Desarrollo de una plataforma basada en micro-servicios en la nube pública para la orquestación de dispositivos IoT y un *gateway* en la red local para el acceso de los dispositivos IoT a la plataforma

Daniela Morales Ponce

UVG UNIVERSIDAD DEL VALLE DE GUATEMALA

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

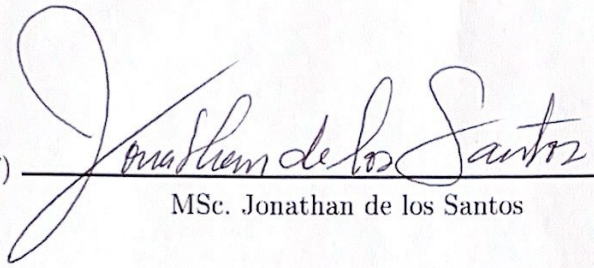
Desarrollo de una plataforma basada en microservicios en la nube pública para la orquestación de dispositivos IoT y un *gateway* en la red local para el acceso de los dispositivos IoT a la plataforma

Trabajo de graduación presentado por Daniela Morales Ponce para optar al grado académico de Licenciada en Ingeniería Electrónica

Guatemala,

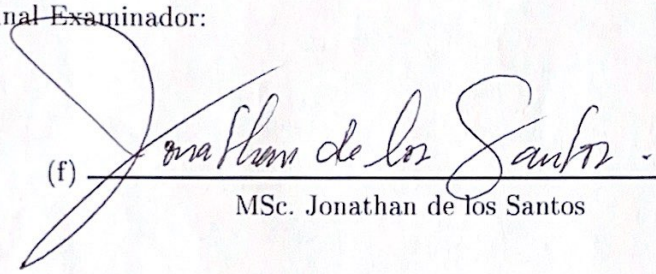
2024

Vo.Bo.:

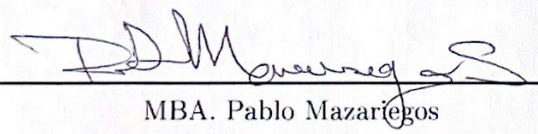
(f) 

MSc. Jonathan de los Santos

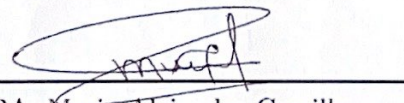
Tribunal Examinador:

(f) 

MSc. Jonathan de los Santos

(f) 

MBA. Pablo Mazariegos

(f) 

MBA. María Alejandra Carrillo

Fecha de aprobación: Guatemala, 13 de enero de 2024.

Índice

Lista de figuras

Lista de cuadros

Lista de códigos

Resumen

Este trabajo consiste en el desarrollo de una plataforma con la capacidad de conectarse a dispositivos IoT y manejar la información necesaria para poder controlarlos, con una interfaz por medio de la cual el usuario pueda gestionar y programar los dispositivos. Esta plataforma será probada en una red IoT conformada por microcontroladores ESP32, la comunicación se realizará por medio de distintos protocolos. El sistema se puede dividir en dos partes, las cuales son el *core* y el *edge*. El *core* se encuentra conformado por ocho microservicios y el *edge* es conformado por un microservicio y la red de dispositivos IoT.

El *core* es conformado por los microservicios siguientes: la base de datos de grafos donde se almacena la información de los dispositivos IoT, el *backend* en el que se maneja el comportamiento de los dispositivos IoT, la base de datos SQL del historiador, API del historiador, cron job para generar estadísticas del día, cron job para enviar correos electrónicos, tablero de Power BI y el *frontend*. Cabe mencionar que en el *edge* de la plataforma se encontrará un microservicio encargado de manejar la comunicación directa con los dispositivos IoT. Este microservicio es el *gateway* de los dispositivos IoT y es capaz de soportar varios protocolos IoT y transformar los mensajes recibidos de los dispositivos IoT a *Requests* HTTP a enviar al *core*.

Abstract

This work consists of the development of a platform with the ability to connect to IoT devices and manage the necessary information to control them, with an interface through which the user can manage and program the devices. This platform will be tested in an IoT network composed of ESP32 microcontrollers, and communication will be carried out through different protocols. The system can be divided into two parts, namely the core and the edge. The core is composed of eight microservices, while the edge consists of one microservice and the network of IoT devices.

The core includes the following microservices: the graph database where information about IoT devices is stored, the backend that handles the behavior of IoT devices, the SQL database of the historian, historian API, cron job for generating daily statistics, cron job for sending emails, Power BI dashboard, and the frontend. It is worth mentioning that in the edge of the platform, there will be a microservice responsible for handling direct communication with IoT devices. This microservice is the gateway for IoT devices and is capable of supporting various IoT protocols and transforming the messages received from IoT devices into HTTP requests to be sent to the core.

CAPÍTULO 1

Introducción

El Internet de las Cosas es una tecnología que permite la interconexión entre dispositivos ordinarios, permitiendo su comunicación para manipular el comportamiento de los mismos y obtener información de ellos. Dichos dispositivos se pueden clasificar en sensores y actuadores, la primera clasificación se encarga de obtener información del entorno y la segunda de manipular el entorno. A partir de estos dispositivos, se pueden crear aplicaciones que permitan automatizar procesos, la magnitud de la red de dispositivos IoT depende de la aplicación. Es decir, se pueden crear redes de dispositivos IoT que tomen como variables de entrada los valores de los sensores y las variables de salida del sistema sean el valor de cada actuador. Sin embargo, en el caso de aplicaciones que requieran una gran cantidad de dispositivos, es necesaria una plataforma que permita la orquestación de los mismos, puesto que la complejidad de la red aumenta y se requiere una herramienta que permita la administración de los dispositivos para evitar la programación manual de cada dispositivo. En este trabajo se desarrolla una plataforma basada en microservicios en la nube pública para la orquestación de dispositivos IoT y un *gateway* en la red local para el acceso de los dispositivos IoT a la plataforma. Por lo tanto, en esta tesis se propone una arquitectura basada en microservicios, los cuales son: *frontend*, *backend*, base de datos de grafos, *gateway*, API del historiador, base de datos SQL, tablero, *cron job* para guardar estadísticas y *cron job* para enviar correos.

Las redes de dispositivos IoT mencionadas previamente conforman sistemas de automatización de procesos, dichos procesos pueden ser de distintas índoles, como la domótica, automatización industrial, automatización de procesos, entre otras. Sin embargo, independientemente del tipo de proceso, dicho sistema puede tener una magnitud que genere la necesidad de una plataforma para la orquestación de los dispositivos IoT. Por lo tanto, dicha plataforma es quien debe de tener acceso al valor de entrada y salida de todos los dispositivos, ya que en esta arquitectura es el *backend* quien contiene la inteligencia del sistema. Como variables de entrada del sistema, se toman los valores de los sensores en el sistema físico y las acciones realizadas por el usuario en el *frontend*. Cabe mencionar que el funcionamiento del *backend* depende de la base de datos de grafos, puesto que es en esta donde se almacenan las variables de entrada y donde se almacenan los resultados de las operaciones realizadas en el *backend*.

Como se mencionó, es el *backend* quien recibe todas las entradas del sistema y actualiza el valor de las salidas del sistema, por lo tanto, también es el encargado de responder a las peticiones del *frontend* y de los dispositivos IoT. Es decir, el *backend* recibe cambios a la lógica del sistema por parte del *frontend* y actualiza la base de datos de grafos, además, recibe los valores de los sensores y actualiza la base de datos, por último, recibe las peticiones de los actuadores y responde con el valor actualizado. En paralelo, se tiene el subsistema historiador, el cual se encarga de almacenar los valores de los sensores en una base de datos SQL. Y esta información se ingesta información al *frontend*, al igual que al tablero de Power BI y el *cron job* que se encarga de enviar correos a los usuarios sobre el uso de su cuenta.

Para simplificar la comunicación entre los dispositivos IoT, el *backend* y el historiador se implementó un *gateway*. Este *gateway* es un servidor que se encuentra en la red local y que recibe todas las solicitudes y actualizaciones de los dispositivos IoT y las reenvía al *backend* como una petición HTTP. Por lo tanto, el *gateway* debe de poder soportar todos los protocolos que son utilizados por los dispositivos IoT y el protocolo utilizado por el *backend*, es decir, HTTP. El uso del *gateway* simplifica la incorporación de protocolos soportados por el sistema y permite que el funcionamiento del *backend* sea independiente de los protocolos utilizados por los dispositivos IoT y de la estabilidad de los mismos, ya que el objetivo es que los microservicios sean ligeramente acoplados. Este componente también se encarga de obtener los *tokens* de autenticación del usuario, a partir de su nombre de usuario y contraseña. Al igual, que envía los *tokens* en cada petición HTTP y los actualiza cuando es necesario. Sin embargo, es posible que los dispositivos IoT se comuniquen con el *backend* directamente únicamente por medio de HTTP, pero este tendría que manejar los *tokens* de autenticación y enviarlos en el *header* de la petición, aumentando la complejidad del sistema.

Por lo tanto, la arquitectura propuesta consta de sensores IoT que envían su valor al *backend* y al historiador por medio del *gateway*, actuadores IoT que solicitan su valor al *backend* por medio del *gateway*, el *frontend* que envía la lógica del sistema al *backend* y el *backend* que recibe las entradas del sistema y actualiza la base de datos. Además, se tienen *cron jobs* que se encargan de enviar correos a los usuarios y de guardar estadísticas del uso de la plataforma y un tablero de Power BI que se conecta a la base de datos SQL del historiador para mostrar las estadísticas de uso de la plataforma.

CAPÍTULO 2

Antecedentes

Actualmente en el departamento de Ingeniería electrónica, mecatrónica y biomédica de la Universidad del Valle de Guatemala no se tiene una línea de investigación de telecomunicaciones, IoT o de las tecnologías a utilizar en este proyecto, las cuales son MQTT, Django, FastAPI, Docker y Kubernetes. En cuanto a antecedentes externos a la universidad, existen diversos proyectos y sistemas desarrollados para la orquestación de dispositivos IoT o que utilicen las tecnologías mencionadas previamente.

Primero, cabe mencionar que actualmente los principales proveedores de nube (Azure, AWS y Google Cloud) cuentan con servicios de orquestación de dispositivos IoT [1]. Dichos servicios permiten a los usuarios tener toda la información de sus dispositivos IoT en la nube y procesarla por medio de otros servicios del mismo u otro proveedor [2] [3] [4].

Existen varios proyectos realizados previamente que implementan arquitecturas de microservicios para la orquestación de dispositivos IoT, utilizando protocolos de comunicación como *MQTT* y bases de datos de grafos. Un ejemplo de esto es el proyecto presentado en el artículo *"MQTT based home automation system using ESP8266"* [5], el cual describe el desarrollo y resultados de un proyecto que implementa un sistema automatizado conformado por sensores y actuadores. Los sensores y actuadores son implementados en el microcontrolador ESP8266 y utilizando el protocolo de comunicación *MQTT* [5]. El objetivo de este prototipo es implementar la comunicación de sensores y actuadores, implementados en ESP8226, utilizando *MQTT*. Esto se realiza conectando los sensores y actuadores al ESP8266 y conectando este a Mosquitto, el cual es un *broker MQTT*, para establecer un monitoreo y control remoto. La red implementada consiste de un cliente *MQTT* implementado en ESP8266. El microcontrolador tiene un sensor y dos actuadores, un led y un *buzzer*. Por medio de esta red se tiene un forma de controlar actuadores y obtener la información de sensores en dispositivos por medio de una interfaz, sin embargo, por medio de esta solución se debe de configurar un *topic* distinto manualmente por cada sensor o actuador en cada microcontrolador.

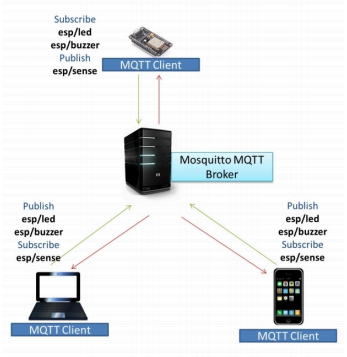


Figura 1: Arquitectura de la red *MQTT* implementada con ESP

En el artículo *"An open IoT framework based on microservices architecture"* se describe un *framework* compuesto por microservicios desarrollados con contenedores Docker y utilizando el orquestador Kubernetes[6]. Este framework es capaz de orquestar dispositivos IoT de una forma simplificada para el usuario, manejando a los dispositivos IoT por medio de sus *plugins* y utilizando comunicación asíncrona para mantener la independencia entre microservicios. La arquitectura y comunicación de microservicios utilizada en este *framework* se muestra en la Figura 2. Cada microservicio mostrado en la arquitectura es uno o varios contenedores Docker ejecutados a partir de una imagen, todos los contenedores son orquestados por medio de Kubernetes. Se menciona en el articulo que el *framework* todavía cuenta con problemas como: procesamiento de transacciones, retardo de la red, fallos en la red, serialización de mensajes, entre otros.

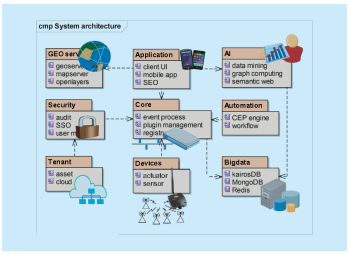


Figura 2: Arquitectura del framework

En cuanto al uso de bases de datos de grafos, esta base datos puede ser utilizada en sistemas conformados por dispositivos IoT para representar distintos factores de este sistema, lo cual se realizó en el proyecto presentado en el artículo *"Real-time Processing of IoT Events using a Software as a Service (SaaS) Architecture with Graph Database"*[7]. El sistema implementado en este proyecto se basa en el hecho que en los sistemas IoT se tiene información relevante de los dispositivos, ubicaciones, usuarios, entre otros y esta información se encuentra interconectada. Por lo tanto, una base de datos de grafos es ideal para representar conexiones entre distintos tipos de entidades, puesto que en las bases de datos de grafos las relaciones incluyen información. Por otro lado, las bases de datos relacionales y otras bases de datos NoSQL no son optimizadas para información de conexiones.

El sistema consiste de tres secciones, la primera es el envío de mensajes a un servidor

de mensajería asíncrona (ejabberd), la segunda es el procesamiento de los eventos en *apache spark* y el último es la ejecución de un *query* en la base de datos Neo4j desde *apache spark*. Es decir, el sistema es orientado a mensajes y la información del sistema IoT se encuentra en una base de datos de grafos para mejorar la eficiencia del almacenamiento y obtención de la información.

A partir de estos proyectos se puede concluir que es posible implementar un sistema de orquestación de dispositivos IoT utilizando microservicios contenerizados en Docker y orquestados en Kubernetes, por medio de una plataforma que utiliza una base de datos de grafos para la gestión de dispositivos y se comunica con los mismos por medio de protocolos IoT como MQTT.

CAPÍTULO 3

Justificación

Este trabajo aporta una forma automatizada de suscribir, programar y obtener información de dispositivos IoT. El trabajo es dirigido hacia aplicaciones con redes de dispositivos IoT de grande escala, por lo que deben de recibir y enviar información a un controlador, el cuál debe de programar el comportamiento de un sistema, es decir, manipular a los actuadores en base a la información de los sensores.

Al implementar la plataforma en un sistema, esta asume el rol de un controlador que puede automatizar el proceso de instanciar sensores y actuadores al sistema. Este trabajo tiene un aporte tanto en el departamento de ingeniería electrónica de la Universidad del Valle de Guatemala como en Guatemala y sus empresas. El aporte de este trabajo para el departamento de ingeniería electrónica de la Universidad del Valle de Guatemala es que se construyeron las bases y realizaron recomendaciones para nuevos proyectos de microservicios e IoT. En el proceso de desarrollar la plataforma se indagaron en tecnologías que no se han explotado en el país, lo cual puede despertar un interés en las empresas locales a usar tecnologías modernas para mejorar y optimizar el uso de sus recursos.

CAPÍTULO 4

Objetivos

4.1. Objetivo general

Desarrollar una herramienta de software para la orquestación de dispositivos IoT conformada por microservicios ligeramente acoplados.

4.2. Objetivos específicos

- Automatizar y simplificar el proceso de registrar dispositivos IoT a una plataforma de orquestación, el proceso de solicitar el valor de sensores y el proceso de cambiar el estado de actuadores.
- Desarrollar en Django y contenerizar en Docker un microservicio encargado del registro y control de los dispositivos.
- Implementar una base de datos de grafos en la cual, el microservicio encargado de la gestión de los dispositivos almacenará la información de dichos dispositivos y su comportamiento.
- Desarrollar una interfaz gráfica por medio de la cual el usuario pueda describir el sistema de dispositivos IoT.
- Desarrollar en Python un microservicio encargado de actuar como un *gateway* hacia la plataforma para los sensores y actuadores.
- Implementar la comunicación de la plataforma con los dispositivos IoT por medio de distintos protocolos IoT.
- Desplegar el *Backend* y *Frontend* de la plataforma en contenedores Docker orquestados en Kubernetes por medio de Okteto.

CAPÍTULO 5

Alcance

El alcance de este proyecto es proporcionar una solución completa para la gestión de sistemas físicos automatizados por medio de dispositivos IoT. Por solución completa se entiende que el usuario solamente tendrá la responsabilidad de registrar los dispositivos a la plataforma, asignarle su comportamiento, levantar el *gateway* en su red local y conectar los dispositivos IoT al *gateway*. El proceso de conexión de los dispositivos IoT al *gateway* que deberá de realizar el usuario depende del protocolo por medio del cual se esta realizando la conexión.

El sistema físico con el que se realizarán las pruebas de la plataforma y *gateway* es un prototipo de escala reducida, sin embargo, el sistema puede ser escalado a un sistema industrial. El *gateway* de los dispositivos IoT soportará una cantidad reducida de protocolos IoT, sin embargo, el *gateway* puede ser extendido para soportar otros protocolos IoT.

CAPÍTULO 6

Marco teórico

6.1. Internet de las Cosas (IoT)

El Internet de las Cosas es una red de objetos con sensores o actuadores embebidos, con la capacidad de enviar y recibir información a través de Internet. [8] IIoT es la interconexión de dispositivos utilizados en aplicaciones industriales para mejorar la eficiencia y procesos [9].

En cuanto a protocolos IoT existen dos tipos, correspondientes a distintas capas del modelo de Internet, los protocolos IoT de la capa de red y los protocolos IoT de la capa de datos [10]. Estas capas mencionadas se basan en el *stack* de el modelo TCP/IP de 5 capas. Este modelo es una forma de dividir las capas que conformar la comunicación entre dispositivos, dicho modelo esta conformado por las siguientes capas:

1. Capa física: la encargada de la conexión física entre dispositivos.
2. Capa de enlace de datos: la encargada de la conexión lógica entre dispositivos conectados directamente.
3. Capa de red: la encargada de enrutar los paquetes de datos a través de la red.
4. Capa de transporte: la encargada de la comunicación entre dispositivos finales. En internet existen dos protocolos de transporte, TCP y UDP.
5. Capa de aplicación: la encargada de la comunicación entre aplicaciones.

Entre los protocolos de la capa de red IoT se encuentran: WiFi, LTE CAT 1, LTE CAT M1, NB-IoT, Bluetooth, Zigbee, LoRaWAN, entre otros[10]. Y entre los protocolos IoT de la capa de datos, comúnmente usados, se encuentran: MQTT, CoAP, AMQP, XMPP, HTTP, entre otros.

6.1.1. MQTT

Message Queuing Telemetry Transport se utiliza para aplicaciones IoT que requieren mensajería confiable y escalable, para dispositivos de baja potencia que deben de conectarse al *broker* y publicar mensajes a un *topic*. MQTT aplica filtrado basado en *topics* y utiliza los *topics* determinar a que clientes debe de ser entregado cada mensajes[11]. Es decir, se utiliza para la transmisión de data en medios con bajo ancho de banda, utilizando recursos mínimos. Los *topics* mencionados consisten de uno o más niveles de *topics* separados por una barra, es decir, como un directorio. Un cliente puede conectarse a uno o varios *topics* haciendo uso de *wildcards* El *broker* de MQTT no permite que los mensajes sean archivados, por lo que los mensajes se envían a los clientes conectados en el momento. Este protocolo es mayormente adecuado para redes grandes conformadas por dispositivos con recursos restringidos que deben de ser monitoreados desde un servidor en Internet[12].

En cuanto a la seguridad del protocolo, los mensajes de la conexión TCP son encriptados con SSL/TLS. Y es posible configurar el *broker* para que los clientes deban de autenticarse con un nombre de usuario y contraseña[13].

Cabe mencionar que la entrega de mensajes se realiza acorde al nivel de *Quality of Service* definido. Hay tres posibles niveles, en el nivel el mensaje es entregado un máximo de 1 vez, en el nivel 1 el mensaje es entregado por lo menos 1 vez y en el nivel 2 el mensaje es entregado exactamente 1 vez.

Los mensajes de control del protocolo MQTT están conformados por tres partes. Las cuales son el *header* fijo, el *header* variable y el *payload*. El *header* fijo se encuentra en todos los paquetes de control MQTT. Mientras que el *header* y *payload* se encuentra en solamente algunos. Por ejemplo, en el caso del paquete de control *PUBLISH* el *payload* contiene el mensaje que se está publicando. Los tipos de paquetes de control son los siguientes[14]:

- *Connect:* Establecer una conexión entre el cliente y el *broker*.
- *Connack:* Confirmación de conexión.
- *Publish:* Publicar un mensaje en un *topic* en el *broker*.
- *Puback:* Publicación recibida, primera parte de la entrega cuando se tiene un QoS de 2.
- *Pubrel:* Liberación de la publicación, segunda parte de la entrega cuando se tiene un QoS de 2.
- *Pubcomp:* Publicación completada, ultima parte de la entrega cuando se tiene un QoS de 2.
- *Subscribe:* Suscribir a un cliente a un *topic* en el *broker*.
- *Suback:* Confirmación de suscripción.
- *Unsubscribe:* Permitir que el cliente se desuscribra de algún *topic* en el *broker*.
- *Unsuback:* Confirmación de cancelación de suscripción.

- *Pingreq:* Solicitud de ping.
- *Pingresp:* Respuestos de ping.
- *Disconnect:* Notificación de desconexión.
- *Auth:* Intercambio de autenticación.

6.1.2. UDP

UDP (*User Datagram Protocol*) es un protocolo de transporte sin conexión, por lo que no se establece una conexión entre el cliente y el servidor, cuyo objetivo es transmitir datagramas[15]. Este protocolo se utiliza para aplicaciones que requieren una transmisión de datos rápida y eficiente, ya que prioriza la velocidad sobre la integridad de los datos. Para priorizar la velocidad, UDP no tiene un mecanismo de control de flujo ni confirmación, por lo que, los datagramas pueden llegar en un orden distinto al receptor y el emisor no tiene una forma de confirmar la recepción de los datagramas.

6.1.3. Bluetooth

Bluetooth es un protocolo de comunicación inalámbrica que permite la comunicación entre dispositivos en distancias cortas[16]. Una ventaja de este protocolo es su bajo consumo de energía, lo que lo hace un protocolo adecuado para dispositivos IoT en ciertos casos.

6.1.4. HTTP

Hypertext Transfer Protocol se utiliza para comunicación basada en web entre servidores y clientes. Existen 2 tipos de mensajes HTTP, estos son *request* y *response*. El *request* es el mensaje que el cliente envía al servidor y el *response* es la respuesta. En HTTP, el servidor envía la *data* a traves de *URI (Universal Resource Identifier)* y el cliente recibe la data por medio del mismo. Cabe mencionar que este protocolo no se recomienda para redes con dispositivos de recursos restringidos, debido a que requiere un alto uso de recursos. Los métodos de este protocolo son[12]:

- *Get:* Recuperar información de un recurso específico.
- *Post:* Cambiar la información de un recurso en específico.
- *Head:* Obtener los *headers* de un recurso.
- *Put:* Reemplazar un recurso en específico.
- *Patch:* Aplicar modificaciones parciales a un recurso en específico.
- *Delete:* Eliminar un recurso en específico.

Los mensajes de tipo *request* están conformados por dos partes, las líneas de *request* y las líneas de *header*. La linea de *request* está conformada por tres campos: el campo del método, el campo de *URL*, el campo de versión HTTP. En el campo del método, mencionado previamente, es donde se indica cuál de los métodos enlistados anteriormente es el *request*. En cuanto al *response*, cabe mencionar que uno de los campos que se incluye en el mensaje es el *status code*. El *status code* es de importancia, puesto que indica el estado del *request* que previamente hizo el cliente, este código está compuesto por tres dígitos. El primero de estos digitos indica la clase de *response*, estas clases son[17]:

- 1xx: *response* informativo
- 2xx: *response* exitoso
- 3xx: Redireccionamiento
- 4xx: Error en el *request*
- 5xx: Error en el servidor

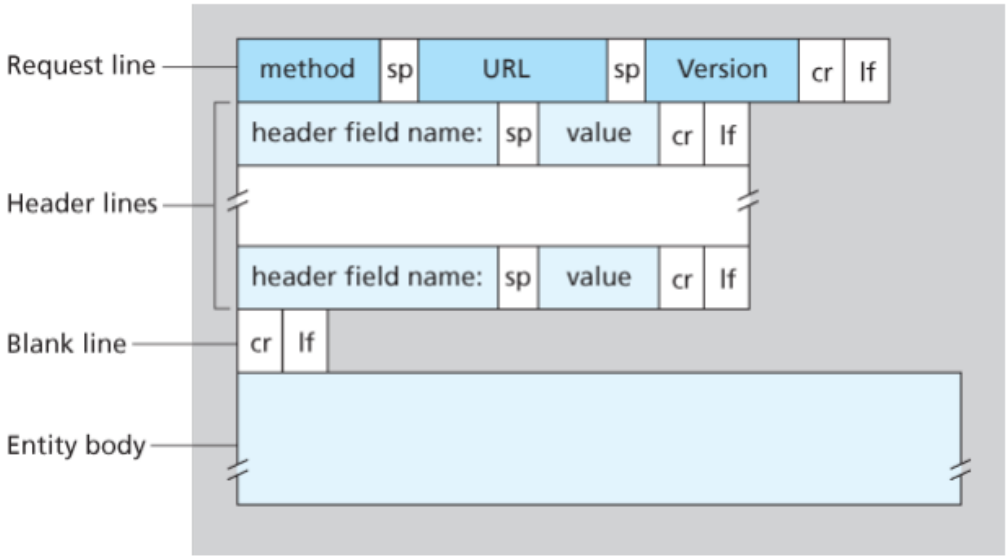


Figura 3: Formato general de un *request* HTTP [15]

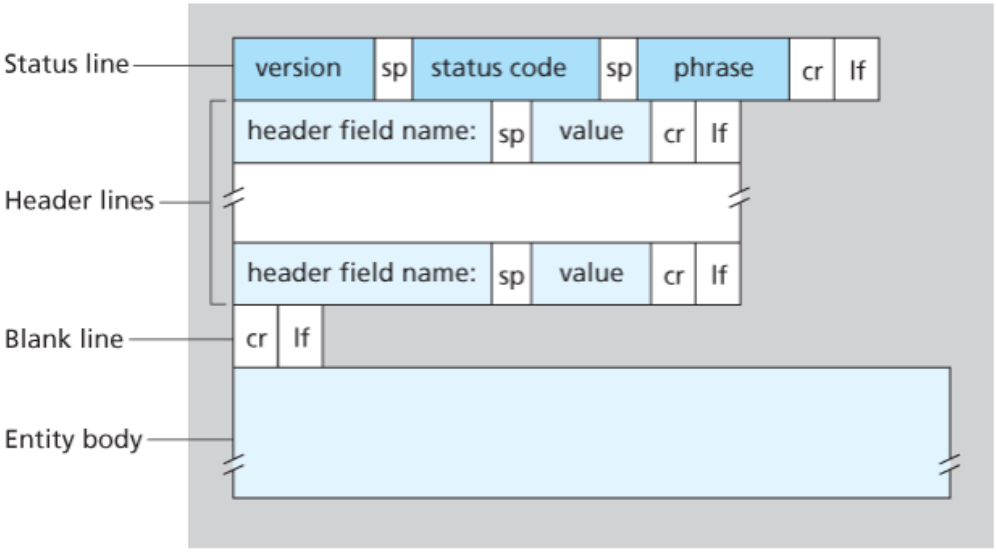


Figura 4: Formato general de un *response* HTTP [15]

Este protocolo se diseña para la comunicación directa entre dos dispositivos, por lo que, no es recomendado para aplicaciones IoT en las que existe un gestionador centralizado, puesto que si es un sistema centralizado el servidor necesita comunicar el mismo mensaje a varios dispositivos IoT[13].

Cabe mencionar que, para una arquitectura en la que potencialmente se requiere enviar el mismo mensaje a varios dispositivos, la utilización de un protocolo con *publish/subscribe*como modelo de comunicación mejoraría el rendimiento de la aplicación puesto que solamente es necesario enviar el mensaje una vez y el *broker* se encarga de distribuirlo a los dispositivos suscritos.

Los dispositivos IoT son dispositivos que conectan objetos físicos a internet. Constan con capacidad computacional, de recopilar datos del entorno, de comunicarse con otros dispositivos y de interactuar con los usuarios[9]. Estos dispositivos pueden ser sensores, actuadores, microcontroladores, entre otros. En el caso de microcontroladores, estos deben de tener capacidad de conectarse a internet, un ejemplo de microcontrolador con capacidad de ser dispositivo IoT sería el ESP32.

Microcontrolador ESP32

ESP32 es un microcontrolador con conectividad WiFi y Bluetooth integrada, lo cual lo hace ideal para implementar comunicación IoT al conectarle sensores y actuadores a sus pines de entrada y salida. Este microcontrolador es desarrollado por Espressif Systems y es una opción popular debido a su versatilidad, bajo consumo de energía y capacidad para conectarse a redes inalámbricas. [18]

El ESP32 DevKit v1 cuenta con los siguientes periféricos:

- 34 pines de entrada/salida.
- SAR-ADC de 12 bits.
- 10 sensores *touch*.
- 4 SPI
- 2 I2S
- 2 I2C
- 3 UART
- 1 *host* SD/eMMC/SDIO
- 1 *slave* SDIO/SPI
- Ethernet MAC *interface*
- TWAI, compatible con ISO 11898-1
- RMT (tx/rx)
- Motor PWM
- LED pwm de hasta 16 canales

Dado que este microcontrolador cuenta con la capacidad de establecer conexión a una red de internet, con pines para conectar sensores y actuadores, resulta idóneo para ser utilizado como dispositivo IoT en redes de prototipo. En la Figura 5, se puede observar el microcontrolador ESP32 DevKitM-1.

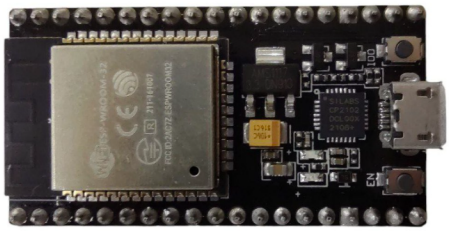


Figura 5: Placa ESP32 DevKit v1

6.2. Restful APIs

Las API REST *(Representational State Transfer)* son un servicio web que utiliza *requests* HTTP para obtener, modificar, crear y borrar data. Las API REST se ajustan a los límites de arquitectura REST, que son los siguientes[19]:

- **Arquitectura cliente-Servidor:** El cliente y el servidor deben de ser independientes entre sí. El cliente es quien realiza las peticiones y el servidor es quien las recibe y procesa.
- **Sin estado:** Cada petición API depende es manejada independientemente, por lo que capa petición debe contener toda la información necesaria.
- **Interfaz uniforme:** La interfaz debe de acomodarse facilmente a todos los *softwares* conectados, para esto debe de establecer reglas sobre como el cliente y el servidor deben de interactuar.
- **Sistema por capas:** Para que la API sea escalable, la arquitectura REST dicta que el diseño debe ser estructurado en capas que operan conjuntamente. Con una jerarquía de capas, las capas deben de comunicarse entre sí pero no con todos los componentes del programa, por lo que, si se desea cambiar una capa, no se afecta a las demás.
- **Cacheable:** El *overhead* de las APIs sin estado generalmente son grandes, por lo que, para peticiones repetidas es recomendable guardar esta información en caché.
- **Código bajo demanda (opcional):** Los servidores pueden proporcionar código ejecutable al cliente para extender su funcionalidad. Es opcional debido a que pueden implicar un riesgo de seguridad.

6.2.1. Django

Django es un framework de código abierto de desarrollo web escrito en Python para el desarrollo de aplicaciones web y está basado en el patrón de diseño Modelo-Vista-Controlador (MVC).[20] Este modelo MVC separa la aplicación en tres componentes interconectados, los cuales son el modelo, la vista y el controlador. El modelo es la representación de la *data* y lógica de negocio de la aplicación. El framework de Django utiliza un archivo llamado `models.py` para definir los modelos de la aplicación para transferirlos al esquema de la base de datos. La vista es la representación visual de la información, es responsable de renderizar la información al usuario. En caso de usar Django para realizar APIs, se utiliza el Django Rest Framework para realizar las vistas en el archivo `views.py` y un archivo llamado `serializers.py` para mostrar la información de los modelos en formato JSON. El controlador es el encargado de recibir las peticiones del usuario y de llamar a los modelos y vistas correspondientes para procesarlas. [20]

6.2.2. FastAPI

FastAPI es un *framework* para construir APIs con Python 3.8 en adelante, con las siguientes características[21]:

- Alto rendimiento
- Reduce al rededor del 40 % de los errores causados por humanos
- Intuitivo, fácil y rápido de desplegar

- Basado en estándares

6.3. Bases de datos

Las bases de datos son una herramienta para almacenar información estructurada. Existen distintos tipos de bases de datos, entre los cuales cabe mencionar los siguientes[22]:

- Relacional: La información es estructurada en tablas, las cuales tienen relaciones entre ellas.
- Orientada a objetos: La información es representada como objetos, similar a la programación orientada a objetos.
- Distribuida: Los archivos se encuentran almacenados en distintas ubicaciones.
- *Data warehouse*: Repositorio de *data* diseñado para poder realizar analisis y *queries* rápidamente.
- NoSQL: Almacena y permite la manipulación de *data* no estructurada y semi-estructurada, al contrario de bases de datos SQL que solamente permite *data* estructurada.
- Grafos: La *data* es estructurada en nodos con etiquetas y existen relaciones con atributos entre los nodos.

6.3.1. Neo4j

La base de datos Neo4j es una base de datos nativa de grafos *open-source*. Debido a que es nativa, implementa un modelo de grafos hasta el nivel de almacenamiento[23]. Esto causa que el rendimiento de Neo4j sea mejor y la base de datos sea más flexible, en comparación con las bases de datos que almacenan la información como una abstracción de grafos. Para realizar *queries* directamente a la base de datos se utiliza Cypher, que es un lenguaje de consulta declarativo similar a SQL pero optimizado para grafos[23]. Sin embargo, por medio de librerías es posible utilizar neo4j desde lenguajes de programación o *frameworks*. Por ejemplo, por medio de la librería Neomodel se pueden realizar clases de Python para representar nodos y relaciones en la base de datos, obtener la información y manipularla. Neomodel es un mapeador gráfico de objectos (OGM) diseñado para Neo4j. Esta librería fue desarrollada sobre el driver de Neo4j para python[24]. Cabe mencionar que AuraDB ofrece una base de datos Neo4j como un servicio manejado, sin embargo, Neo4j se puede utilizar en un ambiente local con la edición *community* o *enterprise*.

6.3.2. PostgreSQL

PostgreSQL es una base de datos relacional *open-source* que utiliza SQL o JSON para realizar consultas[25]. PostgreSQL es una base de datos muy popular, debido a su confiabilidad, rendimiento y más de 20 años de desarrollo comunitario[26]. Varias nubes públicas cuentan

con servicios especializados para desplegar bases de datos PostgreSQL, por ejemplo, Amazon Web Services cuenta con Amazon Relational Database Service (RDS) for PostgreSQL[26] y Azure cuenta con Azure Database for PostgreSQL[27].

En Azure la base de datos se puede desplegar de dos formas: servidor único o servidor flexible, siendo el servidor flexible la opción que ofrece para casos en los que se necesita tener mas control sobre la base de datos y asegurar mayor disponibilidad. El servicio de Azure para el servidor único asegura que la base de datos estará disponible el 99.99 % del tiempo, puede ser ser escalada dinamicamente y seguridad.

6.4. ReactJS

ReactJS es una librería de código libre de JavaScript para el desarrollo de componentes para interfaces de usuario[28]. Por lo tanto, esta librería puede ser utilizada para desarrollar el *frontend* de un sitio web que obtiene su información de una o varias APIs. Entre las ventajas de esta librería se encuentra que tiene un rendimiento altamente eficiente[28], esto se debe a que consta de un DOM virtual, el cual es ligero. Un DOM virtual implica que React interactua con un DOM almacenado en la memoria, no con el DOM generado por el navegador.

6.5. Microservicios

Una arquitectura de microservicios es un software que esta compuesto por varios servicios independientes, cada servicio tiene un solo objetivo y responsabilidad. La comunicación entre los microservicios se realiza por medio de APIs. [29]

A diferencia de las arquitecturas monolíticas, las arquitecturas de microservicios son más rápidas de desarrollar, más fáciles de escalar y mantener. Puesto que, cada servicio es independiente es desarrollado y escalado independientemente. En el caso de una arquitectura monolítica, toda la arquitectura debe de ser escalada si una parte de la arquitectura tiene una carga alta, lo cual resulta en un uso ineficiente de recursos y una arquitectura compleja, ya que todos los servicios son dependientes entre sí.

Algunas de las ventajas de las arquitecturas de microservicios son:

- Agilidad: la independencia entre servicio aumenta la simplicidad del desarrollo de cada microservicio. Por lo tanto, el tiempo del ciclo de desarrollo disminuye.
- Escalado flexible: cada microservicio puede escalar independientemente acorde a la demanda específica de ese servicio.
- Facilidad de despliegue: Cada microservicio es desplegado de forma independiente, por lo tanto, cada microservicio puede ser desplegado en un ambiente de desarrollo, pruebas o producción de forma independiente.
- Libertad tecnológica: Cada microservicio puede ser desarrollado con la tecnología que mejor se adapte a las necesidades del servicio. Por lo tanto, cada microservicio de una

misma arquitectura puede desarrollarse en lenguajes distintos sin afectar la interoperabilidad del sistema.

- Facilidad de mantenimiento: Cada microservicio es mantenido de forma independiente, por lo que, se pueden realizar cambios en el código del microservicio fácilmente sin afectar al resto del sistema.
- Resistencia a los errores: Cada microservicio es independiente, por lo tanto, si un microservicio falla, el resto del sistema no se ve afectado. Es decir, en los sistemas de microservicios no existe un único punto de falla.

Mensajes asíncronos

Los mensajes asíncronos son útiles para lograr la comunicación entre microservicios, mientras se mantiene un acoplamiento ligero y una arquitectura orientada a eventos, puesto que no se espera una respuesta inmediata al enviar un mensaje. Para la comunicación basada en mensajería instantánea entre microservicios es preferible un *broker* de mensajería ligero, por lo que la inteligencia de la aplicación se encuentra en los *endpoints* que producen y reciben los mensajes, es decir, los microservicios.[30] Existen dos tipos de comunicación de mensajería asíncrona: *single receiver message-based communication* y *multiple receivers message-based communication*. La comunicación basada en mensajería con múltiples receptores es la opción mas flexible, con este tipo de comunicación se puede utilizar el mecanismo *publish/subscribe*.

6.6. Virtualización

La virtualización es un tecnología que permite el uso óptimo de una máquina física puesto que permite distribuir los recursos entre varios usuarios o ambientes. Es decir, los servidores se pueden usar más eficientemente, reduciendo los costos de compra y mantenimiento. En cuanto al funcionamiento de virtualización, es necesario un software llamado hipervisor. El hipervisor es el encargado de organizar los recursos físicos para que los ambientes virtuales puedan utilizarlos. Existen distintos tipos de virtualización[31]:

- Virtualización de datos
- Virtualización de escritorios
- Virtualización de servidores
- Virtualización de sistemas operativos
- Virtualización de funciones de red

6.6.1. Contenerización

Un contenedor es un proceso que se ejecuta aisladamente de los demás procesos del sistema operativo. Un contenedor es una instancia de una imagen que se puede crear, iniciar,

ejecutar, detener y eliminar[32]. Siendo una imagen un paquete de software que contiene todo lo necesario para que una aplicación se ejecute. Generalmente se utilizan para empaquetar funciones únicas que realizan una tarea en específico, es decir, para empaquetar microservicios. A diferencia de la virtualización convencional, los contenedores no necesitan la capa del hipervisor puesto que corren directamente sobre el sistema operativo del *host*[33].

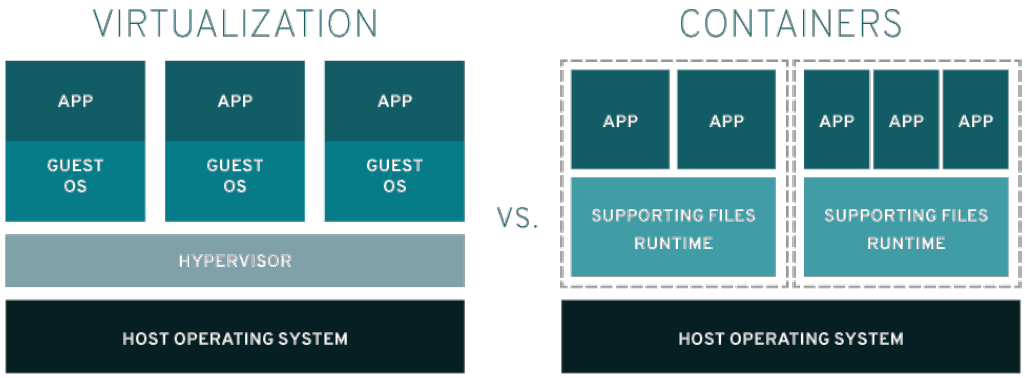


Figura 6: Comparación entre la arquitectura de virtualización y de contenedores

6.6.2. Docker

Docker es una plataforma de código abierto de virtualización a nivel del sistema operativo para la creación, implementación y gestión de aplicaciones en contenedores [32]. Los contenedores son unidades estandarizadas de software que incluyen las dependencias para que la aplicación pueda ser ejecutada. Entre estas dependencias se encuentra el código, librerías, archivos de configuración, entre otros. [34] Cabe mencionar que entre las dependencias de la aplicación incluidas en el contenedor no se encuentra el Kernel, puesto que los contenedores comparten el Kernel del sistema operativo anfitrión, lo cual hace que los contenedores sean más ligeros que una máquina virtual. Docker hace uso de características de aislamiento de recursos del Kernel Linux para mejorar la seguridad de los contenedores. Entre estas características se encuentran los *Namespaces*, grupos de control y *Capabilities*. El objetivo de los *Namespaces* es aislar los sistemas de archivos, red, procesos, de los contenedores. El objetivo de los grupos de control es aislar el uso de recursos (CPU, memoria, disco, entre otros). Y el objetivo de las *capabilities* es restringir el acceso de los contenedores a los recursos del sistema operativo anfitrión. Para contenerizar una aplicación en un contenedor Docker se debe de crear un Dockerfile. El archivo Dockerfile es un archivo de texto sin extensión que contiene un *script* de instrucciones que le indican a Docker como construir la imagen. Luego de crear el Dockerfile se utiliza el comando Docker *build* para construir la imagen. Y el comando Docker *run* se utiliza para crear y ejecutar un contenedor a partir de la imagen creada.

6.6.3. Orquestación de contenedores con Kubernetes

Kubernetes es un sistema para automatizar el lanzamiento, escalamiento y manejo de contenedores. Al tener una gran cantidad de contenedores que orquestar se vuelve necesario

un orquestador de contenedores, debido a la cantidad de microservicios y al escalamiento necesario para manejar el tráfico. Entre las características de Kubernetes importantes se encuentran las siguientes[35]:

- Despliegues y *rollbacks* automatizados
- Sistema de descubrimiento y balanceo de carga
- orquestación de almacenamiento
- Auto-sanación
- Gestión de variables secretas y configuraciones
- Colocar contenedores automáticamente dependiendo de sus requerimientos y limitaciones
- Escalado horizontal

6.6.4. Okteto

Okteto es una plataforma para el despliegue de aplicaciones por medio de kubernetes[36]. Para desplegar microservicios en Okteto, se debe de sincronizar Okteto con un repositorio que contenga un Dockerfile para cada miroservicio a desplegar, al igual que el código a ejecutar de cada microservicio y un Docker-compose para desplegar todos los servicios de la aplicación. A partir de los contenedores, Okteto crea los *clusters* de Kubernetes que contienen los contenedores Docker, según lo indicado en el Dockerfile. Cabe mencionar, que el ambiente el Okteto también se puede desplegar a partir de manifiestos de Kubernetes[37].

6.7. Nube pública

La nube pública es un modelo de computación, en la que los recursos utilizados se encuentran en las instalaciones de un proveedor y se accede a ellos a través de Internet. Dichos recursos se encuentran disponibles para el público en general y se pueden utilizar bajo demanda con el modelo de pago *pay-as-you-go* [38]. Estos recursos son utilizados para ofrecer servicios de distintas categorías, como *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) y *Software as a Service* (SaaS), lo que permite a los usuarios elegir el nivel de administración que desean tener sobre los recursos. Las nubes públicas más utilizadas son Amazon Web Services (AWS), Microsoft Azure y Google Cloud Platform (GCP).

6.7.1. Azure

Azure es una nube pública que ofrece servicios de computación, almacenamiento, redes, análisis, bases de datos, aprendizaje automático e Internet de las cosas (IoT)[39]. Algunos servicios de Azure que pueden ser utilizados para el desarrollo de este proyecto son:

- *Azure Kubernetes Service (AKS)*: Servicio que permite desplegar y administrar contenedores Docker en Kubernetes.
- *Azure Container Registry (ACR)*: Servicio para almacenar imágenes de contenedores Docker.
- *Azure Database for PostgreSQL*: Servicio que permite crear y administrar bases de datos PostgreSQL.
- *Azure Cosmos DB*: Servicio que permite crear y administrar bases de datos NoSQL.
- *Azure IoT Hub*: Servicio que permite administrar dispositivos IoT.
- *Power BI*: Servicio de análisis de datos que permite proporcionar información detallada sobre los datos que se tienen disponibles[40]. Por medio de este servicio se pueden crear tableros para realizar informes y visualizar los datos obtenidos de bases de datos u otras fuentes.
- *Azure Communication Services*: Servicio que ofrece APIs que pueden ser integradas a aplicaciones para enviar mensajes a dispositivos móviles, correos electrónicos, entre otros[41].
- *Azure Functions*: Servicio que permite ejecutar código en la nube de una forma *serverless*.

CAPÍTULO 7

Arquitectura de la plataforma

La arquitectura de la plataforma está conformada por microservicios por lo que es nativa de la nube y es posible desplegarla en la nube pública. La arquitectura completa se puede observar en la Figura 7. Algunos de los microservicio son contenedores Docker y orquestados por medio de Kubernetes, específicamente los microservicios que son desplegados en Okteto son levantados en contenedores Docker. Para desplegar los *pods* de Kubernetes en Okteto, se utiliza el archivo *docker-compose.yml* que se puede observar en el Código 7.1, el cual despliega los *pods* con contenedores Docker generados a partir de archivos Dockerfile. Para cada microservicio, que es desplegado en un contenedor Docker, se crea un archivo Dockerfile que contiene las instrucciones para crear el contenedor a partir de la imagen del microservicio.

```
version: '3'

services:
  backend:
    build: .
    command: python flujos/manage.py runserver 0.0.0.0:8000
    ports:
      - "8000:8000"

  web:
    build: ./frontend
    command: npm start
    ports:
      - "3000:3000"
    depends_on:
      - backend
    environment:
      - REACT_APP_BACKEND_URL=https://backend-tesis-mor19213.cloud.okteto.net/
      - REACT_APP_HISTORICAL_URL=https://historical-tesis-mor19213.cloud.okteto.net

  historical:
    build: ./historical
    command: uvicorn app.main:app --host 0.0.0.0 --port 80
    ports:
      - "8000:80"
```

```
  report:
    build: ./report

  mail:
    build: ./mails
```

Código 7.1: Archivo Docker-compose para los microservicios desplegados en Okteto

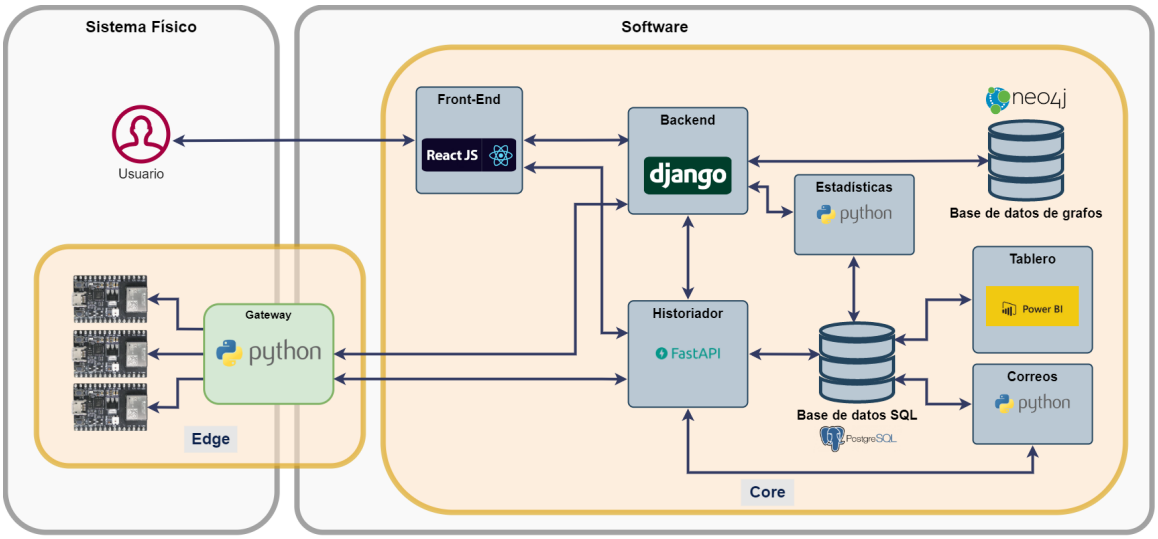


Figura 7: Arquitectura de la plataforma

La plataforma consiste de un *backend*, una base de datos de grafos, el *frontend*, una API programada en FastAPI, una base de datos SQL, dos *cron jobs*, un tablero en Power BI y un *gateway* para los dispositivos IoT. En la base de datos de grafos se almacena en forma de nodos los usuarios y los dispositivos. Y utilizando las relaciones entre nodos, se almacenará la información de cuales son los dispositivos que pertenecen a cada usuario y las dependencias del estado de los actuadores con los sensores. Para desplegar la base de datos de grafos, se utilizó una base de datos de grafos como servicio ofrecida por Neo4j. Dicho servicio es llamado AuraDB, cuenta con un límite de 20,000 nodos y 40,000 relaciones y es posible desplegarlo en distintas nubes públicas, como lo son AWS, GCP y Azure, la nube en la que se encuentra la base de datos de esta plataforma es GCP.

El *backend* es un microservicio desarrollado en el *framework* Django. El objetivo de este es almacenar de una forma ordenada la información de los dispositivos IoT y los usuarios en la base de datos, al igual que proporcionar *endpoints* desde los cuales se puede obtener o modificar la información en la base de datos. Dichos *endpoints* son consumidos por el *frontend*, por el *gateway* y por el *cron job* que guarda estadísticas de la plataforma en la base de datos SQL. El *backend* también se encarga de la autenticación de los usuarios y de la generación de tokens de autenticación.

La API del historiador es un microservicio desarrollado en FastAPI. El objetivo de este es proporcionar *endpoints* para que el *gateway* pueda enviar la información de los sensores a la base de datos SQL y se almacene la data histórica. Esta API también proporciona un *endpoint* por medio de los cuales se puede observar la data histórica de los sensores, filtrada por fecha.

El *frontend* de la plataforma, desarrollado con ReactJS, consume los *endpoints* del backend para obtener la información necesaria según la *URL* que accese el usuario. Al igual que consume el *endpoint* de la API del historiador para desplegarla.

Los *cron jobs* son dos microservicios desarrollados en Python. El objetivo del primero es guardar estadísticas de la plataforma en la base de datos SQL que, como ya se ha mencionado, obtiene del *backend*. El objetivo del segundo es recopilar información del uso de la plataforma y sensores durante el día anterior y enviarla por correo electrónico a cada usuario.

El tablero de Power BI consume la base de datos SQL para desplegar las estadísticas de la plataforma. La información que consume se obtiene por medio de una consula SQL que se puede observar en el Código. El objetivo de este tablero es que el administrador de la plataforma pueda monitorear el uso de la misma.

El *gateway* de la plataforma es el encargado de recibir los paquetes de los dispositivos IoT. El *gateway* envía tres tipos de *requests* al *backend*. El primer tipo es un *put request* para actualizar la información de los sensores en la base de datos. El segundo tipo es un *get request* para obtener la información de los actuadores y enviarla a los dispositivos IoT para que actualicen el estado del mismo. Y el tercer tipo es un *post request* para enviar la información de los sensores a la API del historiador. Cabe mencionar que el *gateway* permite que los dispositivos IoT y la plataforma se encuentren acoplados ligeramente. El *gateway* es capaz de soportar distintos protocolos IoT, para la comunicación entre la plataforma y los dispositivos IoT. Actualmente los protocolos que soporta son UDP, Bluetooth y MQTT, sin embargo, es posible agregar programas en el *gateway* para que este pueda soportar más protocolos.

Finalmente, el sistema físico es un prototipo conformado por microcontroladores ESP32 que tienen conectados sensores y actuadores que se comunican con el *gateway* a través de los protocolos que son soportados.

CAPÍTULO 8

Backend de la plataforma

8.1. Modelos de nodos y relaciones

En la base de datos de grafos se realizaron modelos para tres tipos de nodos, los cuales son: Usuario, Sensor y Actuador. Las relaciones entre estos nodos se pueden observar en la Figura 8. La relación entre los nodos Usuarios y los dispositivos, es decir, los nodos Sensor y Actuador, indica pertenencia. La relación entre los nodos Sensor y Actuador, indica que una condición que debe de cumplirse del valor del sensor para activar al actuador representado por el nodo Actuador. Los atributos de esta relación son:

- El nombre de la relación, la cual índica que tipo de condición debe de cumplirse para que el nodo Actuador se active. Las distintas opciones de nombres de relación reconocidas por la plataforma son:
  - mayor que
  - menor que
  - igual que
  - mayor o igual que
  - menor o igual que
- El valor de la condición, el cual índica el valor que debe de tener la condición mencionada previamente.
- El resultado de la condición

Por lo tanto, en el *backend*, utilizando el *framework* Django, se realizó un modelo para cada tipo de nodo y para cada tipo de relación entre nodos. En la Figura 8 se puede observar

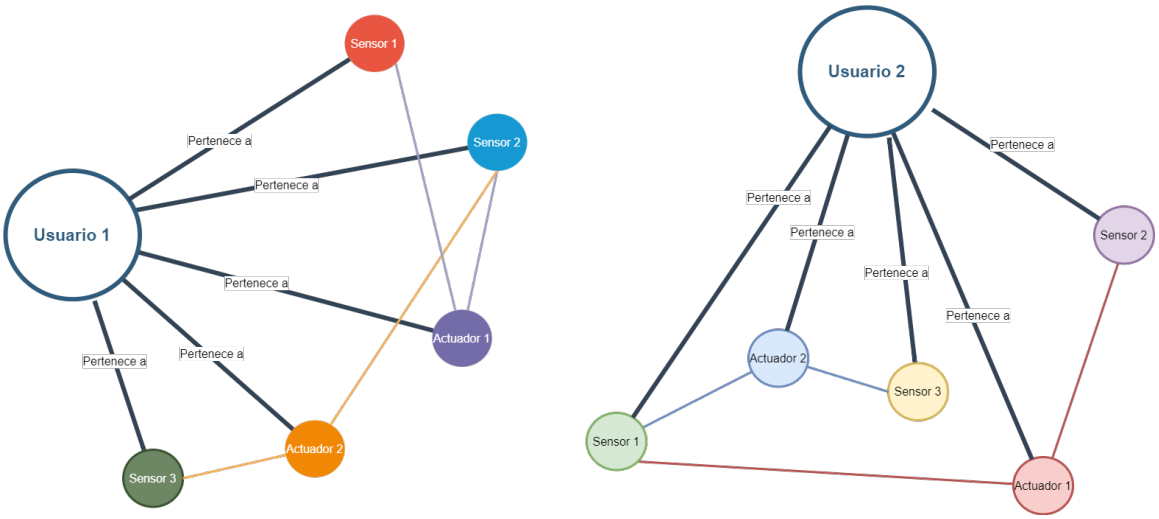


Figura 8: Modelo de la base de datos

un diagrama con el prototipo de los nodos y sus relaciones. Asimismo, en la Figura 9 se puede observar un ejemplo de los nodos y relaciones de un usuario desde la interfaz de la base de datos Neo4j.

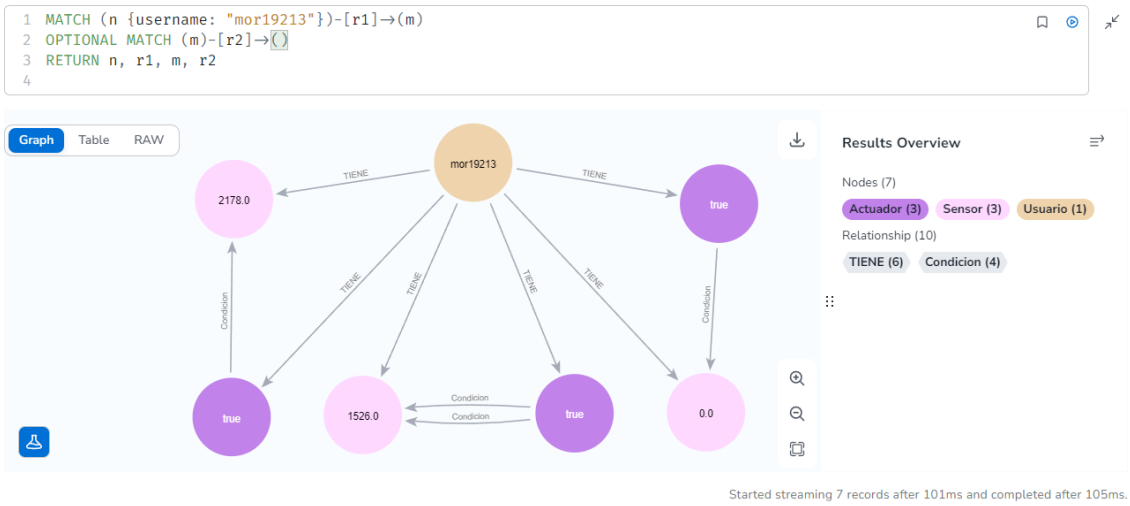


Figura 9: Nodos y relaciones en base de datos

8.1.1. Nodo Usuario

En cuanto al modelo del nodo Usuario se cuenta con los atributos:

- **nombre**
- **apellido**
- **username**

- **email**
- **date_joined**
- **password_hash**

Los atributos **nombre**, **apellido**, **email** y **username** es requerido que el usuario los ingrese al momento de registrarse. De esto atributos, solamente el atributo **username** y **email** son únicos. Cabe mencionar que en la plataforma el correo electrónico se utiliza por el *cron job*, puesto que a este correo es que se envían los reportes diarios. En cuanto al atributo **password_hash**, este es generado por el *backend* al momento de que el usuario se registra a partir del *string* que el usuario ingresa como contraseña. El **hash** de la contraseña se encripta utilizando el algoritmo *SHA256* y el resultado es el *string* que se guarda en el atributo *password_hash*.

8.1.2. Nodo Sensor

En cuanto al modelo del nodo Sensor, este cuenta con los atributos:

- **nombre**
- **valor**
- **salida_num**

Al momento de que el usuario registra un nuevo dispositivo, el usuario ingresa únicamente el nombre del Sensor y el *backend* revisa que el nombre no se encuentre asignado a otro nodo sensor perteneciente a ese usuario y crea un nodo para este dispositivo. El atributo valor es el valor actual del sensor y este es asignado por el *gateway* al momento de que el dispositivo envía un paquete con la información del sensor. Finalmente, el atributo **salida_num** es un booleano que índica si el sensor es de salida numérica o no.

8.1.3. Relación Condición

El modelo de la relación **Condicion** cuenta con los atributos:

- ***Unique Identifier*** **(uid)**
- **nombre**
- **condition**

El **uid** es generado al momento de crear una condición entre un nodo Sensor y un nodo Actuador. El objetivo del atributo **uid** es diferenciar entre relaciones existentes entre los mismos nodos. El atributo **nombre** es un *string* que índica qué tipo de condición se debe de cumplir para que el actuador se active, estas opciones son las mencionadas previamente.

Este atributo es asignado por el usuario a través de la interfaz gráfica. El atributo **condition** es un booleano que índica si la condición se cumple o no. Este atributo es distinto puesto que es el resultado de una función con el decorador **@property** que se ejecuta cada vez que se accede al atributo.

8.1.4. Nodo Actuador

En cuanto al modelo del nodo Actuador, este cuenta con los atributos:

- **nombre**
- **operacion**
- **estado**
- **manual**
- **cond_s**

El atributo **nombre** se asigna de la misma manera que en el modelo del nodo Sensor. El atributo **cond_s** es una relación con el nodo Sensor y de tipo **Condicion**. El atributo operación indica que operación lógica se realiza entre las condiciones con los sensores. El *backend* puede realizar las operaciones *and, or, xor, nand, nor y xnor*. El atributo estado es un booleano que índica si el actuador se encuentra activo o no. Este atributo es calculado en el *backend* cada vez que esta variable es llamada. Para realizar este cálculo primero se guarda en una lista el resultado de cada condición que tiene el actuador. Luego se realiza la operación lógica indicada por el atributo **operacion** para calcular una operación lógica entre los valores booleanos de la lista mencionada anteriormente. Finalmente, el atributo **manual** es un valor booleano que indica si el estado del actuador será definido a partir de la operación lógica de las condiciones mencionada anteriormente, o si este valor será definido directamente por el usuario a partir de un *endpoint* que expone la API.

8.2. Endpoints de la API

El microservicio *backend* es el encargado de llevar a cabo la lógica del sistema, al igual que mantener la comunicación con el *frontend* y la red de dispositivos IoT. Para dicha comunicación, se realizó una API en el *backend* que expone los *endpoints* mostrados en el Cuadro 1 y explicados a posteriormente. En los Códigos 16.1 a 16.8 se pueden observar ejemplos de peticiones.

Cabe mencionar que en los *endpoints* es posible enviar variables, las cuales se utilizan en la lógica realizada por el *backend* cuando se envía una petición a dicho *endpoint*. En el caso específico de los *endpoints* desarrollados para la plataforma, se utilizan tres variables. Estas variables son **username**, **name** y **uid**. El **username** se utiliza en todos los *endpoints* que son para editar u obtener información de un usuario en específico, puesto que a partir del nombre de usuario enviado en dicha variable se recopila la información necesaria. Por

Endpoint	Método	Descripción	Formato request	Formato response
/admin/stats	GET	Información estadística de la plataforma	-	{usuarios, sensores, actuadores}
/signup	POST	Obtener tokens de acceso y actualización si el nombre de usuario y contraseña son correctos	{data: {username, password, nombre, apellido, email} }	-
/usuarios	GET	Obtener el listado de todos los usuarios de la plataforma	-	{[listado de usuarios]}
/usuarios/<username>	PUT	Cambiar la contraseña, nombre o apellido del usuario	{data: {nombre, apellido, password} }	-
	DELETE	Eliminar la cuenta del usuario	-	-
/tokens	POST	Obtener los tokens de acceso y actualización si el nombre de usuario y contraseña son correctos	{username, password}	{refresh, access }
/tokens/refresh	POST	Obtener los tokens de acceso y actualización si el token de actualización es válido	{data: {refresh, username} }	{refresh, access }

/<username>/dispositivos	GET	Obtener una lista de los sensores y actuadores de un usuario	-	{
				usuario,
				sensores:[lista de sensores],
				actuadores:[lista de actuadores]
				}

/<username>/sensores	GET	Obtener una lista de los sensores y actuadores de un usuario	-	[Lista de sensores]
	POST	Crear un nuevo sensor	{nombre}	-
/<username>/sensores/<nombre>	DELETE	Eliminar el sensor con el nombre indicado	-	-
	PUT	Editar el nombre del sensor	{nombre}	-
	POST	Editar el valor del sensor	{valor}	-
/<username>/actuadores	GET	Obtener una lista de los actuadores de un usuario	-	[Lista de actuadores]
	POST	Crear un nuevo actuador	{nombre}	-
/<username>/actuadores/<nombre>	DELETE	Eliminar el actuador con el nombre indicado	-	-
	PUT	Editar el nombre del actuador	{nombre}	-
	GET	Obtener el valor del actuador	-	{actuador }

/<username>/condiciones	GET	Obtener un listado de las condiciones por actuador del usuario	-	{datos: {
				[listado de actuadores: {
				actuador,
				[lista de condiciones]
				}] } }

/<username>/condiciones/<name>	GET	Obtener las condiciones de un actuador en específico del usuario	-	{condicion }
	PUT	Editar las condiciones de un actuador en específico del usuario	{operacion}	{actuador }
/<username>/condiciones/<name>/<uid>	PUT	Editar una condición específica de un actuador específico del usuario	{comparacion, valor, sensor}	-
	DELETE	Eliminar una condición específica de un actuador específico del usuario	-	-
	GET	Obtener una condición específica de un actuador específico del usuario	-	-

Cuadro 1: Resumen de funcionamiento de endpoints

motivos de seguridad, para todos estos *endpoints* se realiza una verificación de que la petición se envíe con un *token* de acceso válido. El **name** se utiliza para enviar el nombre de un nodo en específico de tipo Sensor o Actuador. Finalmente, el **uid** se utiliza solamnete en un *endpoint*. Este *endpoint* actúa sobre la información de una relación específica de un actuador de un usuario, por lo tanto, en este *endpoint* se envía la variable **username** para indicar el usuario, la variable **name** para indicar el actuador y la variable **uid** para indicar la relación saliente de dicho actuador.

8.2.1. *Endpoints* de información de usuarios

/signup

Este *endpoint* permite a un usuario registrarse en la plataforma. Para ello, se envía una petición POST con los parámetros: nombre, apellido, *username*, correo electrónico y contraseña. A partir de la información proporcionada, el *backend* crea el nodo Usuario, guarda la contraseña codificada, guarda el nodo Usuario, obtiene los tokens de autenticación del usuario que se acaba de crear y devuelve los tokens si la operación fue ejecutada exitosamente.

/usuarios

Este *endpoint* permite *get request* y devuelve una lista de todos los usuarios registrados en la plataforma, con su *username* y correo.

/usuarios/*username*

Este *endpoint* permite *put* y *delete requests* sobre la información del usuario en particular. Es decir, a partir de este *endpoint* se puede borrar un usuario, cambiar su contraseña, nombre, apellido o correo.

8.2.2. *Endpoints* de obtención de tokens

/tokens

Este *endpoint* se utiliza en el inicio de sesión de un usuario. Acepta *Post requests* con el nombre de usuario y contraseña, a partir de esta información el *backend* procede a revisar si la contraseña enviada coincide con la contraseña del usuario. En caso de coincidir, este *endpoint* devuelve dos *tokens* uno de acceso y otro de actualización. El *token* de acceso es enviado en el *header* de las peticiones que requieren autenticación y en caso de vencerse, se utiliza el *token* de actualización para obtener nuevos *tokens*. Debido a esto, es que el *token* de acceso se genera con un tiempo de vencimiento menor el *token* de actualización. El *token* de acceso es requerido en todos los *endpoints* que inician con un nombre de usuario. Ya que, estos devuelven información específica de dicho usuario y que sólo este usuario tiene autorizado acceder.

/tokens/refresh

Este *endpoint* permite peticiones *post* en el cual se envía el *token* de actualización y devuelve nuevos *tokens* en caso el *token* de actualización enviado sea válido.

8.2.3. *Endpoints* de información de dispositivos

Los *endpoints* de los dispositivos son para obtener o modificar información de los sensores y actuadores que posee un usuario. Por lo tanto, todos los *endpoints* a continuación inician con el nombre de usuario, ya que, la información es de los dispositivos con una relación hacia el usuario.

/*username*/dispositivos

Este *endpoint* devuelve un diccionario, con las claves: Usuario, Sensores y Actuadores. El valor de la clave Usuario es un *string* con el nombre y apellido del usuario. El valor de la clave Sensores es una lista conformada por diccionarios con los valores de cada sensor que pertenecen al usuario. El valor de la clave Actuadores sigue el mismo formato que el valor de Sensores.

/*username*/sensores

Este *endpoint* devuelve la lista de los sensores que pertenecen al usuario. Por cada sensor se incluye toda la información pertinente en un diccionario.

/*username*/sensores/*name*

Este *endpoint* permite peticiones *post, get, delete y put*. Las peticiones *get* devuelven un diccionario con la información del sensor. Este es el mismo diccionario que se observó en el endpoint anterior. Las peticiones *post* se utilizan para la creaciones y las peticiones *delete* para la eliminación del sensor con el nombre indicado por la variable *name*. Finalmente, las peticiones *put* solamente se ejecutan por el gateway, puesto que se utilizan para enviar el valor actualizado del sensor.

/*username*/actuadores

Este *endpoint* permite únicamente peticiones *get* y devuelven una lista de diccionarios con la información de cada actuador que pertenece al usuario.

*/username/*actuadores*/name*

Este *endpoint* tiene un funcionamiento similar al endpoint *username*/sensores/*name*. Sin embargo, en este caso el *gateway* no ejecuta *put requests*, ejecuta *get requests*. Ya que, el *gateway* busca obtener el valor actual de los actuadores para enviar un mensaje al dispositivo con dicho actuador para cambiar su estado. Este *endpoint* si permite *put requests* los cuales son ejecutados por el *frontend* cuando el actuador está siendo manipulado directamente a través de la interfaz gráfica, es decir, el estado del actuador no está dado por una operación lógica.

8.2.4. Condiciones del estado de los actuadores

De una forma similar a los *endpoints* de los dispositivos, la dirección de los *endpoints* de las condiciones también inician con el nombre de usuario. Esto se debe a que los *endpoints* manejan información de un usuario en específico.

*/username/*condiciones

Este *endpoint* permite *get request* y devuelve una lista de todas las condiciones de datos del usuario en particular. En la Figura 16.6 se puede observar un ejemplo de las condiciones de un usuario listados al hacer un *get request* al *endpoint*. Se observa que la *data* está conformada por un diccionario con una clave "datos", el cual tiene como valor una lista. Esta lista está conformada por diccionarios con dos claves, los cuales son "actuador"y "condiciones". El valor de la clave "actuador"es un diccionario con la información de dicho actuador. Finalmente, el valor de la clave condiciones es una lista de diccionarios con la información de cada condición que tiene el actuador.

*/username/*condiciones*/name*

Este *endpoint* permite *put request* y *delete request* sobre la información del flujo en particular del actuador con nombre indicado por la variables *name* del usuario indicado por medio del *string* username. Este incluye la misma información que la lista anterior, es decir, una lista de diccionarios de las condiciones que tiene el actuador. Por cada condición se obtiene la información: dispositivo, condición, valor, actual, real y uid.

*/username/*condiciones*/name/uid*

Este *endpoint* permite obtener la información de una condición específica. Para esto es necesario indicar, el nombre del actuador que posee esta condición por medio de la variable *name* y el uid de la relación que define la condición.

8.3. Tokens para autenticación de usuarios

Para la autenticación de los usuarios en la plataforma se utilizan JWT (*JSON Web Tokens*). Estos tokens se generan por el *backend* de la plataforma y se envían al *frontend*, cuando este los solicita sí envía una combinación de nombre de usuario y contraseña válida, para que este los almacene en el *local storage* del navegador. El *frontend* es el encargado de enviar el token en cada petición que se haga al *backend*, para que este pueda verificar la identidad del usuario, y mantener los tokens actualizados.

En el *backend* estos tokens se generan por medio de un método de la clase Usuario. Este método recibe como único parámetro la contraseña del usuario, y devuelve un token JWT con la información del usuario, y una fecha de expiración de 25 minutos y un token JWT con expiración de 2 días. El segundo token mencionado, que tiene mayor tiempo de expiración, no contiene información del usuario pero se utiliza para generar nuevos tokens de acceso cuando estos expiran. Este token con mayor tiempo de expiración se llama *refresh token*.

CAPÍTULO 9

Historiador de la plataforma

El objetivo de los microservicios que conforman al historiador es el de almacenar los datos históricos de los sensores, mantener un registro de la cantidad de usuarios, sensores y actuadores en la plataforma. También el objetivo es, a partir de esta información, tener un tablero en PowerBI donde se pueda observar el uso que le dan los usuarios a la plataforma, al igual que enviar correos automáticos a los usuarios para indicarles como fue el comportamiento de sus sensores el día anterior.

Para lograr este objetivo se utilizaron cinco microservicios, los cuales son una base de datos SQL, una API con FastAPI, un *cron job* para almacenar estadísticas del día, un *cron job* para enviar correos automáticos y un tablero en PowerBI. Cabe mencionar que la información que ingresa a este subsistema proviene de las peticiones HTTP realizadas en el *gateway* para actualizar el valor de los sensores y de la base de datos de grafos proviene la información sobre la cantidad de usuarios, sensores y actuadores en la plataforma.

9.1. Base de datos SQL

La base de datos utilizada para almacenar la información histórica de los sensores es una base de datos SQL alojada en un servidor de Azure. Esta base de datos consta de tres tablas: *users*, *data*, *stats*, y *blacklisted_tokens*. La tabla *blacklisted_tokens* almacena los tokens JWT que han sido utilizados por el usuario, con el fin de evitar que se puedan utilizar nuevamente para acceder a la información histórica de los sensores. Por lo tanto, esta tabla únicamente contiene, para cada token, un campo *id*, el campo *token* que almacena el token en sí mismo y el campo *delete_at* que almacena la fecha de expiración del token.

La tabla *users* almacena información sobre los usuarios de la plataforma, con los siguientes campos:

- *username*: Identificador único del usuario.
- *password*: Contraseña del usuario almacenada de forma encriptada.
- *email*: Correo electrónico del usuario.

Esta información se utiliza en dos momentos. Primero, al enviar correos automatizados, ya que se extraen todos los correos de esta tabla para enviar los correos correspondientes. Segundo, al guardar datos históricos de los sensores, donde se verifica la pertenencia de los sensores al usuario indicado mediante tokens JWT. Estos tokens se envían al usuario al iniciar sesión y deben actualizarse periódicamente.

La tabla *data* almacena la información histórica de los sensores con los siguientes campos:

- *year*: Año de ingreso de los datos.
- *month*: Mes de ingreso de los datos.
- *day*: Día de ingreso de los datos.
- *hour*: Hora de ingreso de los datos.
- *minute*: Minuto de ingreso de los datos.
- *username*: Identificador único del usuario.
- *sensor_name*: Identificador del sensor.
- *value*: Valor del sensor.

Finalmente, la tabla *stats* se actualiza diariamente a las 23:50 y almacena la siguiente información:

- *fa*: Fecha de ingreso de los datos.
- *user_count*: Cantidad de usuarios registrados en la plataforma.
- *sensor_count*: Cantidad de sensores registrados en la plataforma.
- *actuator_count*: Cantidad de actuadores registrados en la plataforma.

La Figura 10 muestra la configuración de la base de datos en Azure.

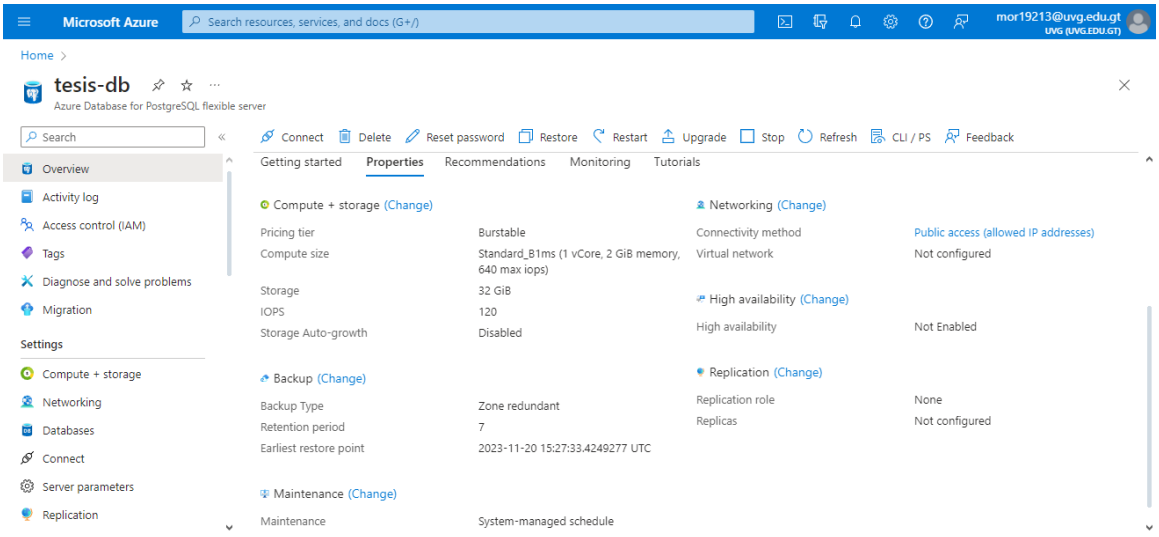


Figura 10: Configuración de la base de datos en Azure

9.2. API de FastAPI

La API desarrollada para el historiador de la plataforma cuenta con los siguientes *endpoints*.

9.2.1. /users/*username*

Este *endpoints* permite peticiones *POST* y *DELETE* para crear o actualizar la información del usuario, al igual que eliminarlo. En el caso del post se ejecuta la consulta SQL observada en el Código 9.1, en la cual se observa que se utiliza la cláusula *ON CONFLICT* para actualizar la información del usuario en caso de que ya exista, puesto que el campo *username* debe de ser único. En el caso del *DELETE* se ejecuta la consulta SQL observada en el Código 9.2, en la cual se observa que se hace un join entre la tabla *data* y *users* para luego eliminar todos los registros que coincidan con el nombre de usuario, de esta manera se eliminan los registros del usuario de ambas tablas al mismo tiempo.

```
    INSERT INTO users (username, password_encrypted)
    VALUES (%s, %s)
    ON CONFLICT (username)
4   DO UPDATE SET password_encrypted = EXCLUDED.password_encrypted
```

Código 9.1: Consulta SQL para crear o actualizar un usuario

```
1 DELETE data, users
  FROM data
  JOIN users ON data.username = users.username
  WHERE TRIM(data.username) = %s;
```

Código 9.2: Consulta SQL para eliminar un usuario

9.2.2. /users/*username*/sensores

Este *endpoint* únicamente permite peticiones *GET* para obtener la lista de sensores del usuario. Esta lista se obtiene de la tabla *data* utilizando la consulta SQL observada en el Código 9.3, en la cual se observa que se utiliza la cláusula *DISTINCT* para obtener el nombre cada sensor solamente una vez. Debido a que se obtiene de la tabla *data*, únicamente se obtienen los sensores de los cuales se han enviado datos al historiador, es decir, los sensores activos.

```
1 SELECT DISTINCT sensor_name FROM data WHERE TRIM(username)=%s
```

Código 9.3: Consulta SQL para obtener los sensores de un usuario

9.2.3. /users/*username*/sensores/*sensor*/valores

Este *endpoint* únicamente permite peticiones *GET* para obtener los valores de un sensor en específico. En la respuesta de la petición se devuelve un diccionario con una única llave, la cual es *valores*, y su valor es una lista de diccionarios, en los cuales se encuentran los valores de cada sensor. Dichos valores son: fecha, hora, *anomaly* y *value*. La fecha y hora, indican cuando fue ingestado el valor del sensor a la base de datos, siendo *value* dicho valor ingestado. Estos tres datos se obtienen por medio de una consulta SQL a la tabla *data*, filtrando entre los valores *begin_date* y *end_date* enviados por el usuario. En cuanto a *anomaly*, es un valor booleano que indica si el valor del sensor es una anomalía o no. Este valor booleano se obtiene por medio de un modelo de *Isolation Forest* entrenado con los valores del sensor. Este modelo es un algoritmo de aprendizaje no supervisado que permite detectar anomalías en los datos utilizando arboles de decisiones.

```
def get_anomalies(username: str, sensor: str, contamination: float = 0.1,
                  begin_date: Optional[str] = None,
                  end_date: Optional[str] = None):

    try:
        query = "SELECT value, year, month, day, hour, minute FROM data WHERE
            TRIM(username)=%s AND TRIM(sensor_name)=%s"
        query_params = [username, sensor]
        print(f"begin date: {begin_date}")
        print(f"end date: {end_date}")
        if begin_date:
            day, month, year = map(int, begin_date.split('/'))
            print(f"year: {year}, month: {month}, day: {day}")
            query += " AND year >= %s AND month >= %s AND day >= %s"
            query_params = query_params + [year, month, day]

        if end_date:
            day, month, year = map(int, end_date.split('/'))
            query += " AND year <= %s AND month <= %s AND day <= %s"
            query_params = query_params + [year, month, day]

        query += " ORDER BY year DESC, month DESC, day DESC, hour DESC, minute DESC"

        rows = db.execute(query, query_params, True)
        print(f"rows: {rows}")

        if not rows:
            return JSONResponse(content=jsonable_encoder({"valores": []}))
```

```
        values = [row[0] for row in rows]
        fechas = [str(row[3])+'/'+str(row[2])+'/'+str(row[1]) for row in rows]
        horas = [str(row[4])+':'+str(row[5]) for row in rows]

        if len(values) < 2:
            anomalies_list = [
            {
                "fecha": fecha,
                "hora": hora,
                "anomaly": False,
                "value": value
            }
            for anomaly, value, fecha, hora in zip(anomalies, values_list, fechas,
                horas)
            ]
            anomalies_json = jsonable_encoder({"valores": anomalies_list})
            return JSONResponse(content=anomalies_json)

        values = np.array(values).reshape(-1, 1)

        model = IsolationForest(random_state=0, contamination=contamination)
        model.fit(values)
        anomalies = model.predict(values)
        values_list = values.flatten().tolist()

        anomalies_list = [
        {
            "fecha": fecha,
            "hora": hora,
            "anomaly": int(anomaly) == 1,
            "value": value
        }
        for anomaly, value, fecha, hora in zip(anomalies, values_list, fechas, horas)
        ]

        anomalies_json = jsonable_encoder({"valores": anomalies_list})
        return JSONResponse(content=anomalies_json)

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

Código 9.4: Función para obtener los valores de un sensor

9.2.4. /users/*username*/sensores/*sensor*/valores/*valor*

Este *endpoint* fue creado para ser accedido por el *gateway* y actualizar el valor de los sensores, por lo tanto, solamente permite peticiones tipo POST. El valor enviado por el *gateway* es insertado a la tabla *data* junto con el año, mes, día, hora y minuto. Cabe mencionar que la fecha y hora es equivalente a la hora en Guatemala, puesto que en el código se realiza la conversión necesaria. Este *endpoint* es el único que puede ser accedido desde fuera de la misma plataforma, por lo que, para evitar que cualquier persona pueda enviar datos al historiador, se utilizan *tokens* JWT para la autenticación. Estos *tokens* se generan por la API y el *gateway* envía el *token* de acceso en el *header* de la petición. El flujo es el mismo al utilizado para la autenticación por medio de *tokens* en el microservicio *backend*.

9.2.5. *Endpoints* para el manejo de *tokens*

Como se ha mencionado previamente, el fin de estos *endpoints* es permitir la autenticación de los usuarios para actualizar el valor de los sensores. Para lograr este objetivo se crearon dos *endpoints*, uno para iniciar sesión y obtener los *tokens* de acceso y actualización, y otro para actualizar los tokens.

El primer *endpoint* es */login*, permite peticiones POST y requiere que el usuario envíe su usuario y contraseña. A partir de esta información valida si el usuario existe en la base de datos y si la contraseña coincide con la constraseña encriptada almacenada en la tabla *users*. En caso de que el usuario exista y la contraseña coincida, se genera un *token* de acceso con tiempo de expiración de 10 miutos y un *token* de actualización con un tiempo de expiración de 4 horas.

El segundo *endpoint* es */refresh*, permite peticiones POST y requiere que el usuario envíe su *token* de actualización. Primero valida si el *token* de actualización es válido, es decir, si no ha expirado. Luego, valida que el *token* de actualización no se encuentre en la tabla *blacklisted_tokens*, la cual almacena los *tokens* de actualización que han sido utilizados anteriormente para obtener un nuevo *token* de acceso. En caso de que essl *token* de actualización sea válido y no se encuentre en la tabla *blacklisted_tokens*, se genera un nuevo *token* de acceso con tiempo de expiración de 10 minutos y un nuevo *token* de actualización con un tiempo de expiración de 4 horas. Finalmente, se agrega el *token* de actualización utilizado a la tabla *blacklisted_tokens* con el fin de evitar que se pueda utilizar nuevamente para obtener un nuevo *token* de acceso, para aumentar la seguridad de la API.

9.3. Cron job para almacenar estadísticas del día

Este microservicio tiene el objetivo de llenar información en la tabla *statistics* de la base de datos. Esto lo realiza todos los días a las 23:50, y almacena la cantidad de usuarios, sensores y actuadores registrados en la plataforma. La información a almacenar la obtiene del endpoint */admin/stats* del microservicio *backend*. Este programa es ejecutado por medio de un cron job en un contenedor docker de Linux con Python. El contenedor es definido por medio del Dockerfile observado en el Código 9.5. Finalmente, para definir que se ejecute el programa *report.py* todos los días a las 23:50, se incluye el Códgio observado en 9.6 en el archivo *crontab*. Cabe mencionar que debido a que el contenedor es ejecutado en un Okteto, cuyos servidores tienen la zona horaria UTC, se debe definir la hora en la que se desea ejecutar el programa en UTC.

```
FROM python:3.7
RUN apt-get update && apt-get -y install cron vim
WORKDIR /app
COPY requirements.txt requirements.txt
RUN pip3 install --no-cache-dir -r requirements.txt
COPY crontab /etc/cron.d/crontab
RUN chmod 0644 /etc/cron.d/crontab
RUN /usr/bin/crontab /etc/cron.d/crontab

COPY report.py report.py
RUN echo $PYTHONPATH
CMD ["cron", "-f"]
```

Código 9.5: Dockerfile del cron job para llenar la tabla *statistics*

```
# ejecutar el programa report.py todos los dias a las 5:50 UTC
50 5 * * * /usr/local/bin/python /app/report.py > /proc/1/fd/1 2>/proc/1/fd/2
```

Código 9.6: Crontab del cron job para llenar la tabla *statistics*

El programa de Python que es ejecutado por el cron job se observa en el Código 9.7. En este código se obtiene la información del endpoint */admin/stats* del microservicio *backend* y se almacena en la tabla *statistics* de la base de datos, junto con la fecha del día anterior, ya que la hora del contendor en Okteto se encuentra en UTC.

```
import psycopg2
import requests
import datetime

db = db_connection()

endpoint = "https://backend-tesis-mor19213.cloud.okteto.net/admin/stats"
try:
    result = requests.get(endpoint).json()
except Exception as e:
    result = {
        "usuarios": None,
        "sensores": None,
        "actuadores": None
    }
    print(f"Error obteniendo info del backend {e}")
    exit()

result["fa"] = int((datetime.date.today() -
    datetime.timedelta(days=1)).strftime("%d/%m/%Y"))

try:
    # guardar a la tabla statistics
    query = "INSERT INTO statistics (fa, user_count, sensor_count, actuador_count)
        VALUES (%s, %s, %s, %s)"
    db.execute(query, (result["fa"], result["usuarios"], result["sensores"],
        result["actuadores"]))
    print("Estadísticas guardadas correctamente")
except Exception as e:
    print(f"Error ingresando los valores: {e}")
```

Código 9.7: Programa de Python para llenar la tabla *statistics*

9.4. Cron job para enviar correos automáticos

Este microservicio tiene el objetivo de obtener información sobre el comportamiento de los sensores de los usuarios y enviar correos automáticos a los usuarios con la información correspondiente a su cuenta. La lógica por medio de la cual se despliega este microservicio, es la misma con la que se despliega el microservicio de generación de estadísticas, ya que ambos son *cron jobs* que ejecutan un programa en Python. El Dockerfile utilizado para desplegar este microservicio se observa en el Código 9.8, la única diferencia que se tiene con el código para el microservicio stats, es que en este caso se copia el archivo *mails.py* en lugar del archivo *report.py*.

```
FROM python:3.7
RUN apt-get update && apt-get -y install cron vim
WORKDIR /app
COPY requirements.txt requirements.txt
RUN pip3 install --no-cache-dir -r requirements.txt
COPY crontab /etc/cron.d/crontab
RUN chmod 0644 /etc/cron.d/crontab
RUN /usr/bin/crontab /etc/cron.d/crontab

COPY mails.py mails.py
RUN echo $PYTHONPATH
CMD ["cron", "-f"]
```

Código 9.8: Dockerfile del cron job para enviar correos automáticos

```
50 5 * * * /usr/local/bin/python /app/report.py > /proc/1/fd/1 2>/proc/1/fd/2
```

Código 9.9: Crontab del cron job para enviar correos automáticos

El archivo *mails.py* ejecutado se observa en el Código 9.10. Este código se conecta a la base de datos para obtener los nombres de usuario y correos. Posteriormente, para cada usuario, se obtiene la cantidad de actualizaciones de los sensores del día anterior y la cantidad de anomalías detectadas por cada sensor. Finalmente, con toda esta información se manda un correo al usuario con la información correspondiente a su cuenta. En la Figura 38 se observa un ejemplo de un correo enviado por este microservicio.

```
from azure.communication.email import EmailClient
import psycopg2
import requests
import datetime

db = db_connection()

query = "SELECT username, correo from users"
result = db.execute(query, None, True)
if result is None:
    exit()
users = [user[0] for user in result]
emails = [email[1] for email in result]
date = (datetime.date.today() - datetime.timedelta(days=1)).strftime("%d/%m/%Y")

for u, email in zip(users, emails):
    query = "SELECT count(*) FROM data WHERE username = %s and year = %s and month =
        %s and day = %s"
    result = db.execute(query, (u, date.split("/")[2], date.split("/")[1],
        date.split("/")[0]), True)

    records = result[0][0]
    # obtener lista de sensores del usuario
    query = "SELECT sensor_name FROM data WHERE username = %s GROUP BY sensor_name"
    result = db.execute(query, (u,), True)
    sensors = [sensor[0] for sensor in result]

    # request para obtener cantidad de anomalías
    anom = {}
    endpoint =
        f"https://historical-tesis-mor19213.cloud.okteto.net/users/{u}/sensores/"
    for sensor in sensors:
        endpoint += f"{sensor}/ml"
        endpoint += f"?begin_date={date}&end_date={date}"
        try:
            result = requests.get(endpoint).json()["valores"]
```

```
        except Exception as e:
            result = []
            print(f"Error getting statistics from backend {e}")

        anom[sensor] = result

    body = f"""- {records} actualizaciones enviadas para {len(sensors)} sensores"""
    if len(sensors) != 0:
        body += f"""<br><h2>Anomalias por sensor:</h2>"""
    for sensor in sensors:
        anomaly_count = sum(1 for res in anom[sensor] if res["anomaly"] == False)
        body += f"""- {sensor}: {anomaly_count} anomalías detectadas<br>"""

    body += f"<br><br>Para más información, ingresa a <a
        href='https://web-tesis-mor19213.cloud.okteto.net'>Plataforma de
        orquestación de dispositivos IoT</a>"
    print(f"Sending email to {email} for date {date}")
    result = send_email(email, u, body, date)
```

Código 9.10: Programa de Python para enviar correos automáticos

9.5. PowerBI como herramienta de visualización del uso de la plataforma

El tablero de Power BI desarrollado se utiliza para poder observar la información histórica de los usuarios y dispositivos en la plataforma, permitiendo que el administrador de la plataforma pueda monitorear el uso de la misma.

```
SELECT
    TO_DATE(CAST(statistics.fa as varchar), 'YYYYMMDD') as fa,
    user_count,
    sensor_count,
    actuador_count,
    COUNT(DISTINCT username) AS active_users,
    COUNT(DISTINCT CONCAT(username, sensor_name)) AS active_sensors,
    COUNT(value) AS trafico
FROM statistics
LEFT JOIN data ON CAST(statistics.fa as VARCHAR) = CONCAT(data.year,
    TO_CHAR(data.month, 'FM00'), TO_CHAR(data.day, 'FM00'))
GROUP BY fa, user_count, sensor_count, actuador_count
```

Código 9.11: Consulta SQL para obtener la información histórica

El tablero final, consta de una sola página, la cual puede ser observada en la Figura 11. Primero cabe mencionar que se cuenta con un elector de fecha que permite seleccionar el rango de fechas que se desea observar. En la parte superior se observan estadísticas generales de la plataforma, como la cantidad de usuarios registrador y activos, sensores registrador y activos, los actuadores registrados en la plataforma y el promedio de actualizaciones de los sensores por día. En la parte inferior se observan gráficas que muestran la cantidad de usuarios activos y registrados por día, la proporción del estado de los sensores y del tipo de dispositivo, y la cantidad de actualizaciones de los sensores por día.

Figura 11: Tablero de Power BI con información histórica de la plataforma

CAPÍTULO 10

Frontend de la plataforma

10.1. Contexto de autenticación utilizando Axios

Como se ha mencionado previamente, en la plataforma, el *frontend* es el encargado de enviar el token de acceso en cada petición que se haga al *backend*. Para lograr esto eficientemente, se utiliza la librería Axios, que permite configurar un interceptor de peticiones HTTP. Este interceptor se encarga de agregar el *token* de acceso a cada petición que se haga al *backend* en el encabezado del paquete, y de refrescar el token de acceso cuando este expira. Es decir, este programa envía la petición HTTP al *backend* y espera a recibir la respuesta. Si dicha respuesta es exitosa, la ejecución de este programa finaliza. Sin embargo, en caso la respuesta del *backend* sea *403 Not Allowed*, se debe de actualizar los *tokens*. Para realizar esto, debe de enviar una petición *Post* al *endpoint /tokens/refresh* con el token de actualización. A partir de esto, existen dos casos. El primero se debe a que el *token* de actualización se encuentre vencido, lo cual causaría que la respuesta de la petición anterior no sea exitosa. En este caso, el programa elimina los *tokens* almacenados localmente y redirige al usuario a la página *Home*. En el segundo caso, el *token* de actualización es válido y el *backend* envía un nuevo *token* de acceso y actualización.

Este interceptor se creó como un *hook* de React, con el nombre *useAxios*.

10.2. Componentes y páginas

El *frontend* consta de las páginas:

- Home

- Crear Cuenta
- Iniciar Sesión
- Ajustes de cuenta
- Dispositivos
- Condiciones
- Historial

Todas las páginas tienen en la parte superior un menú para navegar a otras páginas. Al estar iniciada la sesión, se muestra un *link* para cerrar sesión y los *links* a las páginas de Ajustes de cuenta, Dispositivos y Condiciones. Sin embargo, si no está iniciada la sesión, solamente se muestran *links* a las páginas *Home*, Iniciar sesión y crear cuenta.

10.2.1. *Home*, Iniciar Sesión y Crear Cuenta

La página *Home* es una página de bienvenida y la única diferencia si la sesión está iniciada o no, es que al estar iniciada la sesión se muestra el nombre de usuario, esto se puede observar en la Figura 12.

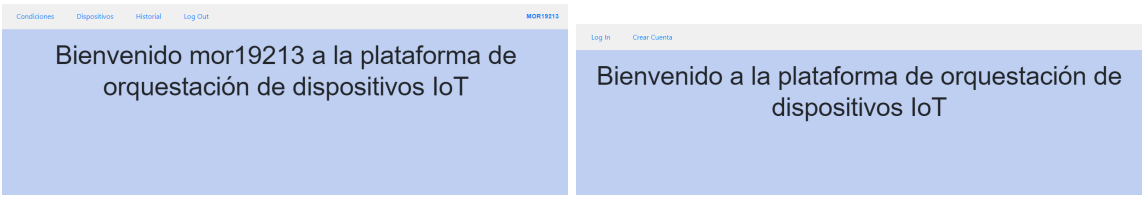


Figura 12: Página Home

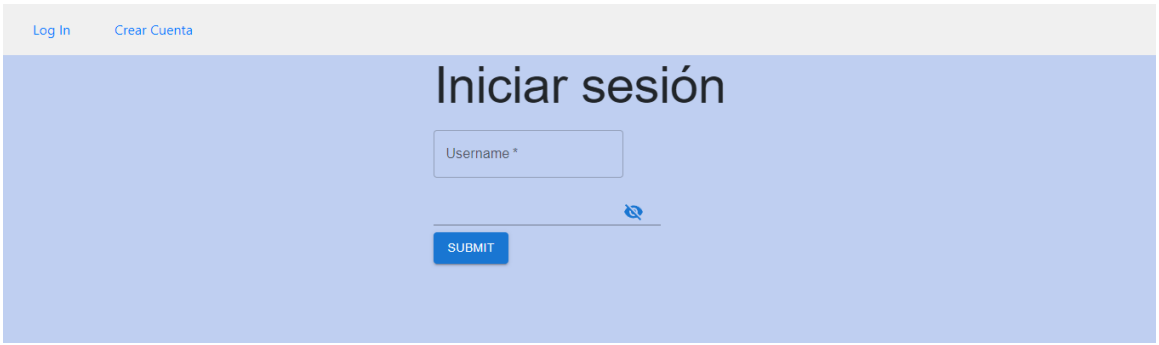


Figura 13: Página Inicio de Sesión

En las páginas Iniciar sesión y Crear cuenta se muestra un formulario, cada una con los campos necesarios para la petición HTTP que deben de enviarse. En el caso de la página de Iniciar sesión, los campos en la interfaz y que se envían son el nombre de Usuario y contraseña. Mientras que, en el caso de la página de Crear Cuenta, los campos en la interfaz son el nombre de usuario, contraseña, nombre, apellido y correo. Este formulario solamente se puede enviar si los dos campos de contraseña contienen el mismo valor, el nombre de

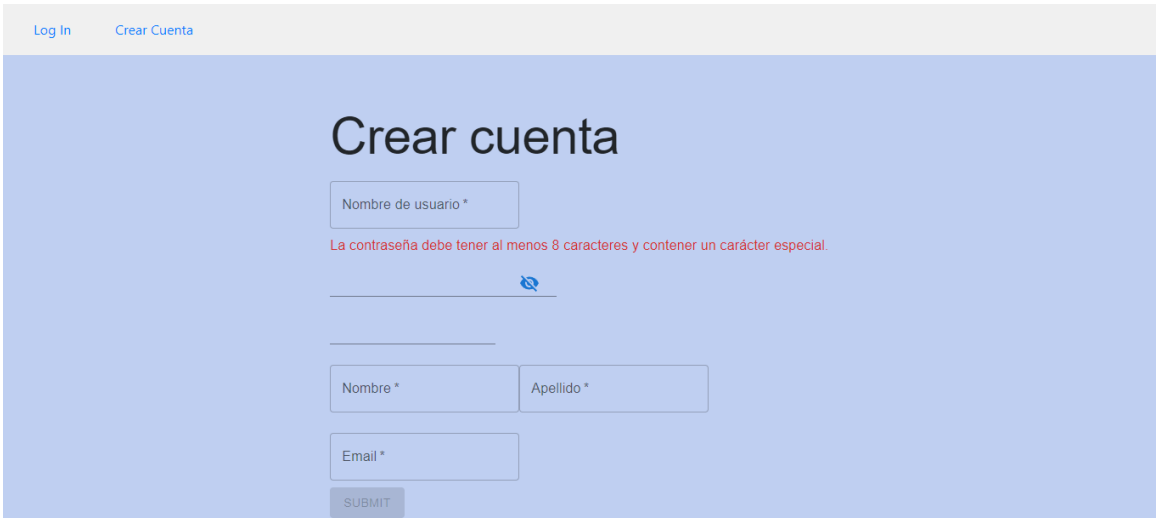


Figura 14: Página Crear Cuenta

usuario no se encuentra repetido y el correo no es usado en alguna otra cuenta. En la Figura 13 y en la Figura 14 se pueden observar la interfaz de las páginas Iniciar Sesión y Crear Cuenta, respectivamente.

Como se ha mencionado en la sección anterior, al realizar la petición *Post* para iniciar sesión el *React Hook* UseAxios se utiliza para almacenar localmente los *tokens* del usuario.

10.2.2. Dispositivos

La página de los dispositivos del usuario se encuentra dividida en dos, en los actuadores y sensores. En cada sección se despliega toda la información pertinente de cada dispositivo que pertenece al usuario individualmente. En está página es donde el usuario debe de crear sus sensores y actuadores. También es posible eliminarlos o editar el nombre de los mismos. Al cargar la página por primera vez se realiza el proceso observado en la Figura 16. En esta figura se observa que al cargar la página se realiza una petición *GET* al *endpoint* /*username*/dispositivos, el cual devuelve un JSON conformado por una lista de sensores y actuadores. Esta información es la fuente de información de los componentes HTML de la página. Este mismo proceso es repetido cada 5 segundos, con el objetivo de mantener actualizado los valores de los sensores y estados de los actuadores. En la Figura 15 se puede observar la página de dispositivos.

En la sección de actuadores se puede observar que hay tres atributos del actuador mostrados, los cuales son: el nombre, si su estado es definido a través de la interfaz o por medio de condicionales definidas por el usuario y el estado actual.

Para definir si el estado del actuador es activado manualmente o por medio de condiciones se muestra un *checkbox* que el usuario puede manipular. En el caso que el estado del actuador sea definido por medio de la interfaz, se observa un *switch* que el usuario puede presionar para manipular el estado del actuador. En el caso contrario, el componente *switch* no es mostrado, y el estado se puede observar a partir del color del círculo al lado del nombre

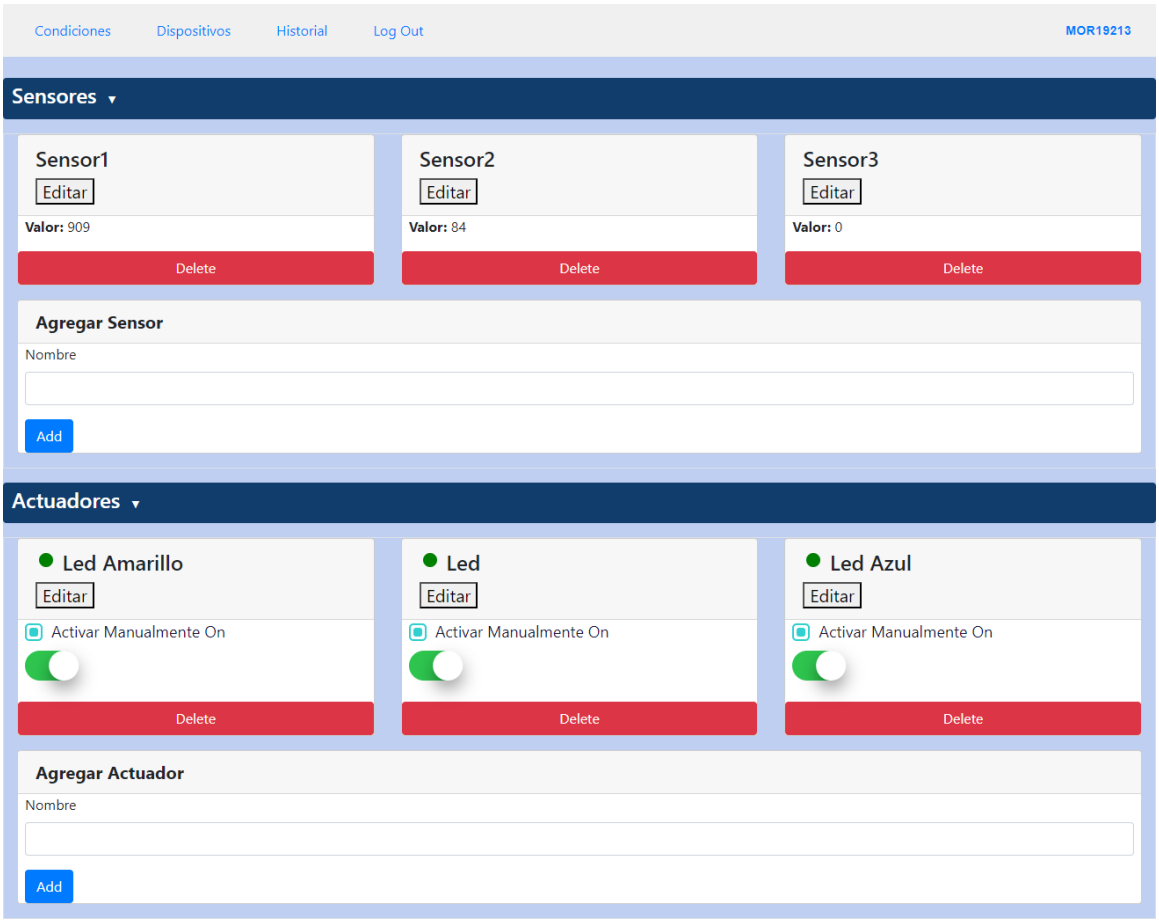


Figura 15: Página Dispositivos

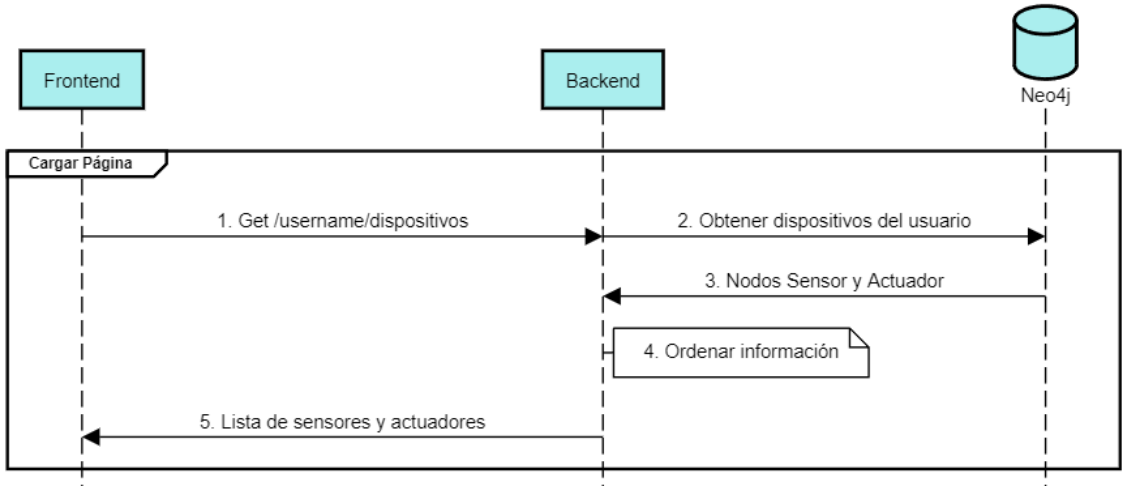


Figura 16: Flujo de comunicación entre microservicios al cargar la página Dispositivos

del actuador. Este círculo se muestra independientemente de si el estado del actuador es establecido manualmente o por medio de las condiciones.

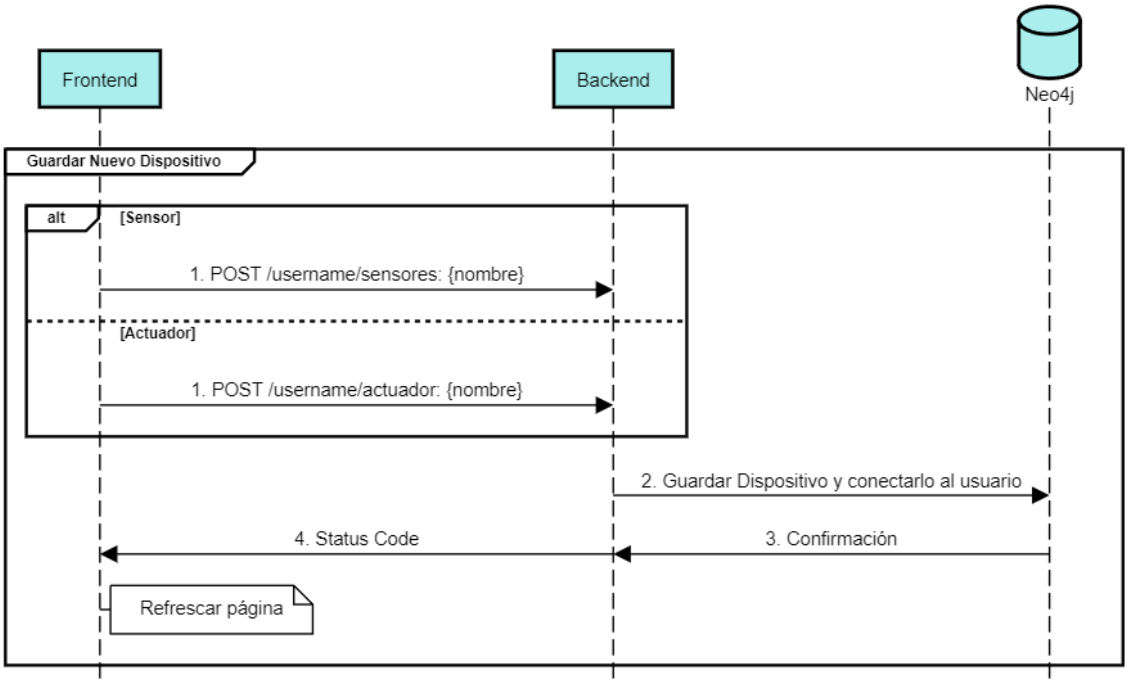


Figura 17: Flujo de comunicación entre microservicios para agregar un nuevo dispositivo

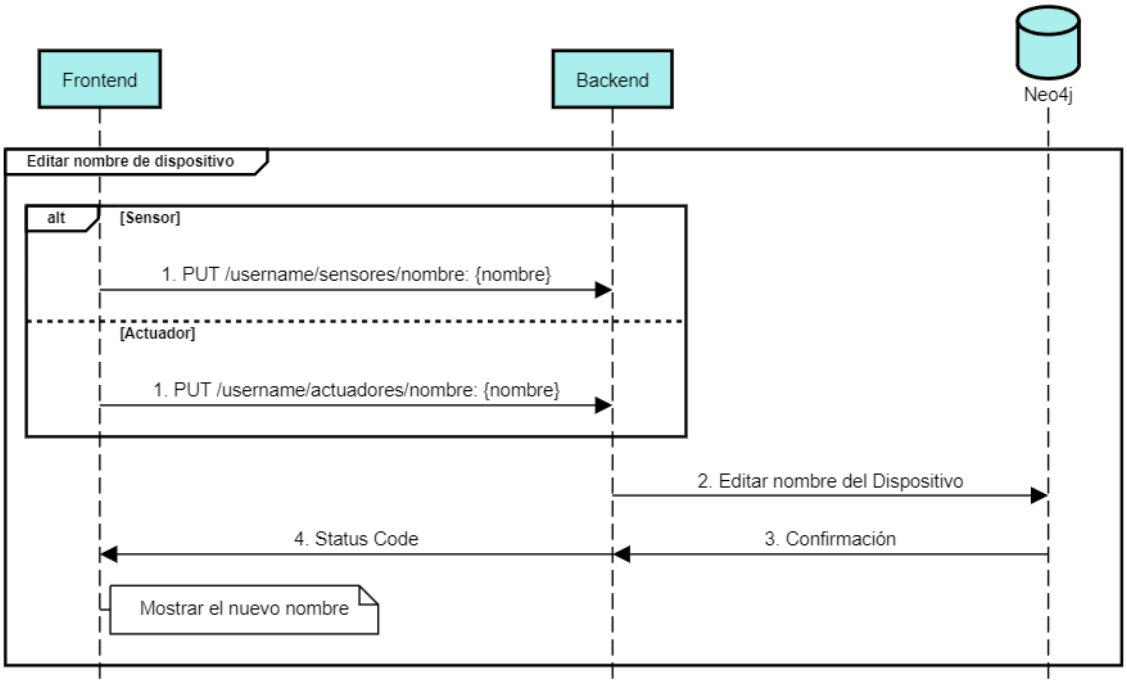


Figura 18: Flujo de comunicación entre microservicios para editar el nombre de un dispositivo

Dentro de esta página se pueden realizar tres operaciones con respecto al sistema IoT, las cuales son: agregar dispositivos, editar el nombre de los dispositivos, editar si el estado del actuador es establecido manualmente o por medio de las condiciones y editar el estado del actuador (en caso esté siendo establecido manualmente).

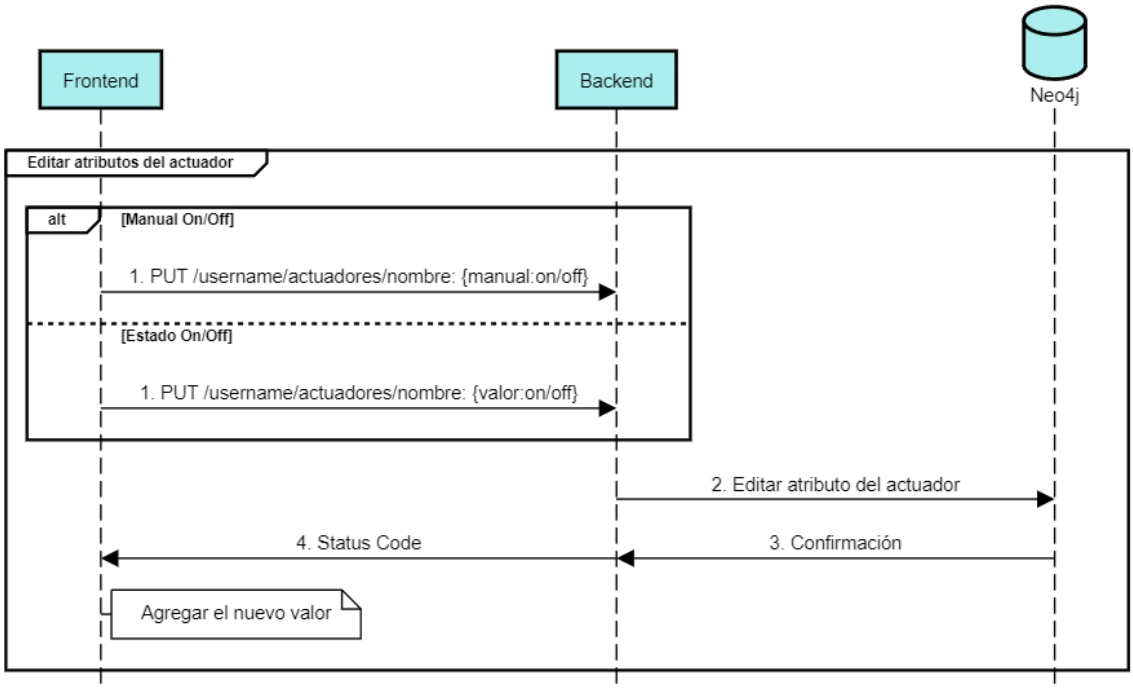


Figura 19: Flujo de comunicación entre microservicios para editar los atributos de un actuador

Para agregar dispositivos al sistema IoT, se debe de llenar alguno de los formularios que se puede observar en la página, los cuales se pueden observar en la Figura 15. En ambos formularios, únicamente se necesita el nombre del dispositivo y envían una petición *POST* que contiene en el *payload* un objeto con el atributo **nombre**, el cual es el nombre nuevo a asignar al dispositivo. El formulario en la sección de sensores envía la petición al *endpoint* /*username*/sensores y el formulario en la sección de actuadores al *endpoint* /*username*/actuadores.

Para la manipulación de los atributos manual y estado mencionada anteriormente, se sigue el proceso observado en la Figura 19. En esta figura se observa que en ambos casos, la petición HTTP se realiza al mismo *endpoint* solamente varía la información enviada en el *payload*. En ambos casos el *payload* es un objeto, pero solamente se envía una llave, esta llave es el atributo del objeto actuador que se desea editar. Este proceso es ejecutado luego de que el usuario presiona el *checkbox* "Activar Manualmente"o el *switch*.

La comunicación entre microservicios para editar el nombre de un dispositivo se puede observar en la Figura 18. En este flujo de comunicación se observa que primero se envía una petición *PUT*, el *endpoint* a donde se envía la petición depende de si el dispositivo es un sensor o un actuador. En caso de ser un sensor se envía la petición al *endpoint* /*username*/sensor/nombre y en caso de ser un actuador se envía al *endpoint* /*username*/actuador/nombre. Cabe mencionar, que el **nombre** enviado en el *endpoint* es el nombre actual del dispositivo. Para ejecutar este proceso, el usuario debe de presionar el botón "Editar" del dispositivo a editar. El cual desplegará un cuadro de entrada de texto y un botón al lado para enviar la información, es decir, iniciar el proceso.

10.2.3. Condiciones

Como se ha mencionado previamente, el estado de un actuador puede definirse por una operación lógica calculada entre el resultado de varias condiciones. Dichas condiciones son la evaluación de una desigualdad entre el valor de un sensor y una constante definida por el usuario. Estas condiciones son almacenadas en la base de datos de grafos como una relación entre un nodo Actuador y un nodo Sensor, con varios atributos. Entre estos atributos se encuentra la constante con la que será comparado el valor del sensor y el nombre de la condición. El atributo **nombre** de la relación indica qué condición será evaluada. Por lo tanto, en la página de condiciones se muestra un recuadro por cada actuador del usuario. En este recuadro se indica el resultado de las condiciones, cuál será la operación lógica que será calculada entre el resultado de las condiciones y la lista de las condiciones evaluadas. Por cada condición se observa el nombre del sensor, la condición que está siendo evaluada, la constante contra cual se compara el valor del sensor indicado, el resultado de la condición y el valor actual del sensor.

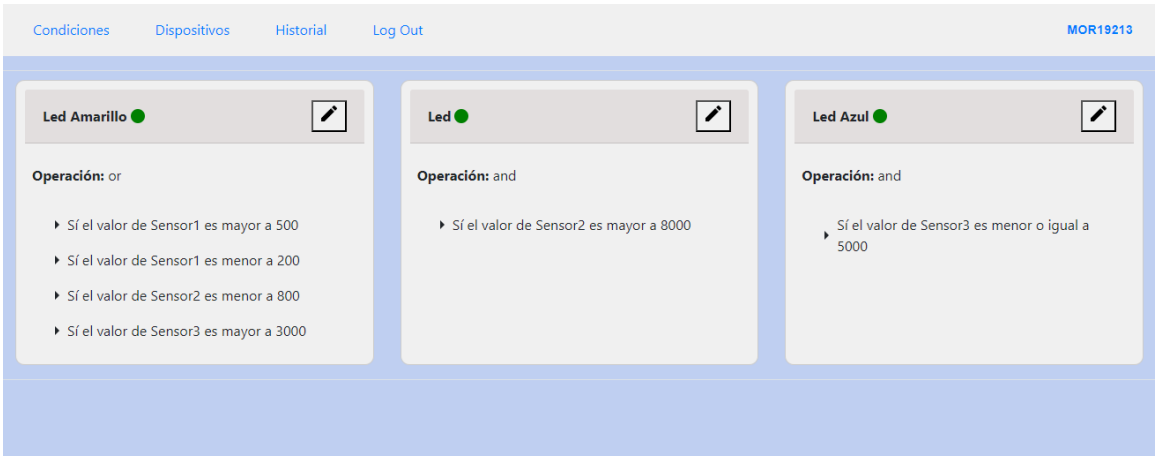


Figura 20: Página Condiciones

Para editar las condiciones y la operación que define a un actuador, se debe de presionar el botón en la esquina del recuadro del actuador a editar. Al presionar este botón se abre el modal que se puede observar en la Figura 21. Como se puede observar, en este modal es que se puede editar qué operación lógica es la que se realiza entre el resultado de todas las condiciones del actuador para definir el estado del mismo. También se puede editar las condiciones ya existentes del actuador o eliminarlas. Para realizar esto se debe de modificar el valor de cada campo de la condición y presionar el botón "Guardar"o presionar el botón "Eliminar", respectivamente. También se puede agregar nuevas condiciones al establecer un valor en cada campo en la sección "Nuevas Condiciones"del modal y luego presionar "Agregar". Finalmente, para cerrar el modal se debe de presionar el botón finalizar, el cual refrescará las páginas para mostrar los cambios realizados. Cabe mencionar, que al presionar este botón no se envía ninguna petición al *backend* puesto que todos los cambios son guardados al presionar los demás botones en el modal, los cuales fueron mencionados anteriormente.

Al cargar la página de condiciones se ejecuta el proceso mostrado en la Figura 22. En esta figura se observa que para obtener la información necesaria en la página se realizan

Figura 21: Modal de la página Condiciones

dos peticiones *GET* HTTP. La primera es al *endpoint* /username/condiciones, para obtener todos los actuadores del usuario y las condiciones de cada uno, independientemente de si el estado del actuador es definido por medio de estas condiciones o por la interfaz gráfica. La segunda petición es al *endpoint* /username/sensor, para obtener una lista de todos los sensores del usuario. Esta información es necesaria para colocar cada uno de estos sensores como una opción en el formulario del modal para editar o agregar condiciones.

Como se mencionó anteriormente, en el modal se pueden realizar cuatro distintas operaciones. La primera operación, es editar la operación lógica del actuador, el flujo de comunicación se puede observar en la Figura 23. En esta figura se observa el proceso que se ejecuta luego de rellenar el campo "Operación lógica"y presionar el botón "Guardar". En este proceso, primero se envía una petición *PUT* al *endpoint* /usuario/condiciones/nombre, siendo nombre el nombre del actuador a editar, y en el *payload* de la condición se envía un objeto con la operación como su único atributo. A partir de esto, el *backend* solicita a la base de datos el actuador a ser editado. Luego de obtener el actuador, se edita el atributo **operacion** del actuador y se guarda. Finalmente, en base al mensaje recibido por la base de datos se devuelve un distinto código de estado distinto.

La segunda operación es editar una condición, dicho flujo de comunicación puede ser observado en la Figura 25. Dicho proceso observado se ejecuta luego de editar los campos de dicha condición y presionar "Guardar". Este proceso inicia con enviar una petición *PUT* al *endpoint* /username/sensor/nombre/uid enviando un objeto con los atributos comparación, valor y sensor en el *payload*. Al recibir esta petición, el *backend* solicita el nodo actuador y sensor implicados en la condición a la base de datos. A partir de tener ambos nodos, se solicita la relación entre estos nodos con un **uid** equivalente al enviado en el *path* de la petición. Luego de que el *backend* ya posee los nodos actuador y sensor, y la relación indicada en la condición, se procesan tres posibles cambios. Estos cambios son: cambio de sensor, de la comparación realizada y el valor al que se compara. Cabe mencionar, que no es necesario que se envíen los tres campos para realizar la edición. Para cambiar el sensor

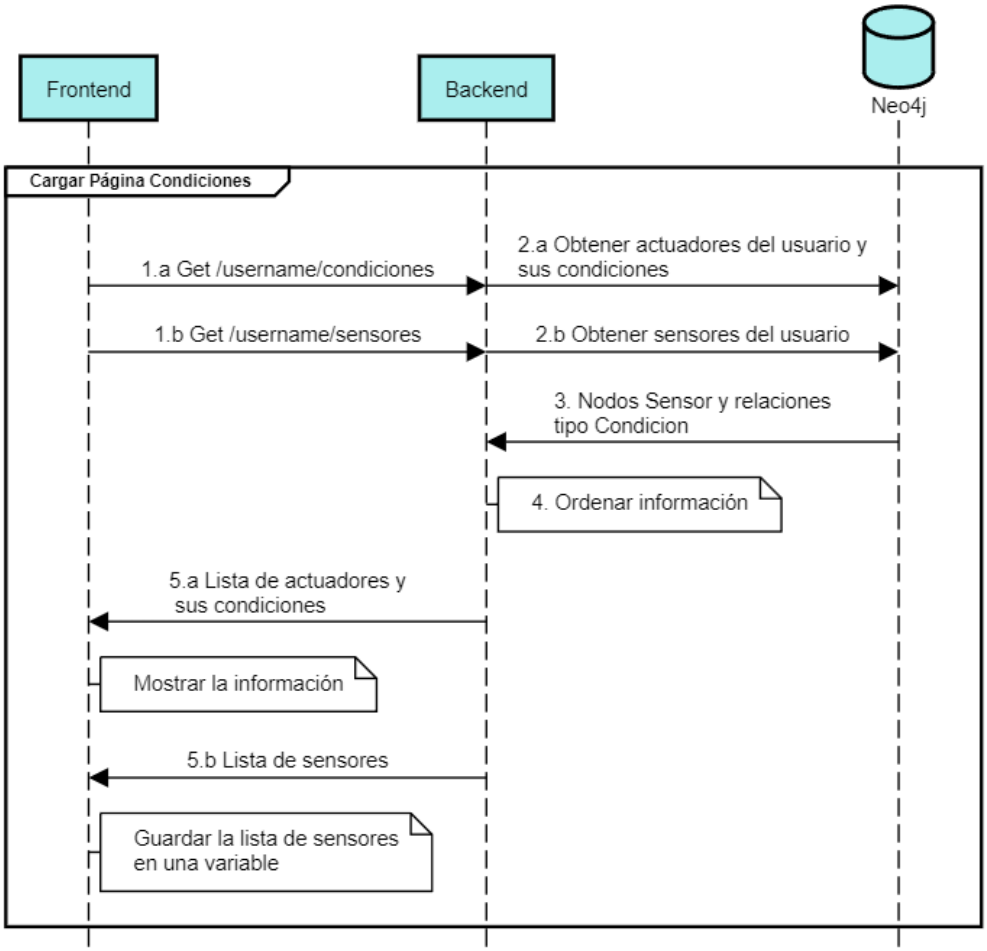


Figura 22: Flujo de comunicación entre microservicios para cargar la página Condiciones

de la condición se debe de eliminar la relación que existe actualmente y crear una nueva relación con los mismos atributos, entre estos incluido el **uid**, hacia el nuevo sensor. En el caso de editar la comparación o el valor a comparar, el proceso es similar, puesto que se debe de editar el atributo de la relación y se procede a guardar la relación. Luego de haber realizado los casos que aplican anteriormente, a partir de lo enviado en el *payload*, se finaliza el proceso enviado al *frontend* el código de estado.

La tercera operación es la eliminación de una condición, este flujo de comunicación puede ser observado en la Figura 25. Como se observa en la figura, este proceso inicia enviando una petición *DELETE* al *endpoint* /username/condiciones/nombre/uid. Al recibir esta petición, el *backend* solicita a la base de datos el nodo actuador, el nodo sensor y la relación entre ellos. A partir de esto, procede a eliminar la relación y enviar el código de estado al *frontend*.

Finalmente, la cuarta operación es la creación de una nueva condición, dicho flujo de comunicación entre microservicios se puede observar en la Figura 26. Este proceso se ejecuta luego de rellenar todos los campos en la sección del modal "Nuevas condiciones"y presionar el botón "Agregar". El primer paso es realizar una petición *POST* al *endpoint* /username/condiciones/nombre, el cual causa que el *backend* solicite los nodos actuador y sensor

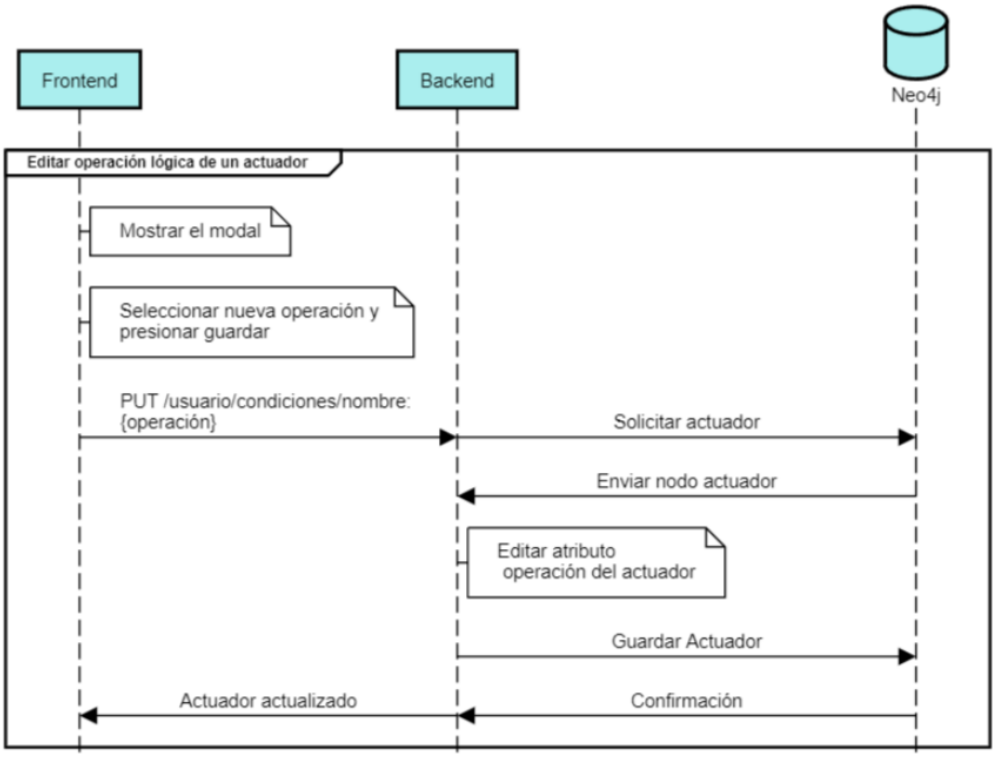


Figura 23: Flujo de comunicación entre microservicios para editar la operación lógica de las condiciones de un Actuador

implicados en la relación a la base de datos y luego crear una relación entre los mismo con los atributos indicados en el *payload* del *POST*. Luego de esto, el *frontend* recibe únicamente el código de estado de la petición.

10.2.4. Historial

En la página de historial se muestra una tabla con el historial de los sensores del usuario. Para desplegar la tabla debe de seleccionarse el sensor, la lista de sensores se obtiene del *endpoint* /users/*username*/sensores de la API del historiador. Por lo tanto, únicamente se muestran los sensores de los cuales se ha ingestado información a la plataforma. A partir de esto, cada vez que se selecciona un nuevo sensor se realiza una petición *GET* al *endpoint* /users/*username*/sensores/*sensor*/ml de la API del historiador con el rango de fecha seleccionado. En la información que se obtiene en la respuesta de la petición se encuentra, la hora y fecha de ingestión de la data, el valor y si es considerado una anomalía o no. Por lo tanto, en el *frontend* los valores catalogados como anomalías que colocan en rojo en la tabla.

10.2.5. Ajustes

En la página de Ajustes de Cuenta es donde el usuario puede eliminar su cuenta, al igual que editar su nombre, apellido u contraseña.

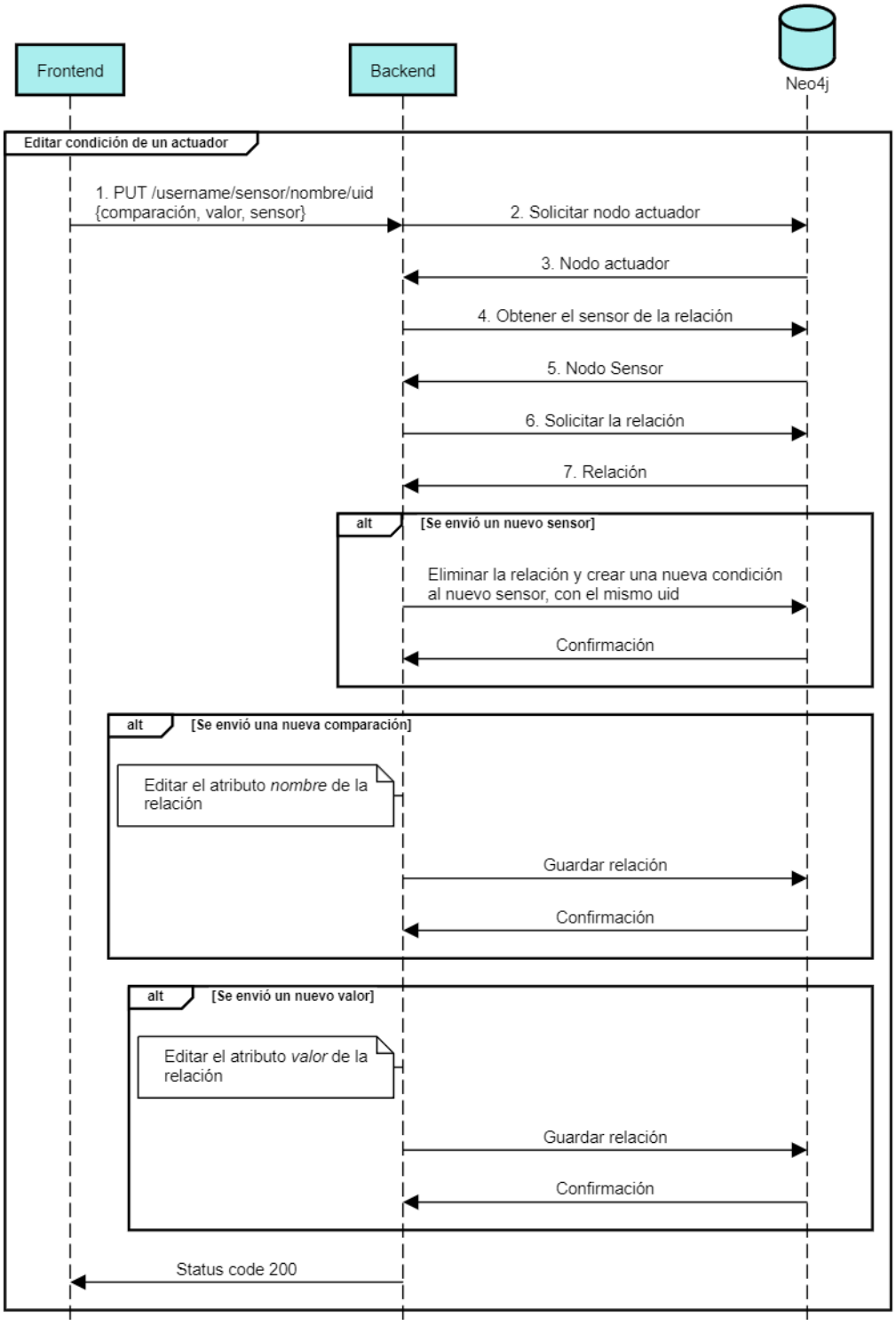


Figura 24: Flujo de comunicación entre microservicios para editar Condición de un Actuador

Cabe mencionar que para eliminar la cuenta primero se debe de seleccionar el *checkbox* que tiene el texto "Eliminar cuenta". Si este *checkbox* se encuentra seleccionado se muestra el botón rojo que elimina la cuenta al ser presionado. El fin del *checkbox* es agregar un nivel

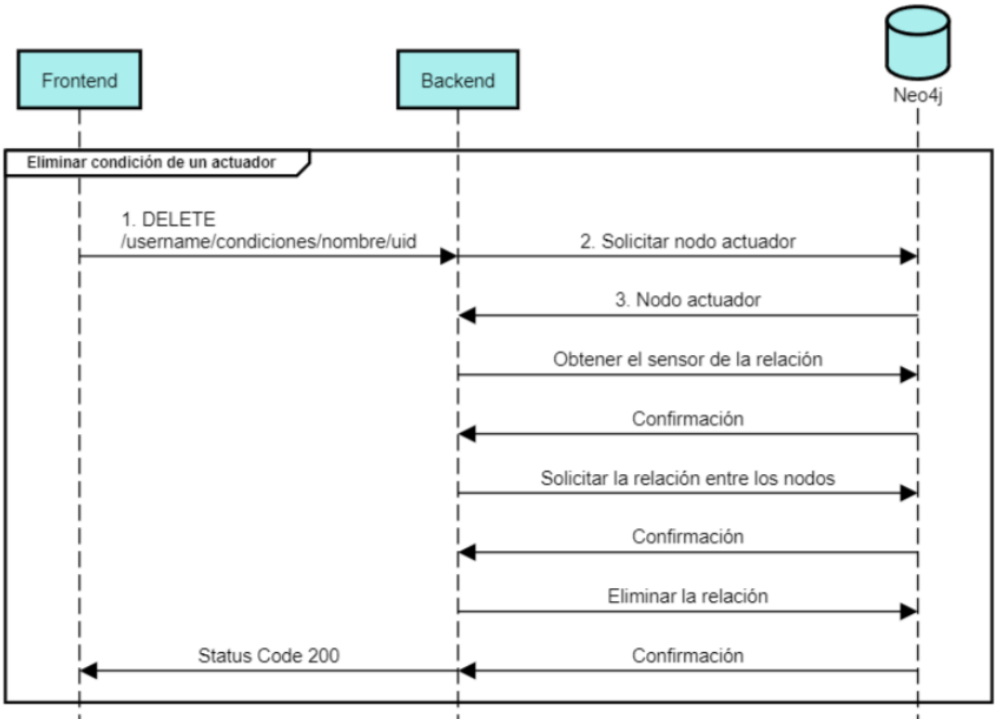


Figura 25: Flujo de comunicación entre microservicios para eliminar Condición de un Actuador

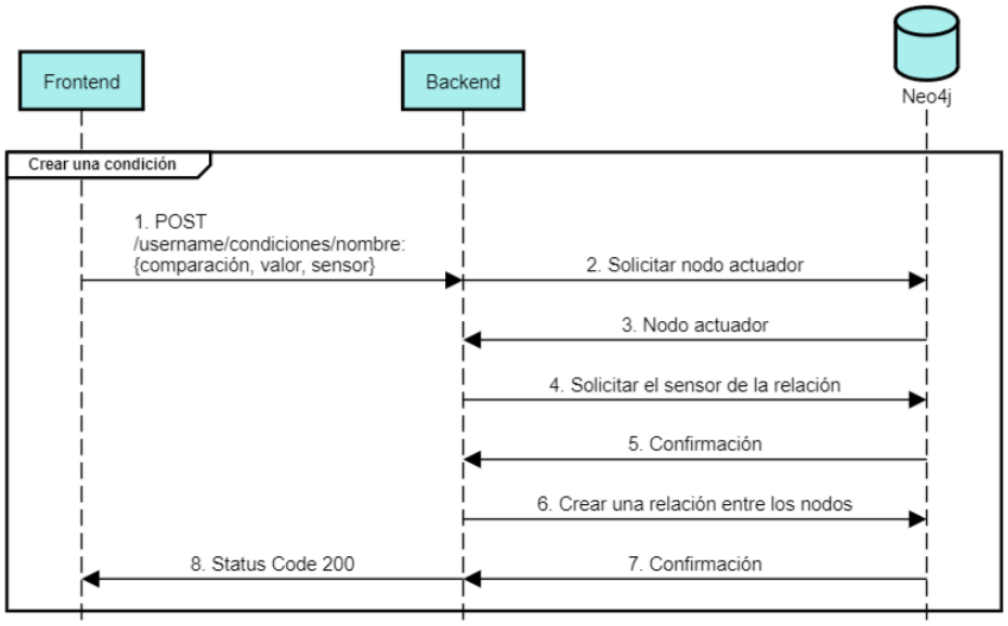


Figura 26: Flujo de comunicación entre microservicios para crear una nueva condición a un Actuador

de seguridad para evitar que el usuario elimine su cuenta accidentalmente. En la Figura 29 se observa el *checkbox* seleccionado y el botón para eliminar la cuenta, al igual que se puede observar los campos por medio de los cuales se pueden editar atributos del usuario. El flujo de comunicación entre microservicios para eliminar la cuenta se puede observar en la

Figura 27: Página Historial de los sensores

Figura 30. En dicho flujo se observa que primero el *frontend* realiza una petición *DELETE* al *endpoint* /usuarios/username. El *backend* al recibir esta petición procede a obtener el nodo del usuario de la base de datos y eliminarlo, a partir de esto espera a recibir la confirmación de la base de datos y devolver el código de estado al *frontend*.

También es posible editar otros atributos de usuario en esta página, estos atributos son el nombre, apellido y la contraseña, la cual, como se observa en la Figura 29 debe de ingresarse dos veces. Cada atributo es actualizado por separado, es decir, por cada atributo que se puede cambiar existe un botón para enviar la actualización al *backend*. A pesar de que los atributos a cambiar son distintos, el proceso ejecutado es el observador en la Figura 31. En este proceso se observa que primero se envía una petición *PUT* al *endpoint* usuarios/username, y en el *payload* coloca el atributo del usuario a ser editado. Luego, el *backend* solicita el nodo del usuario a la base de datos y edita los atributos que fueron enviados en el *payload*, guarda el nodo del usuario y luego de la confirmación de la base de datos le envía el código de estado al *frontend*.

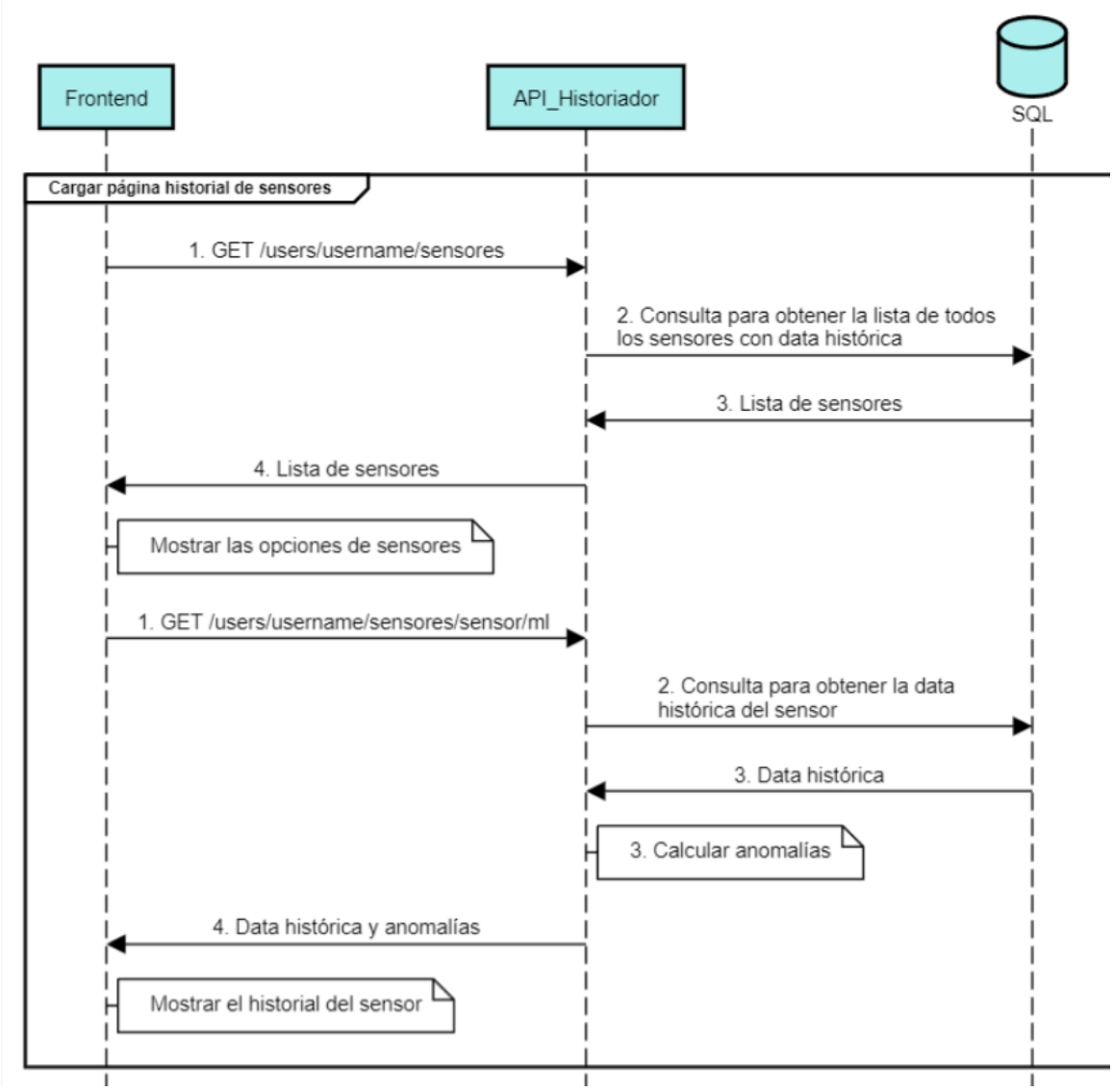


Figura 28: Flujo de comunicación entre microservicios para cargar la página de historial de sensores

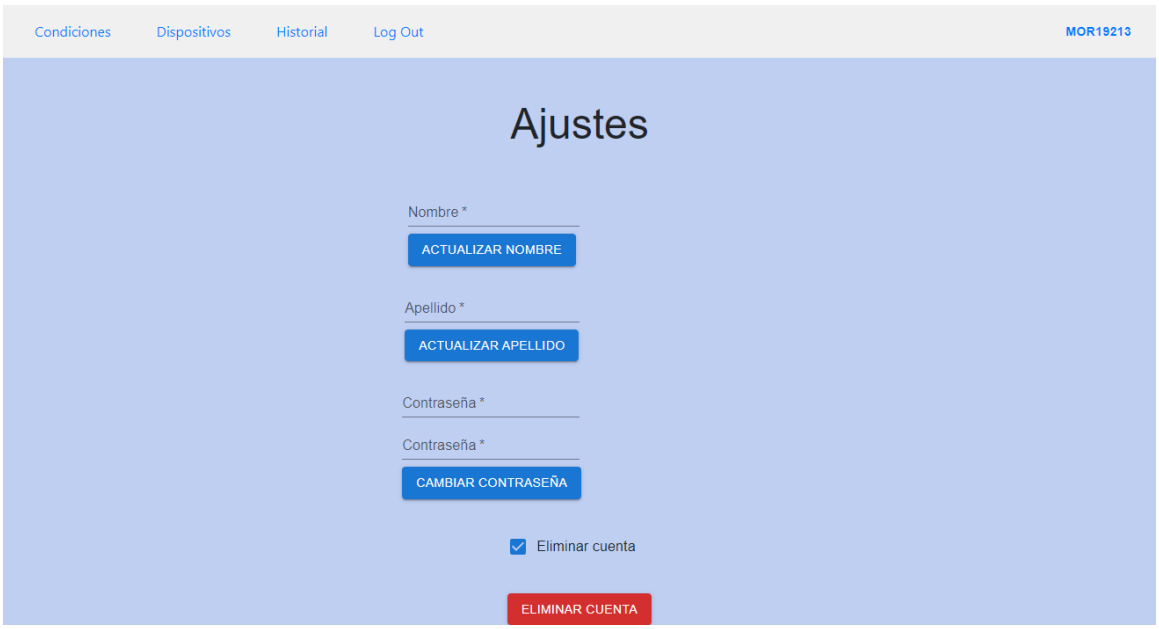


Figura 29: Página Ajustes de Cuenta

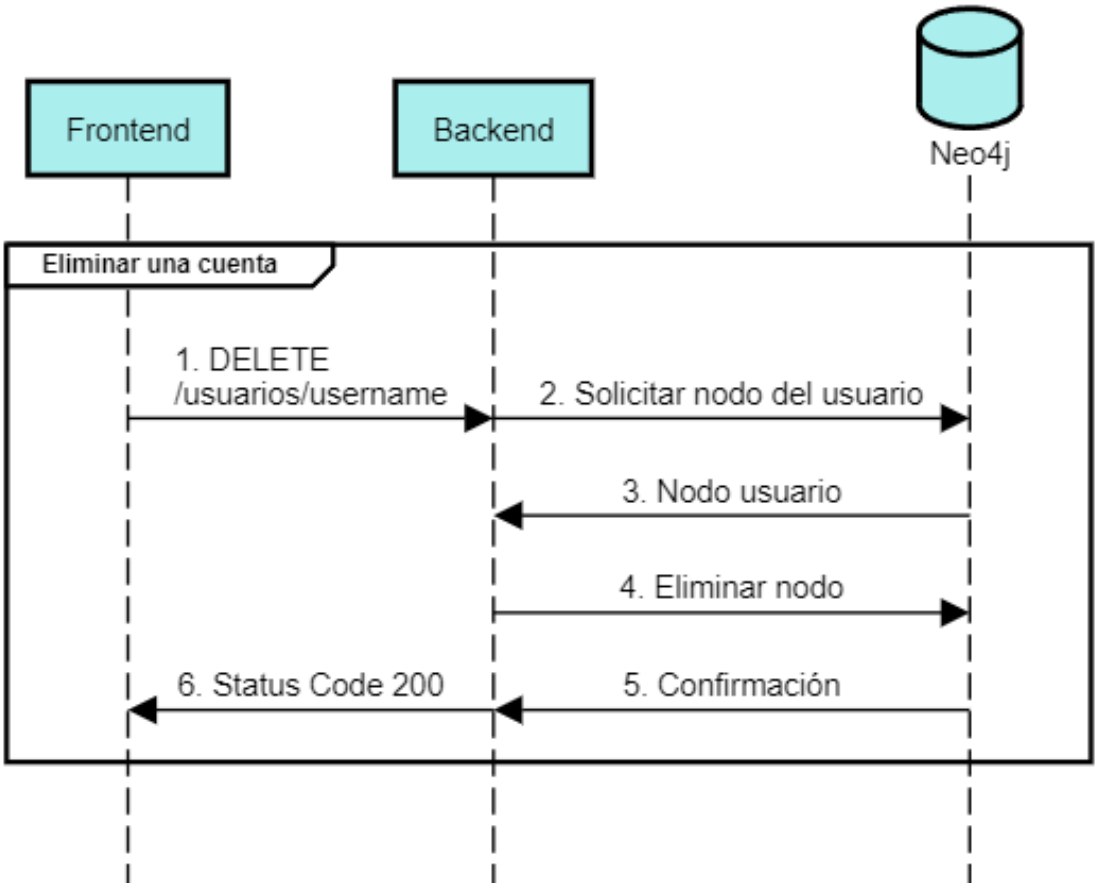


Figura 30: Flujo de comunicación entre microservicios para eliminar una cuenta

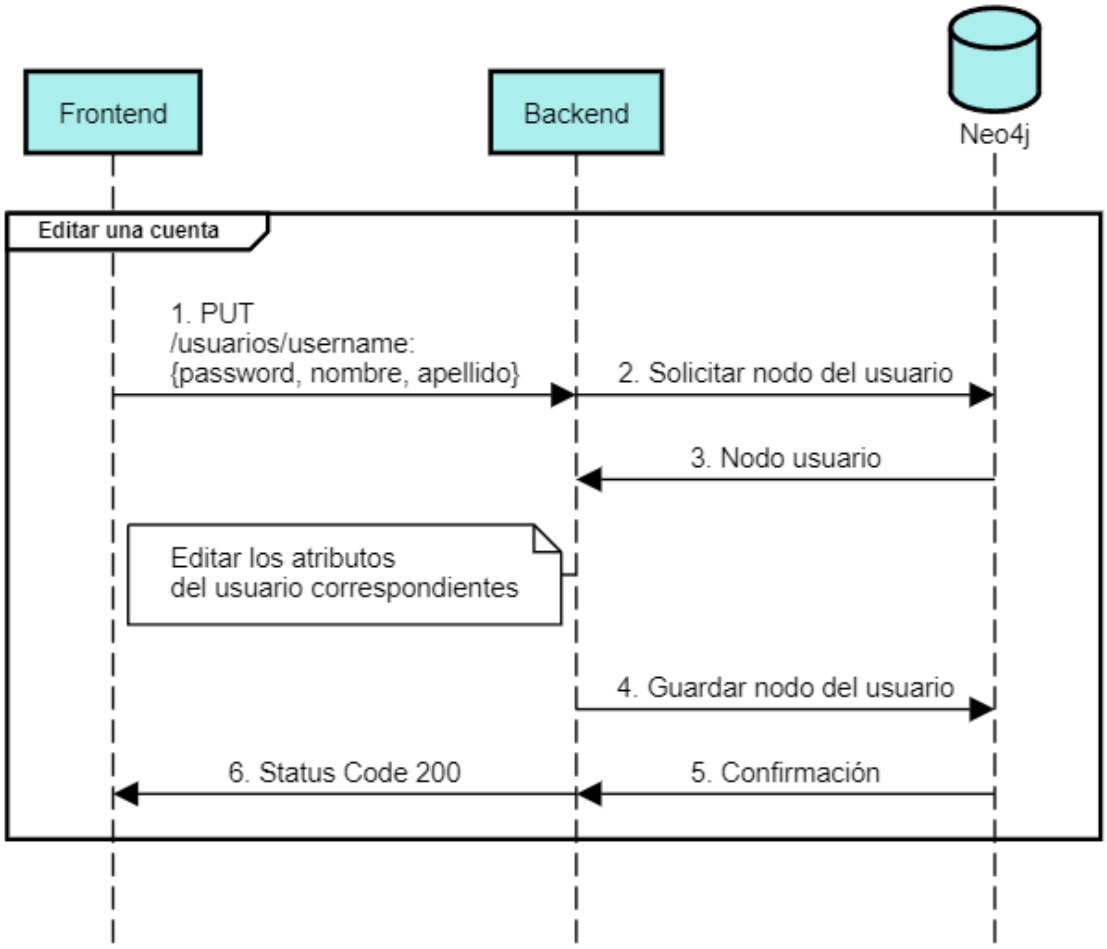


Figura 31: Flujo de comunicación entre microservicios para editar una cuenta

CAPÍTULO 11

Gateway de la plataforma

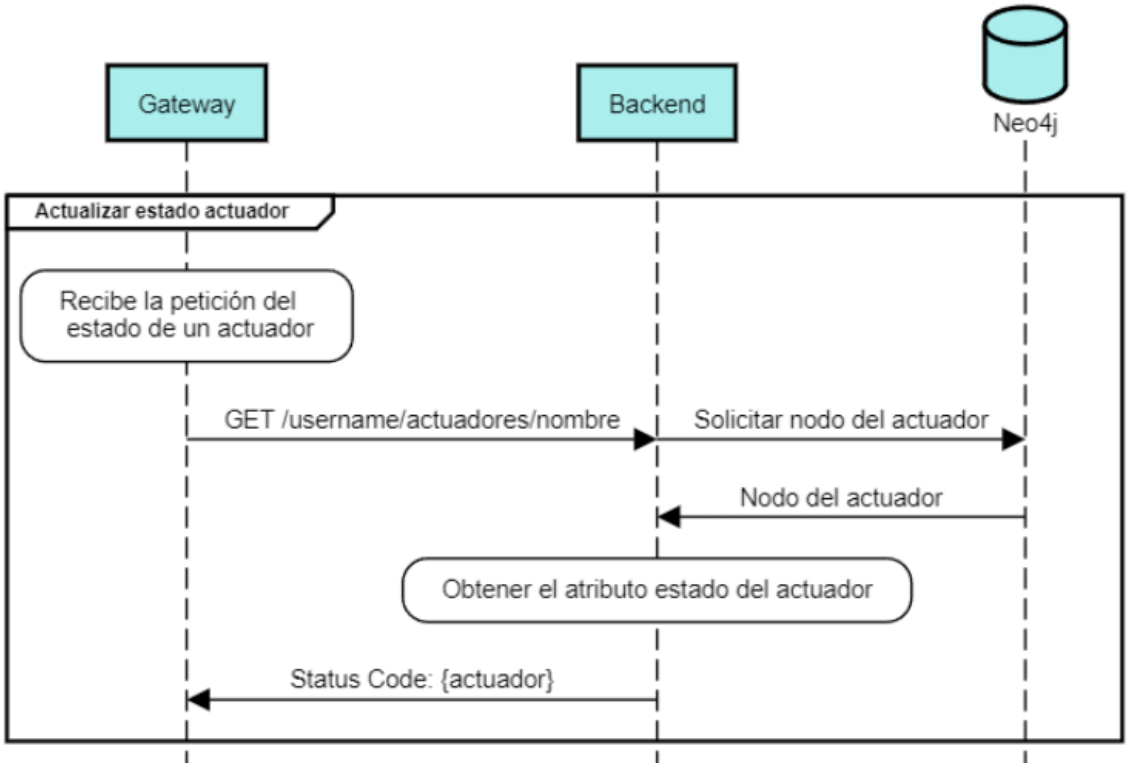


Figura 32: Flujo de comunicación entre microservicios para actualizar el valor de un actuador

El objetivo del *gateway* desarrollado es permitir la comunicación entre la plataforma en la nube y el sistema físico de cada usuario. Dicho *gateway* fue desarrollado en una *Raspberry Pi* con sistema operativo Ubuntu. Sin embargo, el *gateway* puede ser levantado en cualquier

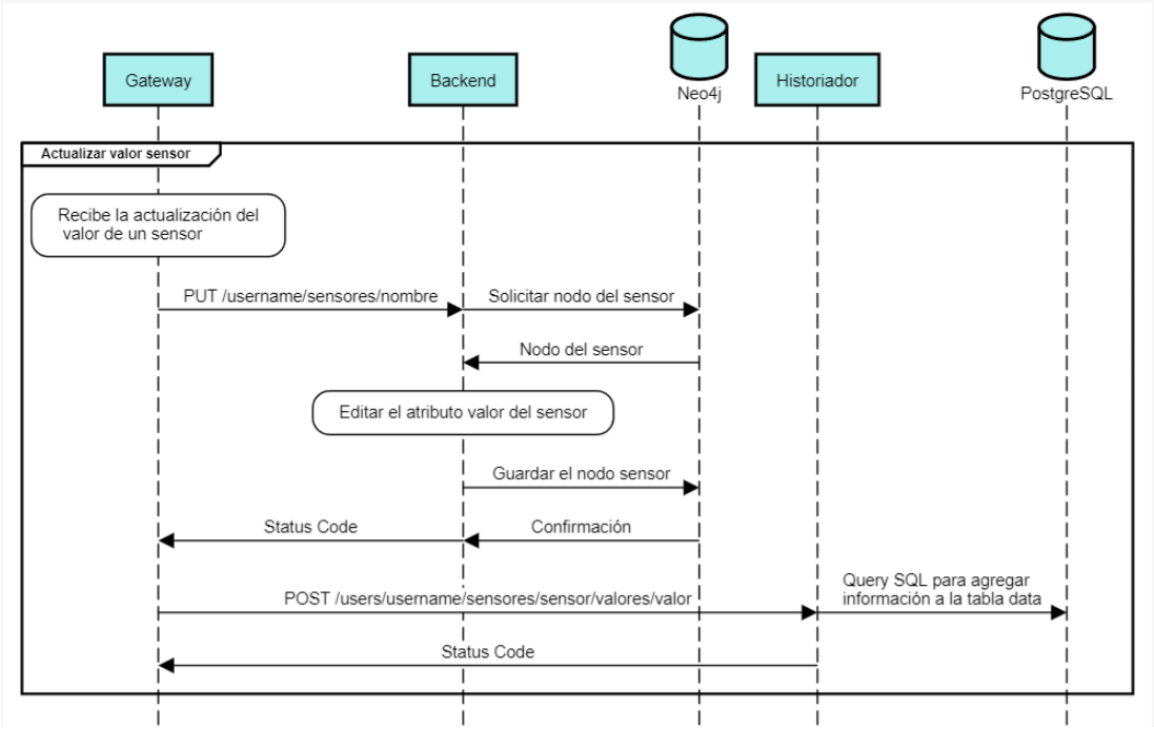


Figura 33: Flujo de comunicación entre microservicios para actualizar el valor de un sensor

sistema operativo con Python y un broker MQTT (en el caso que se desee implementar). Cabe mencionar que los programas para recibir y enviar mensajes en los distintos protocolos soportados fueron desarrollados en *Python*, al igual que el programa para enviar peticiones HTTP a la plataforma. Debido a que se desarrollo un programa por protocolo, el *gateway* es modular y es posible correr los programas de únicamente los protocolos que son de interés para el usuario.

11.1. Protocolos soportados

11.1.1. MQTT

El protocolo MQTT se levantó utilizando el *broker* Mosquitto. El programa de comunicación por medio de MQTT es simple debido al funcionamiento de los *topics* en MQTT. Primero se realizó un programa en *Python* en el que se crearon dos clases: *consumer* y *producer*. Las cuales, como lo indica su nombre, su objetivo es consumir o producir mensajes MQTT.

Luego se realizó el programa donde se manejan todos los mensajes MQTT que se reciben y se envían a los dispositivos del sistema físico. El flujo del manejo de mensajes MQTT se puede observar en la Figura 34. En el flujo se observa que primero el programa inicializa un productor y un consumidor de mensajes MQTT. El consumidor se suscribe a los *topics* "sensor/#"y "actualizar". Cabe mencionar que el símbolo "#"es un comodín que indica que cualquier mensaje enviado a un *topic* de primer nivel "sensor"será recibido por el programa,

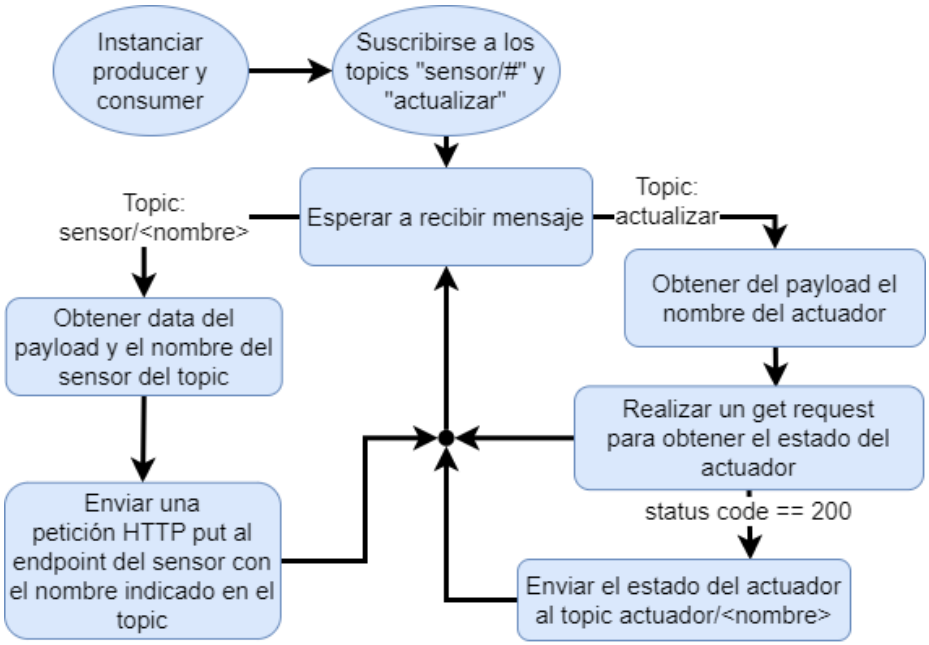


Figura 34: Flujo programa mqtt

independientemente de cual sea el *topic* en los subsiguientes niveles. En el siguiente nivel, luego de "sensor", lo que se coloca es el nombre de dicho sensor.

En el caso de recibir mensajes de un *topic* "sensor"primero se obtiene el *topic* en el segundo nivel, puesto que este es el nombre de dicho sensor. También, del *payload* se obtiene el valor actual del sensor. Ya teniendo el nombre del sensor y su valor actual, se envía una petición HTTP tipo *put* al *endpoint* para actualizar el valor. El *endpoint* para actualizar es "<str:username>/sensores/<str:nombre del sensor>/".

En el caso de recibir mensajes del *topic* "actualizar", se debe de obtener del *payload* el nombre del actuador. A partir de esta información, se envía una petición HTTP tipo *get* al endpoint "<str:username>/actuadores/<str:nombre del actuador>". Al recibir el *response* si el *status code* es distinto a 200, el programa regresa a esperar a recibir mensajes. Y si el *status code* es 200, se obtiene el estado del actuador enviado en el *response*. Este estado es enviado por medio de un mensaje MQTT al actuador, por medio del topic "actuador/<nombre del actuador>.

11.1.2. UDP

Para el protocolo UDP, la configuración necesaria consiste en configurar un *socket* para enviar y otro para recibir, al igual que un hilo para enviar y otro para recibir datagramas. Cabe mencionar que la información de los actuadores se guarda en una lista de diccionarios, la cual contiene el nombre del actuador, la IP del dispositivo que envío el datagrama para obtener información del actuador y el *timestamp* de cuando fue la última vez que el dispositivo envío un datagrama para obtener la información. Se guarda el nombre del actuador y la IP, ya que de esta forma es posible que existan varios dispositivos (con distintas IPs)

suscritas al mismo actuador en la plataforma. El objetivo de guardar la hora de la última vez que un dispositivo solicitó información de un actuador, es para eliminar la instancia de la lista, en caso pasen mas de 100 segundos desde la última vez que se solicitó. Puesto que en ese caso, se toma como que el dispositivo ya no se encuentra activo y no requiere la información.

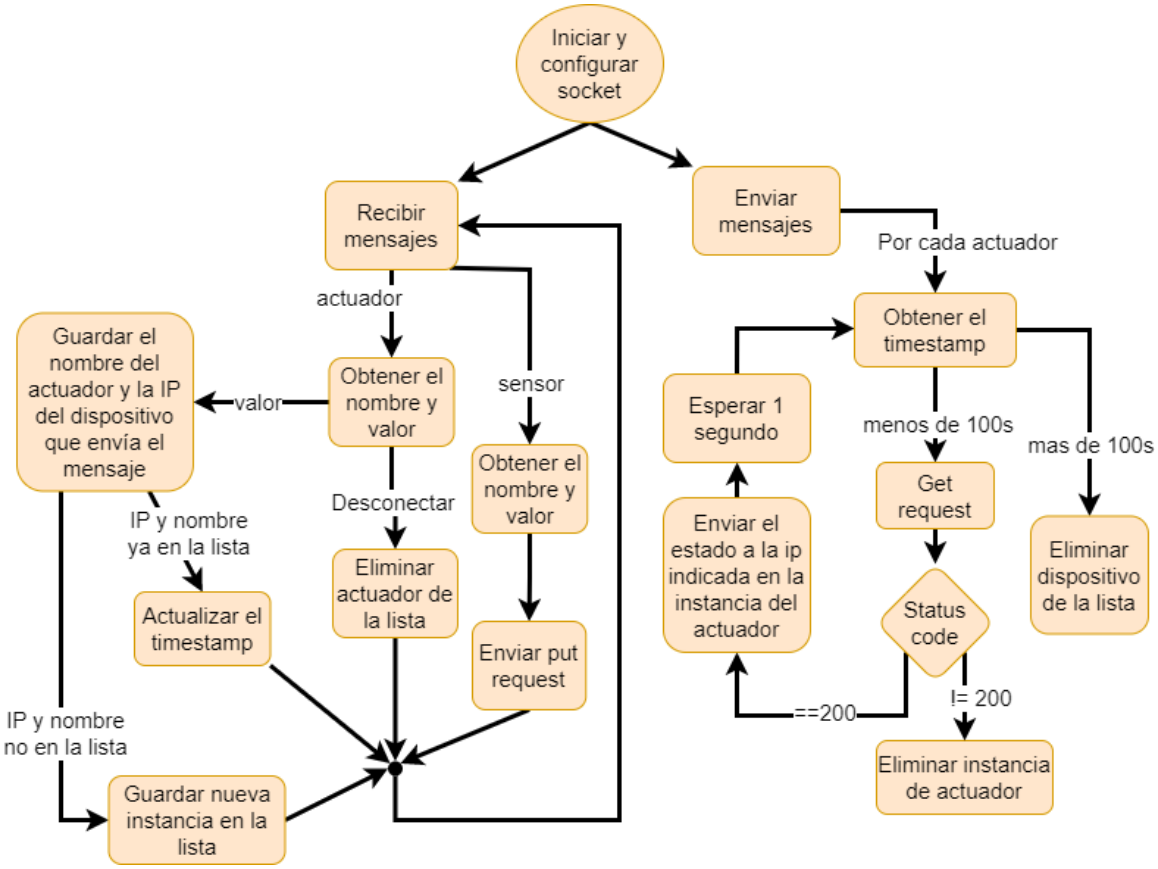


Figura 35: Flujo programa udp

El primer hilo es el que recibe los datagramas y a partir de la información en el datagrama se obtiene si se está enviando información de un actuador o de un sensor, al igual que el nombre de dicho dispositivo independientemente de si el dispositivo es un actuador o un sensor y el valro enviado. En el caso de que el datagrama contenga información de un sensor, se envía una petición HTTP *put* al *endpoint* "<str:username>/sensores/<str:nombre del sensor>"con el valor actual del sensor. Por otro lado, en el caso de que el datagrama contenga información de un actuador, primero se revisa el contenido del valor enviado. Si el valor enviado es "desconectar", se elimina la instancia de dicho dispositivo solicitando ese actuador específicamente. Por otro lado, si el valor enviado es un *string* distinto a "desconectar", se debe de actualizar el estado del actuador. Para esto, primero si en la lista de actuadores ya se encuentra un diccionario con el mismo nombre e IP, se actualiza el *timestamp* del dispositivo a la hora actual. En el caso que la combinación de nombre de actuador e IP no existe en la lista se guarda el nombre del actuador, la IP del dispositivo que lo envió y la hora actual. Como se explicó anteriormente, el único objetivo de este hilo es recibir datagramas, mantener actualizado el valor de los sensores y almacenar en una variable la información de los actuadores.

El segundo hilo tiene como objetivo obtener el estado de todos los actuadores guardados y enviar el estado actualizado del actuador al dispositivo que indica la IP. Por lo tanto, se realiza un ciclo *for* a través de la lista de actuadores. Por cada actuador primero se revisa que el tiempo transcurrido entre el *timestamp* y el tiempo actual sea menor a 100 segundos. En caso, el tiempo transcurrido sea mayor a 100 segundos, se elimina la instancia del actuador en la lista. En el caso contrario, se envía una petición HTTP *get* al *endpoint* "<str:username>/actuadores/<str:nombre del actuador>". Si el *status code* de la respuesta a la petición es 200, se envía un datagrama udp a la IP indicada en el diccionario.

11.1.3. Bluetooth

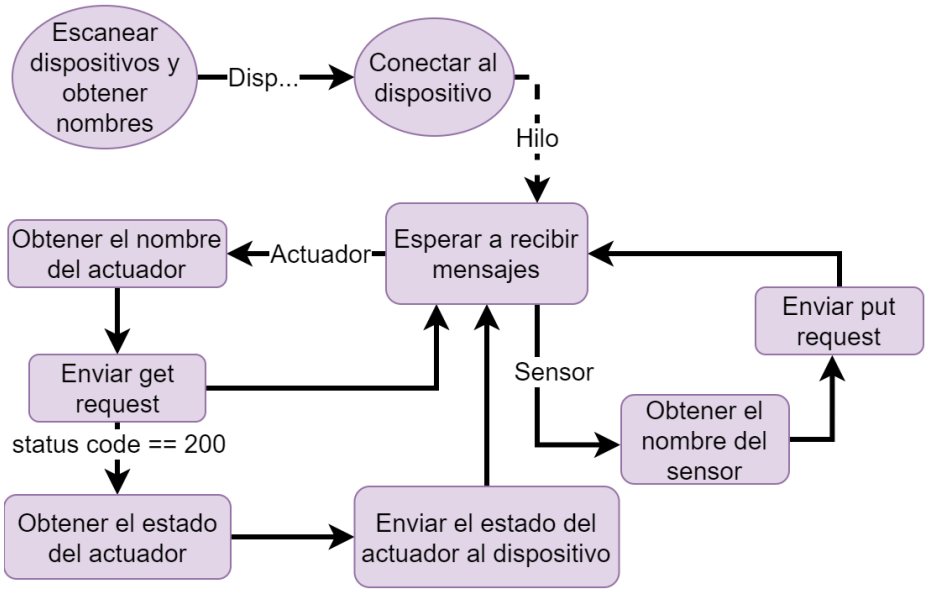


Figura 36: Flujo programa Bluetooth

Para el último protocolo que soporta el *gateway* se realizó un programa que escanea y se conecta con los dispositivos cuyo nombre inica con "Disp". Por cada dispositivo conectado, crea un hilo que mantiene la comunicación con dicho dispositivo y maneja actualización del valor de los sensores y actuadores.

Específicamente, cada hilo se encuentra esperando a recibir un mensaje. Al recibir, revisa si el mensaje recibido contiene el *string* "sensor"o "actuador". En caso de que contenga "sensor"obtiene el nombre de dicho sensor y envía una petición *put* con el valor actual. Por otro lado, si contiene "actuador"se obtiene el nombre de dicho actuador y se procede a enviar una petición *get* para obtener el estado actual. Si el *status code* de la respuesta a la petición es 200, se regresa a esperar a recibir un nuevo mensaje, ya que esta no fue exitosa. En el caso contrario, del *payload* de la respuesta se obtiene el estado del actuador. Y finalmente, se procede a enviar, por medio de bluetooth, el estado del actuador al dispositivo que solicitó dicha información.

CAPÍTULO 12

Sistema físico

El sistema físico utilizado para realizar las pruebas de la plataforma consiste de 6 microcontroladores ESP32. Se tienen 2 microcontroladores por cada protocolo que es soportado por el *gateway*, uno con un sensor y otro con un actuador. Sin embargo, la programación del *gateway* fue realizada de una forma que permitiera que en cada microcontrolador se tuvieran múltiples sensores y actuadores simultáneamente.

Por lo tanto, se desarrollaron 6 programas para los microcontroladores ESP32. Dichos programas fueron desarrollados en la IDE de Arduino en el lenguaje C y utilizando librerías para microncontroladores ESP en C. En el caso de todos los programas fue necesario utilizar la librería "ArduinoJson.h", ya que todos los datos se envían en formato JSON. En el caso de los microcontroladores que se comunican con el *gateway* por medio de bluetooth se utilizó la librería "BluetoothSerial". Para los protocolos MQTT y UDP, fue necesario utilizar la librería "wifi.h". En el caso de MQTT también se utilizó la librería "PubSubClient.h", y en el caso de UDP la librería "WiFiUdp.h".

En los códigos 12.1 y 12.2 se puede observar los códigos necesarios para un microcontrolador ESP32 que tiene conectado un sensor y un actuador, respectivamente. Para la configuración del ESP como *access point* con el nombre necesario se inicia la comunicación serial por medio de Bluetooth. En el caso del sensor se inicia con el nombre "DispPot1"y del actuador se inicia con el nombre "DispEspActuador". Se colocan estos nombres puesto que, como se mencionó previamente, el *gateway* se conecta a los dispositivos Bluetooth que tengan un nombre que inicie con "Disp". En el caso del sensor en el *loop* del programa inicia leyendo el pin analógico, se guarda en una variable llamada **message** el texto a mandar por serial. El formato de dicho mensaje es "sensor/**nombre del sensor**/**valor del sensor**. Y finalmente, el mensaje es enviado al *gateway* y se tiene un *delay* de 1 segundo. Por otro lado, en el caso del actuador, se envía un mensaje serial por medio de Bluetooth al *gateway*, dicho mensaje tiene el formato "actuador/**nombre del actuador**". Luego, se espera a recibir una respuesta del *gateway*, al recibirla se convierte el mensaje de dicha respuesta a formato

JSON. Al tener el JSON se busca el valor enviado en el atributo "valor". En caso de este valor sea "True", se coloca el pin de salida en **HIGH** y en el caso contrario se coloca en **LOW**.

```
#include "BluetoothSerial.h"

#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth no habilitado
#endif

const int analogPin = 34;
String sensorName = "sensor1";

BluetoothSerial SerialBT;


void onBTConnect(esp_spp_cb_event_t event, esp_spp_cb_param_t* param) {
  if (event == ESP_SPP_SRV_OPEN_EVT) {
    // Enviar string cuando se conecte el dispositivo
    SerialBT.println("Conectado"); // Enviar nueva línea
  }
}

void setup() {
  Serial.begin(115200);
  pinMode(analogPin, INPUT);
  SerialBT.begin("DispPot1"); // Nombre del dispositivo Bluetooth
  Serial.println("Dispositivo inicializado, se puede conectar por bluetooth!");

  SerialBT.register_callback(onBTConnect);
}

void loop() {
  int potValue = analogRead(analogPin);

  String message  = "sensor/" + sensorName + "/" + String(potValue) +"\n";
  // Forward data from Serial to SerialBT
  SerialBT.println(message);

  delay(1000);
}
```

Código 12.1: Sensor por medio de Bluetooth

```
#include "BluetoothSerial.h"
#include <ArduinoJson.h>

#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth no está habilitado!
#endif

const int ledPin = 23;

BluetoothSerial SerialBT;
String receivedString = "";
StaticJsonDocument<200> jsonDoc;
String msg;


void onBTConnect(esp_spp_cb_event_t event, esp_spp_cb_param_t* param) {
  if (event == ESP_SPP_SRV_OPEN_EVT) {
```

```
    // Enviar string cuando se conecte el dispositivo
    SerialBT.println("Conectado"); // Nueva línea para separar mensajes
  }
}

void setup() {
  Serial.begin(115200);
  pinMode(ledPin, OUTPUT);
  SerialBT.begin("DispEspActuador"); // Nombre del dispositivo Bluetooth
  Serial.println("Dispositivo inicializado, se puede conectar por bluetooth!");

  SerialBT.register_callback(onBTConnect);
}

void loop() {
  digitalWrite(ledPin, HIGH);

  SerialBT.println("actuador/led amarillo\n"); // Enviar nueva línea

  // Enviar data de SerialBT a Serial
  while (SerialBT.available()) {
    char data = SerialBT.read();
    receivedString += data;

  }
  if (!receivedString.isEmpty()) {
    Serial.println("Recibido: " + receivedString);
     DeserializationError error = deserializeJson(jsonDoc, receivedString);
      if (error) {
        Serial.print("deserializeJson() falló: ");
        Serial.println(error.c_str());
        return;
      }
      // Extract the value from the JSON payload
      if (jsonDoc.containsKey("valor")) {
        msg = jsonDoc["valor"].as<String>();
        Serial.print("Valor recibido: ");
        Serial.println(msg);

        if (msg == "True") {
          digitalWrite(ledPin, HIGH);
          Serial.println("LED encendido");
        } else {
          digitalWrite(ledPin, LOW);
          Serial.println("LED apagado");
        }
      }

    receivedString = ""; // Borrar la variable donde se almacena el string recibido
  }

  delay(3000);
```

Código 12.2: Actuador por medio de Bluetooth

En los códigos 12.3 y 12.4 se puede observar el código necesario para comunicar por medio de MQTT un ESP32 con un sensor y con un actuador, respectivamente. En ambos códigos primero se debe de conectar a WiFi y luego conectarse al *broker* MQTT, el cual se encuentra en el *gateway*. En el caso que el microcontrolador tiene un sensor conectado se debe de leer el valor del sensor, enviar un JSON con el atributo **valor**, el cual contiene el valor del sensor, al *topic* "sensor/**nombre del sensor**"y luego esperar cierto tiempo antes de volver a leer el sensor y enviar el mensaje al *topic*. Por otro lado, en el caso que el microcontrolador tenga un actuador, se debe de suscribir al *topic* "actuador/**nombre del actuador**". Por lo tanto,

cada que le llega al microcontrolador un mensaje a este *topic* se ejecuta la función **callback.** En esta función se transforma a JSON el mensaje recibido, del cual se obtiene el atributo "valor". En caso el valor de este atributo sea "True"se coloca el actuador en "HIGH", y en caso contrario, se coloca el actuador en "LOW". En el *loop* del código se envía cada cierto tiempo un mensaje con el nombre del actuador al *topic* "actualizar/", este mensaje se envía para indicarle al *gateway* que el microcontrolador desea obtener una actualización de cual debe de ser el estado de el actuador. Cabe mencionar que cuando un microcontrolador envía un mensaje al *topic* "actualizar/**nombre del actuador**", el *gateway* responde con un mensaje que incluye el valor que debe de tener el actuador y este mensaje es recibido por todos los dispositivos suscritos al *topic* "actuador/**nombre del actuador**".

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <ArduinoJson.h>

const char* ssid = "Pellecer_Inadria";
const char* password = "fans1234";
const char* mqtt_server = "192.168.0.122"; // IP del gw
const int analogPin = 34;
int potValue = 0;

WiFiClient espClient;
PubSubClient client(espClient);
StaticJsonDocument<200> jsonDoc;
String jsonString;
int value = 0;

void setup() {
  Serial.begin(115200);
  pinMode(analogPin, INPUT);

  // Conectarse a la red Wi-Fi
  WiFi.begin(ssid, password);
  Serial.print("Conectandose a Wi-Fi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.print(".");
  }
  Serial.println();
  Serial.print("Conectado a Wi-Fi, IP: ");
  Serial.println(WiFi.localIP());

  // Conectarse a broker MQTT
  client.setServer(mqtt_server, 1883);
  while (!client.connected()) {
    Serial.print("Conectarse a broker MQTT...");
    if (client.connect("ESP32Client")) {
      Serial.println("conectado!");
    } else {
      Serial.print("falló, rc=");
      Serial.print(client.state());
      Serial.println(" volver a intentar en 5 segundos");
      delay(5000);
    }
  }

  jsonString = "";
}

void loop() {
  if (!client.connected()) {
    // Reconectarse al broker MQTT si se perdió la conexión
    if (client.connect("ESP32Client")) {
```

```
      Serial.println("Reconectado al broker MQTT ");
    }
  }

  potValue = analogRead(analogPin);
  Serial.println(potValue);

  jsonDoc["valor"] = String(potValue);
  serializeJson(jsonDoc, jsonString);
  client.publish("sensor/sensor3", jsonString.c_str());
  Serial.println(jsonString);

  jsonString = "";
  delay(2000);
}
```

Código 12.3: Sensor por medio de MQTT

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <ArduinoJson.h>

const char* ssid = "Pellecer_Inadria";
const char* password = "fans1234";
const char* mqtt_server = "192.168.0.122"; // IP del gw
const int ledPin = 23;
unsigned long previousMillis = 0;
const unsigned long interval = 800;

WiFiClient espClient;
PubSubClient client(espClient);
StaticJsonDocument<200> jsonDoc;
String jsonString;
bool value;
String msg;

void callback(char* topic, byte* payload, unsigned int length) {
  payload[length] = '\0'; // Terminar payload con null
  // Formatear el Json recibido en el payload
  DeserializationError error = deserializeJson(jsonDoc, payload);

  // Verificar errores
  if (error) {
    Serial.print("deserializeJson() falló: ");
    Serial.println(error.c_str());
    return;
  }

  // Extraer el valor del payload
  if (jsonDoc.containsKey("valor")) {
    msg = jsonDoc["valor"].as<String>();
    Serial.print("Valor recibido: ");
    Serial.println(msg);


    if (msg == "True") {
      // Encender led
      digitalWrite(ledPin, HIGH);
      Serial.println("LED encendido");
    } else {
      digitalWrite(ledPin, LOW);
      Serial.println("LED apagado");
    }
```

```
  }
}

void setup() {
  Serial.begin(115200);
  pinMode(ledPin, OUTPUT);
  // Conectarse a la red Wi-Fi
  WiFi.begin(ssid, password);
  Serial.print("Conectandose a Wi-Fi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.print(".");
  }
  Serial.println();
  Serial.print("Conectado a Wi-Fi, dirección IP: ");
  Serial.println(WiFi.localIP());

  // Connect to MQTT broker
  client.setServer(mqtt_server, 1883);
  client.setCallback(callback); // Funcion callback
  while (!client.connected()) {
    Serial.print("Conectandose al broker MQTT...");
    if (client.connect("ESP32Client")) {
      Serial.println("conectado!");
      client.subscribe("actuador/led verde", 0);
    } else {
      Serial.print("Falló, rc=");
      Serial.print(client.state());
      Serial.println(" volver a intentar en 5 segundos");
      delay(5000);
    }
  }

  jsonString = "";
}

void loop() {
  if (!client.connected()) {
    // reconectar al broker MQTT si se desonectó
    if (client.connect("ESP32Client")) {
      Serial.println("Reconectado al broker MQTT");
      client.subscribe("actuador/led", 0); // Resuscribirse al topic
    }
  }

  // Mantener la conexión MQTT y manejar mensajes entrantes
  client.loop();

  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval) {
    previousMillis = currentMillis;

    // Enviar el mensaje al topic "actualizar/"
    String message = "led";
    client.publish("actualizar/", message.c_str());
    Serial.println("Mensaje enviado al topic 'actualizar/'");
  }
}
```

Código 12.4: Actuador por medio de MQTT

Finalmente, en el caso de los microcontroladores ESP que se comunican con el *gateway* por medio de UDP, se observa el código cuando se tiene un sensor y un actuador conectado en el Código 12.5 y 12.6 respectivamente. En ambos códigos se observa que primero se debe de conectar el microcontrolador a internet e iniciar la comunicación por medio de UDP a la

IP del *gateway* en el puerto 5005. En el caso de tener un sensor conectado al microcontrolador en el *loop* únicamente se debe de enviar un datagrama con el formato "sensor/**nombre del sensor**/**valor del sensor** al *gateway* y luego esperar cierto tiempo antes de enviar nuevamente el valor del sensor. Y en el caso de tener un actuador conectado, lo que se debe de realizar en el *loop* es revisar si se ha recibido algún datagrama UDP, en caso si se ha recibido se convierte el datagrama a formato JSON y se obtiene el atributo **valor**. En caso el valor de este atributo sea "True"se coloca el estado del actuador en "HIGH", en caso contrario, se coloca en "LOW". Luego de esto, se revisa si ha transcurrido más de cierta cantidad de segundos desde la ultima vez que se envió un datagrama. En caso ya haya transcurrido el tiempo indicado, se envía al *gateway* un datagrama un mensaje con el formato "actuador/**nombre del actuador**/". Cabe mencionar que la cantidad de segundos a esperar es definida por el usuario en el *gateway*.

```
#include <WiFi.h>
#include <WiFiUdp.h>

const char* ssid = "Pellecer_Inadria";
const char* password = "fans1234";
const char* udpServerIP = "192.168.0.122"; // Dirección IP del servidor UDP
const int udpServerPort = 5005; // Puerto del servidor UDP

const int analogPin = 34;
int potValue;
String sensorName = "sensor2";

WiFiUDP udp;

void setup() {
  Serial.begin(115200);
  pinMode(analogPin, INPUT);

  // Conectar a Wi-Fi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Conectandose a WiFi...");
  }
  Serial.println("Conectado a WiFi");

  // Iniciar comunicación UDP
  udp.begin(udpServerPort);
  Serial.print("UDP iniciado en el puerto local");

}

void loop() {
  int potValue = analogRead(analogPin);
  String message  = "sensor/" + sensorName + "/" + String(potValue);
  Serial.println(message);
  // Send UDP datagram
  udp.beginPacket(udpServerIP, udpServerPort);
  udp.print(message);
  udp.endPacket();


  delay(4000);
}
```

Código 12.5: Sensor por medio de UDP

```
#include <WiFi.h>
#include <WiFiUdp.h>
#include <ArduinoJson.h>

const char* ssid = "Pellecer_Inadria";
const char* password = "fans1234";
const char* udpServerIP = "192.168.0.122"; // Dirección IP del servidor UDP
const int udpServerPort = 5005; // Puerto del gw
const int ledPin = 23;
unsigned long previousMillis = 0;
const unsigned long interval = 4000;

WiFiUDP udp;
StaticJsonDocument<200> jsonDoc;
String jsonString;
bool value;
String msg;
String actuadorName = "led azul";

void setup() {
  Serial.begin(115200);
  pinMode(ledPin, OUTPUT);
  // Conectado a la red Wi-Fi
  WiFi.begin(ssid, password);
  Serial.print("Conectando a Wi-Fi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.print(".");
  }
  Serial.println();
  Serial.print("Conectado a Wi-Fi, IP: ");
  Serial.println(WiFi.localIP());

  // Inicar comunicación UDP
  udp.begin(udpServerPort);
  Serial.print("UDP iniciado en el puerto local");

  jsonString = "";
}

void loop() {

  int packetSize = udp.parsePacket();
  if (packetSize){
    char buffer[255];
    int len = udp.read(buffer, 255);
    if (len > 0){
      buffer[len] = 0;
      Serial.println(buffer);
      DeserializationError error = deserializeJson(jsonDoc, buffer);
      if (error) {
        Serial.print("deserializeJson() falló: ");
        Serial.println(error.c_str());
        return;
      }
      // Extract the value from the JSON payload
      if (jsonDoc.containsKey("valor")) {
        msg = jsonDoc["valor"].as<String>();
        Serial.print("Valor recibido: ");
        Serial.println(msg);

        if (msg == "True") {
          // Encender el led
          digitalWrite(ledPin, HIGH);
          Serial.println("LED encendido");
        } else {
          digitalWrite(ledPin, LOW);
```

```
            Serial.println("LED apagado");
          }
        }
71    }
  }

  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval) {
76  previousMillis = currentMillis;
    String message  = "actuador/" + actuadorName + "/";
    Serial.println(message);
    udp.beginPacket(udpServerIP, udpServerPort);
    udp.print(message);
81  udp.endPacket();
  }
}
```

Código 12.6: Actuador por medio de UDP

12.1. Rendimiento del sistema físico

El sistema físico de prototipo se puede observar en la Figura 37. Como se mencionó previamente, se tienen 6 microcontroladores, 3 de los cuales tienen conectados un sensor y 3 un actuador. Como sensor, se utilizó un potenciómetro y como actuador se utilizaron leds.

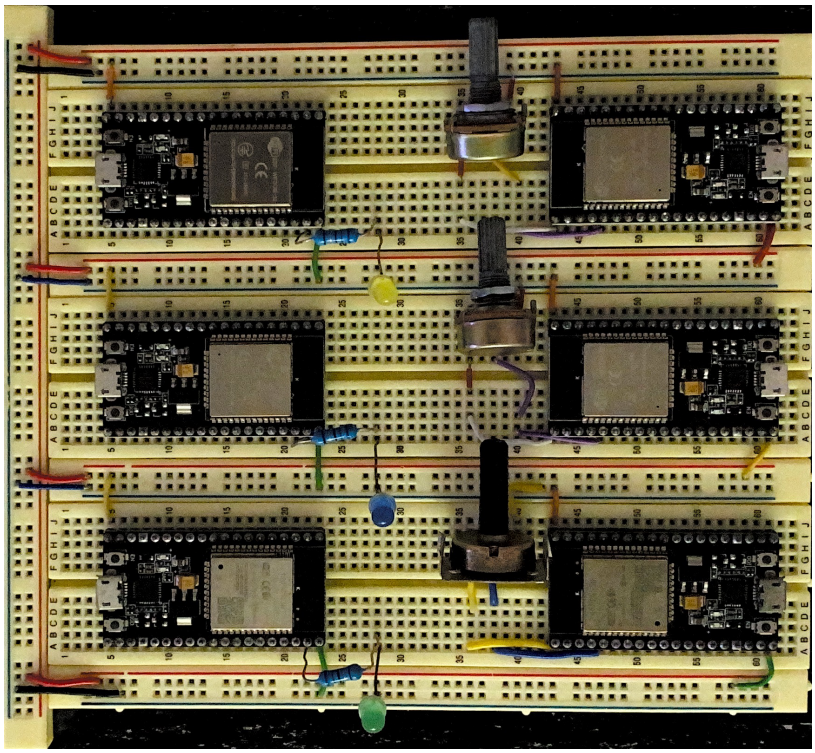


Figura 37: Sistema físico

Protocolo	**Ventajas**	**Desventajas**
Bluetooth	▪ Desde cada microcontrolador se puede mantener un canal de comunicación dedicado	▪ Alcance limitado ▪ Su uso no es escalable a sistemas con una mayor cantidad de dispositivos
MQTT	▪ Protocolo de mensajería ligero ▪ Es escalable a sistemas con una mayor cantidad de dispositivos	▪ Requiere un servidor MQTT en el *gateway* ▪ Latencia variable ▪ Al tener varios actuadores en el sistema físico que tomen el valor del mismo actuador en la plataforma, solamente un dispositivo debe de enviar mensajes al *topic* de actualización.
UDP	▪ Comunicación de baja latencia ▪ No se establece una conexión por lo que se puede dejar de transmitir y no se requiere de programación defensiva para evitar algún error en el programa del *gateway*	▪ Sin garantía de entrega ▪ Sin control de flujo

Cuadro 2: Comparación de protocolos implementados en el sistema físico

CAPÍTULO 13

Conclusiones

- Fue posible, por medio de la plataforma desarrollada, automatizar y simplificar los procesos de agregar dispositivos IoT a una plataforma de orquestación, solicitar el valor de sensores y cambiar el estado de actuadores.
- Se desarrolló exitosamente en Django y contenerizó en Docker el microservicio *backend*, el cual debe de mantener el registro y control de los dispositivos.
- Se implementó exitosamente la base de datos de grafos en la cual, el *backend* almacena la información de los usuarios, sus dispositivos y el comportamiento de los actuadores.
- Se desarrolló exitosamente una interfaz gráfica por medio de la cual el usuario puede describir el sistema de dispositivos IoT.
- El *Backend* y *Frontend* de la plataforma se desplegaron exitosamnete en contenedores Docker orquestados en Kubernetes por medio de Okteto.
- Se desarrolló exitosamente en Python un microservicio encargado de actuar como un *gateway* hacia la plataforma para los sensores y actuadores.
- Se implementó exitosamente la comunicación de la plataforma con los dispositivos IoT por medio los protocolos UDP, MQTT y Bluetooth.
- Se desarrolló exitosamente una API en FastAPI que permite almacenar y manejar la data histórica de los sensores.
- Se implementó exitosamente el modelo de *machine learning isolation forest*, por medio del cual se puede clasificar el comportamiento de los sensores, para identificar anomalías.
- Se desarrollaron dos *cron jobs* que permiten ejecutar tareas de forma automática en la plataforma.

CAPÍTULO 14

Recomendaciones

A partir de la plataforma desarrollada cabe mencionar las siguientes recomendaciones.

- Agregar dispositivos IoT industriales a la red de dispositivos, no solamente ESP32, y desarrollar el código necesario para que el *gateway* pueda soportar dichos dispositivos.
- Desarrollar librerías para que el microcontrolador ESP32 pueda soportar más protocolos IoT y los programas correspondientes en el *gateway*
- Programar librerías de Python para poder integrar proyectos de Django con distintos tipos de bases de datos, no solamente relacionales y de grafos.
- Optimizar el rendimiento del *gateway* desarrollando en C los programas que se comunican con los microcontroladores ESP32 y envian peticiones HTTP.
- Programar un *endpoint* en el historiador por medio del cual se pueda predecir el valor de un sensor.
- Agregar un tablero embebido en el *frontend* por medio del cual el usuario pueda visualizar gráficamente los datos históricos de sus dispositivos.
- Expandir el funcionamiento del historiador para que pueda almacenar datos de actuadores.
- Implementar en el *backend* una tercera forma de definir el estado de un actuador mediante traversals de grafos.
- Ampliar el modelo de la base de datos de grafos para incorporar actuadores con valores continuos
- Ampliar el funcionamiento del historiador para que pueda identificar cuando un dispositivo IoT se desconecta de la red y notificar al usuario.

CAPÍTULO 15

Bibliografía

[1] A. S. Muhammed y D. Ucuz, "Comparison of the IoT Platform Vendors, Microsoft Azure, Amazon Web Services, and Google Cloud, from Users' Perspectives," en *2020 8th International Symposium on Digital Forensics and Security (ISDFS)*, 2020, págs. 1-4. DOI: 10.1109/ISDFS49300.2020.9116254.

[2] *AWS IoT*, Accessed on April 23, 2023. dirección: https://aws.amazon.com/es/iot/.

[3] *Azure IoT Hub*, Accessed on April 23, 2023. dirección: https://azure.microsoft.com/es-mx/products/iot-hub/.

[4] *AWS IoT*, Accessed on April 23, 2023. dirección: https://cloud.google.com/solutions/iot/.

[5] R. K. Kodali y S. Soratkal, "MQTT based home automation system using ESP8266," en *2016 IEEE Region 10 Humanitarian Technology Conference (R10-HTC)*, 2016, págs. 1-5. DOI: 10.1109/R10-HTC.2016.7906845.

[6] L. Sun, Y. Li y R. A. Memon, "An open IoT framework based on microservices architecture," 2017. DOI: 10.1109/cc.2017.7868163.

[7] G. M. D'silva, S. Thakare y V. A. Bharadi, "Real-time processing of IoT events using a Software as a Service (SaaS) architecture with graph database," en *2016 International Conference on Computing Communication Control and automation (ICCUBEA)*, 2016, págs. 1-6. DOI: 10.1109/ICCUBEA.2016.7859984.

[8] Oracle, *What is the Internet of Things (IoT)?* 2023. dirección: https://www.oracle.com/internet-of-things/what-is-iot/.

[9] A. W. Services, *What is IoT?* Accessed: April 3, 2023, 2023. dirección: https://aws.amazon.com/what-is/iot/.

[10] Particle, *A 2022 Guide to IoT Protocols and Standards*, Accessed: April 6, 2023, n.d. dirección: https://www.particle.io/iot-guides-and-resources/iot-protocols-and-standards/.

[11] A. E. Rodriguez, L. M. Kristensen y A. Rutle, "On Modelling and Validation of the MQTT IoT Protocol for M2M Communication," en *PNSE@Petri Nets/ACSD*, 2018.

[12] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," en *2017 IEEE International Systems Engineering Symposium (ISSE)*, 2017, págs. 1-7. DOI: 10.1109/SysEng.2017.8088251.

[13] N. Nikolov, "Research of MQTT, CoAP, HTTP and XMPP IoT Communication protocols for Embedded Systems," en *2020 XXIX International Scientific Conference Electronics (ET)*, 2020, págs. 1-4. DOI: 10.1109/et50336.2020.9238208.

[14] A. Banks, E. Briggs, K. Borgendale y R. Gupta, *MQTT Version 5.0*, Accessed on May 27, 2023, 2019. dirección: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf.

[15] J. F. Kurose y K. W. Ross, *Computer Networking: A Top-Down Approach*, 7th. Pearson, 2016.

[16] Bluetooth, *Bluetooth Technology Overview*, https://www.bluetooth.com/learn-about-bluetooth/tech-overview/.

[17] Mozilla, *HTTP response status codes*, Accessed on May 28, 2023, 2023. dirección: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status.

[18] Espressif, *ESP32*, 2023. dirección: https://www.espressif.com/en/products/socs/esp32.

[19] S.-G. Marica, "Rest API architectural constraints," *WebScrapingAPI*, 2022. dirección: https://www.webscrapingapi.com/rest-api-architecture-constraints.

[20] D. S. Foundation, *Django Documentation*, Accessed: April 3, 2023, 2023. dirección: https://docs.djangoproject.com/en/4.2/.

[21] *FastAPI*, https://fastapi.tiangolo.com, Modern web framework for building APIs with Python 3.8+ based on standard Python type hints, 2023.

[22] Oracle, *What is a Database?* Accessed: May 22, 2023, 2023. dirección: https://www.oracle.com/database/what-is-database/.

[23] Neo4j, *Graph Database Concepts*, Accessed: May 22, 2023, 2023. dirección: https://neo4j.com/developer/graph-database/.

[24] Neomodel, *Neomodel Documentation*, Accessed: May 22, 2023, 2023. dirección: https://neomodel.readthedocs.io/en/latest/.

[25] P. D. Team, *PostgreSQL - Official Website*, Accessed on November 27, 2023, 2023. dirección: www.postgresql.org.

[26] *What is PostgreSQL? – Amazon Web Services*, Accessed: November 27, 2023. dirección: https://aws.amazon.com/rds/postgresql/what-is-postgresql/.

[27] *What is Azure Database for PostgreSQL?* Dirección: https://learn.microsoft.com/en-us/azure/postgresql/single-server/overview.

[28] S. Aggarwal, *Modern Web-Development using ReactJS*, 2023. dirección: http://ijrra.net/Vol5issue1/IJRRA-05-01-27.pdf.

[29] A. W. Services, *Microservices*, 2021. dirección: https://aws.amazon.com/es/microservices/.

[30] Microsoft, *.NET Microservices Architecture for Containerized .NET Applications*. Microsoft Corporation, 2022, Accessed: May 22, 2023. dirección: https://dotnet.microsoft.com/download/e-book/microservices-architecture/pdf.

[31] R. Hat, *What is Virtualization?* Accessed: May 22, 2023, 2023. dirección: https://www.redhat.com/en/topics/virtualization/what-is-virtualization.

[32] W. is a container? "Docker." (), dirección: https://www.docker.com/resources/what-container/ (visitado 10-04-2023).

[33] R. Hat, *Containers vs VMs*, Accessed: May 22, 2023, 2020. dirección: https://www.redhat.com/en/topics/containers/containers-vs-vms.

[34] A. W. Services. "What is Docker?" (2021), dirección: https://aws.amazon.com/es/docker/ (visitado 10-04-2023).

[35] *Kubernetes Documentation*, Accessed: May 22, 2023. dirección: https://kubernetes.io/.

[36] Okteto, *Okteto*, Accessed: May 22, 2023. dirección: https://www.okteto.com/docs/getting-started/.

[37] Okteto, *Okteto Compose Reference*, Accessed: May 22, 2023. dirección: https://www.okteto.com/docs/reference/compose/.

[38] G. Cloud, *What Is a Public Cloud?* https://cloud.google.com/learn/what-is-public-cloud, Accessed: November 30, 2023, 2023.

[39] Azure, *Azure Products*, https://azure.microsoft.com/en-us/products, Accessed: November 30, 2023, 2023.

[40] Azure, *Power BI*, https://www.microsoft.com/en-us/power-platform/products/power-bi/, Accessed: November 30, 2023, 2023.

[41] Azure, *Azure Communication Services*, https://azure.microsoft.com/en-us/products/communication-services, Accessed: November 30, 2023, 2023.

CAPÍTULO 16

Anexos

16.1. Respuestas de peticiones HTTP al backend

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "Usuario": "daniela morales",
    "Sensores": [
        {
            "nombre": "sensor2",
            "valor": 1526.0,
            "uid": null
        },
        {
            "nombre": "sensor3",
            "valor": 0.0,
            "uid": null
        },
        {
            "nombre": "sensor1",
            "valor": 2178.0,
            "uid": null
        }
    ],
    "Actuadores": [
        {
            "nombre": "led amarillo",
            "uid": null,
            "valor": true,
            "operacion": "or",
            "manual": true
        },
        {
            "nombre": "led",
            "uid": null,
            "valor": true,
            "operacion": "and",
            "manual": true
        },
        {
            "nombre": "led azul",
            "uid": null,
            "valor": true,
            "operacion": "and",
            "manual": true
        }
    ]
}
```

Código 16.1: GET /mor19213/dispositivos

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "nombre": "sensor2",
        "valor": 1526.0,
        "uid": null
    },
    {
        "nombre": "sensor3",
        "valor": 0.0,
        "uid": null
    },
    {
        "nombre": "sensor1",
        "valor": 2178.0,
        "uid": null
    }
]
```

Código 16.2: GET /mor19213/sensores

```
HTTP 405 Method Not Allowed
Allow: POST, PUT, DELETE, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

Código 16.3: GET /mor19213/sensores/sensor1

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "nombre": "led amarillo",
        "uid": null,
        "valor": true,
        "operacion": "or",
        "manual": true
    },
    {
        "nombre": "led",
        "uid": null,
        "valor": true,
        "operacion": "and",
        "manual": true
    },
    {
        "nombre": "led azul",
        "uid": null,
        "valor": true,
        "operacion": "and",
        "manual": true
    }
]
```

Código 16.4: GET /mor19213/actuadores

```
HTTP 200 OK
Allow: GET, POST, PUT, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "nombre": "led",
    "uid": null,
    "valor": true,
    "operacion": "and",
    "manual": true
}
```

Código 16.5: GET /mor19213/actuadores/led

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "datos": [
        {
            "actuador": {
                "nombre": "led amarillo",
                "uid": null,
                "valor": true,
                "operacion": "or",
                "manual": true
            },
            "condiciones": [
                {
                    "dispositivo": "sensor1",
                    "condicion": "mayor",
                    "valor": 2000.0,
                    "actual": true,
                    "real": 2178.0,
                    "uid": "20230821214551909483"
                }
            ]
        },
        {
            "actuador": {
                "nombre": "led",
                "uid": null,
                "valor": true,
                "operacion": "and",
                "manual": true
            },
            "condiciones": [
                {
                    "dispositivo": "sensor2",
                    "condicion": "mayor",
                    "valor": 500.0,
                    "actual": true,
                    "real": 1526.0,
                    "uid": "20230823012554604853"
                },
                {
                    "dispositivo": "sensor2",
                    "condicion": "mayor o igual",
                    "valor": 2000.0,
                    "actual": false,
                    "real": 1526.0,
                    "uid": "20230821214616033794"
                }
            ]
        },
        {
            "actuador": {
                "nombre": "led azul",
                "uid": null,
                "valor": true,
                "operacion": "and",
                "manual": true
            },
            "condiciones": [
                {
                    "dispositivo": "sensor3",
                    "condicion": "mayor",
                    "valor": 2000.0,
                    "actual": false,
                    "real": 0.0,
                    "uid": "20230821233523183344"
                }
            ]
        }
    ]
}
```

Código 16.6: GET /mor19213/condiciones

```
HTTP 200 OK
Allow: GET, POST, PUT, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "dispositivo": "sensor2",
        "condicion": "mayor",
        "valor": 500.0,
        "actual": true,
```

```
        "real": 1526.0,
        "uid": "20230823012554604853"
    },
    {
        "dispositivo": "sensor2",
        "condicion": "mayor o igual",
        "valor": 2000.0,
        "actual": false,
        "real": 1526.0,
        "uid": "20230821214616033794"
    }
]
```

Código 16.7: GET /mor19213/condiciones/led

```
HTTP 200 OK
Allow: GET, PUT, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "dispositivo": "sensor2",
    "condicion": "mayor",
    "valor": 500.0,
    "actual": true,
    "real": 1526.0,
    "uid": "20230823012554604853"
}
```

Código 16.8: GET /mor19213/condiciones/led/20230823012554604853

16.2. Cron Job de correos

Figura 38: Correo enviado por el cron job al usuario mor19213