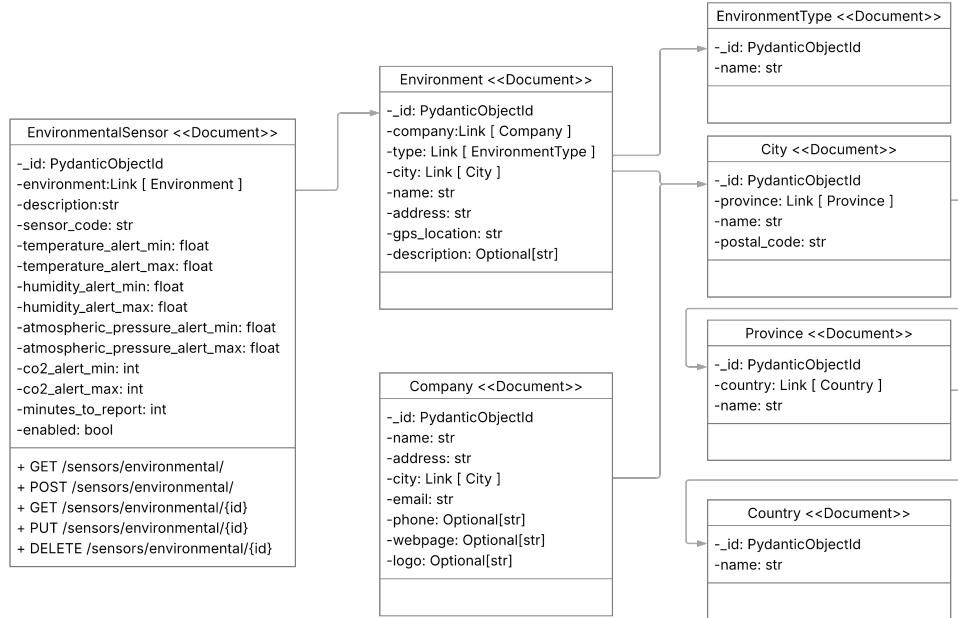


FIGURA 3.5. Diagrama de clases de los modelos implementados.

Como se mencionó anteriormente, los datos se almacenan en MongoDB. Para establecer la conexión con la base de datos, se utiliza el cliente asíncrono de la biblioteca Motor, mientras que la inicialización de los modelos se realiza mediante la función `init_beanie` del ODM Beanie. Esta función configura los modelos y establece la conexión con la base de datos. En la cadena de conexión se especifica el nombre de usuario, la contraseña y la dirección del servidor de MongoDB.

La figura 3.6 ilustra los pasos necesarios para establecer la conexión entre FastAPI y MongoDB.

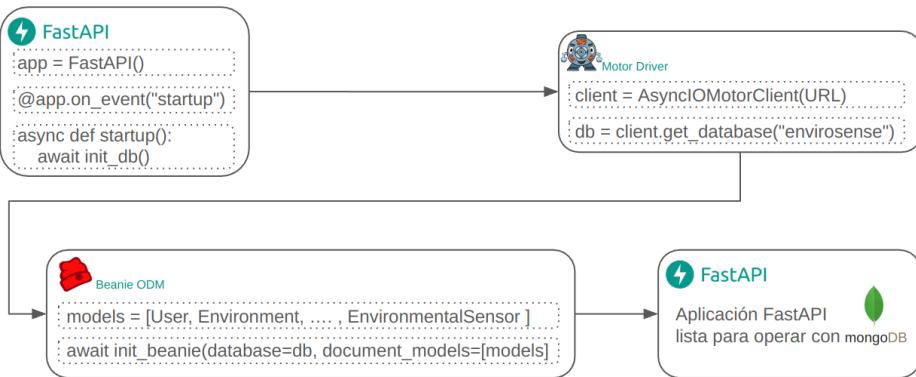


FIGURA 3.6. Pasos para la conexión de FastAPI y MongoDB.

El código para establecer la conexión y la inicialización de los modelos se puede consultar en el apéndice D.

3.4.4. Comunicación con el broker MQTT

A continuación, se describe la implementación de la comunicación con el broker MQTT.

Pasos en AWS IoT Core

Este apartado detalla los pasos realizados en AWS IoT Core para crear y configurar el objeto *Thing* y los certificados de seguridad necesarios para establecer la conexión con el broker MQTT.

1. Se creó un objeto *Thing* en AWS IoT Core, que representa un dispositivo IoT. Este objeto se utiliza para gestionar la conexión y la comunicación con el broker.
2. Se generaron certificados de seguridad y claves privadas para el objeto *Thing*, y se descargó el certificado raíz de Amazon. Estos elementos son necesarios para autenticar la conexión, garantizar el cifrado de los datos transmitidos y establecer una comunicación segura con el broker MQTT.
3. Se definieron las políticas de acceso necesarias para el objeto *Thing*, a fin de permitir publicar y suscribirse a los tópicos correspondientes.

Políticas de acceso: determinan los permisos del cliente para publicar, suscribirse y recibir mensajes en los tópicos. Se aplican a los certificados generados para el cliente y controlan el acceso a los recursos de AWS IoT Core.

Definidas en formato JSON, estas políticas garantizan que solo los clientes autorizados puedan interactuar con los recursos y operar en los tópicos correspondientes. Además, pueden configurarse de manera específica para cada *Thing*, lo que permite ejercer un control granular sobre los permisos de acceso.

En el Apéndice E se puede visualizar un ejemplo de política de acceso definida para el objeto *Thing*.

Implementación de MQTT en FastAPI

Una vez que se creó y se configuró el objeto *Thing* en AWS IoT Core, se procedió a la implementación de la conexión del broker con FastAPI. Para ello, se utilizó la SDK de AWS IoT para Python, que proporciona una interfaz sencilla para conectarse al broker y gestionar la comunicación con los dispositivos IoT.

Se implementó un cliente MQTT que se conecta al broker con los certificados generados previamente. Este cliente permite publicar y suscribirse a los tópicos correspondientes, lo que facilitó la comunicación entre el servidor y los dispositivos IoT.

En la aplicación FastAPI se definieron dos rutas clave para la chequear la comunicación con AWS IoT Core:

1. Una ruta para verificar la conexión con el broker MQTT.
2. Una ruta para enviar mensajes a un tópico y comprobar la comunicación entre el servidor y el broker.

La figura 3.7 muestra los pasos realizados para verificar la conexión con el broker MQTT.

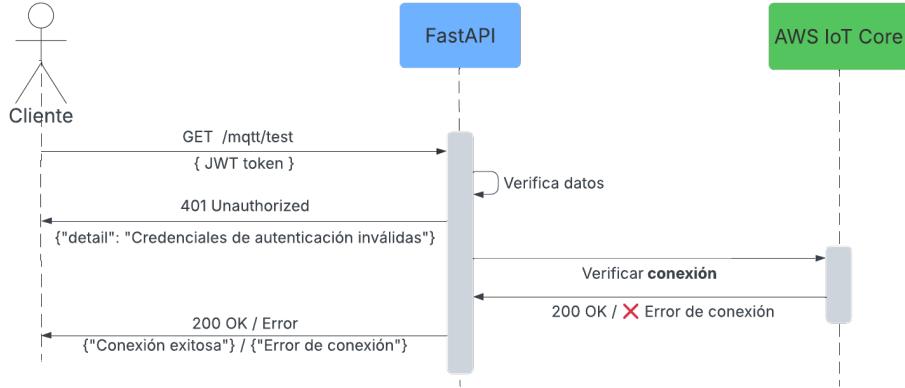


FIGURA 3.7. Pasos para verificar la conexión con el broker MQTT.

La figura 3.8 muestra los pasos realizados para publicar un mensaje en un tópico específico.

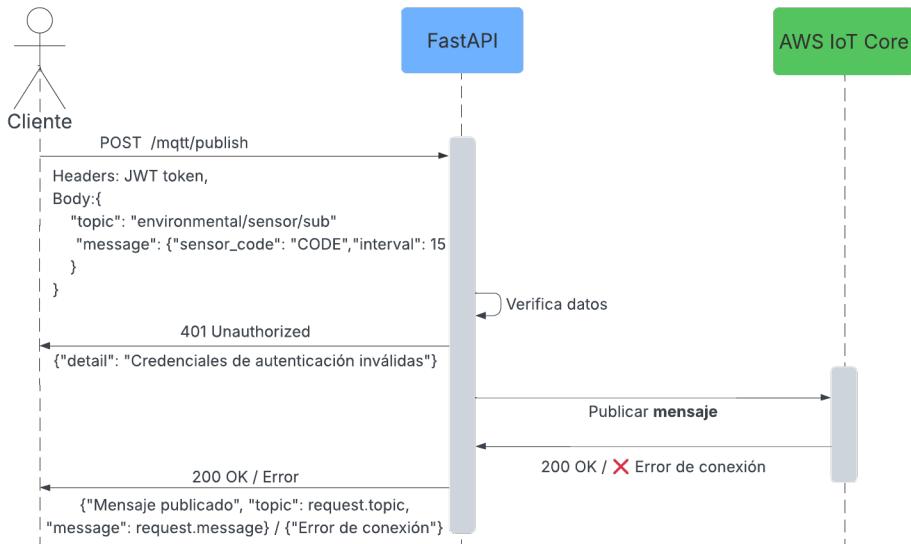


FIGURA 3.8. Pasos para publicar un mensaje en un tópico específico.

Comunicación con MQTT en FastAPI

Además de las rutas anteriores, en FastAPI se implementaron métodos para suscribirse a los tópicos, recibir mensajes de los nodos y publicar en los tópicos correspondientes.

Al iniciarse el servidor, el cliente MQTT establece la conexión con el broker y se suscribe a los tópicos definidos. Los mensajes recibidos son procesados, almacenados en MongoDB y enviados al frontend mediante WebSocket, lo que permite actualizar la interfaz en tiempo real. Este cliente, que maneja la conexión, publicación y suscripción, se inicializa junto con FastAPI para asegurar una comunicación eficiente.

La figura 3.9 muestra los pasos de cómo la aplicación FastAPI se conecta al broker MQTT y se suscribe a los tópicos solicitados.



FIGURA 3.9. Pasos para la conexión del cliente MQTT.

El código completo de la implementación de la conexión con el broker MQTT se encuentra en el Apéndice E.

3.4.5. Implementación de WebSockets

La comunicación en tiempo real se implementó mediante el módulo `websockets` de FastAPI, con una ruta específica para que los clientes se conecten y reciban datos actualizados de sensores y actuadores. La conexión permanece activa mientras sea válida, y se cierra en caso de error o falta de autorización.

El cliente debe enviar un token válido por `Authorization` o `Query Params`. Si es válido, el servidor acepta la conexión y la gestiona a través de la clase `WebSocketManager`.

Esta clase administra las conexiones activas, clientes y el envío de datos en tiempo real. Además, mantiene un caché con los últimos valores por tipo de sensor para optimizar el rendimiento. Al recibir nuevos datos, el servidor actualiza el caché y los transmite a todos los clientes conectados.

El código de la implementación se encuentra en el apéndice F, y la figura 3.10 ilustra el proceso de conexión.

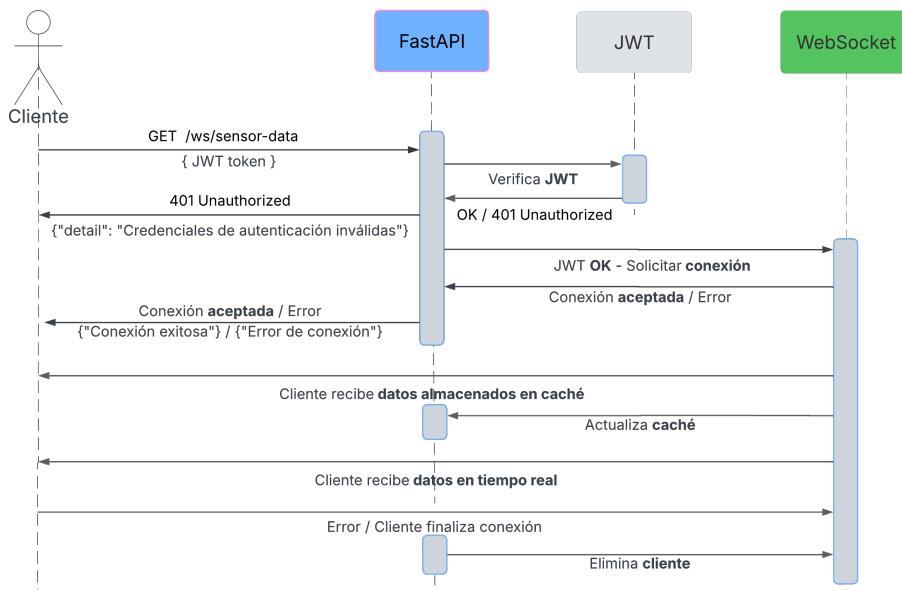


FIGURA 3.10. Pasos para establecer la conexión WebSocket.

3.5. Desarrollo del frontend

3.6. Desarrollo de nodos sensores y actuadores

3.7. Despliegue del sistema

Apéndice A

Modelo de datos implementado en el trabajo

La figura [A.1](#) muestra el modelo de datos implementado en el trabajo.

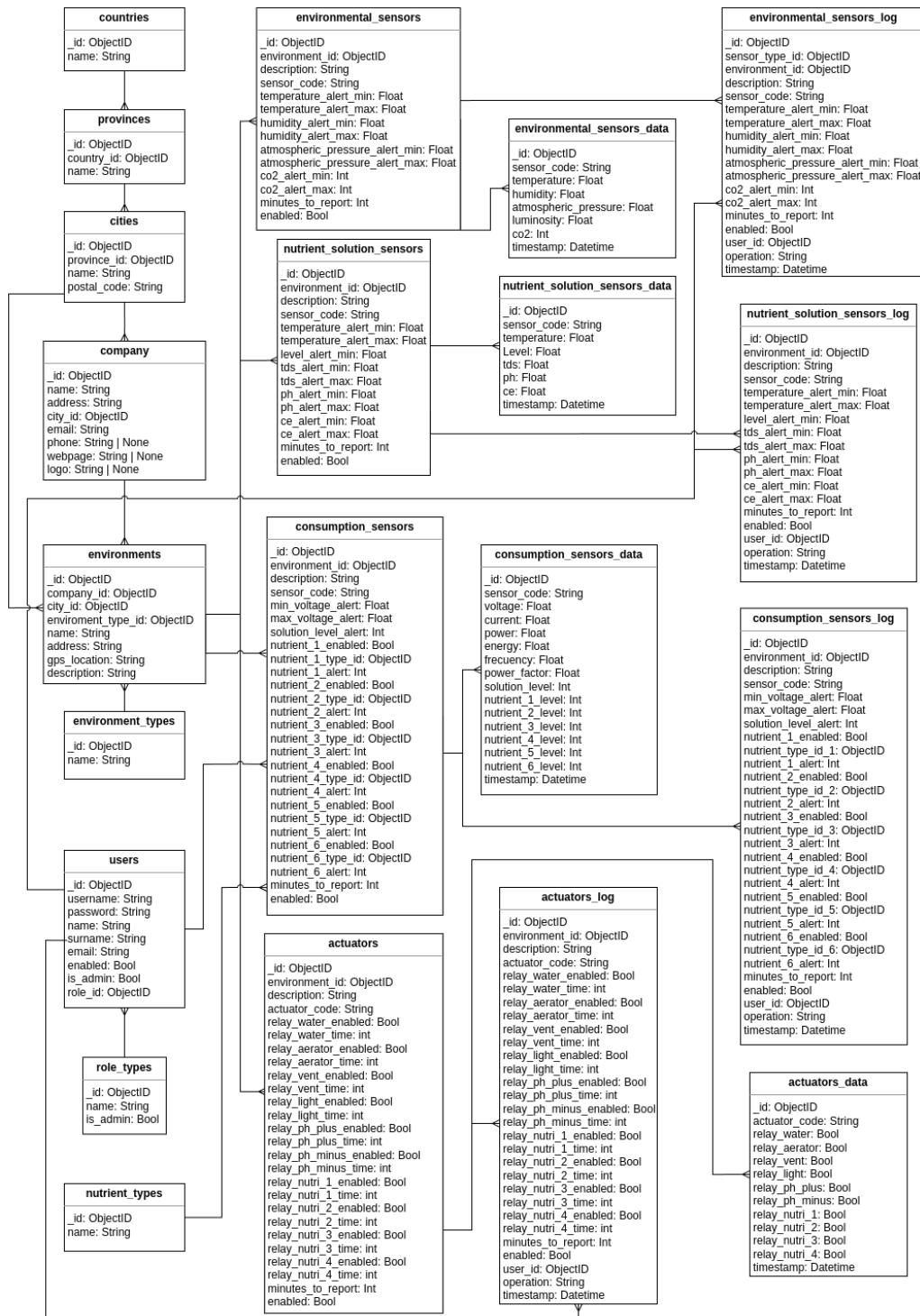


FIGURA A.1. Modelo de datos implementado en el trabajo.

Apéndice B

Resumen de endpoints de la API

Las siguientes tablas presentan un resumen de los endpoints implementados en la API, junto con una breve descripción de la acción y el método HTTP utilizado.

TABLA B.1. Resumen de principales endpoints de la API

Método	Endpoint	Acción
GET	/api/	Ruta por defecto
GET	/mqtt/test	Test conexión cliente MQTT
POST	/mqtt/publish	Publicar en tópico MQTT
POST	/login	Login de usuarios
GET	/renew-token	Renovar token
GET	/roles/	Obtener roles
POST	/roles/	Crear rol
GET	/roles/{id}	Obtener un rol
PUT	/roles/{id}	Actualizar rol
DELETE	/roles/{id}	Eliminar rol
GET	/users/	Obtener usuarios
POST	/users/	Crear usuario
PUT	/users/	Actualizar usuario
GET	/users/{id}	Obtener un usuario
DELETE	/users/{id}	Eliminar usuario
PATCH	/users/	Actualizar username
PATCH	/users/password	Actualizar password
GET	/users/me	Obtener un usuario
PATCH	/users/change/password	Actualizar 'username'
GET	/countries/	Obtener países
POST	/countries/	Crear país
GET	/countries/{id}	Obtener un país
PUT	/countries/{id}	Actualizar país
DELETE	/countries/{id}	Eliminar país
GET	/provinces/	Obtener provincias
POST	/provinces/	Crear provincia
GET	/provinces/{id}	Obtener una provincia
PUT	/provinces/{id}	Actualizar provincia
DELETE	/provinces/{id}	Eliminar provincia

TABLA B.2. Resumen de principales endpoints de la API

Método	Endpoint	Acción
GET	/cities/	Obtener ciudades
POST	/cities/	Crear ciudad
GET	/cities/{id}	Obtener una ciudad
PUT	/cities/{id}	Actualizar ciudad
DELETE	/cities/{id}	Eliminar ciudad
GET	/company/	Obtener empresas
POST	/company/	Crear empresa
GET	/company/{id}	Obtener una empresa
PUT	/company/{id}	Actualizar empresa
DELETE	/company/{id}	Eliminar empresa
POST	/company/uploadLogo	subir logo empresa
GET	/environments/types/	Obtener tipos de ambientes
POST	/environments/types/	Crear tipo de ambiente
GET	/environments/types/{id}	Obtener un tipo de ambiente
PUT	/environments/types/{id}	Actualizar tipo de ambiente
DELETE	/environments/types/{id}	Eliminar tipo de ambiente
GET	/environments/	Obtener ambientes
POST	/environments/	Crear ambiente
GET	/environments/{id}	Obtener un ambiente
PUT	/environments/{id}	Actualizar ambiente
DELETE	/environments/{id}	Eliminar ambiente
GET	/actuators/	Obtener actuadores
POST	/actuators/	Crear actuador
GET	/actuators/{id}	Obtener un actuador
PUT	/actuators/{id}	Actualizar actuador
DELETE	/actuators/{id}	Eliminar actuador
GET	/actuators/log/	Obtener logs de actuadores
POST	/actuators/log/	Crear log de actuador
GET	/actuators/data/	Obtener datos históricos
POST	/actuators/data/	Crear dato histórico
GET	/actuators/data/{id}	obtener un dato histórico
GET	/nutrients/types/	Obtener tipos de nutrientes
POST	/nutrients/types/	Crear tipo de nutriente
GET	/nutrients/types/{id}	Obtener un tipo de nutriente
PUT	/nutrients/types/{id}	Actualizar tipo de nutriente
DELETE	/nutrients/types/{id}	Eliminar tipo de nutriente
GET	/sensors/consumption/	Obtener sensores
POST	/sensors/consumption/	Crear sensor
GET	/sensors/consumption/{id}	Obtener un sensor
PUT	/sensors/consumption/{id}	Actualizar sensor
DELETE	/sensors/consumption/{id}	Eliminar sensor
GET	/sensors/consumption/log/	Obtener logs de sensor
POST	/sensors/consumption/log/	Crear log de sensor
GET	/sensors/consumption/data/	Obtener datos históricos
POST	/sensors/consumption/data/	Crear dato histórico
GET	/sensors/consumption/data/{id}	Obtener un dato histórico

TABLA B.3. Resumen de principales endpoints de la API.

Método	Endpoint	Acción
GET	/sensors/environmental/	Obtener sensores
POST	/sensors/environmental/	Crear sensor ambiental
GET	/sensors/environmental/{id}	Obtener un sensor
PUT	/sensors/environmental/{id}	Actualizar sensor
DELETE	/sensors/environmental/{id}	Eliminar sensor
GET	/sensors/environmental/log/	Obtener logs de sensor
POST	/sensors/environmental/log/	Crear log de sensor
GET	/sensors/environmental/data/	Obtener datos históricos
POST	/sensors/environmental/data/	Crear dato histórico
GET	/sensors/environmental/data/{id}	Obtener un dato histórico
GET	/sensors/nutrients/solution/	Obtener sensores
POST	/sensors/nutrients/solution/	Crear sensor
GET	/sensors/nutrients/solution/{id}	Obtener un sensor
PUT	/sensors/nutrients/solution/{id}	Actualizar sensor
DELETE	/sensors/nutrients/solution/{id}	Eliminar sensor
GET	/sensors/nutrients/solution/log/	Obtener logs de sensor
POST	/sensors/nutrients/solution/log/	Crear log de sensor
GET	/sensors/nutrients/solution/data/	Obtener datos históricos
POST	/sensors/nutrients/solution/data/	Crear dato histórico
GET	/sensors/nutrients/solution/data/{id}	Obtener un dato histórico

Apéndice C

Autenticación con JWT

El control de acceso se realiza mediante la verificación de un token JWT que se envía en las solicitudes. Si el token es válido, se permite el acceso a los recursos protegidos; de lo contrario, se devuelve un error de autenticación.

El código utiliza la librería `passlib` para el manejo de contraseñas y `bcrypt` para el cifrado. Además, se utiliza `fastapi.security` para manejar la autenticación y autorización.

El siguiente código es un ejemplo de cómo implementar el control de acceso en una API REST utilizando FastAPI y JWT.

```

1 from fastapi import APIRouter, Depends, HTTPException, status
2 from fastapi.security import OAuth2PasswordBearer
3 from passlib.context import CryptContext
4 import jwt
5 import bcrypt
6 from models.user import User
7 KEY = "colocar_clave_secreta_aquí"
8 ALG = "HS256"
9 ACCESS_TOKEN_EXPIRE_MINUTES = 30
10 oauth2 = OAuth2PasswordBearer(tokenUrl="login")
11 crypt = CryptContext(schemes=[ "bcrypt" ], deprecated="auto")
12 async def auth_user(token: str = Depends(oauth2)):
13     exception = HTTPException(
14         status_code=status.HTTP_401_UNAUTHORIZED,
15         detail="Credenciales de autenticacion inválidas",
16         headers={"WWW-Authenticate": "Bearer"},
17     )
18     try:
19         user = jwt.decode(token, KEY, algorithms=[ALG]).get("username")
20         if user is None:
21             raise exception
22     except:
23         raise exception
24     user = await User.find_one({ "username": username })
25     if not user:
26         raise exception
27     return user
28     async def current_user(user: User = Depends(auth_user)):
29         if not user.enabled:
30             raise HTTPException(
31                 status_code=status.HTTP_400_BAD_REQUEST, detail="Usuario
32 deshabilitado"
33         )
34         return user

```

CÓDIGO C.1. Pseudocódigo del control de acceso.

Apéndice D

Conexión de la aplicación FastAPI con la base de datos MongoDB

La aplicación FastAPI se conecta con la base de datos MongoDB a través de la biblioteca Motor y el ODM Beanie. La conexión a la base de datos MongoDB se establece con la URL de conexión, que incluye el nombre de usuario, la contraseña y la dirección del servidor MongoDB. La URL de conexión se define en la configuración de la aplicación FastAPI y se utiliza para crear una instancia de Motor y Beanie.

El código D.1 ilustra la conexión a la base de datos MongoDB con Motor y Beanie.

```

1   from fastapi import FastAPI
2   from pymongo import MongoClient
3   from beanie import init_beanie
4   from motor.motor_asyncio import AsyncIOMotorClient
5
6   # Método asíncrono para inicializar la conexión a la base de datos
7   async def init_db():
8       client = AsyncIOMotorClient("mongodb://USER:PASSWORD@URL:PORT/?"
9                                   authSource=admin")
10      db = client.get_database("envirosense")
11      await init_beanie(database=db, document_models=[
12          User, Role, Country, Province, City, Company,
13          EnvironmentType,
14          Environment, NutrientType, ConsumptionSensor,
15          ConsumptionSensorData, ConsumptionSensorLog,
16          EnvironmentalSensor, EnvironmentalSensorData,
17          EnvironmentalSensorLog, NutrientSolutionSensor,
18          NutrientSolutionSensorData, NutrientSolutionSensorLog,
19          Actuator, ActuatorData, ActuatorLog
20      ])
21
22      # Iniciar FastAPI
23      app = FastAPI()
24
25      # Al inicializar la aplicación FastAPI, se ejecuta la función
26      # startup
27      @app.on_event("startup")
28
29      # Método asíncrono que se ejecuta al iniciar la aplicación
30      async def startup():
31          # Se utiliza el método await para esperar a que la conexión se
32          # establezca antes de continuar con la ejecución de la aplicación.
33          await init_db()
```

CÓDIGO D.1. Cliente MQTT.

Apéndice E

Conexión al broker MQTT con FastAPI y AWS IoT SDK para Python

La conexión al broker MQTT se realiza para permitir la comunicación entre el servidor IoT y el broker MQTT.

El código E.1 muestra una política de acceso en AWS IoT Core que habilita al cliente a conectarse, publicar, suscribirse y recibir mensajes en cualquier tópico.

```

1  {
2      "Version": "2012-10-17",
3      "Statement": [
4          {
5              "Effect": "Allow",
6              "Action": [
7                  "iot:Connect",
8                  "iot:Publish",
9                  "iot:Subscribe",
10                 "iot:Receive"
11             ],
12             "Resource":
13                 "arn:aws:iot:*:*::*"
14         }
15     ]
16 }
```

CÓDIGO E.1. Ejemplo de política de acceso en AWS IoT Core.

El código E.2 muestra el proceso de conexión a AWS IoT Core y la implementación de la lógica de publicación y suscripción a los tópicos.

En este código, se definen los métodos para conectar al broker, publicar y suscribirse a tópicos, y manejar los mensajes recibidos. Se implementó un cliente MQTT que interactúa con AWS IoT Core, gestionando la comunicación con los nodos sensores y actuadores. Además, se incorporaron métodos para recibir datos de estos dispositivos, enviarles comandos y almacenar la información en la base de datos MongoDB.

```

1  import asyncio
2  import json
3  import os
```

```

4 # Importamos el cliente MQTT de AWS
5 from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTClient
6 # Importamos los modelos de datos
7 from models.actuator import Actuator
8 from models.actuator_data import ActuatorData
9 from models.sensor_consumption import ConsumptionSensor
10 from models.sensor_consumption_data import ConsumptionSensorData
11 from models.sensor_environmental import EnvironmentalSensor
12 from models.sensor_environmental_data import EnvironmentalSensorData
13 from models.sensor_nutrient_solution import NutrientSolutionSensor
14 from models.sensor_nutrient_solution_data import
NutrientSolutionSensorData
15 # Importamos la configuración de AWS
16 from mqtt.aws_config import AWS_ENDPOINT, MQTT_CLIENT_ID
17 # Importamos el cliente WebSocket
18 from websocket_manager import websocket_manager
19 import datetime
20
21 TOPIC_PUB_MODEL_MAP = {
22     "environmental/sensor/pub": EnvironmentalSensorData ,
23     "nutrient/solution/sensor/pub": NutrientSolutionSensorData ,
24     "consumption/sensor/pub": ConsumptionSensorData ,
25     "actuators/pub": ActuatorData ,
26 }
27
28 TOPIC_SUB_MODEL_MAP = {
29     "environmental/sensor/sub": EnvironmentalSensor ,
30     "nutrient/solution/sensor/sub": NutrientSolutionSensor ,
31     "consumption/sensor/sub": ConsumptionSensor ,
32     "actuators/sub": Actuator ,
33 }
34
35 # Método asíncrono para insertar datos en la base de datos
36 async def insert_sensor_data(payload, model_class):
37     try:
38         sensor_data = model_class(**payload)
39         await sensor_data.insert()
40         print("Datos guardados en la base de datos.")
41     except Exception as e:
42         print(f"Error al guardar datos en la base de datos: {e}")
43
44 # Método asíncrono para actualizar el intervalo de reporte
45 async def update_device_interval(device_code, new_interval,
model_class, device_type="sensor"):
46     try:
47         print(f"Actualizando {device_type} {device_code} a {
new_interval}s en BD...")
48
49         # Convertir a entero explícitamente
50         interval_value = int(new_interval)
51         # Buscar el dispositivo usando Beanie
52         query_field = "sensor_code" if device_type == "sensor" else
"actuator_code"
53         device = await model_class.find_one({query_field:
device_code})
54
55         if device:
56             # Actualizar usando el método set de Beanie
57             await device.set({"seconds_to_report": interval_value})
58             print(f"Intervalo actualizado para {device_type} {
device_code}")
59             return True
60         else:

```

```
61         print(f"{device_type.capitalize()} {device_code} no")
62         encontrado en BD")
63         return False
64     except ValueError as e:
65         print(f"Error de validación: {e}")
66     except Exception as e:
67         print(f"Error actualizando BD: {str(e)}")
68     return False
69
70 # Método para procesar mensajes entrantes de sensores y actuadores
71 async def process_sensor_message_pub(topic, payload):
72     try:
73         print(f"Mensaje recibido desde {topic}")
74         print(f"Data: {payload}")
75
76         model_class = TOPIC_PUB_MODEL_MAP.get(topic)
77         if model_class:
78             print(f"Procesando mensaje de {topic}")
79             await insert_sensor_data(payload, model_class)
80
81         # Determinar el tipo de sensor para WebSocket
82         sensor_type = None
83         if "environmental/sensor/pub" in topic:
84             sensor_type = "environmental"
85         elif "nutrient/solution/sensor/pub" in topic:
86             sensor_type = "nutrient_solution"
87         elif "consumption/sensor/pub" in topic:
88             sensor_type = "consumption"
89         elif "actuators/pub" in topic:
90             sensor_type = "actuators"
91
92         # Verificar si el tipo de sensor es válido
93         if sensor_type:
94             # Actualizar cache y enviar a WebSocket
95             websocket_manager.update_cache(sensor_type, payload)
96             await websocket_manager.broadcast({
97                 "type": sensor_type,
98                 "data": payload,
99                 "timestamp": datetime.datetime.now().isoformat()
100            })
101        else:
102            print(f"Tópico no reconocido: {topic}")
103    except Exception as e:
104        print(f"Error inesperado: {e}")
105
106 # Método para procesar mensajes de control de sensores y actuadores
107 async def process_sensor_message_sub(topic, payload):
108     try:
109         print(f"Mensaje recibido desde {topic}")
110         print(f"Data: {payload}")
111
112         model_class = TOPIC_SUB_MODEL_MAP.get(topic)
113         if not model_class:
114             print(f"Tópico no reconocido: {topic}")
115             return
116
117         # Determinar si es un sensor o actuador
118         is_actuator = "actuators/sub" in topic
119         device_type = "actuator" if is_actuator else "sensor"
120         code_field = "actuator_code" if is_actuator else "
121         sensor_code"
```

```
122     # Verificar que el campo de código exista en el payload
123     if code_field not in payload:
124         print(f"Falló campo {code_field} en el mensaje de {
device_type}")
125         return
126
127     # Solo procesamos si es una confirmación del dispositivo
128     if payload.get("interval") == "OK" and payload.get(
seconds_to_report):
129         print(f"Confirmación válida recibida del {device_type}")
130         await update_device_interval(
131             device_code=payload[code_field],
132             new_interval=payload["seconds_to_report"],
133             model_class=model_class,
134             device_type=device_type
135         )
136     else:
137         print(f"Mensaje recibido (esperando confirmación del {
device_type})")
138     except Exception as e:
139         print(f"Error procesando mensaje: {e}")
140
141 # Clase para manejar la conexión y comunicación con AWS IoT Core
142 class AWSMQTTClient:
143     def __init__(self):
144         CERTS_DIR = os.getenv("CERTS_DIR", "certificates")
145         self.ROOT_CRT = os.path.join(CERTS_DIR, "root.crt")
146         self.CLIENT_CRT = os.path.join(CERTS_DIR, "client.crt")
147         self.CLIENT_KEY = os.path.join(CERTS_DIR, "client.key")
148
149     # Verificar existencia de certificados
150     for cert in [self.ROOT_CRT, self.CLIENT_CRT, self.CLIENT_KEY]:
151         if not os.path.exists(cert):
152             raise FileNotFoundError(f"El archivo {cert} no fue
encontrado.")
153
154     # Configurar cliente MQTT
155     self.client = AWSIoTMQTTClient(MQTT_CLIENT_ID)
156     self.client.configureEndpoint(AWS_ENDPOINT, 8883)
157     self.client.configureCredentials(self.ROOT_CRT, self.
CLIENT_KEY, self.CLIENT_CRT)
158     self.client.configureConnectDisconnectTimeout(10)
159     self.loop = asyncio.get_event_loop()
160
161     def connect(self):
162         print("Estableciendo conexión con AWS IoT Core.")
163         self.client.connect()
164         print("Cliente MQTT Conectado.")
165
166     def disconnect(self):
167         self.client.disconnect()
168         print("Cliente Desconectado")
169
170     def publish(self, topic, message):
171         payload = json.dumps(message) if isinstance(message, dict)
172         else str(message)
173         self.client.publish(topic, payload, 0)
174         print(f"Mensaje enviado a {topic}: {payload}")
175
176     def subscribe(self, topic, callback):
177         def wrapper(client, userdata, message):
178             try:
```

```
178     payload_str = message.payload.decode()
179     payload = json.loads(payload_str)
180     if topic in [ "environmental/sensor/sub", "nutrient/
181 solution/sensor/sub", "consumption/sensor/sub", "actuators/sub"]:
182         # process_sensor_message_sub(topic, payload)
183         self.loop.create_task(process_sensor_message_sub(
184             (topic, payload)))
185     else:
186         self.loop.create_task(process_sensor_message_pub(
187             (message.topic, payload)))
188     except json.JSONDecodeError:
189         print("Error al decodificar el mensaje JSON.")
190     except Exception as e:
191         print(f"Error inesperado en wrapper: {e}")
192
193     self.client.subscribe(topic, 1, wrapper)
194     print(f"Suscripto a {topic}")
195
196     def unsubscribe(self, topic):
197         self.client.unsubscribe(topic)
198         print(f"Desuscripto de {topic})
```

CÓDIGO E.2. Definición de Clase para cliente MQTT.

El código F.3 muestra la integración del cliente MQTT con la aplicación en FastAPI. Se define un cliente MQTT y se suscribe a los tópicos de publicación de datos de sensores y actuadores y tópicos de envío de parámetros a sensores y actuadores. Además, se definen los endpoints para probar la conexión MQTT y publicar mensajes desde la aplicación FastAPI.

```
1 # Fragmento de código para la integración del cliente MQTT con FastAPI
2
3
4 # Inicializar cliente MQTT
5 mqtt_client = AWSMQTTClient()
6
7 @app.on_event("startup")
8 async def startup():
9     await init_db()
10    mqtt_client.connect()
11
12    # Suscripción a tópicos de publicación de sensores y actuadores
13    mqtt_client.subscribe("environmental/sensor/pub",
14                          process_sensor_message_pub)
15    mqtt_client.subscribe("nutrient_solution/sensor/pub",
16                          process_sensor_message_pub)
17    mqtt_client.subscribe("consumption/sensor/pub",
18                          process_sensor_message_pub)
19    mqtt_client.subscribe("actuators/pub", process_sensor_message_pub)
20
21    # Suscripción a tópicos de envío de parámetros a sensores y
22    # actuadores
23    mqtt_client.subscribe("environmental/sensor/sub",
24                          process_sensor_message_sub)
25    mqtt_client.subscribe("nutrient_solution/sensor/sub",
26                          process_sensor_message_sub)
27    mqtt_client.subscribe("consumption/sensor/sub",
28                          process_sensor_message_sub)
29    mqtt_client.subscribe("actuators/sub", process_sensor_message_sub)
30
31
32 @app.on_event("shutdown")
33 async def shutdown():
```

```
26     mqtt_client.disconnect()
27
28 # Clase para manejar la publicación de mensajes MQTT
29 class PublishRequest(BaseModel):
30     topic: str
31     message: dict
32
33 # Endpoint para publicar mensajes MQTT
34 @app.post("/mqtt/publish")
35 def publish_message(request: PublishRequest, user: dict = Depends(
36     current_user)):
37     mqtt_client.publish(request.topic, request.message)
38     return {"status": "Mensaje publicado", "topic": request.topic, "message": request.message}
39
40 # Endpoint para probar la conexión MQTT
41 @app.get("/mqtt/test")
42 def test_mqtt_connection(user: dict = Depends(current_user)):
43     try:
44         mqtt_client.connect()
45         return {"status": "Conexión exitosa"}
46     except Exception as e:
47         return {"status": "Error de conexión", "error": str(e)}
```

CÓDIGO E.3. Cliente MQTT en FastAPI.

Apéndice F

Conexión por WebSocket en FastAPI

F.1. Introducción

Para establecer una conexión WebSocket se emplea el módulo `websockets` de FastAPI, el cual permite crear un servidor que gestiona conexiones en tiempo real con múltiples clientes. En este caso, el cliente WebSocket se conecta a la ruta `/ws/sensor-data` para recibir información de sensores y actuadores. La conexión permanece activa mientras se intercambian datos, y se cierra automáticamente si ocurre un error o si el usuario no está autorizado.

F.2. Autorización para WebSocket

El código F.1 muestra la función encargada de gestionar la autenticación del usuario a través del WebSocket. El token de acceso puede enviarse en el encabezado `Authorization` o como `Query Params`. Si el token es válido, se autentica al usuario; en caso contrario, se cierra la conexión.

```

1 from fastapi import WebSocket, status
2 from utils.authentication import auth_user, current_user
3
4 async def websocket_current_user(websocket: WebSocket):
5     try:
6         # Obtener el token del header o query parameter
7         token = websocket.headers.get("authorization") or websocket.
8         query_params.get("token")
9         if not token:
10             await websocket.close(code=status.WS_1008_POLICY_VIOLATION)
11             return None
12
13         # Limpiar el token (eliminar 'Bearer ' si existe)
14         if token.startswith("Bearer "):
15             token = token[7:]
16
17         user = await auth_user(token)
18         return await current_user(user)
19
20     except Exception as e:
21         print(f"Error en la autenticación del WebSocket: {e}")
22         await websocket.send_text( f"Error en la autenticación del
23         WebSocket: {str(e)}")
24         await websocket.close(code=status.WS_1008_POLICY_VIOLATION)

```

23 return None

CÓDIGO F.1. Autorización para WebSocket.

F.3. Gestión de conexiones WebSocket

En el código F.2 se presenta la clase `WebSocketManager`, encargada de gestionar las conexiones WebSocket. Esta clase sigue el patrón Singleton y mantiene una lista de conexiones activas, además de un caché con los últimos datos enviados por tipo de sensor. También gestiona la conexión y desconexión de los clientes, el envío de datos en tiempo real, y la actualización del caché con cada nuevo mensaje recibido.

```

1 import datetime
2 from typing import List, Dict, Optional
3 from fastapi import WebSocket
4
5 class WebSocketManager:
6     _instance: Optional['WebSocketManager'] = None
7
8     # Método de clase para obtener la instancia única (Singleton)
9     def __new__(cls):
10         if cls._instance is None:
11             cls._instance = super().__new__(cls)
12             cls._instance.active_connections: List[Dict] = []
13             cls._instance.sensor_data_cache: Dict[str, dict] = {
14                 "environmental": {},
15                 "nutrient_solution": {},
16                 "consumption": {},
17                 "actuators": {}
18             }
19         return cls._instance
20
21     # Método para manejar la conexión de un nuevo cliente WebSocket
22     async def connect(self, websocket: WebSocket, user: dict):
23         self.active_connections.append({
24             "websocket": websocket,
25             "user": user
26         })
27         # Enviar datos almacenados al nuevo cliente
28         for sensor_type, data in self.sensor_data_cache.items():
29             if data:
30                 await websocket.send_json({
31                     "type": sensor_type,
32                     "data": data["data"],
33                     "timestamp": data["timestamp"]
34                 })
35
36     # Método para manejar la desconexión de un cliente WebSocket
37     def disconnect(self, websocket: WebSocket):
38         self.active_connections = [
39             conn for conn in self.active_connections
40             if conn["websocket"] != websocket
41         ]
42
43     # Método para enviar un mensaje a un cliente WebSocket específico
44     async def broadcast(self, message: dict):
45         for connection in self.active_connections[:]:
46             try:
47                 await connection["websocket"].send_json(message)

```

```

48         except Exception as e:
49             print(f"Error broadcasting to WebSocket: {e}")
50             self.disconnect(connection)
51
52     # Método para enviar un mensaje a todos los clientes WebSocket
53     def update_cache(self, sensor_type: str, data: dict):
54         self.sensor_data_cache[sensor_type] = {
55             "data": data,
56             "timestamp": datetime.datetime.now().isoformat()
57         }
58
59     # Método para obtener datos almacenados en caché
60     def get_cached_data(self, sensor_type: str) -> dict:
61         return self.sensor_data_cache.get(sensor_type, {})
62
63     # Método para obtener la cantidad de clientes conectados
64     @property
65     def connected_clients(self) -> int:
66         return len(self.active_connections)
67
68     # Instancia del WebSocketManager
69     websocket_manager = WebSocketManager()

```

CÓDIGO F.2. Definición de clase WebSocket.

F4. Integración con FastAPI

Finalmente, el fragmento F.3 muestra cómo se integra WebSocket en FastAPI. Cuando un cliente intenta conectarse a /ws/sensor-data, se autentica al usuario, se acepta la conexión y se agrega al gestor. La conexión se mantiene abierta mientras no se produzca un error o una desconexión por parte del cliente. En ambos casos, el gestor elimina al cliente de la lista de conexiones activas.

```

1  # Fragmento de código para la integración del cliente MQTT con FastAPI
2
3
4  # Importamos el cliente WebSocket
5  from utils.websocket import websocket_manager
6  # Importamos el cliente WebSocket para autenticación
7  from utils.websocket_auth import websocket_current_user
8
9  # WebSocket para recibir datos de sensores y actuadores
10 @app.websocket("/ws/sensor-data")
11 async def websocket_endpoint(websocket: WebSocket):
12     # Aceptar la conexión WebSocket
13     await websocket.accept()
14     # Obtener el usuario actual
15     user = await websocket_current_user(websocket)
16     # Verificar si el usuario está autenticado
17     if not user:
18         # Cerrar la conexión si no está autenticado
19         return
20
21     # Conectar el WebSocket al gestor
22     await websocket_manager.connect(websocket, user)
23
24     # Mantener la conexión abierta
25     try:
26         while True:
27             await websocket.receive_text()
28     except WebSocketDisconnect:

```

```
29     websocket_manager.disconnect(websocket)
30 except Exception as e:
31     print(f"Error en WebSocket: {e}")
32     websocket_manager.disconnect(websocket)
```

CÓDIGO F.3. Cliente WebSocket en FastAPI.

Bibliografía

- [1] Global Agricultural Productivity (GAP). *2016 Global Agricultural Productivity Report*. Inf. téc. Documento en línea. Global Agricultural Productivity, 2016. URL: https://globalagriculturalproductivity.org/wp-content/uploads/2019/01/2016_GAP_Report.pdf (visitado 20-03-2025).
- [2] Raquel Salazar-Moreno, Abraham Rojano-Aguilar e Irineo Lorenzo López-Cruz. «La eficiencia en el uso del agua en la agricultura controlada». En: *Tecnología y ciencias del agua* 5.2 (2014). Documento en línea, págs. 177-183. URL: http://www.scielo.org.mx/scielo.php?script=sci_arttext&pid=S2007-24222014000200012&lng=es&tlang=es (visitado 20-03-2025).
- [3] Misiones Online. *Horticultura en Misiones*. URL: <https://misionesonline.net/2024/06/14/horticultura-en-misiones-2/> (visitado 20-03-2025).
- [4] Primera Edición. *Misiones: la hidroponía, cada vez más presente*. Primera Edición. URL: <https://www.primeraedicion.com.ar/nota/100627758/misiones-la-hidroponia-cada-vez-mas-presente/> (visitado 20-03-2025).
- [5] Lucas A. Garibaldi et al. «Seguridad alimentaria, medio ambiente y nuestros hábitos de Consumo». En: *Ecología Austral* 28.3 (2018), págs. 572-580. URL: https://www.scielo.org.ar/scielo.php?script=sci_arttext&pid=S1667-782X2018000400011&lng=es&tlang=es (visitado 20-03-2025).
- [6] Hidropónia FIL. URL: <https://hidroponiafil.com.ar/> (visitado 20-03-2025).
- [7] Hidrosense. URL: <https://www.hidrosense.com.br/> (visitado 20-03-2025).
- [8] iPonia. URL: <https://iponia.com.br/> (visitado 20-03-2025).
- [9] Growcast. URL: <https://www.growcast.io/> (visitado 20-03-2025).
- [10] Shanna Li. «Comparative analysis of infrastructure and Ad-Hoc wireless networks». En: *ITM Web of Conferences*. Proceedings of the International Conference on Intelligent Computing, Communication & Information Technologies (ICICCI 2018) 25 (2019). Article number 01009, pág. 01009. ISSN: 2271-2097. DOI: <https://doi.org/10.1051/itmconf/20192501009>. URL: https://www.itm-conferences.org/articles/itmconf/pdf/2019/02/itmconf_icicci2018_01009.pdf.
- [11] OASIS Open. *Foundational IoT Messaging Protocol MQTT Becomes International OASIS Standard*. OASIS Open. URL: <https://www.oasis-open.org/2014/11/13/foundational-iot-messaging-protocol-mqtt-becomes-international-oasis-standard/> (visitado 25-03-2025).
- [12] Amazon Web Services. ¿Qué es MQTT? AWS. URL: <https://aws.amazon.com/es/what-is/mqtt/> (visitado 25-03-2025).
- [13] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Internet Requests for Comments. RFC. DOI: [10.17487/RFC8446](https://doi.org/10.17487/RFC8446). URL: <https://datatracker.ietf.org/doc/html/rfc8446>.
- [14] IBM Corporation. *Protocolos TCP/IP*. International Business Machines (IBM). URL: <https://docs.aws.amazon.com/iot/latest/developerguide/transport-security.html> (visitado 25-03-2025).
- [15] I. Fette y A. Melnikov. *The WebSocket Protocol*. Inf. téc. IETF. DOI: [10.17487/RFC6455](https://doi.org/10.17487/RFC6455). URL: <https://tools.ietf.org/html/rfc6455>.

- [16] Amazon Web Services. *Transport Security in AWS IoT Core*. URL: <https://docs.aws.amazon.com/iot/latest/developerguide/transport-security.html> (visitado 25-03-2025).
- [17] Espressif Systems (Shanghai) Co., Ltd. *ESP32-WROOM-32 Datasheet*. URL: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf (visitado 26-03-2025).
- [18] Bosch Sensortec. *BME280 - Combined humidity, pressure and temperature sensor*. URL: <https://www.bosch-sensortec.com/products/environmental-sensors/humidity-sensors-bme280/> (visitado 26-03-2025).
- [19] ROHM Semiconductor. *BH1750 - Ambient Light Sensor (ALS) - Datasheet*. URL: https://www.mouser.com/catalog/specsheets/Rohm_11162017_ROHMS34826-1.pdf (visitado 26-03-2025).
- [20] Zhengzhou Winsen Electronics Technology Co., Ltd. *MH-Z19C NDIR CO₂ Sensor - Terminal Type Manual*. URL: https://www.mouser.com/datasheet/2/1398/Soldered_108994_co2_sensor_mh_z19-3532446.pdf (visitado 26-03-2025).
- [21] DFRobot. *Datos técnicos del sensor de pH líquido PH-4502C*. URL: <https://image.dfrobot.com/image/data/SEN0161/PH%20composite%20electrode%20manual.pdf> (visitado 26-03-2025).
- [22] METTLER TOLEDO. *Sensores de Conductividad*. METTLER TOLEDO International Inc. URL: <https://www.mt.com/es/es/home/products/Process-Analytics/conductivity-resistivity-analyzers/conductivity-sensor.html> (visitado 26-03-2025).
- [23] EC Buying. *Datos técnicos del sensor de CE*. URL: https://es.aliexpress.com/item/1005003479288815.html?spm=a2g0o.order_list.order_list_main.40.42e8194dYjUbr1&gatewayAdapt=glo2esp (visitado 26-03-2025).
- [24] Hach Company. *Sólidos (totales y disueltos)*. URL: <https://es.hach.com/parameters/solids/> (visitado 26-03-2025).
- [25] DFRobot. *Datos técnicos Sensor TDS*. URL: <https://www.dfrobot.com/product-1662.html?srsltid> (visitado 26-03-2025).
- [26] Dallas Semiconductor. *Datos técnicos Sensor DS18B20*. URL: <https://cdn.sparkfun.com/datasheets/Sensors/Temp/DS18B20.pdf> (visitado 26-03-2025).
- [27] Sparkfun Electronics. *Datos técnicos del sensor ultrasónico*. URL: <https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf> (visitado 26-03-2025).
- [28] Unit Electronics. *Datos técnicos del sensor de energía eléctrica PZEM-004T*. URL: <https://uelectronics.com/wp-content/uploads/2024/06/AR4189-AR4190-Medidor-de-Energia-Electrica-AC-100A-Manual.pdf> (visitado 26-03-2025).
- [29] Songle Relay. *Datos técnicos del Relay Songle*. URL: https://datasheet4u.com/pdf-down/S/R/D/SRD-12VDC-xx-x_ETC.pdf (visitado 26-03-2025).
- [30] Damien P. George y contributors. *MicroPython*. URL: <https://micropython.org/> (visitado 26-03-2025).
- [31] CTA Electronics. *MicroPython - Recursos y Guías*. URL: <https://www.ctaelectronics.com/es/micropython/> (visitado 26-03-2025).
- [32] Tiangolo. *FastAPI*. URL: <https://fastapi.tiangolo.com/> (visitado 26-03-2025).
- [33] MongoDB, Inc. *MongoDB Documentation*. URL: <https://www.mongodb.com/> (visitado 26-03-2025).
- [34] Meta (formerly Facebook) and contributors. *React: Biblioteca de JavaScript para interfaces de usuario*. URL: <https://es.react.dev/> (visitado 26-03-2025).

- [35] Docker, Inc. *Docker: Desarrollo acelerado de aplicaciones en contenedores*. URL: <https://www.docker.com/> (visitado 26-03-2025).
- [36] Amazon.com, Inc. *AWS IoT Core*. URL: <https://aws.amazon.com/es/iot-core/> (visitado 26-03-2025).
- [37] Amazon.com, Inc. *EC2, Nube de cómputo elástica de Amazon*. URL: <https://aws.amazon.com/es/ec2/> (visitado 26-03-2025).
- [38] Microsoft Corporation. *Visual Studio Code*. URL: <https://code.visualstudio.com/> (visitado 26-03-2025).
- [39] Postman, Inc. *Postman API Platform*. URL: <https://www.postman.com/> (visitado 26-03-2025).
- [40] Git. *Sistema de control de versiones Git*. URL: <https://git-scm.com/> (visitado 26-03-2025).
- [41] GitHub, Inc. *GitHub*. URL: <https://github.com/> (visitado 26-03-2025).
- [42] Beanie: *Asynchronous Python ODM for MongoDB*. URL: <https://beanie-odm.dev/> (visitado 01-04-2025).
- [43] Inc. MongoDB. *Motor: Asynchronous Python driver for MongoDB*. URL: <https://motor.readthedocs.io/en/stable/index.html> (visitado 01-04-2025).
- [44] Amazon Web Services. *AWS IoT SDKs - Guía del desarrollador*. URL: https://docs.aws.amazon.com/es_es/iot/latest/developerguide/iot-sdks.html (visitado 02-04-2025).
- [45] *FastAPI WebSockets Reference*. URL: <https://fastapi.tiangolo.com/reference/websockets/> (visitado 02-04-2025).
- [46] *Socket.IO: Real-time application framework*. URL: <https://socket.io/> (visitado 02-04-2025).