

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	9
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	10
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	12
2.5.1. React	12
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	13

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
2.2. 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

2.7.1. Visual Studio Code	13
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	16
3.1.2. Capa de red	16
3.1.3. Capa de aplicación	16
3.2. Modelo de datos	17
3.2.1. Pruebas iniciales de sensores	17
3.2.2. Diseño del modelo de datos	17
3.3. Servidor IoT	19
3.3.1. Tecnologías utilizadas	19
Backend	19
Base de datos	19
Frontend	19
Despliegue	19
3.3.2. Arquitectura del servidor	20
Backend	20
Capa de datos	21
Frontend	21
3.4. Desarrollo del backend	21
3.4.1. Diseño de la API	22
3.4.2. Autenticación y autorización	23
3.4.3. Persistencia de datos	24
3.4.4. Comunicación con el broker MQTT	25
Pasos en AWS IoT Core	25
Implementación de MQTT en FastAPI	26
Comunicación con MQTT en FastAPI	26
3.4.5. Implementación de WebSockets	28
3.5. Desarrollo del frontend	28
3.6. Desarrollo de nodos sensores y actuadores	28
3.7. Despliegue del sistema	28
A. Resumen de endpoints de la API	29
B. Autenticación con JWT	33
C. Conexión al broker MQTT con FastAPI y AWS IoT SDK para Python	35
Bibliografía	39

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	24
3.6. Desarrollo de nodos sensores y actuadores	24
3.7. Despliegue del sistema	24
A. Modelo de datos implementado en el trabajo	25
B. Resumen de endpoints de la API	27
C. Autenticación con JWT	31
D. Conexión de la aplicación FastAPI con la base de datos MongoDB	33
E. Conexión al broker MQTT con FastAPI y AWS IoT SDK para Python	35
Bibliografía	39

Índice de figuras

1.1. Diagrama en bloques del sistema.	4
2.1. Arquitectura del protocolo MQTT.	8
2.2. Microcontrolador ESP-WROOM-32.	9
2.3. Sensor BME280.	9
2.4. Sensor BH1750.	9
2.5. Sensor MHZ19C.	9
2.6. Sensor PH-4502C.	10
2.7. Sensor CE.	10
2.8. Sensor TDS.	10
2.9. Sensor de temperatura DS18B20.	11
2.10. Sensor HC-SR04.	11
2.11. Sensor de medición de consumo eléctrico.	11
2.12. Relay de 2 Canales 5 V 10 A	11
3.1. Arquitectura de la solución propuesta.	15
3.2. Modelo de datos implementado.	18
3.3. Arquitectura del servidor del sistema IoT.	20
3.4. Esquema de autenticación y autorización.	23
3.5. Ejemplo de políticas de acceso en AWS IoT Core.	25
3.6. Pruebas en Postman. Test de conexión al broker MQTT.	26
3.7. Pruebas en Postman. Publicación de un mensaje en un tópico.	27
3.8. Cliente MQTT en FastAPI.	27
3.9. Datos almacenados en MongoDB.	28

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
A.1. Modelo de datos implementado en el trabajo.	26

Índice de tablas

1.1. Características de la competencia.....	2
3.1. Principales sensores y librerías utilizadas	17
3.2. Resumen de principales endpoints de la API	22
A.1. Resumen de principales endpoints de la API	29
A.2. Resumen de principales endpoints de la API	30
A.3. Resumen de principales endpoints de la API	31

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	27
B.2. Resumen de principales endpoints de la API	28
B.3. Resumen de principales endpoints de la API	29

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 **proyecto** 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.

1.2. 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.

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.

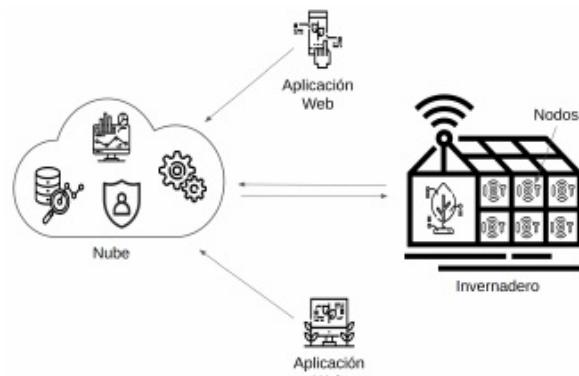


FIGURA 1.1. Diagrama en bloques del sistema.

1.5. Requerimientos

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

1. Requerimientos de los nodos:

- Utilizar microcontroladores basados en ESP32.
- Implementar certificados TLS para seguridad en las comunicaciones.
- Permitir conexión Wi-Fi.
- Identificador único por nodo dentro del sistema.
- Configuración remota del intervalo de envío de datos.
- Los nodos sensores deben transmitir al servidor IoT:

- 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.

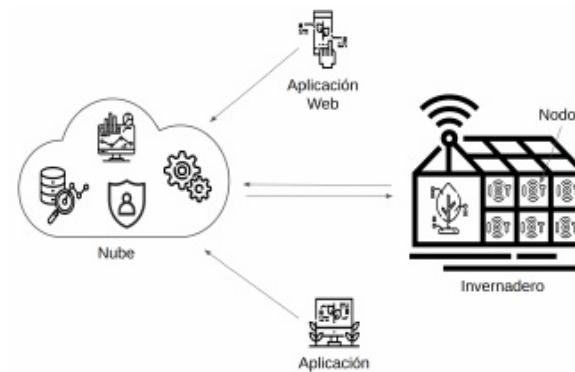


FIGURA 1.1. Diagrama en bloques del sistema.

1.5. Requerimientos

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

1. Requerimientos de los nodos:

- Utilizar microcontroladores basados en ESP32.
- Implementar certificados TLS para seguridad en las comunicaciones.
- Permitir conexión Wi-Fi.
- Identificador único por nodo dentro del sistema.
- Configuración remota del intervalo de envío de datos.
- Los nodos sensores deben transmitir al servidor IoT:

- 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.

La figura 2.1 muestra la arquitectura del protocolo MQTT.

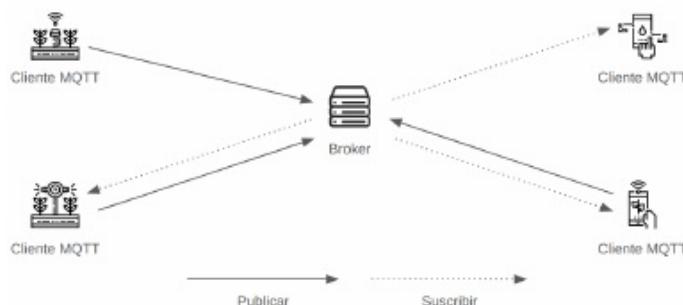


FIGURA 2.1. Arquitectura del protocolo MQTT.

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 Wi-Fi, 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.

2.2. Componentes de hardware

9

FIGURA 2.2. Microcontrolador **ESP-WROOM-32**.

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

El BME280 (figura 2.3) 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].



FIGURA 2.3. Sensor BME280.

2.2.3. Sensor de luz digital

El BH1750 (figura 2.4) 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 2.4. Sensor BH1750.

2.2.4. Sensor de dióxido de carbono

El sensor MH-Z19C (figura 2.5) 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 2.5. Sensor MHZ19C.

2.2. Componentes de hardware

9

FIGURA 2.2. Sensor **BME280**².

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 **BH1750**³.

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 **MHZ19C**⁴.

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](#).

2.2.5. Sensor de detección de pH

El sensor PH-4502C (figura 2.6) 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].



FIGURA 2.6. Sensor PH-4502C.

2.2.6. Sensor de conductividad eléctrica

El sensor de CE (figura 2.7) mide la capacidad de una solución para conducir electricidad, lo cual 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 2.7. Sensor CE.

2.2.7. Sensor de sólidos disueltos totales

El sensor TDS (figura 2.8) 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 2.8. Sensor TDS.

2.2.8. Sensor de temperatura digital sumergible

El DS18B20 (figura 2.9) 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].



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 2.9. Sensor de temperatura DS18B20.

2.2.9. Sensor ultrasónico

El sensor HC-SR04 (figura 2.10) 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 2.10. Sensor HC-SR04.

2.2.10. Sensor de medición de consumo eléctrico

El sensor PZEM-004T (figura 2.11) 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 2.11. Sensor de medición de consumo eléctrico.

2.2.11. Módulo Relay

El módulo Relay (figura 2.12) 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].



FIGURA 2.12. Relay de 2 Canales 5 V 10 A

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.

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].

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 librería 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



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.

2.6. Infraestructura y despliegue

13

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].

2.5. Desarrollo frontend

13

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].

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].

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.

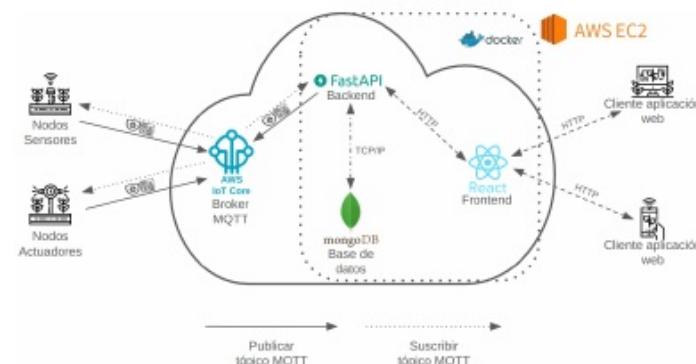


FIGURA 3.1. Arquitectura de la solución propuesta.

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

- **Capa de percepción:** formada por nodos sensores y actuadores que **recopilan datos del entorno y ejecutan acciones de acuerdo con la configuración establecida**.
- **Capa de red:** encargada de gestionar la comunicación entre los dispositivos IoT y el backend. Los sensores y actuadores transmiten datos a través de Wi-Fi, los cuales son gestionados por un broker MQTT.

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.

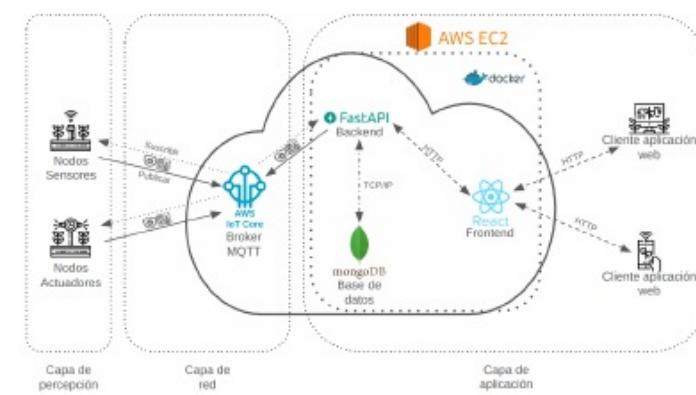


FIGURA 3.1. Arquitectura de la solución propuesta.

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.

- Capa de aplicación: plataforma en la nube responsable del procesamiento, almacenamiento y visualización de datos. Facilita la interacción con los dispositivos, la gestión de la información y la presentación de datos mediante una interfaz accesible para el usuario.

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, el cual 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 cuales 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.

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.

3.2. Modelo de datos

17

3.2. Modelo de datos

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

3.2.1. Pruebas iniciales de sensores

Para diseñar el modelo adecuado, se llevó a cabo una prueba inicial con los sensores y se registraron los datos generados por cada uno de ellos. La tabla 3.1 muestra los datos obtenidos de cada sensor, tal como lo devuelve la librería utilizada para su configuración y lectura.

TABLA 3.1. Principales sensores y **librerías utilizadas**

Componente	Datos
Sensor BME280	La librería utilizada [42] devuelve los valores de temperatura ambiente, humedad relativa y presión atmosférica como Float.
Sensor BH1750	La librería utilizada [43] devuelve el valor de lux como Float.
Sensor MH-Z19C	La librería utilizada [44] devuelve el valor de ppm de CO ₂ como Int.
Sensor PH-4502	La librería utilizada [45] devuelve el valor de TDS como Float.
Sensor de CE	La librería utilizada [46] devuelve el valor de TDS como Float.
Sensor de TDS	La librería utilizada [47] devuelve el valor de TDS como Float.
Sensor DS18B20	La librería utilizada [48] devuelve el valor de la temperatura como Float.
Sensor HC-SR04	La librería utilizada [49] devuelve el valor de distancia en centímetros como Int.
Sensor PZEM-004T	La librería utilizada [50] devuelve los valores de voltaje, corriente, potencia, cálculo de potencia y factor de potencia como Float.

3.2.2. Diseño del modelo de datos

El diseño del modelo de datos se desarrolló de acuerdo a los tipos de datos proporcionados por los sensores, así como los requerimientos técnicos establecidos para el sistema.

La estructura se organizó en colecciones dentro de MongoDB, donde cada colección representa un tipo de dato específico. Cada colección contiene documentos que almacenan las lecturas de los sensores y actuadores, usuarios, ambientes, tipos de ambientes, entre otros. Las colecciones se vinculan mediante identificadores únicos, lo que facilita la conexión entre las diferentes colecciones.

La figura 3.2 muestra el modelo de datos implementado en el sistema.

3.2. Modelo de datos

17

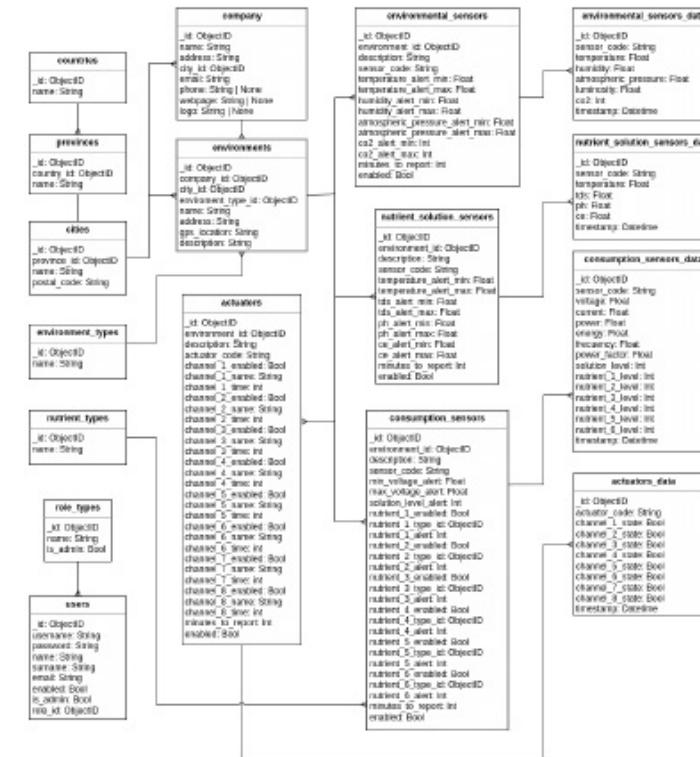


FIGURA 3.2. Modelo de datos implementado.

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.

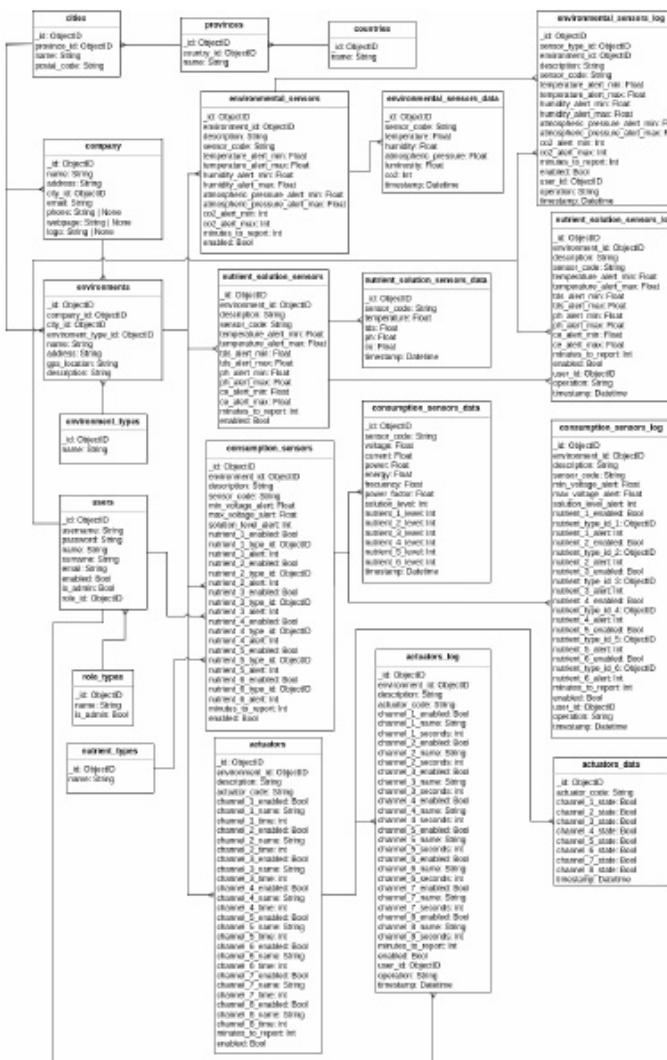


FIGURA 3.2. Modelo de datos implementado.

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.

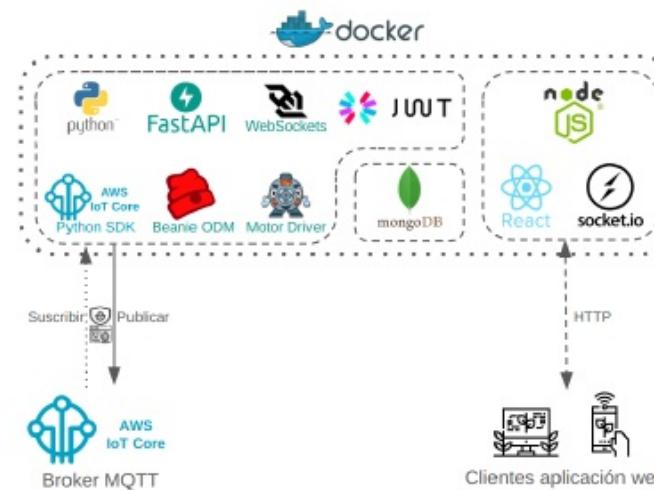


FIGURA 3.3. Arquitectura del servidor del sistema IoT.

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

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. Tecnologías utilizadas

Backend

Para el desarrollo del backend se optó por utilizar el framework FastAPI basado en Python, que permite crear APIs RESTful de manera rápida y eficiente. La comunicación entre el frontend y el backend se realizó a través de la API REST expuesta con HTTP y el formato JSON para el intercambio de datos.

Para el envío de los datos en tiempo real desde el backend al frontend, se utilizó la librería Websocket [51] de FastAPI, que permite establecer una conexión bidireccional entre el servidor y el cliente.

La comunicación entre el backend y el broker MQTT se implementó mediante el SDK (del inglés, *Software Development Kit*) de AWS IoT para Python [52], que simplifica la integración de FastAPI con el protocolo MQTT. Este SDK permite gestionar la conexión con el broker, así como publicar y suscribirse a tópicos.

Base de datos

Como se mencionó anteriormente, se utilizó MongoDB como base de datos para almacenar los datos generados por los nodos sensores y actuadores, así como la información relacionada con los requerimientos del sistema.

La comunicación entre el backend y la base de datos se realizó a través de la librería Motor [53], que proporciona una interfaz asíncrona para interactuar con MongoDB. Además se utilizó el ODM (del inglés, *Object Document Mapper*), a través de la librería Beanie [54], que permite definir modelos y realizar consultas y operaciones sobre la base de datos de manera sencilla.

Frontend

El frontend se desarrolló mediante la librería React de Facebook. Para el diseño de la interfaz, se utilizó la librería Bootstrap para React [55] lo que permitió crear una aplicación web responsive y fácil de usar.

La comunicación entre el frontend y el backend se realizó a través de los endpoints de la API REST. Para la visualización de los datos en tiempo real, se utilizó la librería Socket.IO [56], que permite establecer una conexión WebSocket entre el frontend y el backend.

Despliegue

Por último, para el despliegue del sistema se utilizó Docker, que permite crear contenedores para cada uno de los componentes del sistema. Esto facilita la gestión y el despliegue de la aplicación, ya que cada componente se ejecuta en su propio contenedor, lo que permite una mayor flexibilidad y escalabilidad.

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.

3.3.2. Arquitectura del servidor

La arquitectura se diseñó para ser escalable, flexible y fácil de mantener. Está compuesta de los siguientes componentes principales:

- Backend.
 - API REST.
 - Autenticación y autorización.
 - Integración con la base de datos.
 - Conexión con el broker MQTT.
 - Comunicación con frontend en tiempo real.
- Capa de datos.
- Frontend.

- Interfaz de usuario.
- Comunicación con backend en tiempo real.

La figura 3.3 muestra la arquitectura del servidor del sistema IoT.

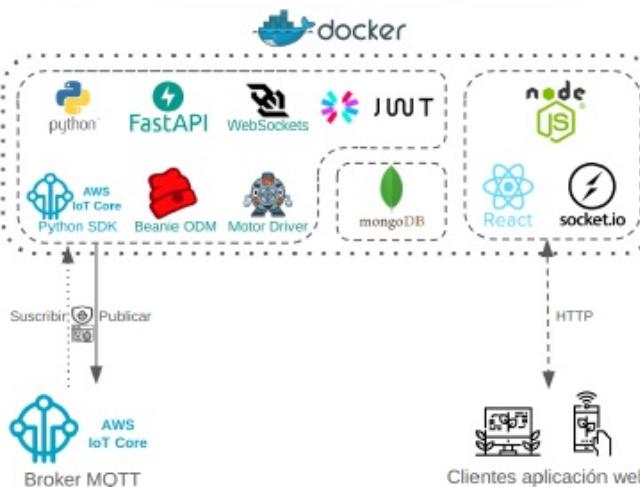


FIGURA 3.3. Arquitectura del servidor del sistema IoT.

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

Backend

API REST: Es el servicio principal que gestiona las peticiones del frontend. Permite la creación, lectura, actualización y eliminación de datos relacionados con los sensores, actuadores, ambientes y usuarios, entre otros.

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 diseñó 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.

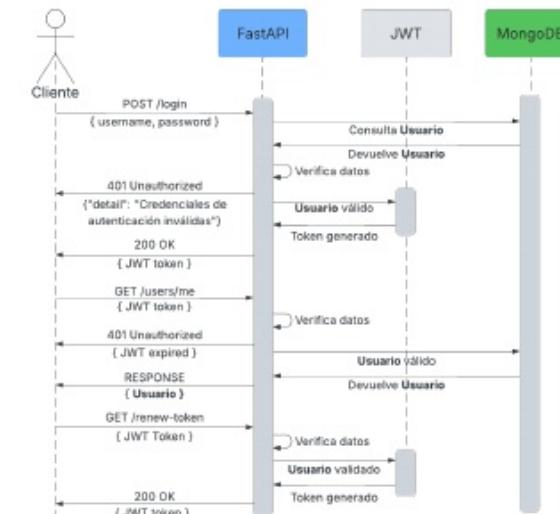


FIGURA 3.4. Esquema de autenticación y autorización.

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

3.4. Desarrollo del backend

21

La API REST posee los endpoints necesarios para realizar las operaciones de lectura de datos históricos y configuración de los nodos sensores y actuadores.

Para implementar el modelo de datos representado en la figura 3.2, se utilizó la librería Beanie, que permite definir modelos de datos de manera declarativa y realizar consultas de forma sencilla.

Autenticación y autorización: Es el componente encargado de gestionar la autenticación y autorización de los usuarios. Se implementó un sistema de autenticación basado en tokens con JWT, que permite a los usuarios acceder a la API de manera segura. Se estableció una duración para los tokens de acceso y de refresh, lo que permitió que los usuarios mantengan su sesión activa sin necesidad de autenticarse nuevamente con sus credenciales.

Integración con la base de datos: Es la integración de FastAPI con MongoDB para persistir la información. Se utilizó la biblioteca Motor para la conexión asíncrona con la base de datos. Beanie ODM proporciona una capa de abstracción sobre Motor, lo que facilita la interacción con la base de datos y permite definir modelos de datos de manera declarativa.

Conexión con el broker MQTT: Es el componente encargado de la comunicación bidireccional con los dispositivos IoT. Se implementó un cliente MQTT que se conecta a AWS IoT Core y gestiona la publicación y suscripción a los tópicos correspondientes.

Comunicación con frontend en tiempo real: Es el componente encargado de gestionar la comunicación en tiempo real entre el servidor y el cliente. Se implementó la librería WebSocket en el backend que permite a los clientes conectarse y recibir actualizaciones en tiempo real sobre los datos recopilados por los nodos.

Capa de datos

Es el componente encargado de persistir la información en la base de datos MongoDB. Se implementó un esquema de datos que permite almacenar los datos recopilados por los nodos sensores y actuadores, así como la información relacionada con los usuarios, ambientes y tipos de ambientes, entre otros. Los datos se almacenan en colecciones, donde cada colección representa un tipo de dato.

Frontend

Es la interfaz de usuario desarrollada en React. Permite a los usuarios interactuar con el sistema, visualizar los datos en tiempo real y gestionar la configuración de los nodos sensores y actuadores. Se implementó una interfaz responsive que se adapta a diferentes dispositivos y tamaños de pantalla.

La comunicación entre el frontend y el backend se realiza a través de la API REST expuesta por el servidor. Además, se implementó la comunicación en tiempo real mediante WebSocket, lo que permite a los usuarios recibir actualizaciones en tiempo real sobre los datos recopilados por los nodos sensores y actuadores.

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. Desarrollo del backen

21

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.

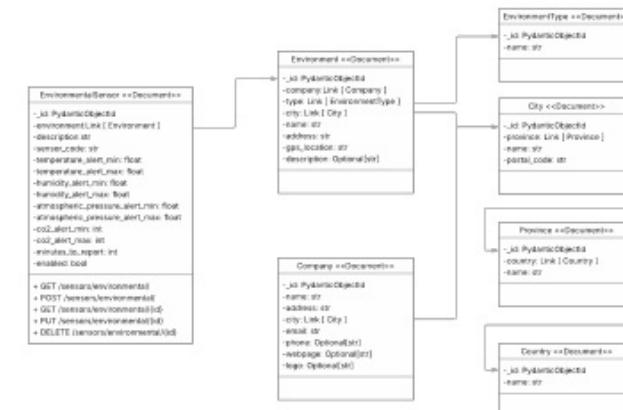


FIGURA 3.5. Diagrama de clases de los modelos implementados

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.

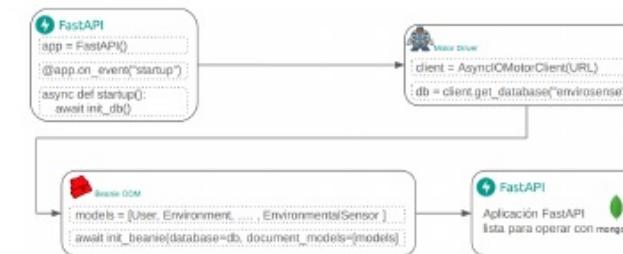


FIGURA 3.6. Pasos para la conexión de FastAPI y MongoDB

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

3.4.1. 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 3.2 presenta un resumen de los principales endpoints de la API, junto con una breve descripción de la acción y el método HTTP utilizado.

TABLA 3.2. Resumen de principales endpoints de la API

Método	Endpoint	Acción
POST	/login	hacer login
GET	/renew-token	renovar token
GET	/users/	obtener usuarios
POST	/users/	crear usuario
PUT	/users/	actualizar usuario
GET	/users/{id}	obtener un usuario
DELETE	/users/{id}	eliminar usuario
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	/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/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/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	/actuators/data/	datos históricos
GET	/sensors/environmental/data/	datos históricos
GET	/sensors/consumption/data/	datos históricos
GET	/sensors/nutrients/solution/data/	datos históricos

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.

3.4. Desarrollo del backend

23

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

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 B.

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

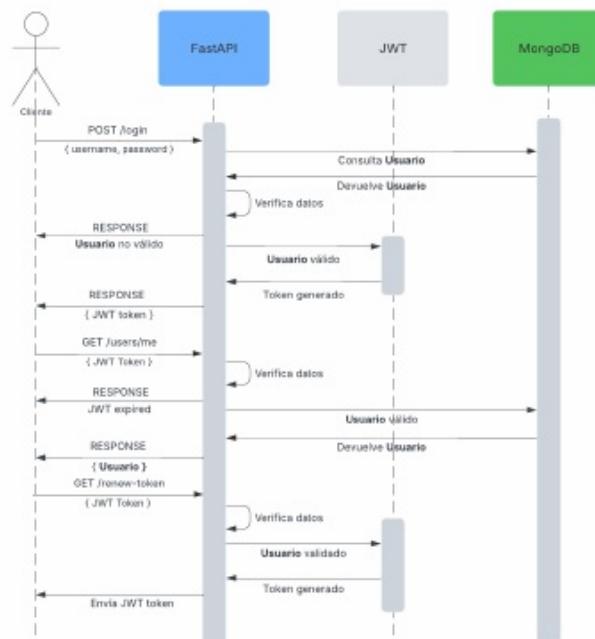


FIGURA 3.4. Esquema de autenticación y autorización.

3.4. Desarrollo del backend

23

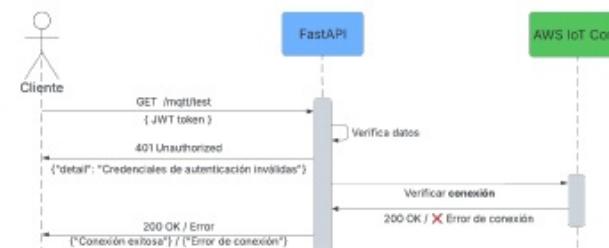


FIGURA 3.7. Pasos para verificar la conexión con el broker MQTT.

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

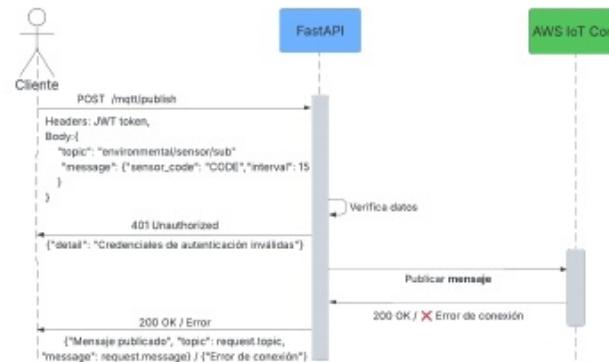


FIGURA 3.8. Pasos para publicar un mensaje en un tópico específico.

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.

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. Para su definición, se utilizó Beanie, que además permite establecer relaciones entre modelos, facilitando la creación de un esquema de datos estructurado y coherente.

El código 3.1 muestra un ejemplo de cómo se definen los modelos con Beanie y se exemplifica una relación entre los modelos con la clase *EnvironmentalSensor* y la clase *Environment* a través de un enlace (del inglés, *Link*).

```
from beanie import Document, Link, PydanticObjectId
from typing import Optional
from models.environment import Environment

class EnvironmentalSensor(Document):
    id: Optional[PydanticObjectId] = None
    environment: Link[Environment]
    description: str
    sensor_code: str
    temperature_alert_min: float
    temperature_alert_max: float
    humidity_alert_min: float
    humidity_alert_max: float
    atmospheric_pressure_alert_min: float
    atmospheric_pressure_alert_max: float
    co2_alert_min: float
    co2_alert_max: float
    minutes_to_report: int
    enabled: bool

    class Settings:
        collection_name = "environmental_sensors"

CÓDIGO 3.1. Ejemplo de definición de modelos con Beanie
```

Como se mencionó anteriormente, los datos se almacenan en MongoDB. La conexión a la base de datos se estableció con Motor, un cliente asíncrono para MongoDB, y Beanie, que facilitó la inicialización de modelos y la ejecución de consultas. La cadena de conexión consta de una URL que incluye el nombre de usuario, la contraseña y la dirección del servidor.

El código 3.2 muestra un ejemplo de cómo establecer la conexión e inicializar los modelos con Beanie.

```
async def init_db():
    client = AsyncIOMotorClient("mongodb://USER-PASSWORD@URL:PORT/")
    authSource="admin"
    db = client.get_database("envirosense")
    await init_beanie(database=db, document_models=[
        User, Role, Country, Province, City, Company, EnvironmentType,
        Environment, NutrientType, ConsumptionSensor,
        ConsumptionSensorData, ConsumptionSensorLog,
        EnvironmentalSensor, EnvironmentalSensorData,
        EnvironmentalSensorLog, NutrientSolutionSensor,
        NutrientSolutionSensorData, NutrientSolutionSensorLog,
        Actuator, ActuatorData, ActuatorLog
    ])

CÓDIGO 3.2. Ejemplo de conexión a MongoDB
```

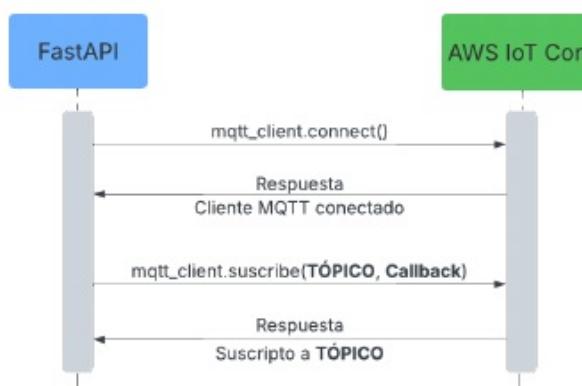


FIGURA 3.9. Pasos para la conexión del cliente MQTT.

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

3.5. Desarrollo del frontend

3.6. Desarrollo de nodos sensores y actuadores

3.7. Despliegue del sistema

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

A continuación, se detallan los pasos realizados en AWS IoT Core para configurar 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*, que permiten autenticar la conexión y cifrar los datos transmitidos. Estos certificados son necesarios para establecer una conexión 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.

La figura 3.5 muestra un ejemplo de cómo se definen las políticas de acceso en AWS IoT Core.

The screenshot shows the AWS IoT Core Policies page. The policy name is 'backend_connection-Policy'. It has two versions: Version 1 and Version 2, with Version 2 being the active one. The policy document is displayed in JSON format:

```

{
    "version": "2012-12-01",
    "statement": [
        {
            "effect": "Allow",
            "action": [
                "iot:Publish",
                "iot:Subscribe",
                "iot:Connect"
            ],
            "resource": "arn:aws:iot:us-east-1:554257980003:topic/backend_connection"
        }
    ]
}
  
```

FIGURA 3.5. Ejemplo de políticas de acceso en AWS IoT Core.

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 cliente, permitiendo un control granular sobre los permisos de acceso.

Apéndice A

Modelo de datos implementado en el trabajo

La figura A.1 muestra el modelo de datos implementado en el trabajo.

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 checar 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.

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 3.6 muestra una prueba de conexión al broker MQTT con Postman.

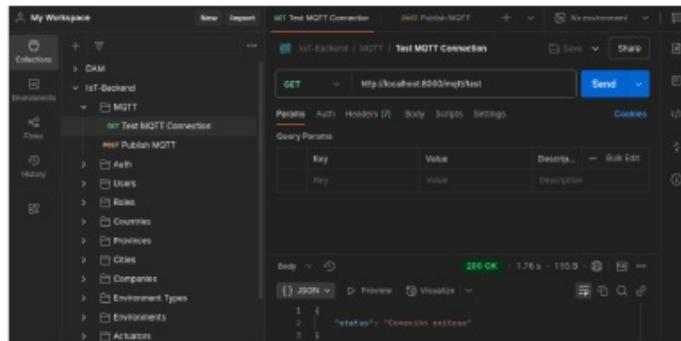


FIGURA 3.6. Pruebas en Postman. Test de conexión al broker MQTT.

La figura 3.7 muestra una prueba de publicación de un mensaje en un tópico con Postman.

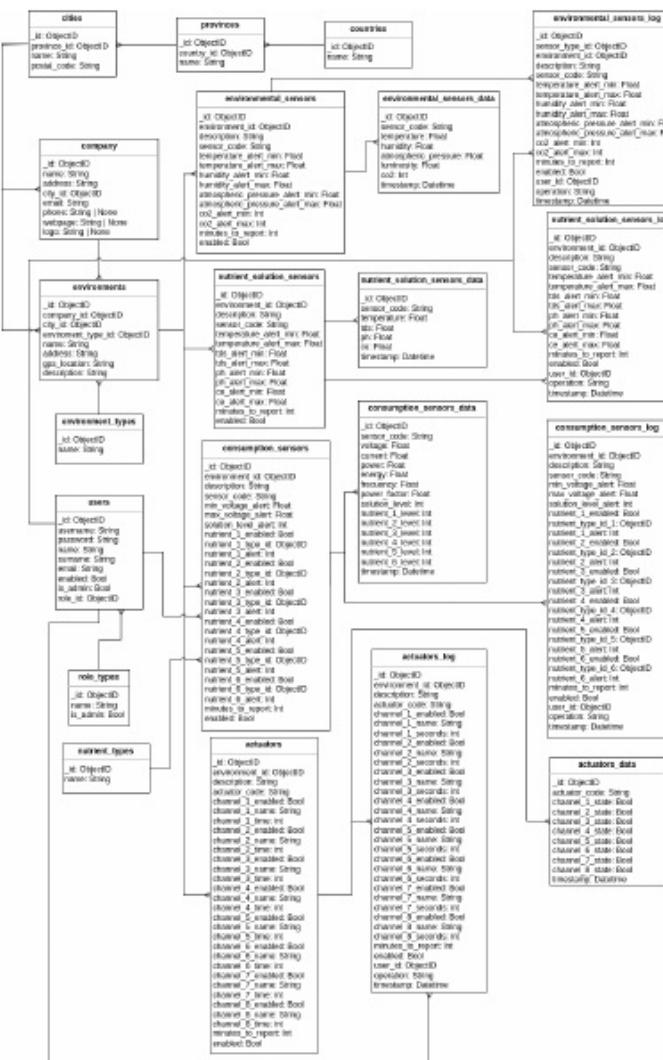


FIGURA A.1. Modelo de datos implementado en el trabajo.

3.4. Desarrollo del backend

27

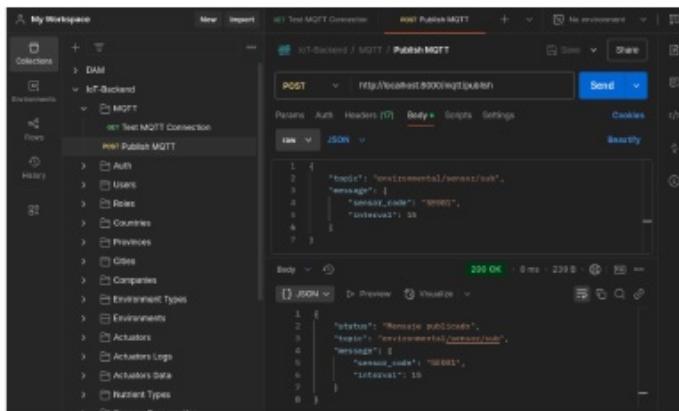


FIGURA 3.7. Pruebas en Postman. Publicación de un mensaje en un tópico.

La figura 3.8 muestra cómo la aplicación FastAPI se conecta al broker MQTT y se suscribe a los tópicos configurados. Se visualiza la interacción del cliente MQTT al recibir mensajes de los nodos sensores y actuadores, los cuales se almacenan en MongoDB. Además, se ejemplifica cómo la aplicación monitorea y registra un nuevo intervalo de envío de datos de un sensor, con lo que se destaca la comunicación entre el cliente MQTT y el servidor FastAPI.

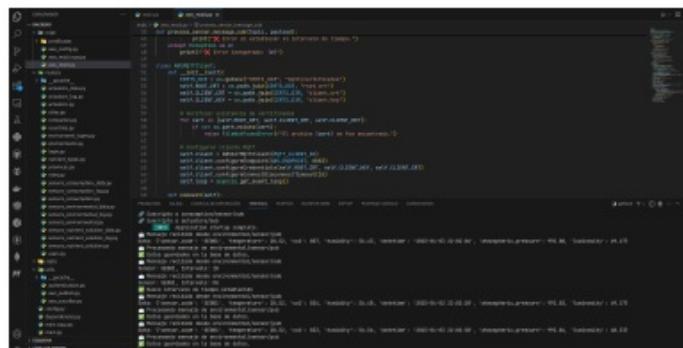


FIGURA 3.8. Cliente MQTT en FastAPI.

Para finalizar, la figura 3.9 muestra los datos almacenados en la base de datos MongoDB, que refleja la información presentada en la Figura 3.8.

27

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

The screenshot shows the MongoDB Compass interface with a connection to 'localhost:27017'. The left sidebar lists collections: 'My Queries', 'CONNECTIONS (0)', and 'EnvironmentalSensorData'. The main area displays three documents from the 'EnvironmentalSensorData' collection. Each document includes fields such as 'sensor_code', 'temperature', 'humidity', 'atmospheric_pressure', 'luminosity', 'cod', and 'dateTime'.

```

    {
      "_id": "64c81ec14f147e40556da30a57514f4a001",
      "sensor_code": "SEB01",
      "temperature": 28.52,
      "humidity": 50.43,
      "atmospheric_pressure": 991.88,
      "luminosity": 68.17,
      "cod": 1024,
      "dateTime": "2023-04-03T22:02:04.089+00:00"
    },
    {
      "_id": "64c81ec14f147e40556da30a57514f4a001",
      "sensor_code": "SEB01",
      "temperature": 28.52,
      "humidity": 50.43,
      "atmospheric_pressure": 991.83,
      "luminosity": 68.17,
      "cod": 1024,
      "dateTime": "2023-04-03T22:02:04.089+00:00"
    },
    {
      "_id": "64c81ec14f147e40556da30a57514f4a001",
      "sensor_code": "SEB01",
      "temperature": 28.52,
      "humidity": 50.43,
      "atmospheric_pressure": 991.84,
      "luminosity": 68.33,
      "cod": 1024,
      "dateTime": "2023-04-03T22:02:04.089+00:00"
    }
  
```

FIGURA 3.9. Datos almacenados en MongoDB.

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

3.4.5. Implementación de WebSockets

3.5. Desarrollo del frontend

3.6. Desarrollo de nodos sensores y actuadores

3.7. Despliegue del sistema

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

Apéndice A

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 A.1. Resumen de principales endpoints de la API

Método	Endpoint	Acción
GET	/api/	ruta por defecto
POST	/login	hacer login
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.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

TABLA A.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 A.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 autenticación 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": user})
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 deshabilitado"
32         )
33     return user

```

CÓDIGO C.1. Pseudocódigo del control de acceso

Apéndice B

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 autenticación 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": user})
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 B.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 C

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 C1 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 from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTClient
5 from models.sensor_environmental_data import EnvironmentalSensorData
6 from mqtt.aws_config import AWS_ENDPOINT, MQTT_CLIENT_ID
7
8 # Método asíncrono para insertar datos en la base de datos
9 async def insert_sensor_data(payload):
10     try:
11         sensor_data = EnvironmentalSensorData(**payload)
12         await sensor_data.insert()
13         print("Datos guardados en la base de datos.")
14     except Exception as e:
15         print("Error al guardar datos en la base de datos:", e)
16
17 # Método para procesar mensajes de sensores y actuadores
18 async def process_sensor_message_pub(topic, payload):
19     try:
20         print(f"Mensaje recibido desde {topic}")
21         print(f"Data: {payload}")
22
23         if topic in [
24             "environmental/sensor/pub",
25             "nutrient_solution/sensor/pub",
26             "consumption/sensor/pub",
27             "actuators/pub",
28         ]:
29             print(f"Procesando mensaje de {topic}")
```

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.

```
 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
```

36 Apéndice C. Conexión al broker MQTT con FastAPI y AWS IoT SDK para Python

```

30         await insert_sensor_data(payload)
31     except Exception as e:
32         print(f"Error inesperado: {e}")
33
34     # Método para procesar mensajes de control de sensores
35     def process_sensor_message_sub(topic, payload):
36         try:
37             sensor_code = payload.get("sensor_code")
38             interval = payload.get("interval")
39
40             print(f"Mensaje recibido desde {topic}")
41             print(f"Sensor: {sensor_code}, Intervalo: {interval}")
42
43             if str(interval) == "OK":
44                 print("Nuevo intervalo de tiempo establecido")
45             else:
46                 # print("Error al establecer el intervalo de tiempo.")
47         except Exception as e:
48             print(f"Error inesperado: {e}")
49
50     class AWSMQTTClient:
51         def __init__(self):
52             CERTS_DIR = os.getenv("CERTS_DIR", "mqtt/certificates")
53             self.ROOT_CRT = os.path.join(CERTS_DIR, "root.crt")
54             self.CLIENT_CRT = os.path.join(CERTS_DIR, "client.crt")
55             self.CLIENT_KEY = os.path.join(CERTS_DIR, "client.key")
56
57             # Verificar existencia de certificados
58             for cert in [self.ROOT_CRT, self.CLIENT_CRT, self.CLIENT_KEY]:
59                 if not os.path.exists(cert):
60                     raise FileNotFoundError(f"El archivo {cert} no fue encontrado.")
61
62             # Configurar cliente MQTT
63             self.client = AWSIoTMQTTClient(MQTT_CLIENT_ID)
64             self.client.configureEndpoint(AWS_ENDPOINT, 8883)
65             self.client.configureCredentials(self.ROOT_CRT, self.
66             CLIENT_KEY, self.CLIENT_CRT)
67             self.client.configureConnectDisconnectTimeout(10)
68             self.loop = asyncio.get_event_loop()
69
70         def connect(self):
71             print("Estableciendo conexión con AWS IoT Core...")
72             self.client.connect()
73             print("Cliente MQTT Conectado.")
74
75         def disconnect(self):
76             self.client.disconnect()
77             print("Cliente Desconectado")
78
79         def publish(self, topic, message):
80             payload = json.dumps(message) if isinstance(message, dict)
81         else str(message)
82             self.client.publish(topic, payload, 0)
83             print(f"Mensaje enviado a {topic}: {payload}")
84
85         def subscribe(self, topic, callback):
86             def wrapper(client, userdata, message):
87                 try:
88                     payload_str = message.payload.decode()
89                     payload = json.loads(payload_str)
90                     if topic in [ "environmental/sensor/sub",

```

36 Apéndice E. Conexión al broker MQTT con FastAPI y AWS IoT SDK para Python

```

4 from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTClient
5 from models.sensor_environmental_data import EnvironmentalSensorData
6 from mqtt.aws_config import AWS_ENDPOINT, MQTT_CLIENT_ID
7
8 # Método asíncrono para insertar datos en la base de datos
9 async def insert_sensor_data(payload):
10     try:
11         sensor_data = EnvironmentalSensorData(**payload)
12         await sensor_data.insert()
13         print("Datos guardados en la base de datos.")
14     except Exception as e:
15         print("Error al guardar datos en la base de datos:", e)
16
17     # Método para procesar mensajes de sensores y actuadores
18     def process_sensor_message_pub(topic, payload):
19         try:
20             print(f"Mensaje recibido desde {topic}")
21             print(f"Data: {payload}")
22
23             if topic in [
24                 "environmental/sensor/pub",
25                 "nutrient_solution/sensor/pub",
26                 "consumption/sensor/pub",
27                 "actuators/pub",
28             ]:
29                 print(f"Procesando mensaje de {topic}")
30                 await insert_sensor_data(payload)
31         except Exception as e:
32             print(f"Error inesperado: {e}")
33
34     # Método para procesar mensajes de control de sensores
35     def process_sensor_message_sub(topic, payload):
36         try:
37             sensor_code = payload.get("sensor_code")
38             interval = payload.get("interval")
39
40             print(f"Mensaje recibido desde {topic}")
41             print(f"Sensor: {sensor_code}, Intervalo: {interval}")
42
43             if str(interval) == "OK":
44                 print("Nuevo intervalo de tiempo establecido")
45             else:
46                 # print("Error al establecer el intervalo de tiempo.")
47         except Exception as e:
48             print(f"Error inesperado: {e}")
49
50     class AWSMQTTClient:
51         def __init__(self):
52             CERTS_DIR = os.getenv("CERTS_DIR", "mqtt/certificates")
53             self.ROOT_CRT = os.path.join(CERTS_DIR, "root.crt")
54             self.CLIENT_CRT = os.path.join(CERTS_DIR, "client.crt")
55             self.CLIENT_KEY = os.path.join(CERTS_DIR, "client.key")
56
57             # Verificar existencia de certificados
58             for cert in [self.ROOT_CRT, self.CLIENT_CRT, self.CLIENT_KEY]:
59                 if not os.path.exists(cert):
60                     raise FileNotFoundError(f"El archivo {cert} no fue encontrado.")
61
62             # Configurar cliente MQTT
63             self.client = AWSIoTMQTTClient(MQTT_CLIENT_ID)
64             self.client.configureEndpoint(AWS_ENDPOINT, 8883)

```

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

37

```

90         "nutrient_solution/sensor/sub",
91         "consumption/sensor/sub",
92         "actuators/sub"]:
93     process_sensor_message_sub(topic, payload)
94   else:
95     self.loop.create_task(process_sensor_message_pub
96       (message.topic, payload))
97     except json.JSONDecodeError:
98       print("Error al decodificar el mensaje JSON.")
99     except Exception as e:
100       print(f"Error inesperado en wrapper: {e}")
101
102     self.client.subscribe(topic, 1, wrapper)
103     print(f"Suscripto a {topic}")
104
105   def unsubscribe(self, topic):
106     self.client.unsubscribe(topic)
107     print(f"Desuscripto de {topic}")

```

CÓDIGO C.1. Cliente MQTT

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

```

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

37

```

63   self.client.configureCredentials(self.ROOT_CRT, self.
64     CLIENT_KEY, self.CLIENT_CRT)
65   self.client.configureConnectDisconnectTimeout(10)
66   self.loop = asyncio.get_event_loop()
67
68   def connect(self):
69     print("Estableciendo conexión con AWS IoT Core...")
70     self.client.connect()
71     print("Cliente MQTT Conectado!")
72
73   def disconnect(self):
74     self.client.disconnect()
75     print("Cliente Desconectado")
76
77   def publish(self, topic, message):
78     payload = json.dumps(message) if isinstance(message, dict)
79   else str(message)
80     self.client.publish(topic, payload, 0)
81     print(f"Mensaje enviado a {topic}: {payload}")
82
83   def subscribe(self, topic, callback):
84     def wrapper(client, userdata, message):
85       try:
86         payload_str = message.payload.decode()
87         payload = json.loads(payload_str)
88         if topic in [ "environmental/sensor/sub",
89           "nutrient_solution/sensor/sub",
90           "consumption/sensor/sub",
91           "actuators/sub"]:
92           process_sensor_message_sub(topic, payload)
93         else:
94           self.loop.create_task(process_sensor_message_pub
95             (message.topic, payload))
96         except json.JSONDecodeError:
97           print("Error al decodificar el mensaje JSON.")
98         except Exception as e:
99           print(f"Error inesperado en wrapper: {e}")
100
101       self.client.subscribe(topic, 1, wrapper)
102       print(f"Suscripto a {topic}")
103
104   def unsubscribe(self, topic):
105     self.client.unsubscribe(topic)
106     print(f"Desuscripto de {topic}")

```

CÓDIGO E.2. Cliente MQTT

El código E.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 # Inicializar cliente MQTT
4 mqtt_client = AWSMQTTClient()
5
6 @app.on_event("startup")
7 async def startup():

```

38 Apéndice C. Conexión al broker MQTT con FastAPI y AWS IoT SDK para Python

```

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):
36     mqtt_client.publish(request.topic, request.message)
37     return {"status": "Mensaje publicado", "topic": request.topic, "message": request.message}
38
39 # Endpoint para probar la conexión MQTT
40 @app.get("/mqtt/test")
41 def test_mqtt_connection():
42     try:
43         mqtt_client.connect()
44         return {"status": "Conexión exitosa"}
45     except Exception as e:
46         return {"status": "Error de conexión", "error": str(e)}
    
```

CÓDIGO C.2. Cliente MQTT en FastAPI

38 Apéndice E. Conexión al broker MQTT con FastAPI y AWS IoT SDK para Python

```

1   await init_db()
2   mqtt_client.connect()
3
4   # Suscripción a tópicos de publicación de sensores y actuadores
5   mqtt_client.subscribe("environmental/sensor/pub",
6                         process_sensor_message_pub)
7   mqtt_client.subscribe("nutrient_solution/sensor/pub",
8                         process_sensor_message_pub)
9   mqtt_client.subscribe("consumption/sensor/pub",
10                        process_sensor_message_pub)
11  mqtt_client.subscribe("actuators/pub", process_sensor_message_pub)
12
13  # Suscripción a tópicos de envío de parámetros a sensores y
14  # actuadores
15  mqtt_client.subscribe("environmental/sensor/sub",
16                        process_sensor_message_sub)
17  mqtt_client.subscribe("nutrient_solution/sensor/sub",
18                        process_sensor_message_sub)
19  mqtt_client.subscribe("consumption/sensor/sub",
20                        process_sensor_message_sub)
21  mqtt_client.subscribe("actuators/sub", process_sensor_message_sub)
22
23  @app.on_event("shutdown")
24  async def shutdown():
25      mqtt_client.disconnect()
26
27  # Clase para manejar la publicación de mensajes MQTT
28  class PublishRequest(BaseModel):
29      topic: str
30      message: dict
31
32  # Endpoint para publicar mensajes MQTT
33  @app.post("/mqtt/publish")
34  def publish_message(request: PublishRequest, user: dict = Depends(
35                       current_user)):
36      mqtt_client.publish(request.topic, request.message)
37      return {"status": "Mensaje publicado", "topic": request.topic, "message": request.message}
38
39  # Endpoint para probar la conexión MQTT
40  @app.get("/mqtt/test")
41  def test_mqtt_connection(user: dict = Depends(current_user)):
42      try:
43          mqtt_client.connect()
44          return {"status": "Conexión exitosa"}
45      except Exception as e:
46          return {"status": "Error de conexión", "error": str(e)}
    
```

CÓDIGO E.3. Cliente MQTT en FastAPI

- [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] Robert Hammelrath. *MicroPython Driver for BME280 Sensor*. URL: <https://github.com/robert-hh/BME280> (visitado 10-12-2024).
- [43] PinkInk. *BH1750 Light Sensor Library for MicroPython*. URL: <https://github.com/PinkInk/upylib/tree/master/bh1750> (visitado 10-12-2024).
- [44] overflo23. *MH-Z19 CO₂ Sensor Library for MicroPython*. URL: https://github.com/overflo23/MH-Z19_MicroPython (visitado 10-12-2024).
- [45] vezorgoat. *TDS Sensor Library for MicroPython*. URL: <https://github.com/vezorgoat/-Raspberry-Pi-TDS-Sensor> (visitado 10-12-2024).
- [46] GreenPonik. *EC Sensor Library for MicroPython*. URL: https://github.com/GreenPonik/GreenPonik_EC_Python (visitado 10-12-2024).
- [47] vezorgoat. *TDS Sensor Library for MicroPython*. URL: <https://github.com/vezorgoat/-Raspberry-Pi-TDS-Sensor> (visitado 10-12-2024).
- [48] Robert Hammelrath. *MicroPython DS18B20 OneWire Library*. URL: https://github.com/robert-hh/Onewire_DS18X20 (visitado 10-12-2024).
- [49] Roberto Sánchez C. *MicroPython Library for HC-SR04 Ultrasonic Sensor*. URL: <https://github.com/rsc1975/micropython-hcsr04r> (visitado 10-12-2024).
- [50] Jacopo Rodeschini. *PZEM-004T Library for MicroPython*. URL: <https://github.com/jacoporodeschini/PZEM-004T> (visitado 10-12-2024).
- [51] *FastAPI WebSockets Reference*. URL: <https://fastapi.tiangolo.com/reference/websockets/> (visitado 02-04-2025).
- [52] 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).
- [53] Inc. MongoDB. *Motor: Asynchronous Python driver for MongoDB*. URL: <https://motor.readthedocs.io/en/stable/index.html> (visitado 01-04-2025).
- [54] Beanie. *Asynchronous Python ODM for MongoDB*. URL: <https://beanie-odm.dev/> (visitado 01-04-2025).
- [55] React-Bootstrap. *Bootstrap components built with React*. URL: <https://react-bootstrap.netlify.app/> (visitado 02-04-2025).
- [56] Socket.IO. *Real-time application framework*. URL: <https://socket.io/> (visitado 02-04-2025).

- [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).