Trabajo Fin de Grado

Grado en Ingeniería de las Tecnologías de Telecomunicación

Sistema domótico basado en NodeRED y servicio REST para el control y monitorización del consumo eléctrico

Autor: Sergio García López

Tutora: María Teresa Ariza Gómez

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2024

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Sistema domótico basado en NodeRED y servicio REST para el control y monitorización del consumo eléctrico

Autor:
Sergio García López

Tutora:
María Teresa Ariza Gómez

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2024

Trabajo Fin de Grado: Sistema domótico basado en NodeRED y servicio REST para el control y monitorización del consumo eléctrico

Autor: Sergio García López

Tutora: María Teresa Ariza Gómez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2024

El Secretario del Tribunal

A mi familia y amigos

Agradecimientos

En primer lugar, quiero expresar mi profundo agradecimiento por el apoyo incondicional de mi familia durante todos estos años de carrera, y por darme consejos para seguir adelante a pesar de las dificultades. A mi madre, por su profundo amor, y por apoyarme y animarme durante este curso tan duro. A mi padre, también por su afecto y apoyo, y por haberme dado indicaciones y orientación durante el desarrollo del proyecto. A mi hermano Nacho, cuya curiosidad e interés por mi trabajo ha supuesto una motivación más que me ha permitido llevarlo a cabo. A mis abuelos, tíos y primos, por sus mensajes de apoyo y cariño desde Jaén. Mi familia merece un lugar especial en esta dedicatoria, pues son el pilar fundamental de mi vida.

Gracias a mis amigos de toda la vida: Carmen, Marina, Inés, Irene y Alejandro. A pesar de los distintos caminos que hemos tomado, quiero agradecer que sigamos manteniendo nuestra profunda amistad. Quiero agradecer los años que hemos pasado juntos y los que están por venir, pues junto a mi familia, sois quienes habéis definido mi persona y mi ser.

También quiero hacer una mención especial a mi amigo Álvaro, quien pronto será ingeniero al igual que yo, pero en su caso, ingeniero aeroespacial. Nuestro paso por la ETSI ha sido prácticamente paralelo, y al final, él ha sido uno de los pocos que ha llegado hasta aquí junto a mí, desde que entramos en la escuela allá por el 2019. Quiero agradecer su amistad, y especialmente los buenos momentos que hemos pasado hablando sobre la carrera, tecnología e incluso filosofía.

De la carrera no solo salgo con conocimientos y experiencias, también con buenos amigos. Quiero agradecer a Chema, Kiko, Pedro y Felipe por haberme acompañado durante este camino. También a quién ya era mi amigo del instituto, Sergio Montañés, a quién le deseo lo mejor en sus últimos años de carrera. Además, quiero hacer una mención especial a mis compañeros del Máster en Ingeniería de Telecomunicación, en especial a Juan Luis, Juanjo y Aitor, por los buenos momentos pasados en clase.

Por último, quiero mostrar mi sincero agradecimiento a mi tutora, María Teresa Ariza Gómez. Le estoy profundamente agradecido por haber aceptado mi idea inicial, por sus conocimientos, consejos y orientaciones, y por haberme ayudado a encaminar el proyecto.

Sergio García López

Sevilla, 2024

Resumen

El constante avance en el desarrollo de las TIC está provocando una profunda transformación en la sociedad del siglo XXI. Esta evolución se refleja en conceptos como el hogar digital, ciudades inteligentes, optimización de la movilidad y transporte, agricultura de precisión, gestión inteligente de la energía, o la Industria 4.0. Muchos de estos conceptos se basan en la masiva recolección de datos mediante dispositivos IoT, seguida de su procesamiento y análisis para la toma de decisiones y gestión de sistemas en tiempo real.

Este Trabajo Fin de Grado se desarrolla en el ámbito del hogar digital y la domótica, centrándose en la implementación de un sistema de control y monitorización del consumo eléctrico. Para ello, se ha hecho uso de NodeRED para implementar la lógica de control y monitorización de un enchufe inteligente Shelly Plug S, empleando para ello el protocolo de comunicaciones MQTT. Paralelamente, se ha desarrollado un servicio REST para generar y almacenar datos de consumo y coste energético utilizando la información de potencia instantánea proveniente del enchufe inteligente, junto con los datos de la tarifa PVPC proporcionados por la API REST de Red Eléctrica de España.

Además, se ha hecho uso de Docker y Docker-Compose para la encapsulación y gestión de los componentes software del sistema mediante contenedores, realizándose el despliegue del sistema final en una máquina virtual en Microsoft Azure.

Abstract

The constant advance in the development of ICT is causing a deep transformation in the society of the 21st century. This evolution is reflected in concepts such as the digital home, smart cities, optimization of mobility and transport, precision agriculture, intelligent energy management, or Industry 4.0. Many of these concepts are based on the massive collection of data through IoT devices, followed by their processing and analysis for decision making and management of systems in real time.

This Final Degree Project is developed in the field of digital home and domotics, focusing on the implementation of a system for control and monitoring of electricity consumption. For this purpose, NodeRED has been used to implement the control and monitoring logic of a Shelly Plug S smart plug, using the MQTT communications protocol. At the same time, a REST service has been developed to generate and store energy consumption and cost data using the instantaneous power information from the smart plug, along with the PVPC prices data provided by the REST API of Red Eléctrica de España.

In addition, Docker and Docker-Compose have been used for the encapsulation and management of the software components of the system through containers, deploying the final system in a virtual machine in Microsoft Azure.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Glosario

TIC	Tecnologías de la Información y Comunicación
API	Application Programming Interface
MQTT	Message Queuing Telemetry Transport
REST	Representational State Transfer
IoT	Internet of Things
IP	Internet Protocol
PVPC	Precio Voluntario para el Pequeño Consumidor
SQL	Structured Query Language
JSON	JavaScript Object Notation
YAML	Yet Another Markup Language
GNU	GNU is Not Unix
TCP	Transmission Control Protocol
IT	Information Technology
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
URL	Uniform Resource Locator
ORM	Object Relational Mapping
CA	Certificate Authority
TLS	Transport Layer Security
DNS	Domain Name System
SSH	Secure Shell

1 INTRODUCCIÓN

Caminante, no hay camino,
se hace camino al andar.
- Antonio Machado -

1.1 Motivación

En los últimos años, el término "hogar inteligente" ha dejado de ser una proyección de futuro y está empezando a convertirse en una realidad para nuestra sociedad actual. La domótica, o automatización del hogar, ha emergido como una tecnología revolucionaria que transforma la forma en que interactuamos con nuestro entorno doméstico. Desde el encendido automático de las luces, puertas automáticas, neveras inteligentes, cámaras de vigilancia IP e incluso la preparación del café por la mañana. En la figura 1-1 se muestran algunas estimaciones del número de dispositivos domésticos inteligentes en el mundo. En el año 2022, aproximadamente 131 millones de hogares disponían de altavoces inteligentes, siendo los más populares Amazon Echo con Alexa como asistente virtual y Google Home. No obstante, se prevé que para el año 2026 esta cifra aumente hasta los 300 millones de hogares. Además, se espera que en los próximos años aumente la integración de otras clases de dispositivos en los hogares, tales como neveras y lavadoras inteligentes, robots aspiradores, detectores de humo, cámaras de seguridad, etc [1].

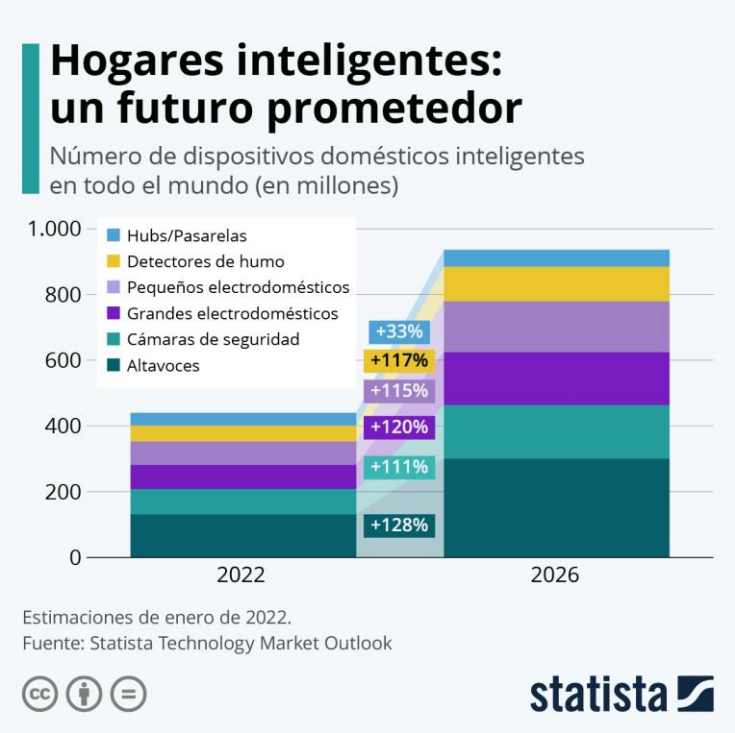


Figura 1-1. Previsiones en el número de dispositivos inteligentes en hogares para el año 2026

Uno de los factores fundamentales que contribuye a los avances en la domótica es el desarrollo del *Internet of Things*. Para el año 2024 se estima que el tamaño de mercado de dispositivos IoT sea de unos 98 mil millones de dólares, con previsiones de crecimiento de hasta unos 336 mil millones de dólares en los próximos 5 años. El desarrollo de nuevas tecnologías de conectividad como el 5G está favoreciendo la adopción de estos dispositivos en múltiples sectores, tales como hogares inteligentes, vehículos conectados, IoT industrial o aplicaciones médicas, teniendo un importante impacto en los países del sudeste asiático. Como resultado, se espera que el número de dispositivos IoT conectados continúe creciendo a un ritmo elevado, con previsiones de más de 27 mil millones de dispositivos para el año 2026, siendo la mayoría de ellos dispositivos IoT de corto alcance [2]. En la figura 1-2 se representa la evolución del número de dispositivos IoT en el mundo en los próximos años:

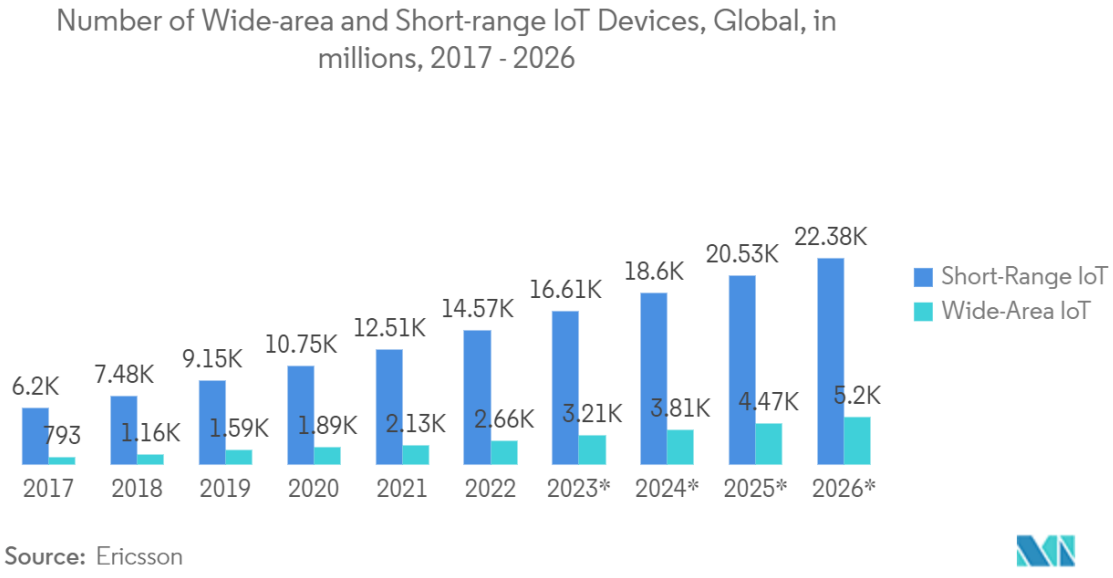


Figura 1-2. Número de dispositivos IoT en el mundo

La domótica ofrece una serie de ventajas para sus usuarios, entre las que podríamos destacar la comodidad que supone automatizar tareas rutinarias, el control remoto y monitorización de dispositivos desde aplicaciones móviles o plataformas web, y el ahorro energético. Este Trabajo Fin de Grado hará un especial énfasis en esta última ventaja, la eficiencia energética. La integración de dispositivos inteligentes en el hogar contribuye significativamente a reducir el consumo de energía, cuestión fundamental en la actualidad debido a las consecuencias del cambio climático. Además, existe una clara motivación económica debido a la reducción de costes que supone el ahorro energético.

Dentro de este contexto es donde surge la motivación de investigar y usar tecnologías *open source* para desarrollar un sistema de control y monitorización del consumo eléctrico a partir de datos proporcionados por dispositivos IoT. En particular, para este proyecto se ha hecho uso de un enchufe inteligente modelo Shelly Plug S [3], al cual se le puede configurar la dirección IP de un broker MQTT para enviarle comandos de control y obtener la potencia instantánea consumida por los dispositivos conectados. Para implementar este sistema es necesario disponer de un nodo centralizado que pueda comunicarse con el enchufe a través del broker MQTT, y que a su vez pueda utilizar los servicios proporcionados por otros sistemas más complejos, que en el caso de este proyecto serán APIs REST. Dicho nodo debe ser capaz de implementar mecanismos para controlar y monitorizar al enchufe inteligente, así como integrar la información proporcionada por el enchufe con los servicios REST.

La herramienta software escogida tanto para este propósito como para la implementación de una interfaz de usuario ha sido NodeRED [4]. Como se explicará más adelante en la definición de la arquitectura del sistema, se ha tomado la decisión de desarrollar un servicio REST en el que se implementarán las funcionalidades de generación y almacenamiento de datos de consumo y de coste energético. Para obtener la información de precios de la electricidad se hará uso de la API REST de Red Eléctrica de España [5], que publica cada 24 horas los precios de la tarifa regulada PVPC [6]. Por otro lado, se utilizarán tecnologías de contenedores para los componentes software del sistema, tales como Docker [7] y Docker-Compose [8]. Finalmente, el despliegue final de dichos componentes se realizará en una máquina virtual en la nube, en concreto, en Microsoft Azure [9].

1.2 Antecedentes

Al analizar proyectos previos, se encuentra relevante el trabajo realizado por Belén María Lozano Jurado en su Trabajo de Fin de Grado titulado "Integración de OpenHAB y Home Connect en una aplicación Android para el control domótico de los electrodomésticos y la estimación de su consumo energético" [10], bajo la dirección de María Teresa Ariza Gómez. El enfoque principal del proyecto consistió en la creación de una aplicación Android para interactuar con un sistema de automatización del hogar mediante el uso de una API REST proporcionada por OpenHAB. Dicha API dirigía las solicitudes hacia Home Connect, una plataforma que posibilita la comunicación con electrodomésticos compatibles con esta tecnología. Además, también se desarrolló un servicio REST para llevar el seguimiento de consumo de los electrodomésticos.

Las diferencias fundamentales de este proyecto con el de Belén María Lozano son las siguientes:

- Uso de NodeRED para la implementación de la lógica de control y monitorización del consumo eléctrico. NodeRED tendrá el rol de nodo centralizado para interconectar al enchufe inteligente con los servicios proporcionados por la API REST de generación y almacenamiento de datos de consumo y costes, así como la información de precios proporcionada por la API REST de Red Eléctrica.
- Interfaz web de usuario implementada mediante NodeRED.
- Comunicación con el enchufe inteligente mediante el protocolo MQTT.
- Cálculo de costes de consumo eléctrico a partir de la información de precios de la tarifa PVPC proporcionada por la API REST de Red Eléctrica.
- Uso de tecnologías de contenedores como Docker y Docker-Compose para los componentes software del sistema.
- Despliegue del sistema en una máquina virtual en la nube de Microsoft Azure, permitiendo el acceso remoto a la interfaz de usuario y la instalación del enchufe inteligente en cualquier lugar, siempre que

se disponga de conexión a Internet.

1.3 Objetivos

El objetivo principal del proyecto consiste en crear una solución domótica que permita a los usuarios controlar y monitorear el consumo eléctrico y los costes asociados de sus electrodomésticos o cualquier otro dispositivo que requiera estar conectado a una toma de corriente para ser alimentado. Para ello, en esta sección definimos una serie de objetivos específicos que deben alcanzarse para lograr este propósito con éxito:

- **Despliegue y configuración de un broker MQTT:** Para enviar comandos de encendido y apagado, así como para obtener datos de potencia instantánea del dispositivo conectado al enchufe, se hará uso del protocolo MQTT, siendo necesario para ello el despliegue y configuración de un broker.
- **Implementación de la lógica de control y monitorización en NodeRED**: NodeRED se empleará para la implementación y automatización de los siguientes procesos:
  - Recopilación de los datos de potencia proporcionados por el enchufe inteligente.
  - Proporcionar mecanismos de encendido y apagado del enchufe, así como el seguimiento de su estado.
  - Actualización en tiempo real de los datos de consumo y costes mediante el envío de peticiones HTTP a un servicio REST encargado de almacenar estas estadísticas.
  - Consulta de los precios de la electricidad mediante peticiones HTTP a la API REST de Red Eléctrica de España.
  - Cálculo de costes a partir de datos de consumo y de precios.
  - Consulta y actualización continua de los datos de potencia, consumo y costes mostrados en la interfaz de usuario.
- **Creación de una interfaz de usuario mediante NodeRED**: Se desarrollará una interfaz web en el propio NodeRED gracias a la facilidad que ofrece en la creación de interfaces gráficas, especialmente para elementos relacionados con la visualización de datos, tales como series temporales, medidores, gráficas de barras, etc.
- **Implementación de un servicio REST para la generación y almacenamiento de datos de consumo y costes**: Se implementará un servicio REST propio para generar datos de consumo a partir de los datos de potencia del enchufe. Para ello, el servicio REST facilitará el almacenamiento de dichos datos de potencia, así como el almacenamiento de datos de consumo y costes para la creación de históricos.
- **Pruebas automáticas del servicio REST**: Se desarrollarán tests automatizados mediante un script para realizar pruebas en las rutas expuestas por el servicio REST.
- **Almacenamiento en una base de datos**: Para el almacenamiento y gestión de los datos de potencia, consumo y costes se hará uso de una base de datos SQL.
- **Uso de contenedores para los componentes del sistema**: El sistema seguirá una arquitectura basada en microservicios, encapsulándose cada uno de ellos en contenedores Docker para garantizar su portabilidad y facilitar su gestión.
- **Despliegue del sistema final en la nube**: Una vez que se hayan desarrollado todos los componentes del sistema, estos se desplegarán en una máquina virtual en la nube de Microsoft Azure. De esta forma, el sistema será accesible de manera remota, lo que permitirá a los usuarios controlar y monitorear al enchufe desde cualquier ubicación con conexión a Internet.

1.4 Funcionalidades

Una vez que se hayan cumplido los objetivos específicos del proyecto, se espera que el sistema final ofrezca

las siguientes funcionalidades a sus usuarios:

- **Control remoto de dispositivos:** La interfaz web permitirá a los usuarios encender y apagar los dispositivos conectados al enchufe inteligente sin necesidad de que los usuarios estén conectados a la misma red local que el enchufe.
- **Monitorización en tiempo real de la potencia instantánea:** Los usuarios podrán visualizar la potencia consumida a lo largo del tiempo por los dispositivos conectados al enchufe.
- **Consumo energético en diferentes franjas horarias:** Los usuarios podrán consultar cuál ha sido su consumo energético en el día en las siguientes franjas horarias: en la madrugada, por la mañana, por la tarde y por la noche.
- **Consumo energético en cada hora del día:** De forma similar, los usuarios tendrán información de consumo para cada una de las 24 horas del día.
- **Histórico de consumo:** Los usuarios podrán ver un histórico de consumo en el que se representarán los consumos totales de días anteriores.
- **Consulta de precios de la electricidad:** Los usuarios tendrán disponible la información de precios de la tarifa regulada PVPC proporcionada por Red Eléctrica de España para cada hora del día.
- **Coste energético en cada hora del día:** Se mostrarán los costes del consumo eléctrico para cada hora del día.
- **Histórico de costes:** Los usuarios podrán consultar un histórico con los costes totales de días anteriores.
- **Limpieza de registros en la base de datos:** En caso de que lo deseen, los usuarios podrán limpiar los registros de potencia, consumo y costes mencionados con anterioridad.

1.5 Arquitectura del sistema

En la figura 1-3 se representan los diferentes elementos que componen el sistema domótico en su totalidad, los cuales describiremos en esta sección:

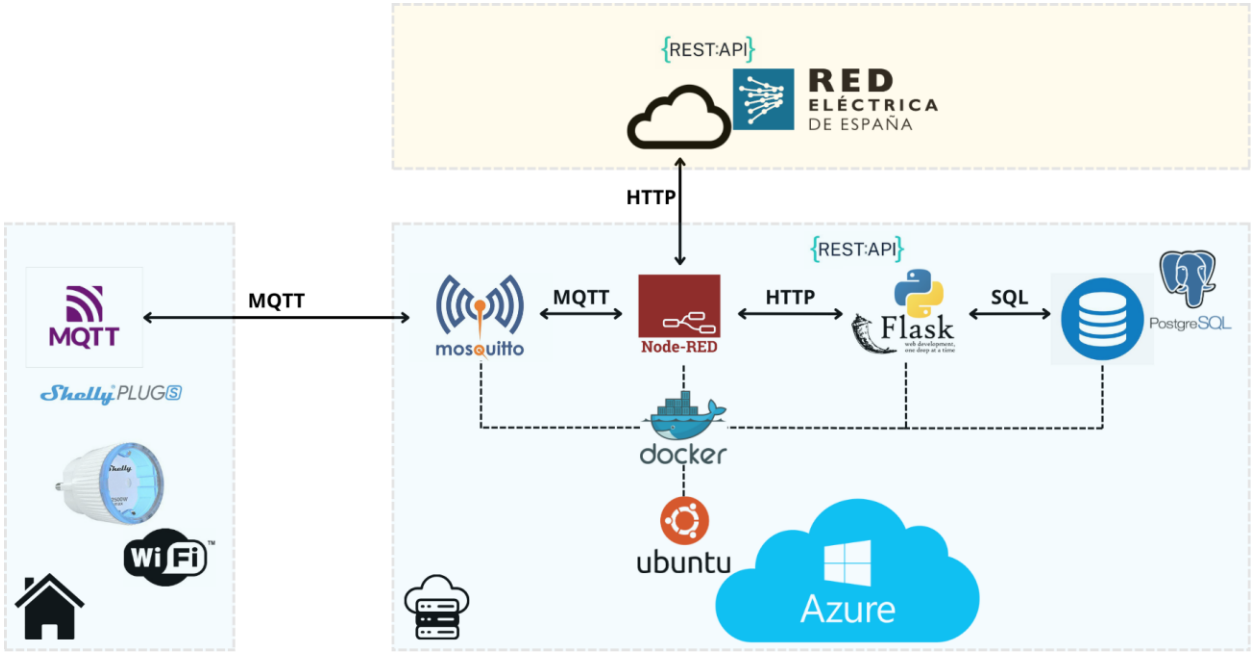


Figura 1-3. Arquitectura del sistema domótico

- **Enchufe inteligente:** El enchufe modelo Shelly Plug S utilizado en el sistema dispone de un cliente MQTT configurable. Entre las opciones de configuración destacan la dirección IP del broker MQTT y los tópicos en los que el enchufe publicará los datos de potencia y se suscribirá para recibir comandos de control. El enchufe se instalará en el domicilio del usuario final y sólo requerirá la configuración de conexión a una red Wi-Fi con acceso a Internet.
- **Máquina virtual alojada en Microsoft Azure:** Para poder utilizar el sistema domótico de forma remota, se han utilizado los servicios de la nube de Microsoft Azure para crear una máquina virtual Ubuntu bajo la dirección IP 20.250.161.128. En dicha máquina se han desplegado los siguientes servicios en contenedores Docker siguiendo una arquitectura basada en microservicios:
    - **Broker Mosquitto:** Para el intercambio de mensajes entre el cliente MQTT del enchufe y NodeRED, tales como datos de potencia instantánea o comandos de control, se hace uso de un broker Mosquitto. El puerto TCP utilizado por este servicio es el 1883.
    - **NodeRED:** En este sistema, NodeRED se utiliza como un nodo centralizado que interactúa de forma directa o indirecta con todos los elementos del sistema. En él se han implementado los aspectos de control y monitorización de consumo mencionados en la Sección 1.3 (Objetivos). Para ello, NodeRED se comunica con el enchufe inteligente a través del broker MQTT, con el servicio REST de consumo y costes, y con el servicio REST de Red Eléctrica de España. Además, también se ha hecho uso de NodeRED para la implementación de la interfaz de usuario. Como se explicará más adelante, NodeRED funciona como un servicio web, y para garantizar la seguridad en las comunicaciones se ha configurado el uso del protocolo HTTPS. Por lo tanto, el puerto TCP utilizado por este servicio es el 443.
    - **Servicio REST de consumos y costes:** Este servicio REST desarrollado en Python ofrece una API para la generación y almacenamiento de datos relacionados con el consumo eléctrico y los costes asociados. Este servicio utiliza el puerto TCP 10000, sin embargo, sólo se utiliza de forma interna en la máquina virtual y no es accesible desde Internet.
    - **Base de datos PostgreSQL:** Para almacenar los datos de potencia generados por el enchufe, y los datos de consumo y costes, se hace uso de una base de datos PostgreSQL. Este servicio utiliza el puerto TCP 5432, y al igual que el servicio REST mencionado anteriormente, no es accesible desde Internet.
- **Servicio REST de Red Eléctrica de España:** La empresa Red Eléctrica de España, que gestiona la red de transporte del sistema eléctrico español, ofrece una API REST llamada REData API. Esta API REST permite la consulta de una amplia gama de datos relacionados con la red eléctrica española [5]. En nuestro sistema será de interés conocer los precios de la tarifa regulada PVPC, cuyos precios son publicados para cada hora del día y son actualizados al finalizar el día [6].

En la figura 1-3, aquellos elementos pertenecientes a rectángulos celestes quedan dentro del dominio de implementación y configuración de este proyecto. En el caso del servicio REST de Red Eléctrica, que se encuentra dentro de un rectángulo amarillo, se trata de un servicio externo utilizado en el sistema.

1.6 Estructura de la memoria

En esta sección se proporciona un resumen de los contenidos fundamentales abordados en la memoria, ofreciendo una visión general de su contenido:

1. **Introducción:** En este capítulo se expone la motivación para realizar este proyecto, destacando la creciente relevancia de la domótica, el IoT y la eficiencia energética en los próximos años. Además, se describen los antecedentes de proyectos similares, los objetivos y funcionalidades que se quieren alcanzar con este proyecto y una descripción de la arquitectura del sistema desarrollado.
2. **Tecnologías utilizadas:** Se realiza una enumeración y descripción de las tecnologías utilizadas para la realización del proyecto, tanto software como hardware.
3. **Control y monitorización mediante Node-RED:** Se describe de forma detallada el desarrollo

del sistema de control y monitorización del consumo eléctrico mediante Node-RED, haciendo un especial énfasis en la comunicación con el enchufe inteligente utilizando el protocolo MQTT y las interacciones con los servicios REST usados en el sistema.

4. **Servicio REST de consumo y costes:** En este capítulo se detalla la implementación y funcionalidades del servicio REST empleado para la generación y almacenamiento de datos de consumo energético y costes. Además, se aborda la interacción con la base de datos, el modelo de datos, la generación de la documentación, y los tests automáticos desarrollados para probar el servicio.
5. **Despliegue del sistema en Microsoft Azure:** Se explican los pasos realizados para el despliegue del sistema en una máquina virtual en la nube de Microsoft Azure, así como aspectos de su operación mediante el uso de Docker-Compose.
6. **Conclusiones y líneas futuras:** Se exponen las conclusiones derivadas del desarrollo del proyecto, destacando los objetivos alcanzados, las lecciones aprendidas y las posibles líneas futuras para la mejora y ampliación del sistema domótico.

2 TECNOLOGÍAS UTILIZADAS

Vivimos en una sociedad extremadamente dependiente de la ciencia y la tecnología, en la que casi nadie tiene un mínimo conocimiento en ciencia y tecnología.

- Carl Sagan -

2.1 Tecnologías hardware

2.1.1 Enchufe inteligente Shelly Plug S

El Shelly Plug S [3] es un enchufe inteligente fabricado por Shelly Group, una empresa especializada en la fabricación de dispositivos domóticos, entre los que destacan enchufes, bombillas, sensores de humedad y temperatura, interruptores, detectores de ventanas y puertas, etc. Este enchufe está diseñado para permitir la conexión y desconexión a la red eléctrica de los dispositivos conectados a él. Se puede configurar y controlar a través de una red Wi-Fi mediante su servidor web local o mediante una aplicación móvil. Puede soportar una potencia máxima de 2500 W. Ofrece integración con servicios *cloud* como Amazon Alexa o Google Assistant. Además, el Shelly Plug S es compatible con el protocolo MQTT, lo que facilita su integración en sistemas domóticos más complejos. Se trata del elemento fundamental que nos permite hacer el control y monitorización del consumo eléctrico en este sistema. En la figura 2-1 puede verse una imagen del enchufe:

Figura 2-1. Enchufe Shelly Plug S

2.1.2 Portátil OMEN 16

El equipo de trabajo empleado durante el desarrollo del proyecto, incluyendo la implementación y configuración de los componentes software del sistema, ha sido un portátil OMEN 16 [11] de la marca HP, mostrado en la figura 2-2:

Figura 2-2. Portátil OMEN 16

Entre sus especificaciones técnicas podemos destacar:

- Procesador Intel Core i7-13700HX
- Tarjeta gráfica NVIDIA GeForce RTX 4080 (GDDR6 de 12 GB dedicada)
- Memoria RAM DDR5-5600 MHz 32 GB (2 x 16 GB)
- Almacenamiento SSD 2 TB PCIe Gen4 NVMe TLC M.2
- Sistema operativo Windows 11 Home
- Pantalla QHD de 16 pulgadas (hasta 2560 x 1440 píxeles), 240 Hz de frecuencia de refresco.

2.2 Tecnologías software

2.2.1 MQTT

MQTT [12] es un protocolo de intercambio de mensajes diseñado específicamente para dispositivos IoT. Se trata de un protocolo ligero que sigue el modelo de comunicación publicador-suscriptor, lo cual permite la publicación y suscripción a tópicos específicos, creando para ello colas de mensajes. Está diseñado para dispositivos con bajos recursos y ancho de banda limitado, lo cual lo hace ideal para dispositivos IoT. Como protocolo de transporte utiliza TCP y ofrece varios niveles de calidad de servicio. En este proyecto, es el protocolo de comunicaciones que nos permite interactuar con el enchufe inteligente.

Figura 2-3. MQTT

2.2.2 Eclipse Mosquitto

Eclipse Mosquitto [13] es un broker MQTT de código abierto que implementa las versiones 5.0, 3.1.1 y 3.1 del protocolo. Se trata de un broker ligero y adecuado para cualquier tipo de dispositivo, desde sistemas embebidos de baja potencia hasta servidores de alta capacidad. Este proyecto forma parte del Eclipse IoT Working Group y está gestionado por la Fundación Eclipse. Es el broker MQTT que se ha elegido para poder enviar comandos de encendido y apagado al enchufe, así como para obtener los datos de potencia instantánea medidos por este.

Figura 2-4. Eclipse Mosquitto

2.2.3 NodeRED

Node-RED [4] es una herramienta de programación basada en flujos para aplicaciones orientadas a eventos. Permite interconectar dispositivos hardware, APIs y servicios web, entre otros. Ofrece un editor gráfico accesible desde el navegador que facilita la creación de flujos mediante una amplia gama de nodos y una programación sencilla. Está basado en Node.js y puede ejecutarse en multitud de plataformas, tanto en hardware de bajo coste como en la nube. Además, tiene una biblioteca integrada que permite guardar componentes para su reutilización, así como compartir flujos fácilmente en formato JSON e instalar módulos creados por la comunidad de Node-RED. Se ha utilizado en este proyecto para implementar los aspectos de control y monitorización del sistema, la comunicación con el enchufe inteligente a través del broker MQTT, la interacción con los servicios REST del sistema, y la implementación de la interfaz de usuario.

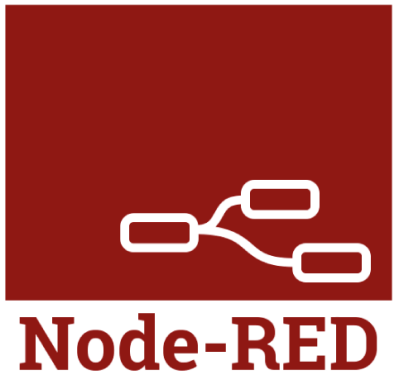


Figura 2-5. NodeRED

2.2.4 Flask

Flask [14] es un *framework* ligero de Python usado para crear aplicaciones web de forma rápida y sencilla. Se trata de un *framework* minimalista y flexible que permite desarrollar desde APIs REST sencillas hasta complejos sistemas web. Además, ofrece una amplia gama de extensiones para agregar funcionalidades según las necesidades de la aplicación. Se ha empleado para desarrollar el servicio REST de consumo y costes del sistema domótico.

Figura 2-6. Flask

2.2.5 Flassger

Flassger [15] es una extensión de Flask que permite generar una documentación interactiva siguiendo las especificaciones OpenAPI [16]. Facilita la documentación automática de una API REST a partir de comentarios YAML en el código de la aplicación Flask. Además, tiene integrada una interfaz Swagger que permite la interacción y validación de las rutas del servicio REST. Ha sido de utilidad para generar de forma automática la documentación de la API del servicio REST de consumo y costes.

Figura 2-7. Flassger

2.2.6 SQLAlchemy

SQLAlchemy [17] es una librería de Python que proporciona una forma flexible de interactuar con bases de datos relacionales. Permite a los desarrolladores trabajar con bases de datos SQL mediante programación orientada a objetos, lo cual facilita la creación, manipulación y consulta de datos sin tener que escribir directamente sentencias SQL. En este proyecto se ha empleado para simplificar la interacción con la base de datos desde el servicio REST de consumo y costes, al basarse en un enfoque ORM.

Figura 2-8. SQLAlchemy

2.2.7 Pytest

Pytest [18] es un *framework* de pruebas que permite el desarrollo y ejecución de pruebas unitarias, de integración y funcionales para aplicaciones desarrolladas en Python. Ofrece una sintaxis simple y sencilla para escribir pruebas, una amplia gama de *asserts* y la generación de informes detallados sobre los resultados de las pruebas. En este proyecto se ha elegido este *framework* para realizar las pruebas de las rutas expuestas por el servicio REST de consumo y costes.

Figura 2-9. Pytest

2.2.8 PostgreSQL

PostgreSQL [19] es un sistema de gestión de bases de datos relacionales de código abierto conocido por su fiabilidad, robustez y rendimiento. Debido a estas características, es ampliamente usado en aplicaciones web, empresariales y científicas. Se ha usado para almacenar los datos de potencia generados por el enchufe, los datos de consumo y los de costes.

Figura 2-10. PostgreSQL

2.2.9 Docker

Docker [7] es un software de código abierto utilizado para desplegar aplicaciones en contenedores mediante tecnologías de virtualización del *kernel* de Linux, tales como los *cgroups* y *namespaces*. Permite encapsular aplicaciones y sus dependencias en un entorno aislado, lo que garantiza la portabilidad de la aplicación y la consistencia del entorno de ejecución, independientemente de la infraestructura que haya por debajo. Se ha utilizado un contenedor Docker para cada uno de los servicios que componen el sistema, facilitando así su gestión y portabilidad.

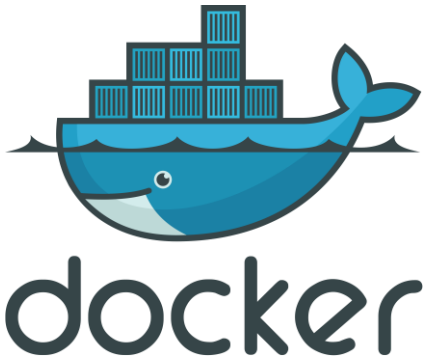


Figura 2-11. Docker

2.2.10 Docker-Compose

Docker-Compose [8] es una herramienta usada para definir y ejecutar aplicaciones multi-contenedor en Docker. Facilita la configuración y gestión de servicios, redes y volúmenes a partir de un fichero de configuración YAML. Además, proporciona comandos para crear, iniciar, detener y administrar contenedores de manera conjunta, simplificando así el proceso de desarrollo y despliegue de sistemas basados en múltiples servicios. Docker-Compose ha sido de gran utilidad para gestionar de forma sencilla la definición, configuración y operación de los contenedores del sistema domótico, sin tener que recurrir directamente a comandos Docker.

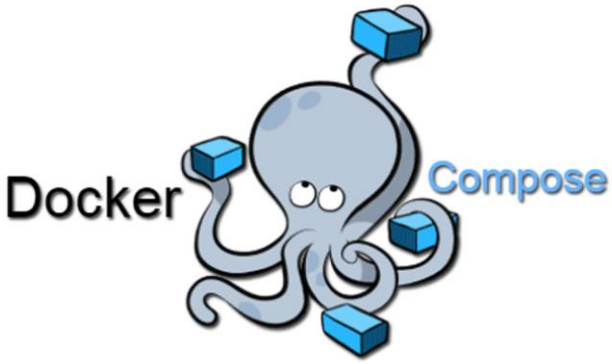


Figura 2-12. Docker-Compose

2.2.11 Ubuntu

Ubuntu [20] es una distribución GNU/Linux basada en Debian. Es ampliamente conocido por su facilidad de uso, estabilidad y seguridad, ofreciendo una amplia gama de software gratuito y de código abierto. Es el sistema operativo que se ha usado tanto para la implementación del sistema domótico como para su despliegue en una máquina virtual en la nube de Azure.

Figura 2-13. Ubuntu

2.2.12 Virtualbox

Virtualbox [21] es un software de virtualización de código abierto que permite ejecutar y crear máquinas virtuales. Permite tener sistemas operativos invitados (*guests*) dentro de un sistema operativo anfitrión (*host*). Es el hipervisor que se ha elegido para crear y ejecutar una máquina virtual Ubuntu en un sistema operativo anfitrión Windows 11, la cual se ha usado para implementar el sistema domótico.

Figura 2-14. Virtualbox

2.2.13 Microsoft Azure

Microsoft Azure [9] es una plataforma *cloud* que ofrece servicios de tipo "infraestructura como servicio" y "plataforma como servicio". Entre estos destacan servicios de almacenamiento, redes, creación de máquinas virtuales, bases de datos o *backend* para aplicaciones móviles. Es la plataforma que se ha elegido para crear una máquina virtual en la nube en la que se ha desplegado el sistema domótico.

Figura 2-15. Microsoft Azure

2.2.14 Wireshark

Wireshark [22] es una herramienta *open source* que permite capturar y analizar tráfico de red en tiempo real. Es utilizado por ingenieros de redes, analistas de ciberseguridad y desarrolladores de aplicaciones para diagnosticar problemas de red, realizar pruebas de seguridad, depurar aplicaciones, y analizar y probar protocolos en una red. Se ha utilizado para solucionar los problemas de comunicación con el enchufe inteligente y depurar el servicio REST de consumo y costes.

Figura 2-16. Wireshark

2.2.15 Sublime Text

Sublime Text [23] un editor de código caracterizado por su interfaz de usuario limpia y minimalista. Es compatible con múltiples lenguajes de programación y ofrece características avanzadas como resaltado de sintaxis, autocompletado inteligente, búsqueda y reemplazo avanzados, y una gran cantidad de extensiones para ampliar su funcionalidad. Es el editor que se ha elegido para trabajar en la implementación del sistema domótico.

Figura 2-17. Sublime Text

3 CONTROL Y MONITORIZACIÓN MEDIANTE NODE-RED

Si no puedes medirlo no puedes mejorarlo.

- Peter Drucker -

En este capítulo se detalla la implementación del sistema de control y monitorización del consumo eléctrico mediante Node-RED, incluyendo los mecanismos de comunicación con el enchufe inteligente utilizando el protocolo MQTT. También se abordan las interacciones con el servicio REST de consumo y costes, y con el servicio REST de Red Eléctrica, así como la creación de la interfaz de usuario.

3.1 Despliegue y configuración del broker MQTT

Uno de los aspectos fundamentales del sistema es la comunicación con el enchufe inteligente a través del protocolo MQTT. Para ello, es necesario disponer de un broker MQTT, que puede ser uno público, como es el caso del broker de HiveMQ [24], o uno privado. Para este proyecto se ha elegido esta última opción, utilizar un broker MQTT propio usando un contenedor de Eclipse Mosquitto.

3.1.1 Despliegue del contenedor Mosquitto

Para cada uno de los contenedores del sistema, se hará uso de imágenes Docker que pueden ser obtenidas desde Docker Hub [25], el repositorio público de imágenes Docker más conocido y utilizado en la industria IT. Para desplegar un contenedor Mosquitto, podemos seguir los pasos indicados en la página de la imagen oficial de Eclipse Mosquitto [26] , mostrada en la figura 3-1:

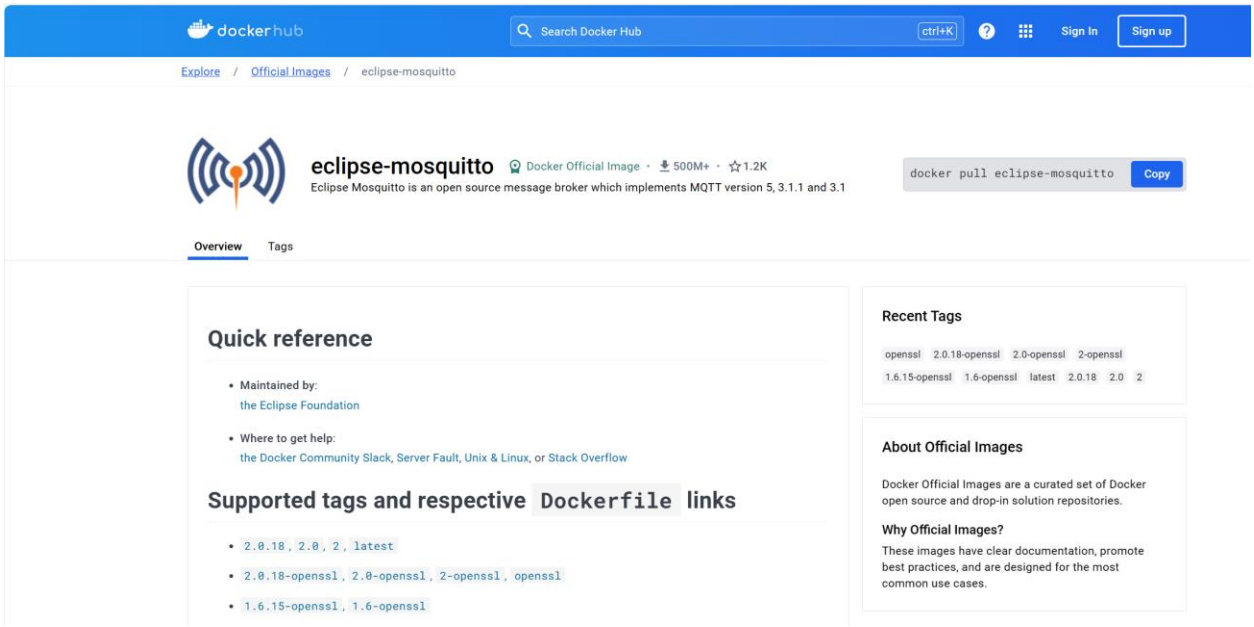


Figura 3-1. Página de la imagen oficial de Eclipse Mosquitto

En las indicaciones mostradas en la página se recomienda utilizar tres directorios para la persistencia de la información generada o utilizada por el contenedor Mosquitto:

- **/mosquitto/config:** Es el directorio donde Mosquitto almacena sus ficheros de configuración, siendo de especial relevancia el fichero **mosquitto.conf.**
- **/mosquitto/data:** En este directorio se persiste información sobre conexiones, mensajes en cola, mensajes publicados, etc.
- **/mosquitto/log:** Es el directorio donde Mosquitto almacena los archivos de registro generados durante su ejecución.

No obstante, en este proyecto solo ha sido necesario mantener la persistencia del directorio /mosquitto/config para poder utilizar un fichero de configuración de Mosquitto personalizado. Para ello, hay que compartir un directorio del *host* mediante un volumen para que pueda ser accesible por el contenedor Mosquitto, lo que en la terminología de Docker se le conoce como un *bind mount*. En la figura 3-2 se muestra un extracto del fichero **docker-compose.yaml** utilizado en este proyecto, mostrando la definición completa del contenedor Mosquitto:

```
20  broker_mqtt:
21
22    container_name: mosquitto
23
24    image: eclipse-mosquitto:latest
25
26    ports:
27      - "1883:1883"
28      - "9001:9001"
29
30    restart: always
31
32    volumes:
33      - ./mosquitto/config:/mosquitto/config
34
35    environment:
36      - TZ=Europe/Madrid
```

Figura 3-2. Definición del contenedor Mosquitto en Docker-Compose

En dicha figura 3-2 se le indica a Docker-Compose que haga lo siguiente:

- Crear un servicio llamado **broker_mqtt**.
- Crear un contenedor llamado **mosquitto**, asociado al servicio **broker_mqtt**.
- Crear dicho contenedor a partir de la última versión de la imagen de **eclipse-mosquitto**, indicado por la etiqueta **latest**.
- Abrir los puertos 1883 y 9001 de la máquina anfitriona y redirigirlos a los puertos 1883 y 9001 del contenedor, respectivamente.
- Reiniciar automáticamente el contenedor en caso de que se detenga por cualquier motivo, a menos que sea detenido explícitamente usando el comando "**docker compose stop**". De esta manera, se garantiza que el servicio se esté ejecutando en todo momento, incluso si se reinicia la máquina anfitriona.
- Compartir el directorio **./mosquitto/config** del *host* (partiendo del directorio de trabajo donde está el fichero docker-compose.yaml) con el directorio **/mosquitto/config** del contenedor. De esta forma, podremos editar de forma sencilla la configuración de Mosquitto modificando el fichero **mosquitto.conf** desde la propia máquina anfitriona.
- Establecer la variable de entorno **TZ** (*Time Zone*) a la zona horaria de Madrid.

3.1.2 Configuración de Mosquitto

En la configuración por defecto de Mosquitto solo se aceptan conexiones que provengan desde la propia máquina donde se está ejecutando (en nuestro caso, sería desde el *localhost* del contenedor). Por lo tanto, es necesario editar el fichero **mosquitto.conf** con el contenido que se muestra en la figura 3-3:

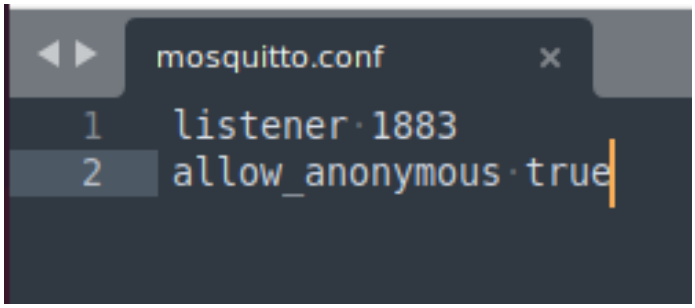


Figura 3-3. Contenido del fichero mosquitto.conf

En dicho fichero se configura un *listener* en el puerto 1883 (puerto TCP estándar de un broker MQTT) para aceptar conexiones por dicho puerto, y se permite que los clientes MQTT puedan conectarse al broker sin credenciales de autenticación. Mosquitto ofrece una amplia gama de opciones de configuración, tales como autenticación mediante usuario y contraseña, encriptación mediante TLS, uso de certificados, etc. Sin embargo, por limitaciones del cliente MQTT del enchufe Shelly Plug S, no es posible utilizar este tipo de medidas de seguridad ya que no tiene implementadas estas funcionalidades. No obstante, en un entorno de producción real sí deberían configurarse estas opciones para garantizar la confidencialidad e integridad de los datos de los usuarios.

3.2 Configuración del enchufe inteligente

El enchufe Shelly-Plug S dispone de un servidor web que permite su configuración desde un navegador accediendo a través de la dirección IP que se le ha asignado en la red local. La página web ofrecida por el enchufe muestra la potencia instantánea que está consumiendo el dispositivo conectado al enchufe, y además tiene integrado un botón que permite habilitar o deshabilitar la conexión a la red eléctrica. Por otra parte, la

interfaz web ofrece diversas opciones, como la configuración de temporizadores, envío de peticiones HTTP a una URL específica tras cambiar el estado del enchufe, o limitar la potencia máxima del enchufe, entre otras, como puede verse en la figura 3-4:

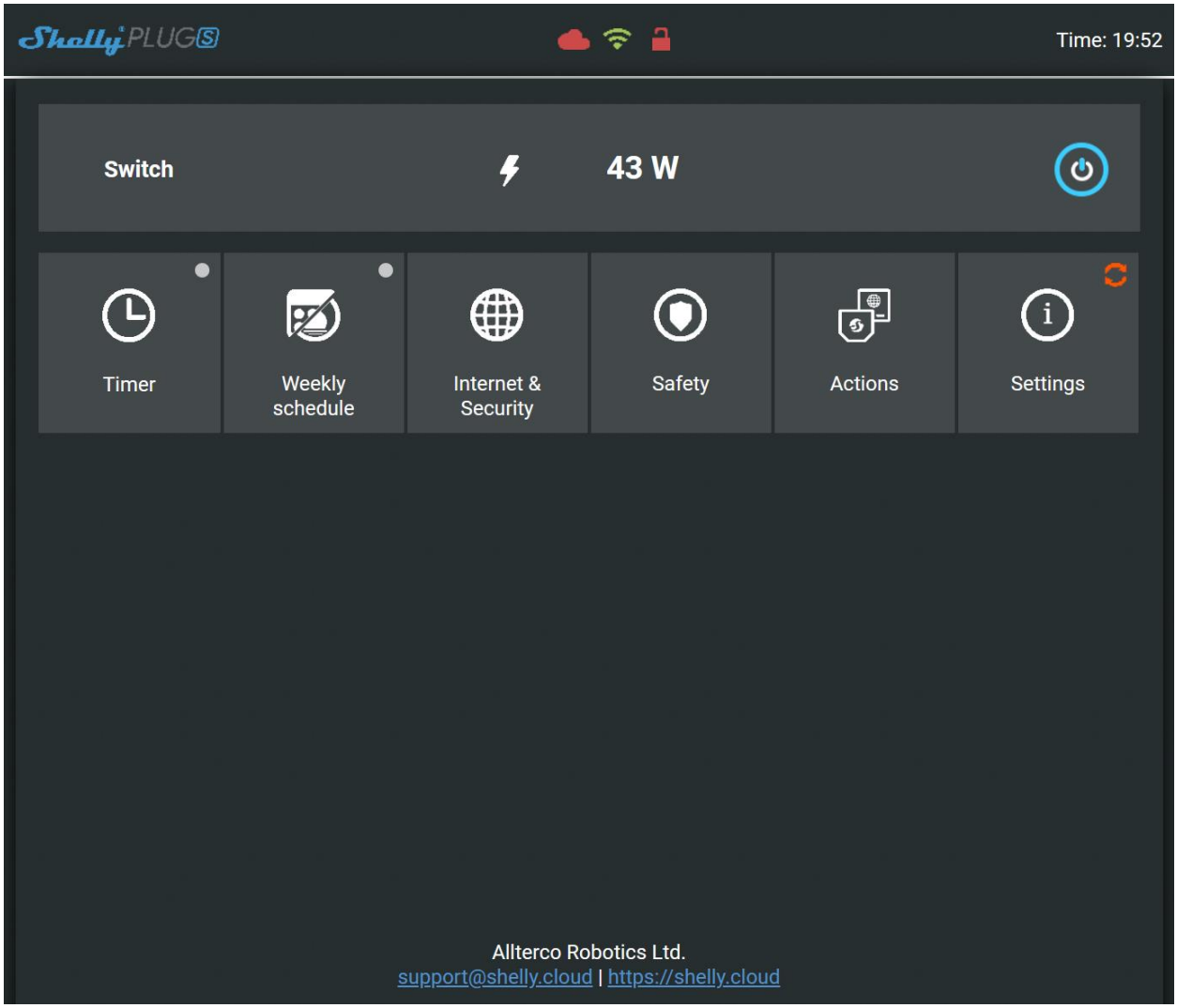


Figura 3-4. Interfaz web de configuración del Shelly Plug S

En el ámbito de nuestro proyecto ha sido de interés el apartado **"Internet & Security"**, ya que ofrece unas opciones avanzadas de desarrollador en el subapartado **"ADVANCED-DEVELOPER SETTINGS"** que nos permitirán el uso de MQTT en el enchufe. Dichas opciones se muestran en las figuras 3-5, 3-6 y 3-7:

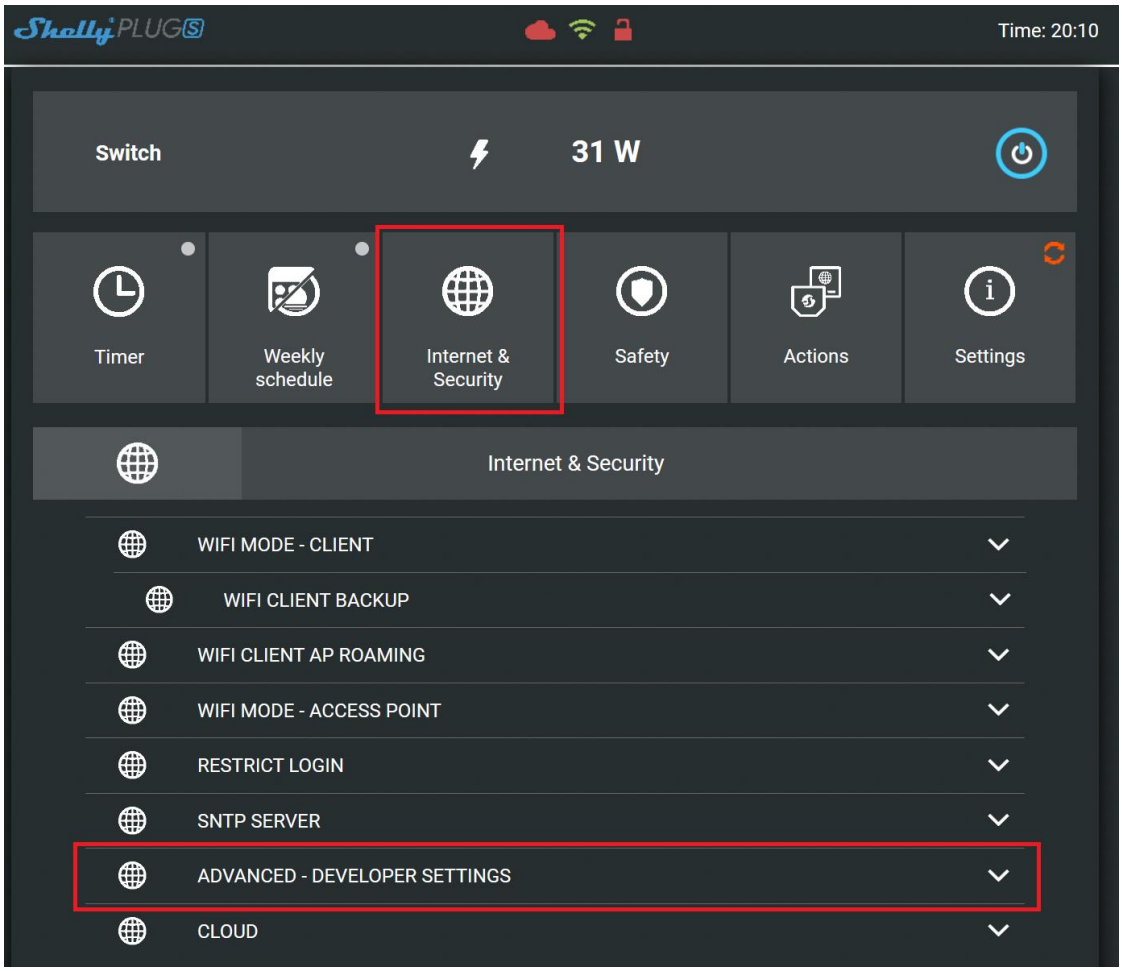


Figura 3-5. Opciones avanzadas de desarrollador del Shelly Plug S

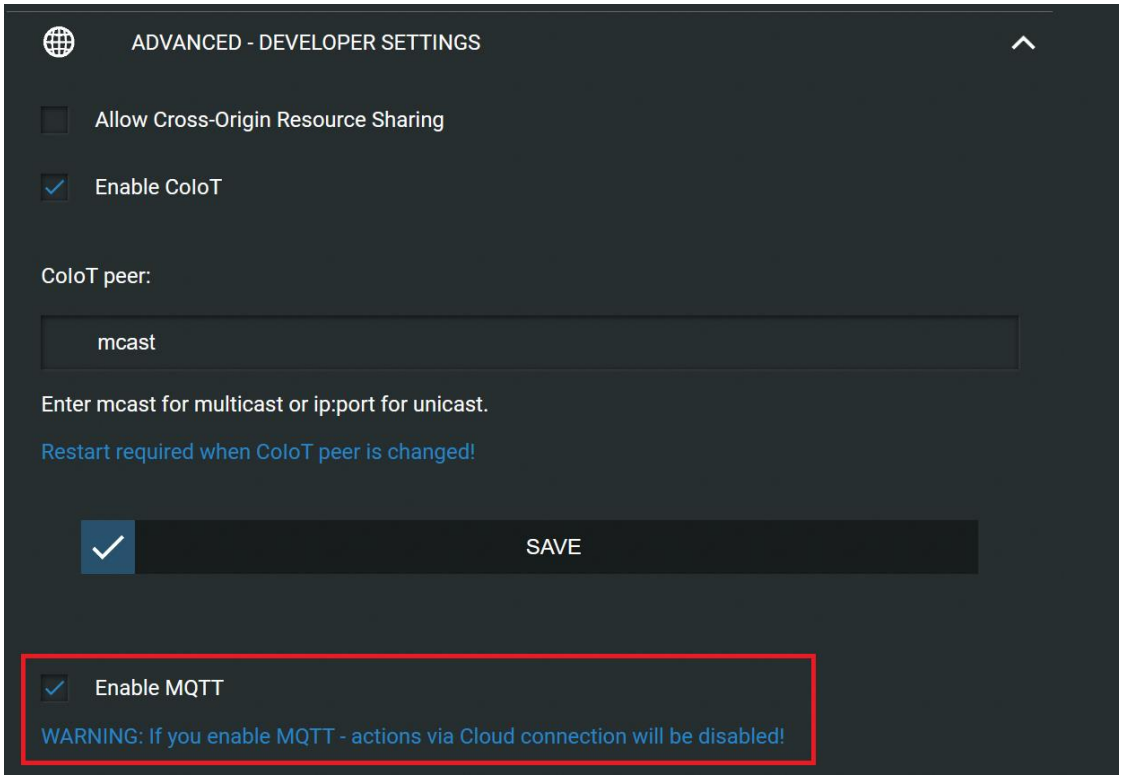


Figura 3-6. Activación del uso de MQTT en el Shelly Plug S

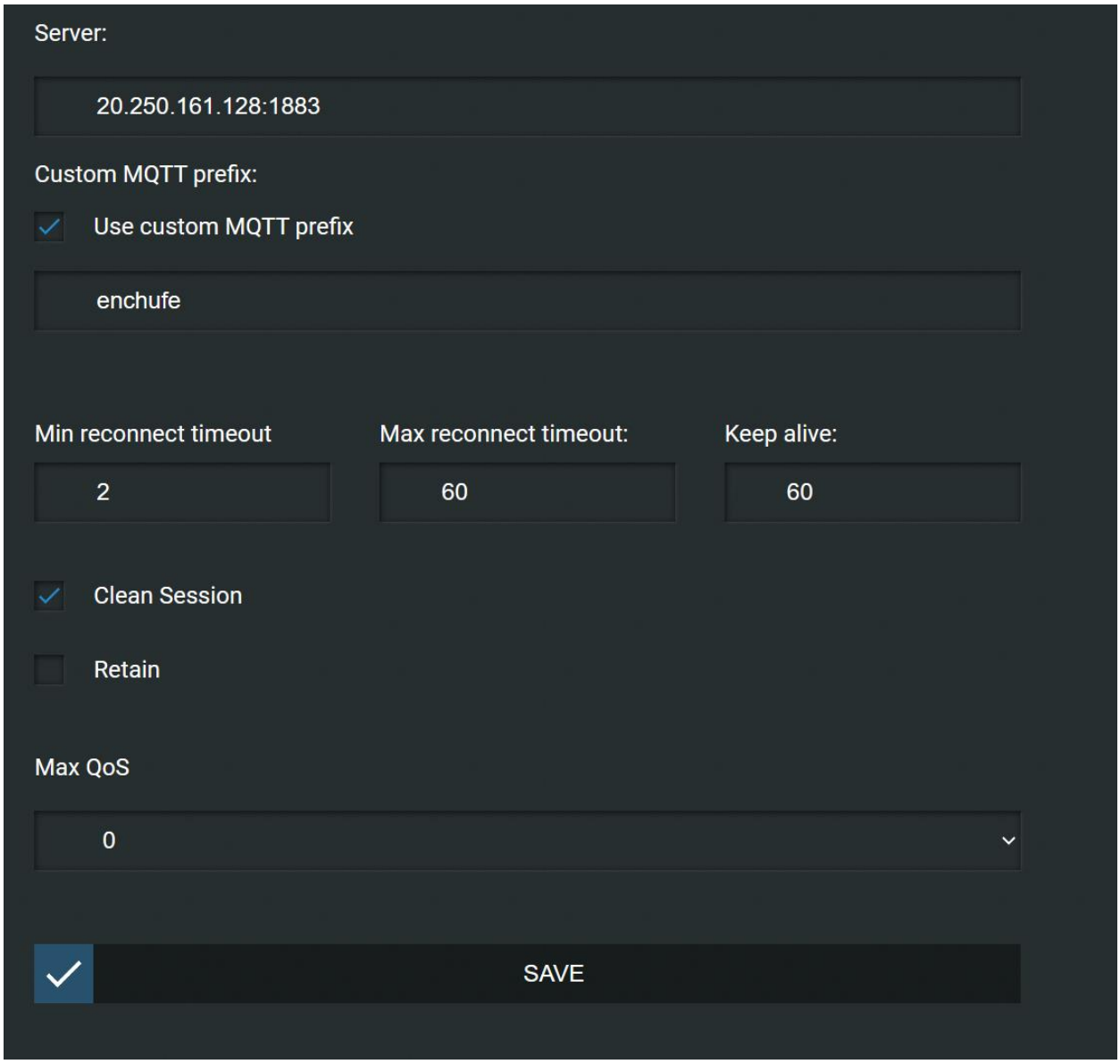


Figura 3-7. Configuración de la dirección del broker MQTT en el Shelly Plug S

Para habilitar el uso de MQTT en el enchufe es necesario activar dicha opción, como puede verse en la figura 3-6. No obstante, esto impedirá el uso de funcionalidades *cloud* como podrían ser la integración con Alexa o con Google Home. En la figura 3-7 se muestra la configuración específica del cliente MQTT del enchufe. En esta configuración lo más relevante es indicar la dirección IP y puerto del broker MQTT, y el uso de un prefijo personalizado para los tópicos MQTT en los que el enchufe publicará datos y se suscribirá para recibir comandos de control. De esta forma, será mucho más sencillo identificar qué tópicos está usando el enchufe y relacionarlos de forma unívoca con él.

La dirección IP y puerto configurados se corresponden con el broker Mosquitto mencionado en la Sección 3.1 (Configuración del broker MQTT). Dicha IP (20.250.161.128) es la IP pública de la máquina virtual en la que está desplegado el sistema domótico. Además, el prefijo para los tópicos MQTT que se ha elegido es "enchufe".

3.3 Despliegue y configuración de NodeRED

El componente más importante del sistema domótico es NodeRED, ya que nos permite programar los aspectos de control y monitorización del sistema siguiendo un enfoque basado en eventos, lo cual es fundamental

cuando estamos recopilando datos generados por dispositivos IoT. Además, nos va a permitir interactuar con servicios REST de forma sencilla y automatizada, potenciando las funcionalidades del sistema.

3.3.1 Despliegue del contenedor NodeRED

A pesar de que NodeRED se puede ejecutar de forma nativa, la propia página oficial de NodeRED tiene una guía recomendada para ejecutar NodeRED como un contenedor Docker [27] , como puede verse en la figura 3-8:

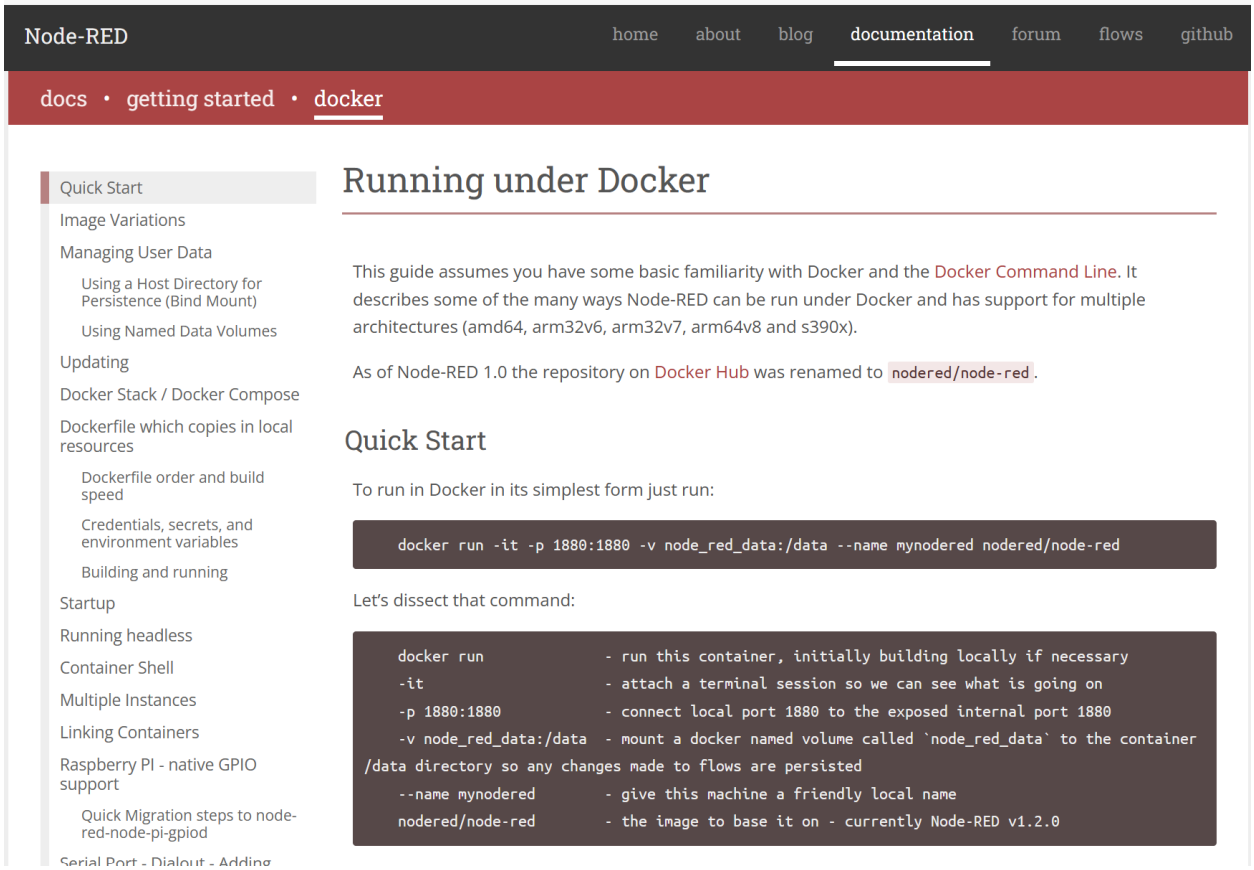


Figura 3-8. Guía para ejecutar NodeRED como contenedor Docker

En el caso de NodeRED, el directorio más importante que debe persistirse es el directorio **/data**, puesto que en este directorio es donde se almacenan los flujos programados en NodeRED, así como ficheros de configuración, certificados, paquetes instalados, etc. En la figura 3-9 de muestra la definición del contenedor NodeRED realizada mediante Docker-Compose:

```
services:

  nodered:

    container_name: nodered

    image: nodered/node-red:3.1-debian

    ports:
      - "443:1880"

    restart: always

    volumes:
      - ./nodered/data:/data

    environment:
      - TZ=Europe/Madrid

```

Figura 3-9. Definición del contenedor NodeRED en Docker-Compose

La definición de dicho contenedor consiste en:

- Crear un servicio en Docker-Compose llamado **nodered**.
- Crear un contenedor llamado **nodered**, asociado al servicio **nodered**.
- Crear dicho contenedor a partir de una versión concreta de la imagen de NodeRED. A diferencia del caso del contenedor Mosquitto, no utilizaremos la etiqueta latest para evitar posibles incompatibilidades con versiones futuras de NodeRED que impidan utilizan los flujos que se han programado. Para este proyecto se está utilizando la etiqueta **3.1-debian**.
- Abrir el puerto 443 de la máquina anfitriona y redirigirlo al puerto 1880 del contenedor. Por defecto, el puerto estándar de NodeRED es el 1880. Sin embargo, para simplificar el acceso mediante navegador al editor de flujos y a la interfaz de usuario, se hará uso de un puerto estándar web, que en el caso de HTTPS es el 443.
- Reiniciar automáticamente el contenedor en caso de que se detenga, garantizando que el servicio se esté ejecutando en todo momento.
- Compartir el directorio **./nodered/data** del *host* (partiendo del directorio de trabajo donde está el fichero docker-compose.yaml) con el directorio **/data** del contenedor. De esta forma, podremos acceder de forma sencilla a los ficheros de configuración de NodeRED, a los ficheros de flujos y almacenar certificados digitales desde la propia máquina anfitriona.
- Establecer la variable de entorno **TZ** (*Time Zone*) a la zona horaria de Madrid.

3.3.2 Configuración de NodeRED

El fichero que nos permite cambiar la configuración de NodeRED es el fichero **settings.js**, que se encuentra dentro del directorio **/data** del contenedor. NodeRED proporciona una guía de usuario básica [28] que aborda de forma general lo que se puede configurar en este fichero. No obstante, existen guías más detalladas y especializadas sobre este fichero, como por ejemplo, medidas para securizar NodeRED [29]. En la figura 3-10 se muestra un extracto del fichero settings.js con las modificaciones que se han realizado en este proyecto:

```
76
77  ····/**********************************************************************/
78  ····/**********************·CAMBIOS·EN·LA·CONFIGURACIÓN·**********************/
79  ····/**********************************************************************/
80
81  ····//·Configuramos·un·usuario·administrador·para·acceder·a·la·interfaz·web·del·editor·de·Node-RED.
82  ····
83  ····adminAuth:·{
84  ····    type:·"credentials",
85  ····    users:·[{
86  ····        username:·"sergio",
87  ····        password:·"$2b$08$UgW7zzBbh9Jqk4U66QQ4uua6J9qbaAWUbzuCT/W2Xmz5RZK7rxdJu",
88  ····        permissions:·"*"
89  ····    }]
90  ····},
91  ····
92  ····//·Cambiamos·la·ruta·del·editor·de·Node-RED·a·"/editor"
93  ····httpAdminRoot:·"/editor",
94
95  ····//·Cambiamos·la·ruta·del·interfaz·de·usuario·a·"/"
96  ····ui:·{
97  ····    path:·"/",
98  ····},
99
100 ····//·Configuramos·HTTPS·proporcionando·la·clave·privada·y·el·certificado
101 ····https:·{
102 ····    key:·require("fs").readFileSync('/data/certs/clave_priv.pem'),
103 ····    cert:·require("fs").readFileSync('/data/certs/cert.pem')
104 ····},
105
106 ····/**********************************************************************/
107 ····/**********************************************************************/
108 ····/**********************************************************************/
109
```

Figura 3-10. Modificaciones del fichero settings.js

Como puede verse en la figura 3-10, la configuración de NodeRED se realiza creando y modificando objetos JSON en el fichero settings.js.

El primer cambio en la configuración consiste en la creación de un usuario administrador para securizar el acceso a la interfaz web del editor de flujos de NodeRED. Para ello, se establece un nombre de usuario, una contraseña y los permisos del usuario. En NodeRED, a las contraseñas se les debe aplicar un *hash* para almacenarlas de una forma más segura. Por defecto, NodeRED utiliza el algoritmo Bcrypt [30], y provee una herramienta de línea de comandos llamada "**node-red admin hash-pw**" para generar el *hash* de una contraseña en texto claro. En la figura 3-11 se muestra un ejemplo de uso de dicha utilidad de NodeRED, obteniendo el *hash* Bcrypt de la contraseña "hola123":

```
sergio@ubuntu:~$ docker exec -it nodered bash
node-red@f5444821044d:~$ node-red admin hash-pw
Password:
$2b$08$bCP7Zq.nrUkvHjdhsLZgmeWOi7TOvNJkKnd1lwbIOd4Bl2UDf6LeW
node-red@f5444821044d:~$
```

Figura 3-11. Ejemplo de uso de la herramienta de hashing Bcrypt de NodeRED

Además, a este usuario administrador se le están asignando permisos de lectura y escritura del editor de NodeRED. También cabría la posibilidad de crear un usuario con acceso al editor que solo tuviera permisos de lectura.

Como resultado, en el sistema domótico únicamente el usuario administrador tiene acceso al editor de flujos de NodeRED, teniendo que autenticarse con su usuario y contraseña, como puede verse en la figura 3-12:

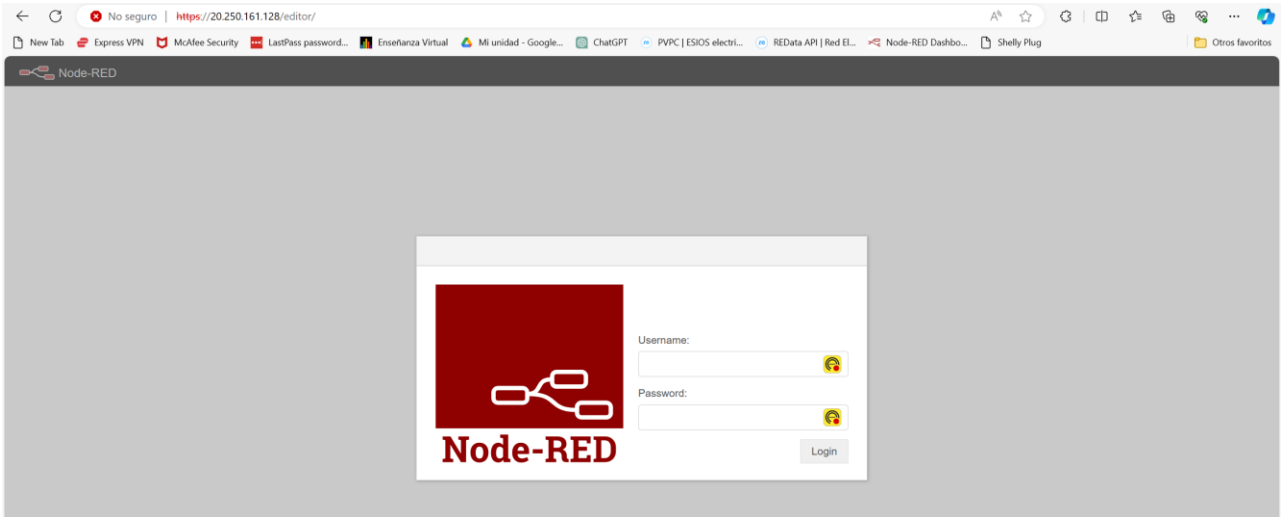


Figura 3-12. Login mediante usuario y contraseña para acceder al editor de flujos de NodeRED

Al estar basado en Node.js, NodeRED funciona como un servicio web. Por lo tanto, para acceder tanto al editor de flujos como a la interfaz de usuario, debemos navegar a las rutas correspondientes. Por defecto, el editor de flujos de NodeRED está expuesto en la ruta raíz (ruta /), mientras que la interfaz de usuario está expuesta en la ruta **/ui**. Para hacer que el acceso a estos recursos sea más intuitivo, se han modificado sus rutas. Con la configuración mostrada en la figura 3-10, la interfaz de usuario es accesible directamente desde la ruta raíz, mientras que para acceder al editor de flujos debemos navegar a la ruta **/editor**.

El último aspecto de configuración del fichero settings.js mostrado en la figura 3-10 es la configuración del uso de HTTPS en NodeRED, con el objetivo de garantizar la seguridad de la información de los usuarios al utilizar la interfaz del sistema domótico. Para ello, es necesario el uso de certificados digitales x509 [31]. En este proyecto se ha hecho uso de la herramienta OpenSSL [32], que permite crear y firmar certificados digitales desde la línea de comandos. No obstante, este certificado al haber sido autofirmado, no está firmado o reconocido por una CA. Por lo tanto, en la gran mayoría de navegadores se mostrará un aviso crítico de seguridad indicando que el certificado proporcionado por el servidor web no es válido, como puede verse en la figura 3-13. Sin embargo, esto no supone ningún impedimento a la hora de usar el sistema domótico, y simplemente habría que utilizar un certificado reconocido por una CA en un entorno de producción real.

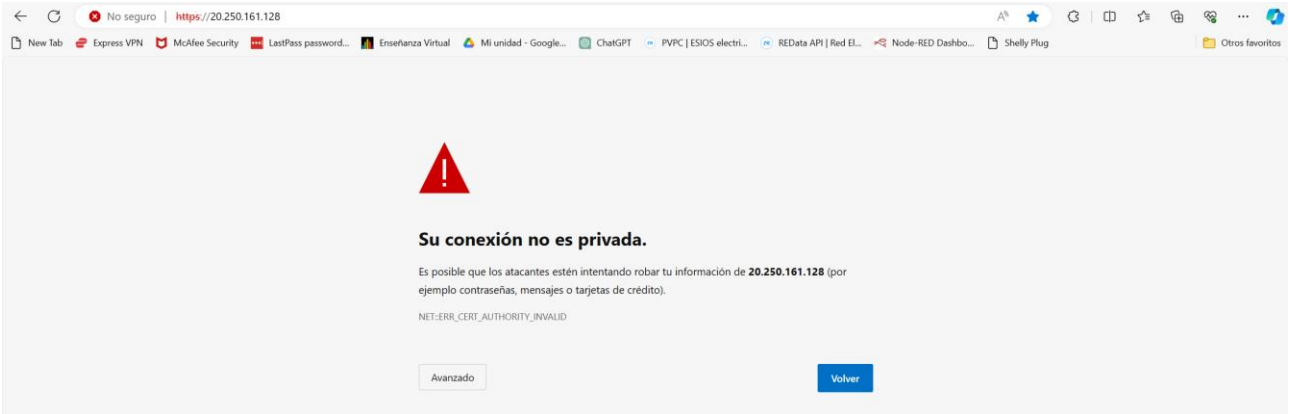


Figura 3-13. Aviso de certificado no válido al acceder a NodeRED

Como comentábamos, para poder utilizar HTTPS, NodeRED necesita tanto el certificado como la clave privada con la que se firmó el certificado. Para ello, se ha creado un directorio llamado **certs** dentro de la ruta **/data** donde se han almacenado estos ficheros generados mediante OpenSSL. Una vez que el contenedor puede acceder a estos ficheros, en la configuración de NodeRED ya podemos indicar cuál es el certificado y la

clave privada que se deben usar.

3.4 Comunicación con el enchufe inteligente mediante MQTT

Una vez que se han desplegado los contenedores de Mosquitto y NodeRED, y se le ha configurado un broker MQTT al enchufe inteligente, ya somos capaces de comunicarnos con el enchufe. El primer paso consiste en identificar los tópicos MQTT en los cuales el enchufe está publicando información. Para ello, podemos consultar la documentación del Shelly Plug S en la que se especifican los tópicos usados por el enchufe inteligente [33]. No obstante, para identificar dichos tópicos, también podemos usar una herramienta de línea de comandos proporcionada por Mosquitto llamada "mosquitto_sub" [34]. Esta herramienta nos permite suscribirnos a tópicos MQTT para que el broker nos reenvíe los mensajes que se han publicado en dichos tópicos. En la figura 3-14 se muestra un ejemplo de uso de "mosquitto_sub" realizando una suscripción a los mensajes publicados por el enchufe inteligente. Para ello, podemos suscribirnos al tópico comodín de MQTT, que es el símbolo almohadilla: #, el cual supone suscribirse a los mensajes de cualquier tópico.

```
sergio@ubuntu:~$ mosquitto_sub -v -t "#" -h 20.250.161.128
shellies/enchufe/relay/0/power 79.85
shellies/enchufe/relay/0/energy 2289
shellies/enchufe/relay/0/power 75.95
shellies/enchufe/relay/0/energy 2289
shellies/enchufe/relay/0/power 74.52
shellies/enchufe/relay/0/energy 2289
shellies/enchufe/relay/0 on
shellies/enchufe/temperature 25.53
shellies/enchufe/temperature_f 77.95
shellies/enchufe/overtemperature 0
```

Figura 3-14. Ejemplo de uso de mosquitto_sub para recibir los mensajes del Shelly Plug S

En la documentación del Shelly Plug S [33] se nos explica que todos los tópicos usados por el enchufe empiezan por el prefijo **shellies**. Para identificar al dispositivo en los mensajes publicados, por defecto se usa el identificador del dispositivo: **shellies/<model>-<deviceid>**. Sin embargo, en la configuración del enchufe realizada en la Sección 3.2 (Configuración del enchufe inteligente) se puede cambiar ese prefijo por uno personalizado, que en nuestro caso es el prefijo **enchufe**, por simplicidad. Como resultado, todos los tópicos de los mensajes MQTT del Shelly Plug S empiezan por **shellies/enchufe**.

El enchufe Shelly Plug S publica diversa información en el broker: estado del enchufe, potencia instantánea, energía consumida total, temperatura medida en el enchufe, sobrecalentamiento, etc. En este proyecto se ha hecho uso de los tópicos MQTT mostrados en la tabla 3-1:

Tópico	Acción realizada por el Shelly Plug S	Descripción
shellies/enchufe/relay/0/power	Publicación	Potencia instantánea consumida en vatios (W)
shellies/enchufe/relay/0	Publicación	Estado del enchufe. Puede estar encendido (on), apagado (off) o en sobrecorriente (overpower)
shellies/enchufe/relay/0/command	Suscripción	Recepción de comandos de control. Pueden ser encender (on), apagar (off) o conmutar (toggle)

Tabla 3-1. Tópicos MQTT del Shelly Plug S

Los tópicos mencionados en la tabla 3-1 nos permiten obtener la potencia instantánea en vatios medida por el enchufe, conocer si el enchufe está apagado o encendido, y controlar al enchufe encendiéndolo o apagándolo mediante comandos de control.

3.5 Recolección de las medidas de potencia del enchufe inteligente

Como ya se comentó en la Sección 2.2 (Tecnologías software), NodeRED es una herramienta de programación basada en la creación de flujos. Un flujo en NodeRED consiste en una secuencia de nodos conectados entre sí que realizan tareas de procesamiento y que van pasando mensajes al siguiente nodo, creando así un flujo lógico de información. Usualmente, el flujo de información se origina en un evento externo a NodeRED, como podría ser la recepción de un mensaje MQTT, o una petición HTTP. Sin embargo, también es muy común que se originen eventos dentro del propio NodeRED, típicamente mediante el uso de temporizadores. En la figura 3-15 se muestra la interfaz web del editor de NodeRED que nos permite programar este tipo de flujos de forma gráfica:

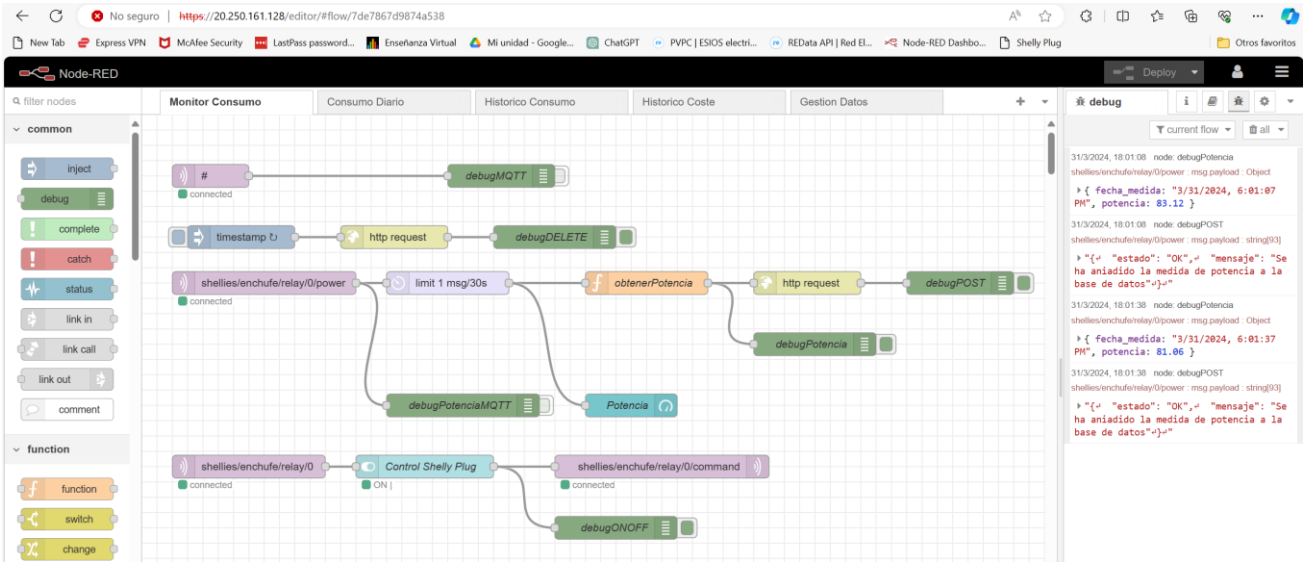


Figura 3-15. Interfaz web del editor de NodeRED

Para recibir en NodeRED las medidas de potencia instantánea publicadas en el broker MQTT, debemos utilizar uno de los nodos que nos permite la conexión a un broker y la suscripción a tópicos. Para ello, debemos instanciar este nodo y configurarlo. En la figura 3-16 se muestra dicho nodo y en la figura 3-17 se muestra su configuración:

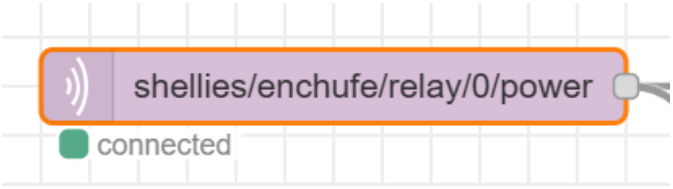


Figura 3-16. Nodo de conexión a un broker MQTT y suscripción a tópicos

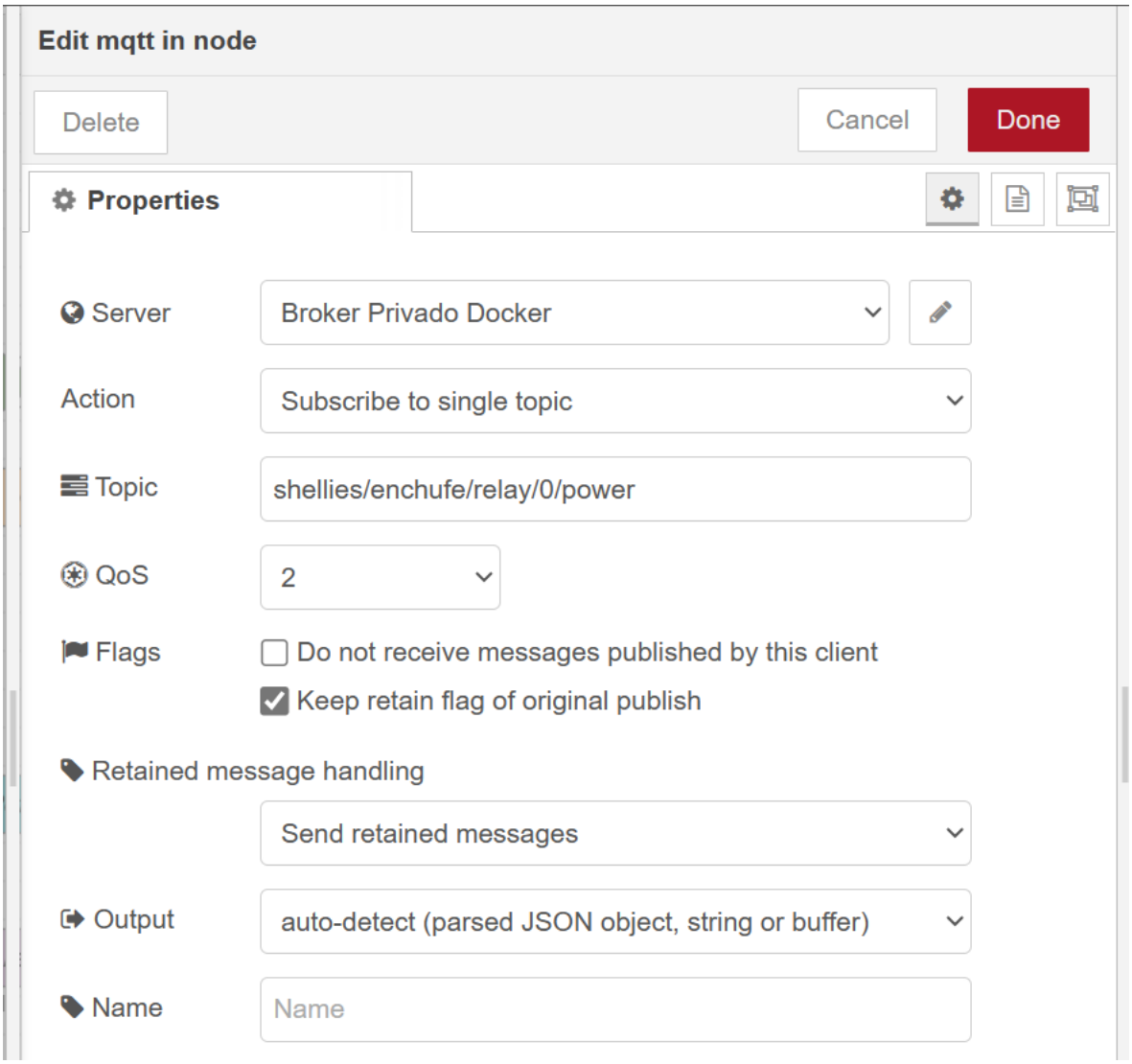


Figura 3-17. Configuración del nodo MQTT de suscripción

Lo más relevante en esta configuración consiste en elegir el broker MQTT, la acción a realizar (que consiste en suscribirse a un único tópico), y escribir el tópico al cual queremos suscribirnos. Como ya se explicó en la Sección 3.4 (Comunicación con el enchufe inteligente mediante MQTT), el tópico en el que el Shelly Plug S publica las medidas de potencia que va tomando es el tópico **shellies/enchufe/relay/0/power**.

En cuanto a qué broker debe usarse, debemos definirlo y configurarlo. En este proyecto lo hemos identificado

como **Broker Privado Docker**, ya que se trata del broker Mosquitto creado mediante Docker que se explicó en la Sección 3.1 (Despliegue y configuración del broker MQTT). Para ello, indicamos cuál es su dirección IP o su nombre de dominio, y el puerto que debe utilizarse. Gracias a que Docker-Compose incorpora un servidor DNS interno [35], podemos tener conectividad entre contenedores utilizando simplemente el nombre del contenedor correspondiente, en vez de tener que especificar su dirección IP. Por ello, en la configuración del broker MQTT mostrada en la figura 3-18, nos basta con indicar cuál es el nombre del contenedor con el broker MQTT, que es **mosquitto**:

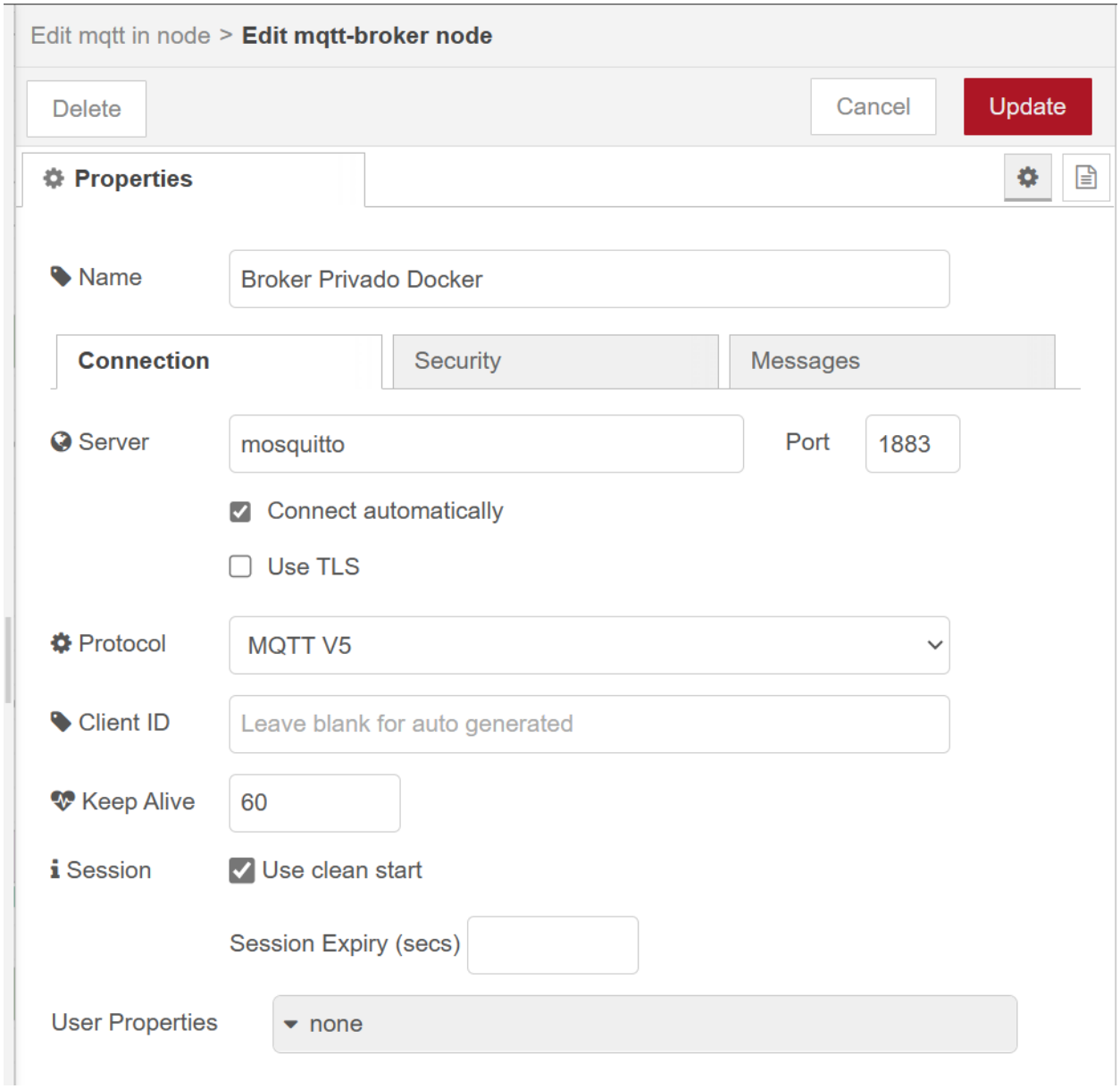


Figura 3-18. Configuración del broker MQTT en NodeRED

Una vez realizada esta configuración y mediante el uso de nodos de depuración, ya podemos visualizar desde NodeRED las medidas de potencia proporcionadas por el Shelly Plug S, como puede verse en la figura 3-19:

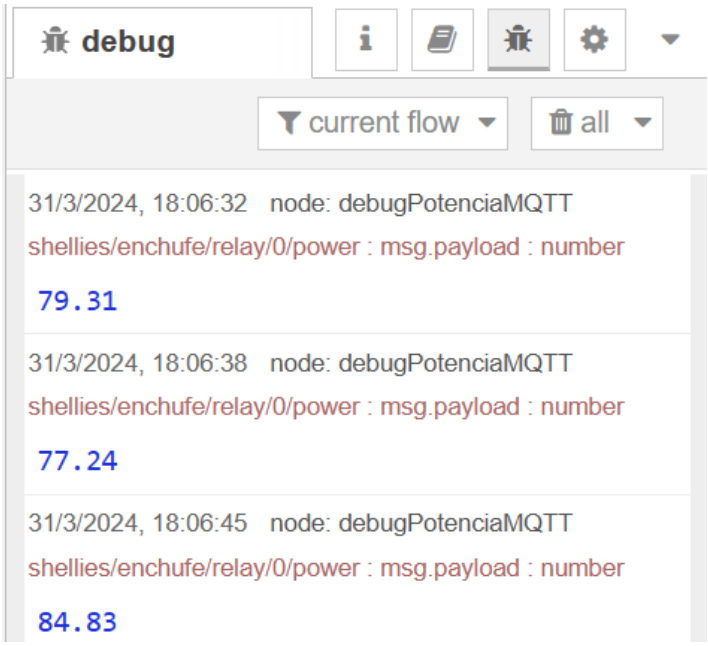


Figura 3-19. Medidas de potencia visualizadas mediante la herramienta de depuración de NodeRED

Tras realizar estos pasos, ya podemos hacer un seguimiento periódico de la potencia que se está consumiendo a través del enchufe. Lo siguiente que debemos hacer es almacenar las medidas de potencia en una base de datos a través del servicio REST de consumo y costes. La implementación y funcionamiento en detalle de dicho servicio REST se explica en el Capítulo 4 (Servicio REST de consumo y costes). Para ello, hacemos uso del flujo NodeRED mostrado en la figura 3-20:

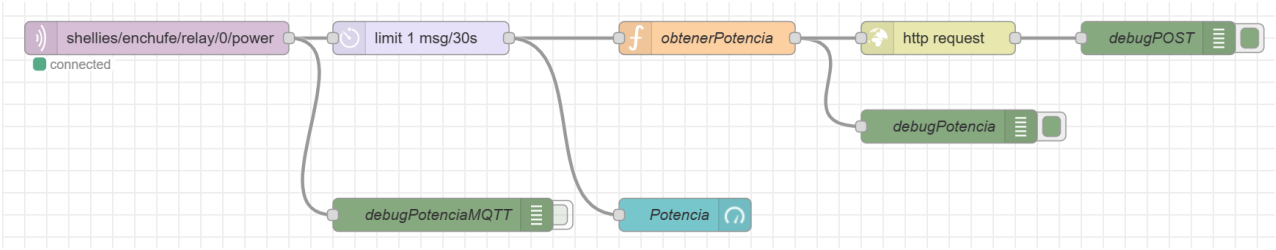


Figura 3-20. Flujo NodeRED para la recolección y almacenamiento de medidas de potencia

Para empezar, en dicho flujo limitamos la cantidad de mensajes recibidos desde el broker MQTT a un mensaje cada 30 segundos (nodo **limit 1 msg/30s**), para evitar tener una cantidad de medidas de potencia excesiva en la base de datos. No obstante, hay que recalcar que las medidas de potencia son borradas de la base de datos tras finalizar el día, puesto que dejan de ser de utilidad una vez que se ha calculado el consumo eléctrico y los costes de ese día en concreto. Con este límite, en la base de datos habrá un máximo de 2880 medidas de potencia cada día.

El siguiente nodo consiste en una función JavaScript llamada **obtenerPotencia,** cuyo código se muestra en la figura 3-21:

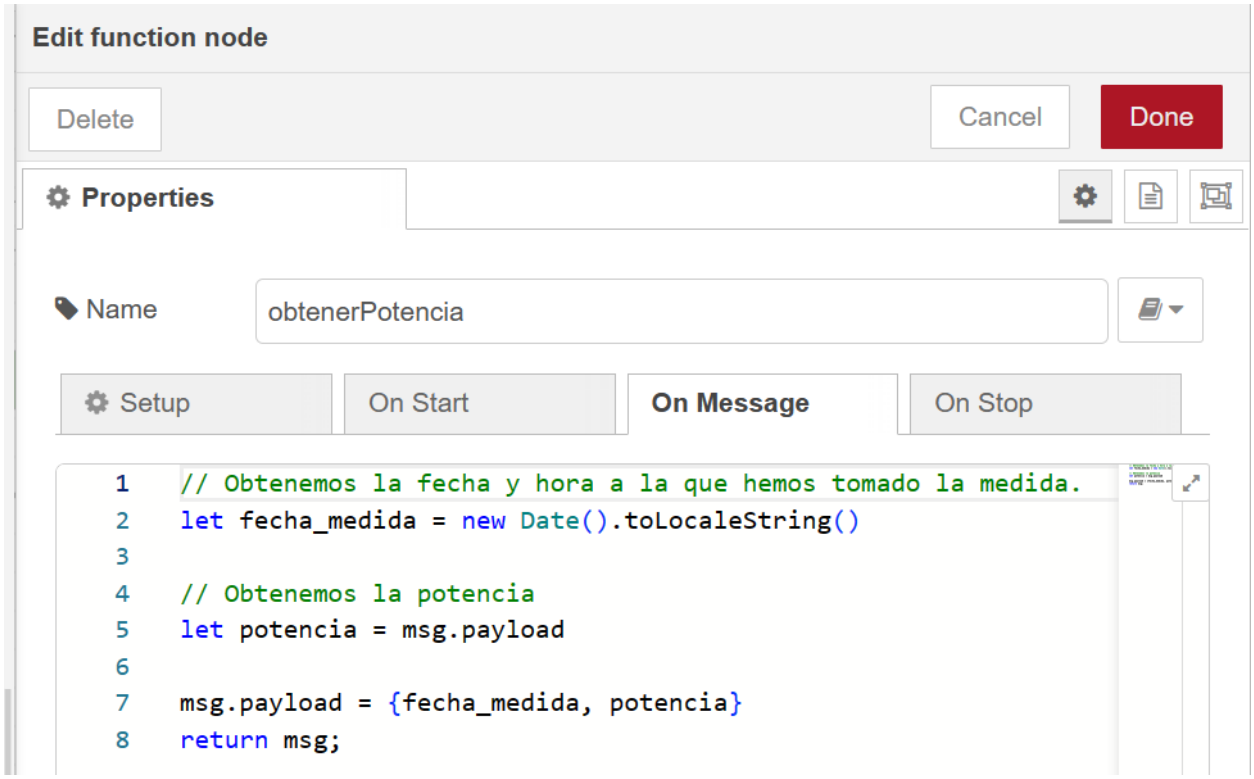


Figura 3-21. Código de la función obtenerPotencia

En dicha función, obtenemos la fecha y hora a la que hemos obtenido la medida de potencia desde el broker MQTT y creamos un objeto JSON compuesto por la fecha de la medida y su valor de potencia. El objeto que típicamente se intercambia entre los nodos de NodeRED es el objeto **msg**. De forma general, la información del mensaje se almacena en el atributo **payload**. Por lo tanto, en esta función lo que hacemos es obtener el valor de potencia accediendo al atributo **msg.payload** y modificamos dicho atributo para que el siguiente nodo del flujo obtenga un objeto JSON con la fecha de la medida y el valor de potencia, como puede verse en la figura 3-22:

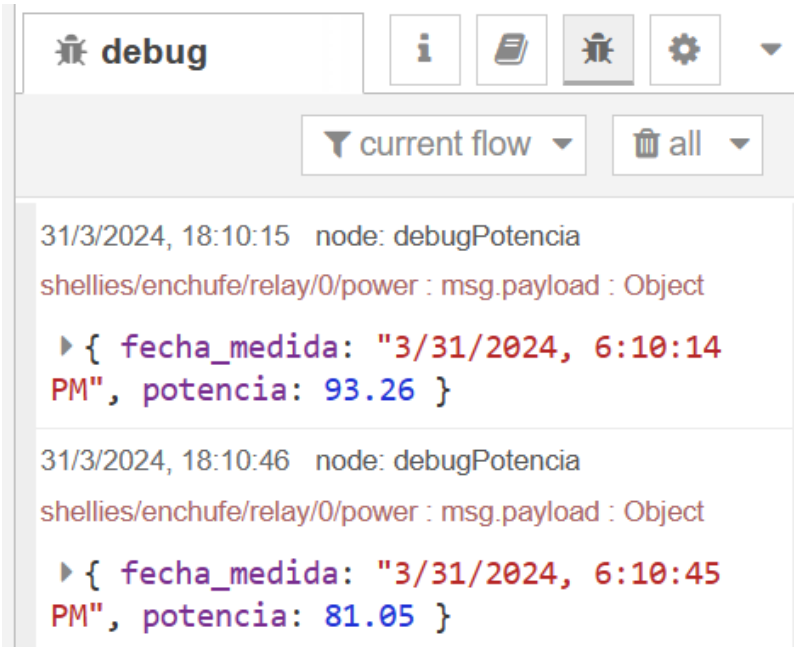


Figura 3-22. Medidas de potencia con fecha de medición en NodeRED

El último nodo del flujo mostrado en la figura 3-20 consiste en una petición HTTP POST al servicio REST de consumo y costes para almacenar la medida de potencia en la base de datos. La configuración de este nodo se muestra en la figura 3-23:

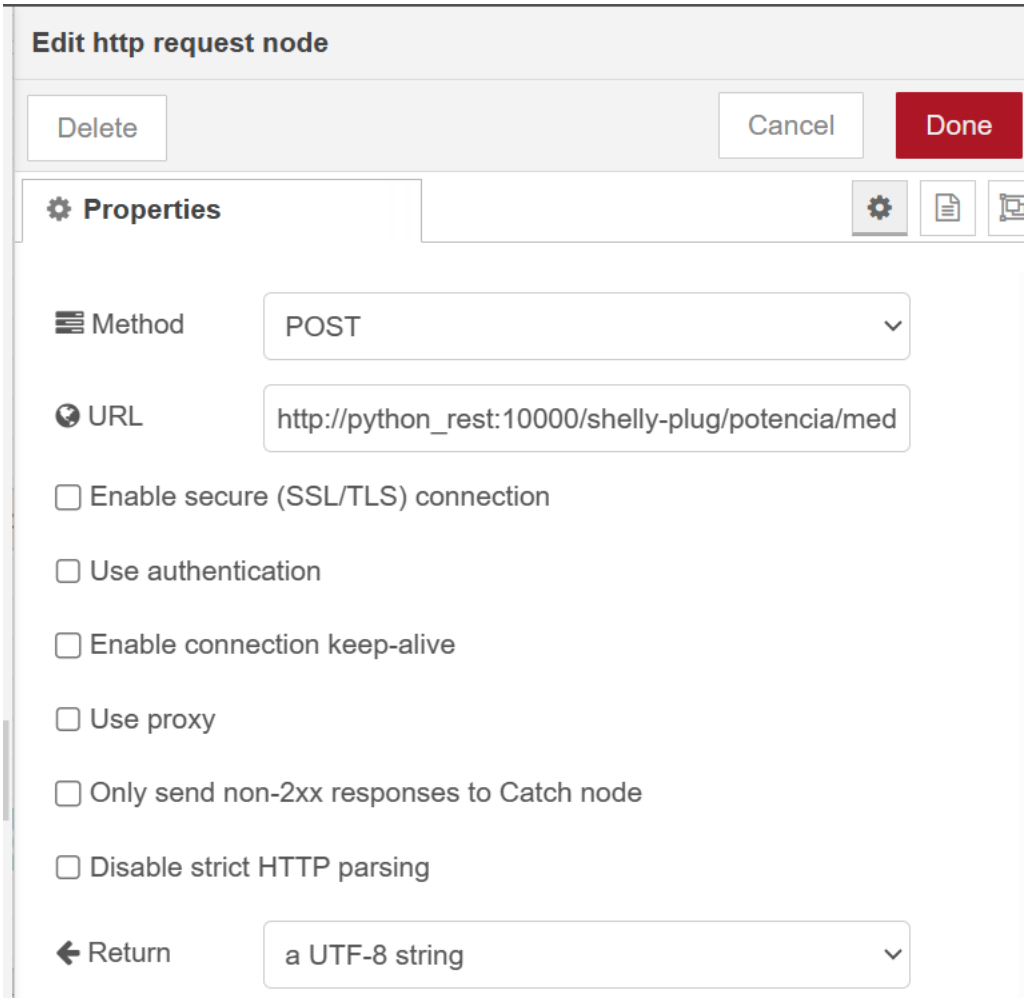


Figura 3-23. Configuración de un nodo HTTP POST para publicar medidas de potencia

En esta configuración indicamos que se debe realizar una petición POST a la URL http://python_rest:10000/shelly-plug/potencia/medidas. El nombre de dominio **python_rest** se corresponde con el nombre del contenedor en el que se está ejecutando el servicio REST de consumo y costes (aceptando peticiones en el puerto 10000). La ruta **shelly-plug/potencia/medidas** es la ruta del servicio REST que nos permite almacenar las medidas de potencia, como se explicará en el Capítulo 4 (Servicio REST de consumo y costes).

Como resultado, una vez que se ha almacenado con éxito una medida de potencia en la base de datos, obtenemos una respuesta satisfactoria desde el servicio REST, como puede verse en la figura 3-24:

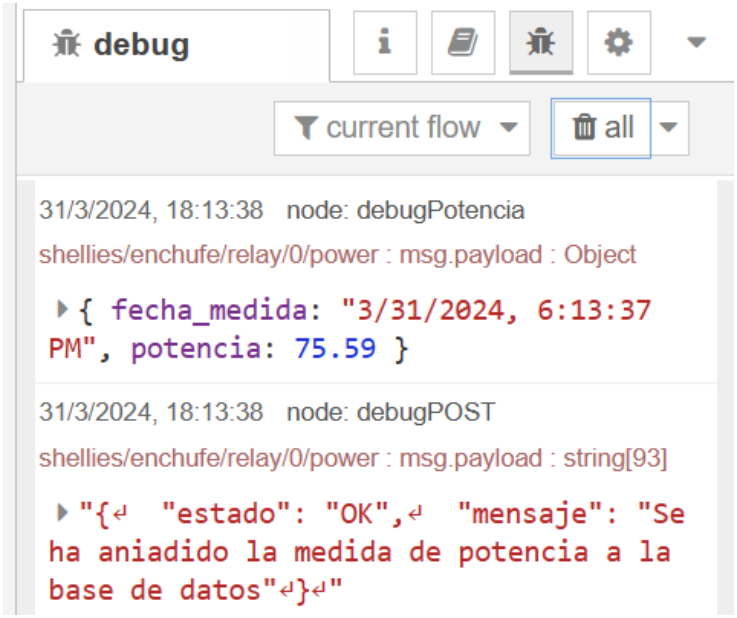


Figura 3-24. Respuesta satisfactoria tras almacenar una medida de potencia

De forma resumida, podemos ver los mensajes intercambiados durante el proceso de recolección de medidas de potencia en el diagrama de paso de mensajes mostrado en la figura 3-25:

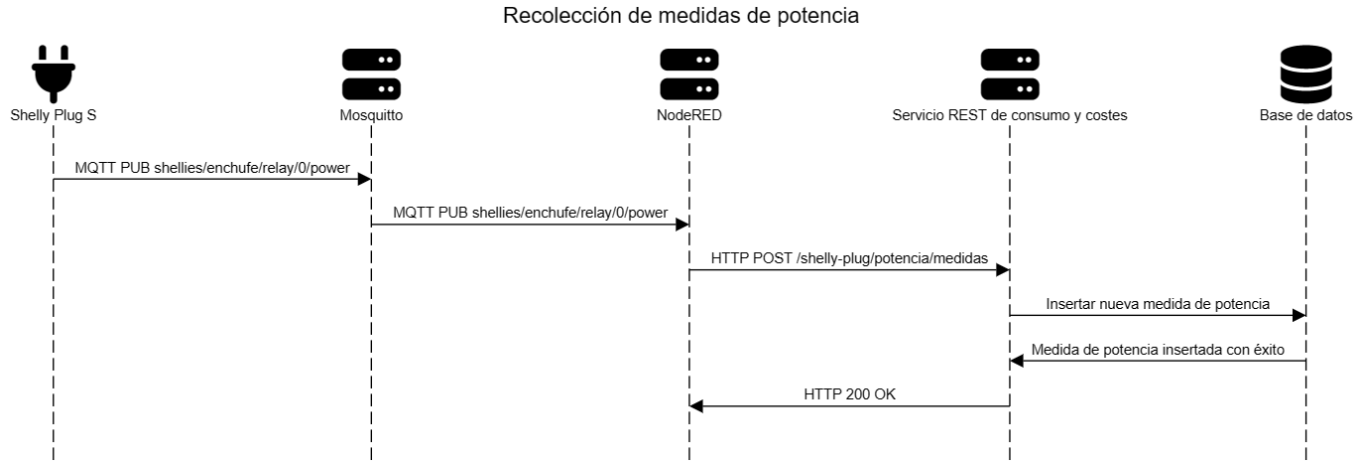


Figura 3-25. Diagrama de paso de mensajes para la recolección de medidas de potencia

Como se comentó previamente, también se ha programado el borrado automático de medidas de potencia de la base de datos para evitar almacenar medidas que ya han dejado de tener utilidad. Para ello, se ha usado el flujo mostrado en la figura 3-26:

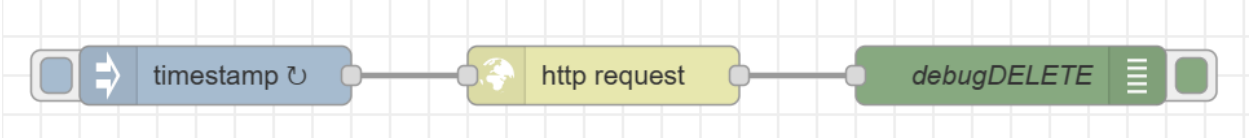


Figura 3-26. Flujo NodeRED para el borrado de medidas de potencia

En el nodo **timestamp** programamos un evento inicial que dará comienzo al flujo de información mediante el

envío de un mensaje (dicho mensaje contiene una marca de tiempo en su propiedad payload, aunque en este flujo no tiene demasiada utilidad). En la figura 3-27 puede verse que se ha programado el envío de ese mensaje todos los días a las 00:05 horas:

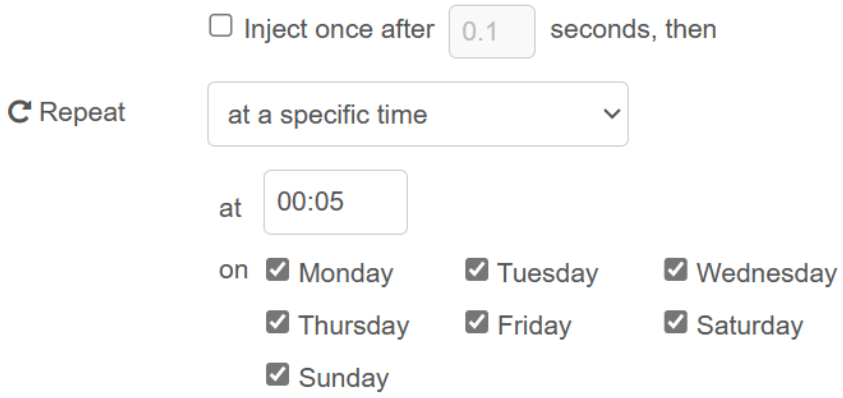


Figura 3-27. Programación de un evento de borrado de medidas periódico

Este evento periódico accionará el envío de una petición HTTP DELETE a la ruta /shelly-plug/potencia/medidas al servicio REST de consumo y costes, como puede verse en la figura 3-28:

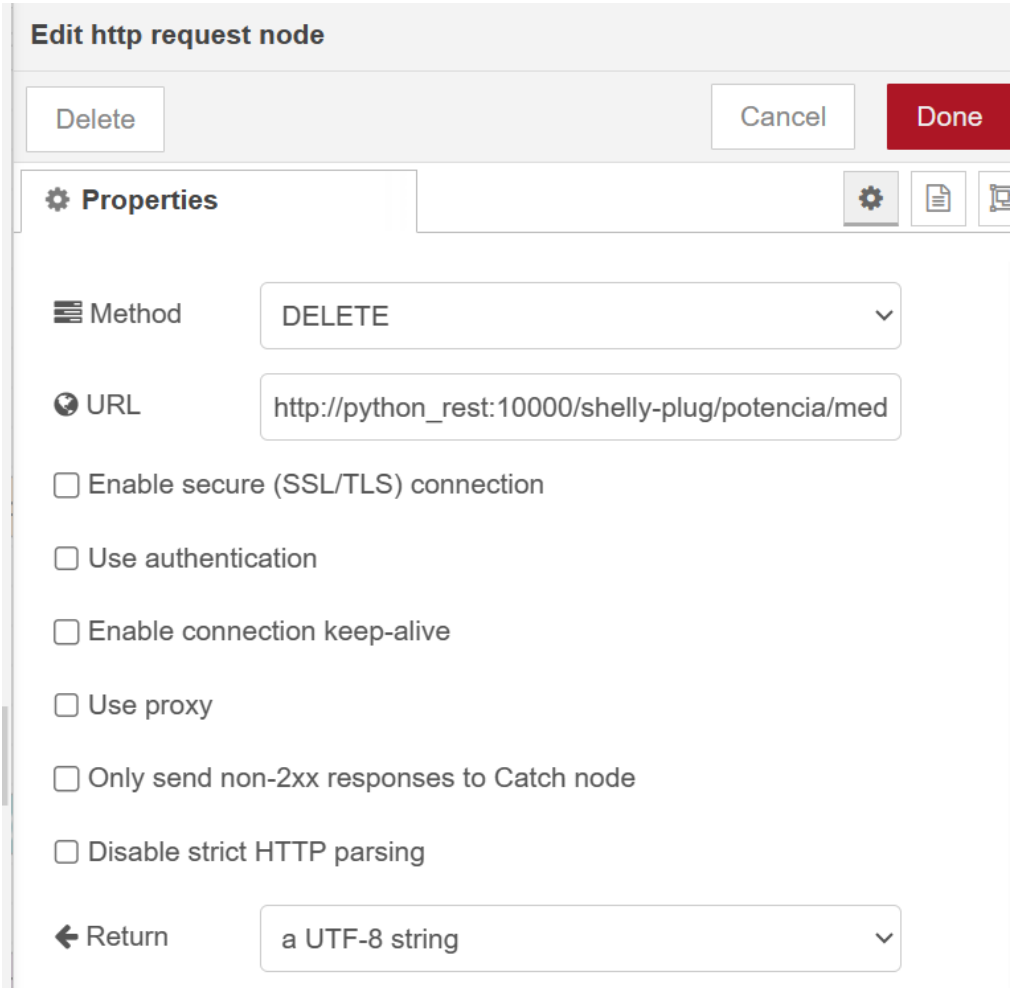


Figura 3-28. Configuración de un nodo HTTP DELETE para el borrado de medidas de potencia

Finalmente, si todo ha salido bien, el servicio REST de consumo y costes responderá con un mensaje indicándonos que se han borrado todas las medidas de potencia de la base de datos. Dicha respuesta se muestra en la figura 3-29:

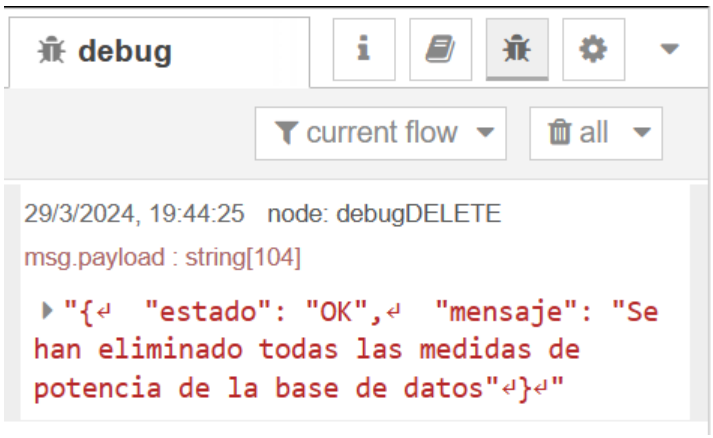


Figura 3-29. Respuesta satisfactoria tras borrar las medidas de potencia

3.6 Control del enchufe inteligente

De forma similar a como hemos obtenido las medidas de potencia del Shelly Plug S, haremos uso de MQTT para conocer el estado del enchufe y controlar su encendido y apagado. Para ello, empleamos el siguiente flujo NodeRED mostrado en la figura 3-30:

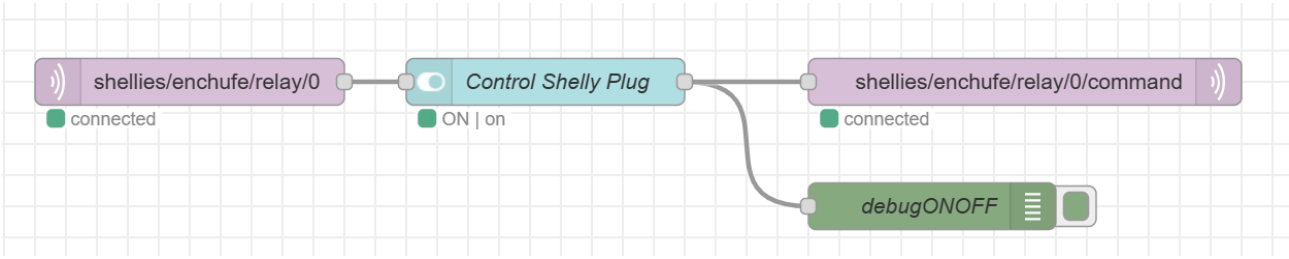


Figura 3-30. Flujo NodeRED para el control del enchufe inteligente

En primer lugar, debemos saber si el enchufe está apagado o encendido. Por lo tanto, debemos suscribirnos al tópico **shellies/enchufe/relay/0**, como ya se explicó en la Sección 3.4 (Comunicación con el enchufe inteligente mediante MQTT) para recibir mensajes con el estado del enchufe (on, off u overpower).

El siguiente nodo en el flujo es un nodo de interfaz de usuario que consiste en un interruptor de apagado y encendido, llamado **Control Shelly Plug**. En la Sección 3.13 (Interfaz de usuario) se explicará con más detalle cómo se ha realizado la interfaz de usuario. Este nodo permite que el usuario pueda controlar el apagado y encendido del enchufe desde la propia interfaz de usuario mediante un botón de tipo interruptor. Cuando el usuario pulse el interruptor, el nodo enviará el correspondiente mensaje de apagado (off) o encendido (on) al siguiente nodo. Además, este interruptor actualiza automáticamente su estado cuando le llega un mensaje desde el broker con el estado del enchufe. Esto resulta de gran utilidad para que el usuario pueda saber en todo momento si el enchufe está apagado o encendido, especialmente si el enchufe se ha apagado de forma manual en vez de haberlo hecho mediante la interfaz de usuario. En la figura 3-31 se muestra la configuración de este nodo interruptor:

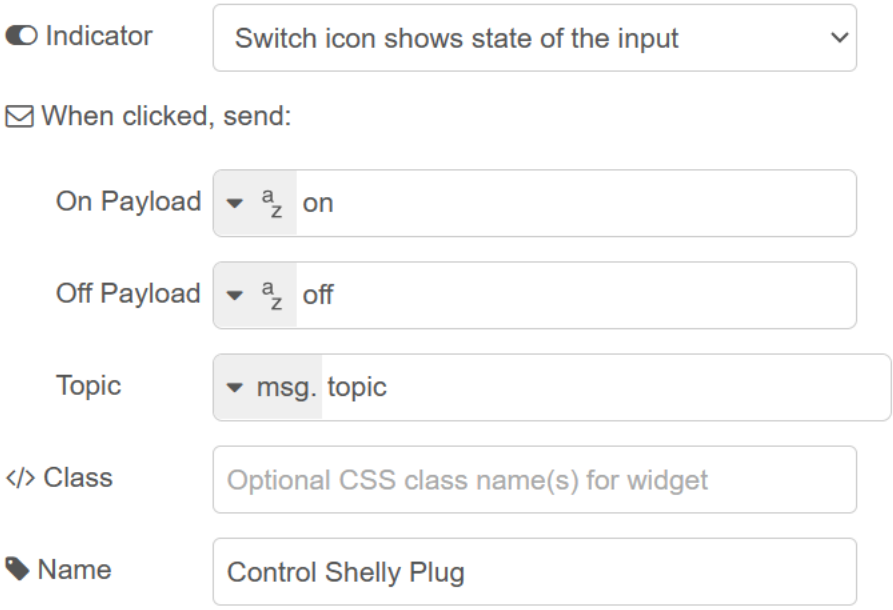


Figura 3-31. Configuración del nodo de control de apagado y encendido

En esta configuración especificamos que cuando se encienda el interruptor se debe enviar un mensaje con el contenido "on", y que cuando se apague, debe enviarse otro mensaje con el contenido "off". Además, para mostrar el estado del enchufe, el interruptor debe mostrar el estado indicado en el mensaje de entrada, que se corresponde con el mensaje publicado en el broker por el enchufe en el tópico shellies/enchufe/relay/0. En la figura 3-32 se muestra el aspecto del interruptor en la interfaz de usuario:

Figura 3-32. Botón para el control de apagado y encendido del enchufe en la interfaz de usuario

El último nodo consiste en el envío de comandos de control al Shelly Plug S en el tópico **shellies/enchufe/relay/0/command**. Los comandos enviados serán de apagado (off) y encendido (on), según lo que haya pulsado el usuario en la interfaz. Cuando se envíe uno de estos comandos, el enchufe publicará su estado actual en el tópico shellies/enchufe/relay/0 tras ejecutar dicha acción. Este intercambio de mensajes durante el encendido y apagado del enchufe se muestra en la figura 3-33:

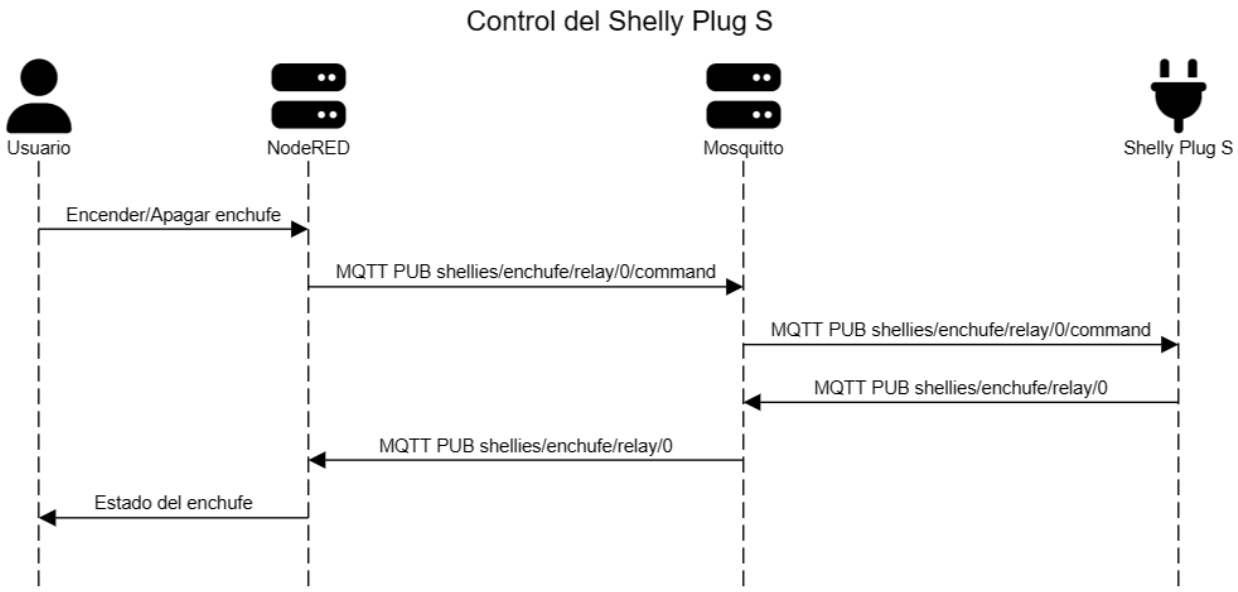


Figura 3-33. Diagrama de paso de mensajes para el control del Shelly Plug S

3.7 Monitorización de la potencia instantánea

La forma más sencilla de monitorizar la potencia instantánea consumida por el dispositivo conectado al enchufe es mostrar directamente los valores de potencia que vamos recibiendo del broker. Para ello, podemos utilizar un nodo de interfaz gráfica de tipo contador, que irá mostrando el valor actual de potencia e irá cambiando ese valor conforme lleguen los mensajes del broker. La configuración de dicho nodo se muestra en la figura 3-34, y el aspecto de dicho contador en la interfaz de usuario se muestra en la figura 3-35.

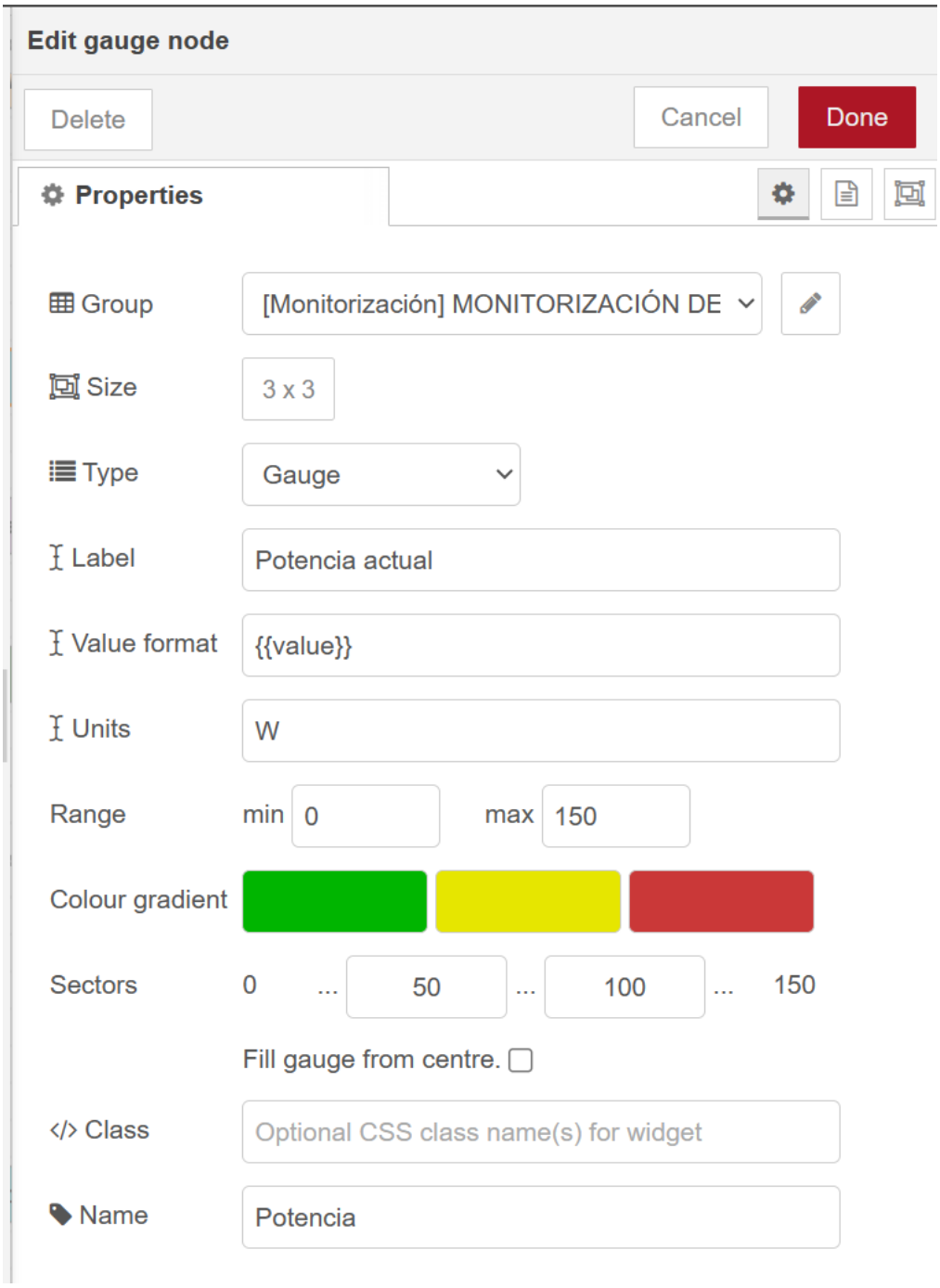


Figura 3-34. Configuración del nodo contador de potencia

En este nodo podemos configurar rangos de valores para cambiar el color del contador en función del valor de potencia recibido. Por ejemplo, si la potencia está entre 0 y 50 vatios el contador se pondrá de color verde, si está entre 50 y 100 se pondrá de color amarillo y si es superior a 100 vatios se pondrá de color rojo. De esta forma, podemos informar al usuario si el enchufe está midiendo un consumo bajo o si se está registrando un valor de potencia excesivo.

Figura 3-35. Contador de potencia en la interfaz de usuario

Sin embargo, a partir de las medidas de potencia instantánea almacenadas en la base de datos, podemos hacer un seguimiento de la potencia consumida a lo largo del tiempo. Para ello, en el servicio REST de consumo y costes se han implementado rutas que devuelven esos datos de potencia. Además, haremos uso de nodos de interfaz gráfica que nos permitirán graficar la potencia instantánea medida por el enchufe a lo largo del tiempo. El flujo NodeRED encargado de realizar estas tareas se muestra en la figura 3-36:

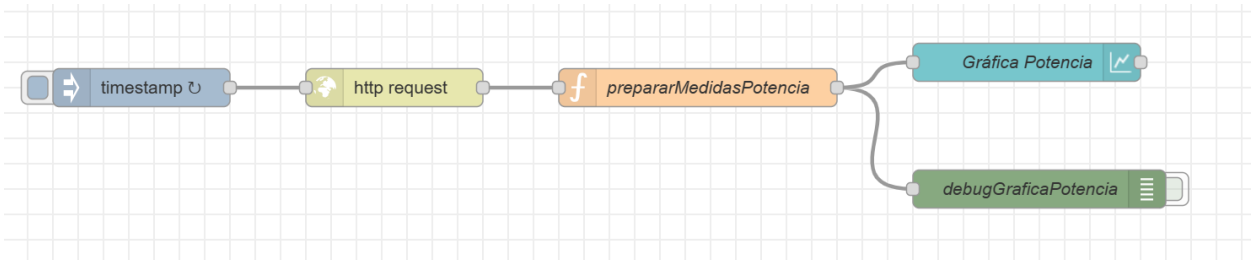


Figura 3-36. Flujo NodeRED para visualizar la potencia consumida a lo largo del tiempo

Cada 30 segundos actualizaremos la gráfica que muestra la potencia instantánea a lo largo del tiempo. Para ello, se hará una petición HTTP GET a la ruta **/shelly-plug/potencia/medidas** del servicio REST de consumo y costes. Si a esta ruta no se le especifica el rango horario en el que se quieren estas medidas en el *query string*, se obtendrán las medidas de potencia registradas en la última hora.

En el siguiente nodo, utilizamos una función JavaScript llamada **prepararMedidasPotencia** para preparar los datos de potencia en un formato de serie temporal, como puede verse en la figura 3-37. Este formato es necesario para poder representar de forma gráfica la potencia a lo largo del tiempo.

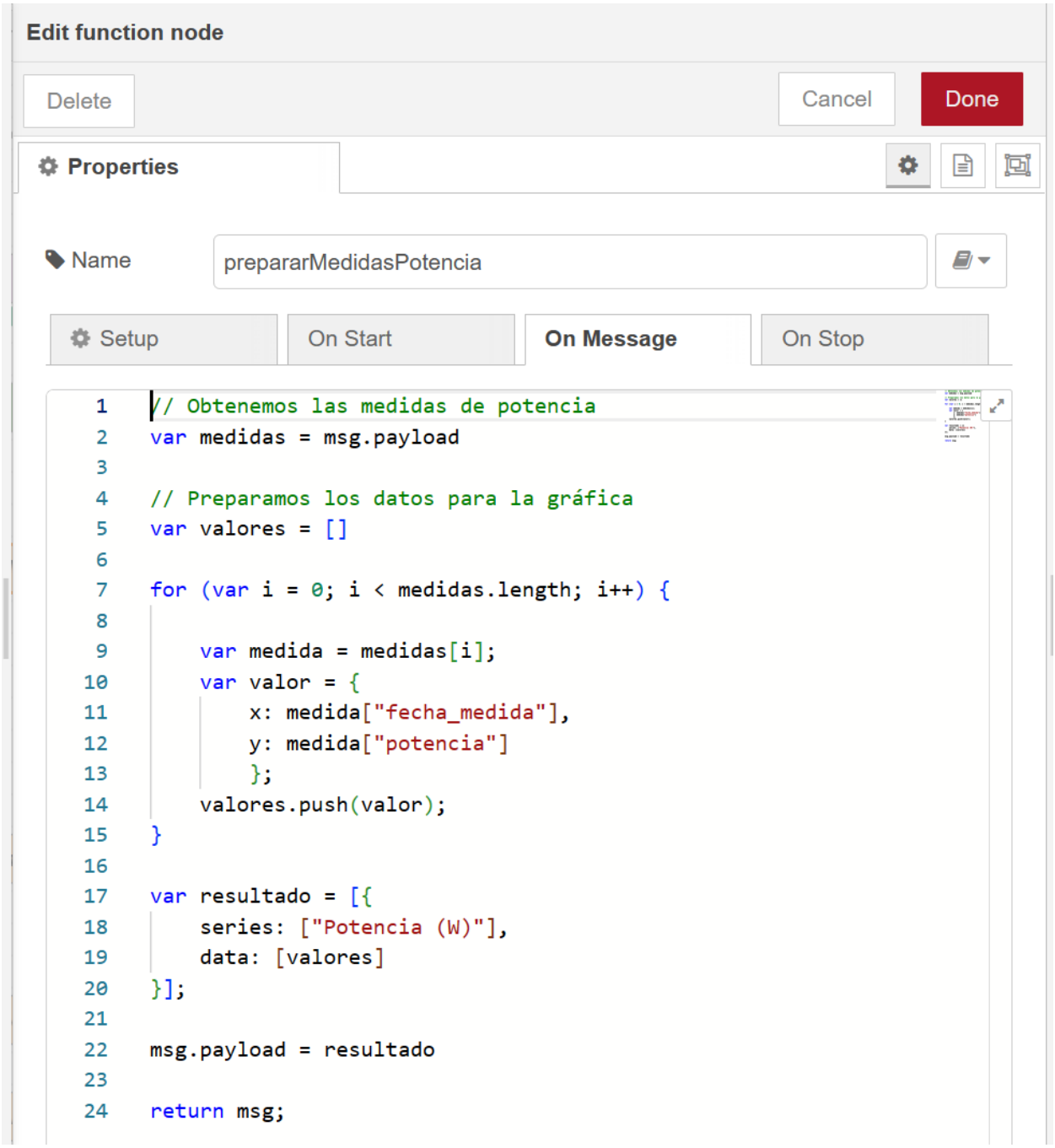


Figura 3-37. Código de la función prepararMedidasPotencia

El último nodo es el que nos permite graficar una serie temporal. La configuración de este nodo se muestra en la figura 3-38. En este caso, como se están obteniendo las medidas de potencia registradas en la última hora, los límites del eje horizontal de la gráfica se establecen desde hace una hora en adelante. Además, el valor del eje horizontal indica el tiempo en horas y minutos, mientras que el valor del eje vertical indica la potencia consumida en vatios. Por último, la estrategia de interpolación elegida es **step**. Como se están tomando medidas cada 30 segundos, se asume que la potencia consumida no cambiará drásticamente entre varias medidas seguidas, manteniendo un valor constante (lo cual suele ser lo común en la mayoría de los electrodomésticos). Por este motivo, esta estrategia de interpolación es la más acertada para mostrar cómo se está midiendo la potencia en este sistema. Sin embargo, otra opción de interpolación que se podría haber elegido es **linear**.

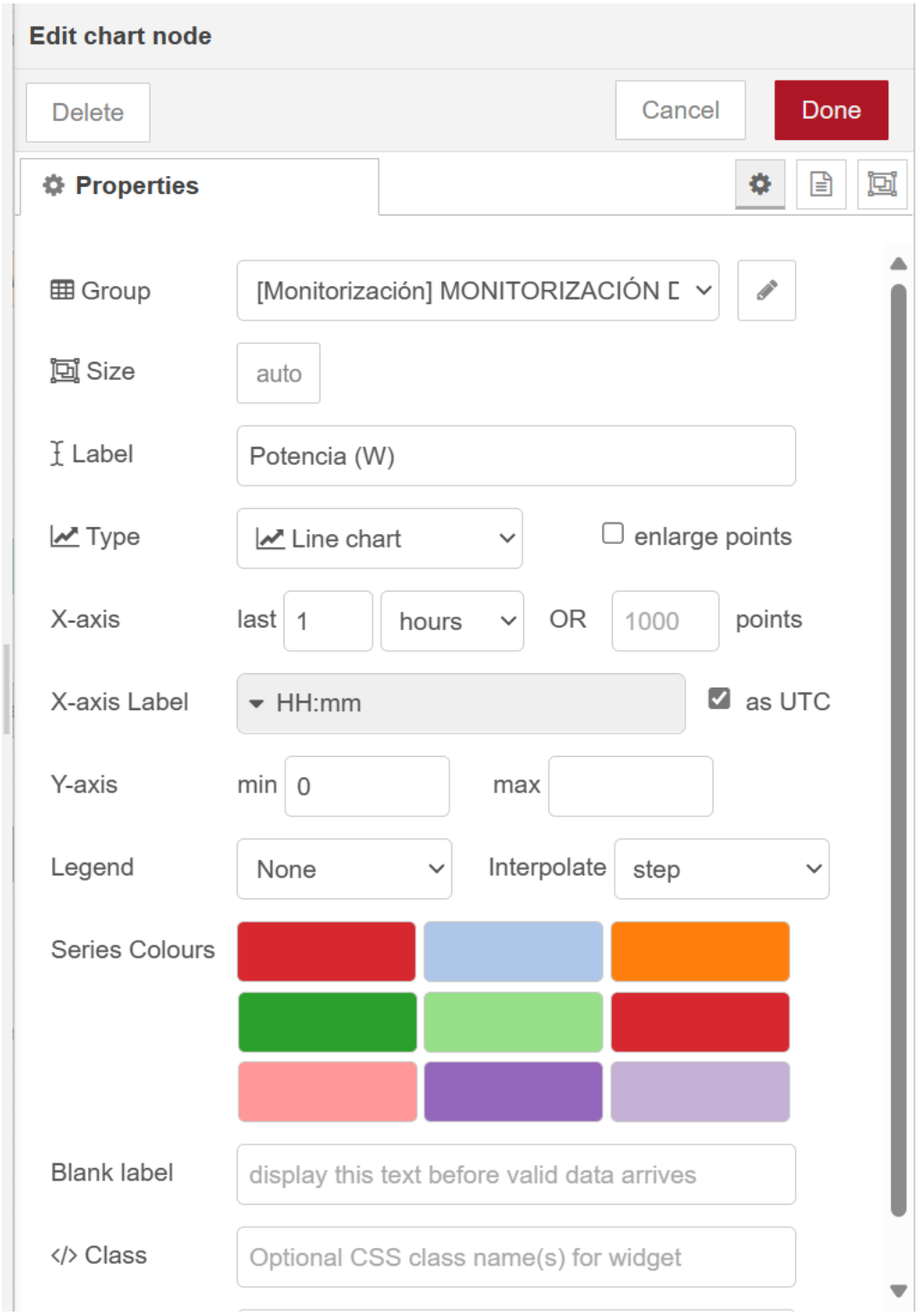


Figura 3-38. Configuración del nodo de gráfica para representar la potencia a lo largo del tiempo

Gracias a estas configuraciones, en la interfaz de usuario podremos visualizar de forma muy práctica la potencia en vatios consumida en el enchufe a lo largo del tiempo, como puede verse en la figura 3-39:

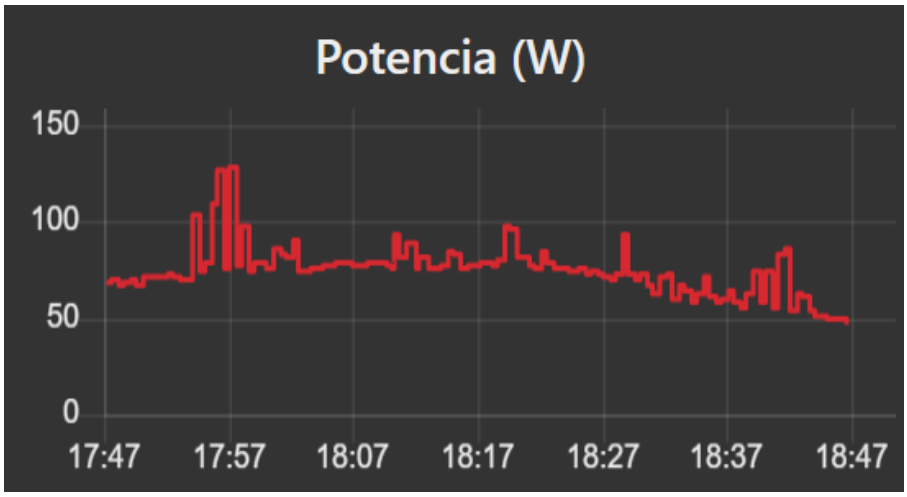


Figura 3-39. Representación gráfica de la potencia consumida en el enchufe a lo largo del tiempo

De forma general, siempre que el usuario visualice datos de potencia, consumos o costes en NodeRED, se realizará un intercambio de mensajes similar al de la figura 3-40. En dicha figura se muestra un diagrama de paso de mensajes para la visualización de la potencia consumida en el enchufe:

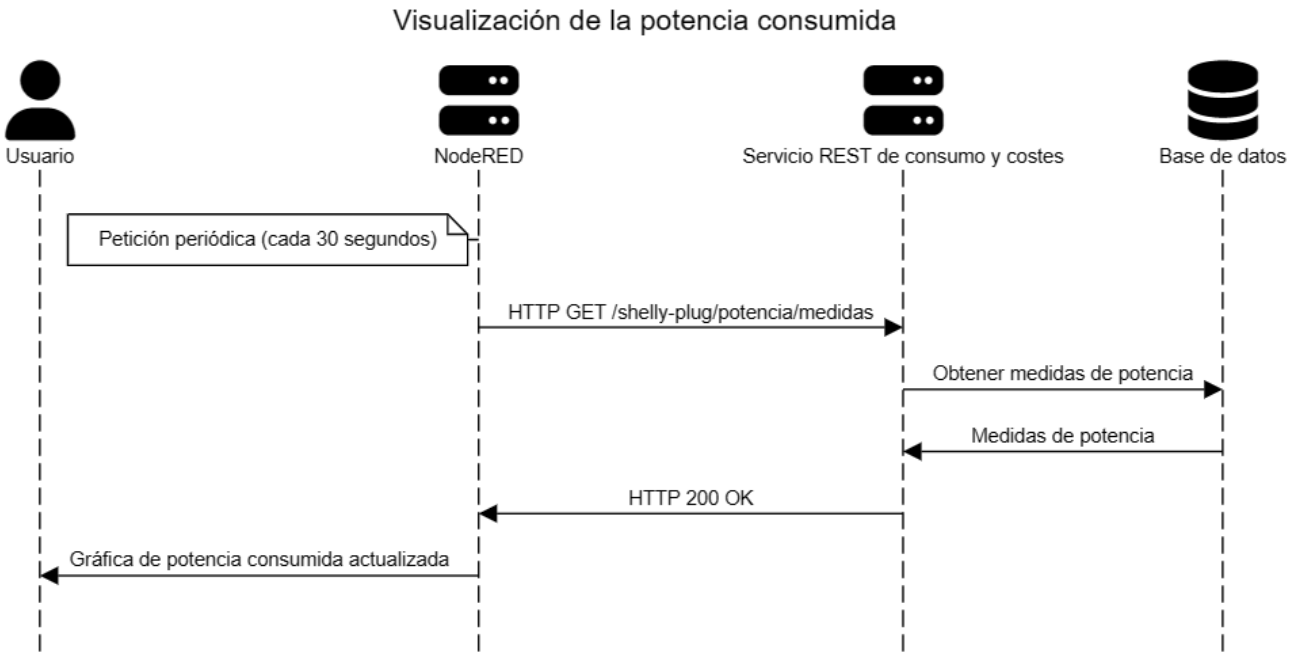


Figura 3-40. Diagrama de paso de mensajes para la visualización de la potencia consumida

3.8 Monitorización del consumo

Uno de los aspectos fundamentales de este proyecto es el seguimiento del consumo eléctrico. Como bien es sabido, el consumo eléctrico no es más que la potencia eléctrica que se ha consumido durante un periodo de tiempo. El servicio REST de consumo y costes explicado en el Capítulo 4 (Servicio REST de consumo y costes) proporciona una serie de rutas que realizan el cálculo del consumo a partir de las medidas de potencia que se han almacenado en la base de datos. Por lo tanto, podremos visualizar de forma relativamente sencilla el consumo energético en NodeRED mediante consultas a las rutas del servicio REST, delegando las tareas de cálculo a dicho servicio.

3.8.1 Consumo en la última hora

Para la consulta del consumo energético que ha habido durante la última hora, se ha implementado un flujo en NodeRED que de forma periódica realiza una petición HTTP GET a la ruta **/shelly-plug/consumo** del servicio REST de consumo y costes. Esta ruta permite obtener el consumo energético en vatios-hora calculado desde una hora inicio a una hora fin si ambas son proporcionadas en el *query string* de la petición HTTP. Si no se proporciona ningún parámetro, dicha ruta devuelve el consumo calculado en la última hora. Posteriormente, se utiliza una función JavaScript llamada **prepararConsumo** que simplemente obtiene el valor de consumo de la respuesta proporcionada por el servicio REST de tal forma que pueda ser representado en un nodo de tipo contador. La estructura de dicho flujo se muestra en la figura 3-41:

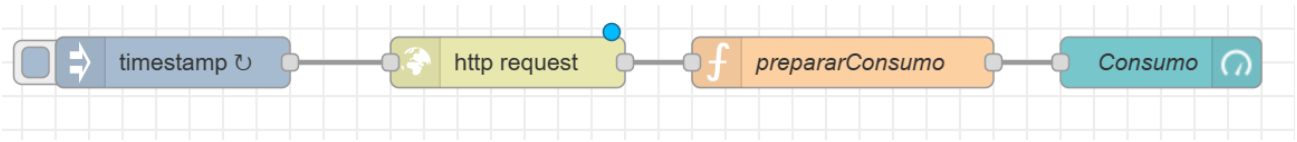


Figura 3-41. Flujo NodeRED para la visualización del consumo en la última hora

De forma similar al contador que permite visualizar la potencia medida por el enchufe, en la figura 3-42 se muestra un contador que permite ver el consumo en vatios-hora que se ha producido en la última hora:

Figura 3-42. Contador de consumo en la interfaz de usuario

3.8.2 Consumo por tramos horarios

El flujo NodeRED mostrado en la figura 3-43 permite consultar de forma periódica el consumo que se ha producido en el día desglosado en cuatro tramos horarios: **madrugada**, **mañana**, **tarde** y **noche**. Como se comentó previamente, si en la petición GET a la ruta **/shelly-plug/consumo** proporcionamos parámetros de hora de inicio y fin, obtendremos el consumo que se ha producido en ese intervalo de tiempo.

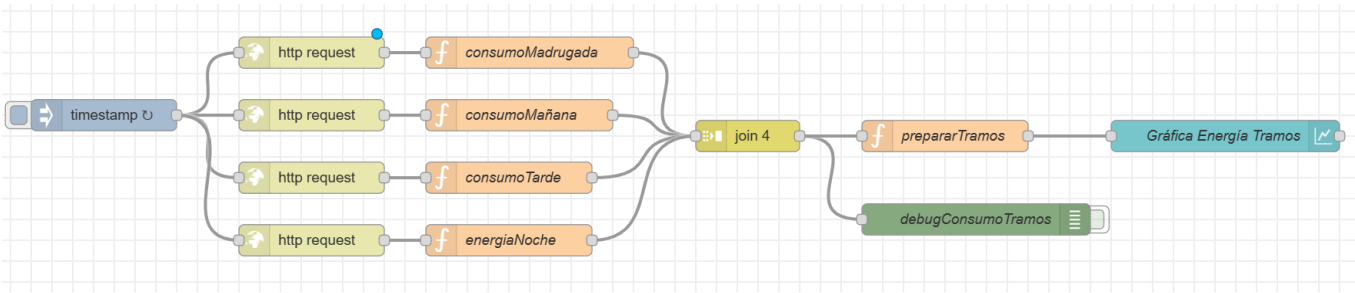


Figura 3-43. Flujo NodeRED para la visualización del consumo por tramos horarios

En este flujo se realizan cuatro peticiones HTTP GET a las rutas mostradas en la tabla 3-2 para obtener el consumo del tramo correspondiente:

Tramo	Horas	Ruta
Madrugada	00:00-05:59	/shelly-plug/consumo?hora_inicio=00:00&hora_fin=05:59
Mañana	06:00-11:59	/shelly-plug/consumo?hora_inicio=06:00&hora_fin=11:59
Tarde	12:00-18:59	/shelly-plug/consumo?hora_inicio=12:00&hora_fin=18:59
Noche	19:00-23:59	/shelly-plug/consumo?hora_inicio=19:00&hora_fin=23:59

Tabla 3-2. Rutas para consultar el consumo por tramos horarios

Posteriormente, se utiliza una función JavaScript en cada caso para crear un objeto JSON que contiene el tramo horario y el valor de consumo de ese tramo. En la figura 3-43 se muestra el código JavaScript para el caso del consumo en la madrugada:

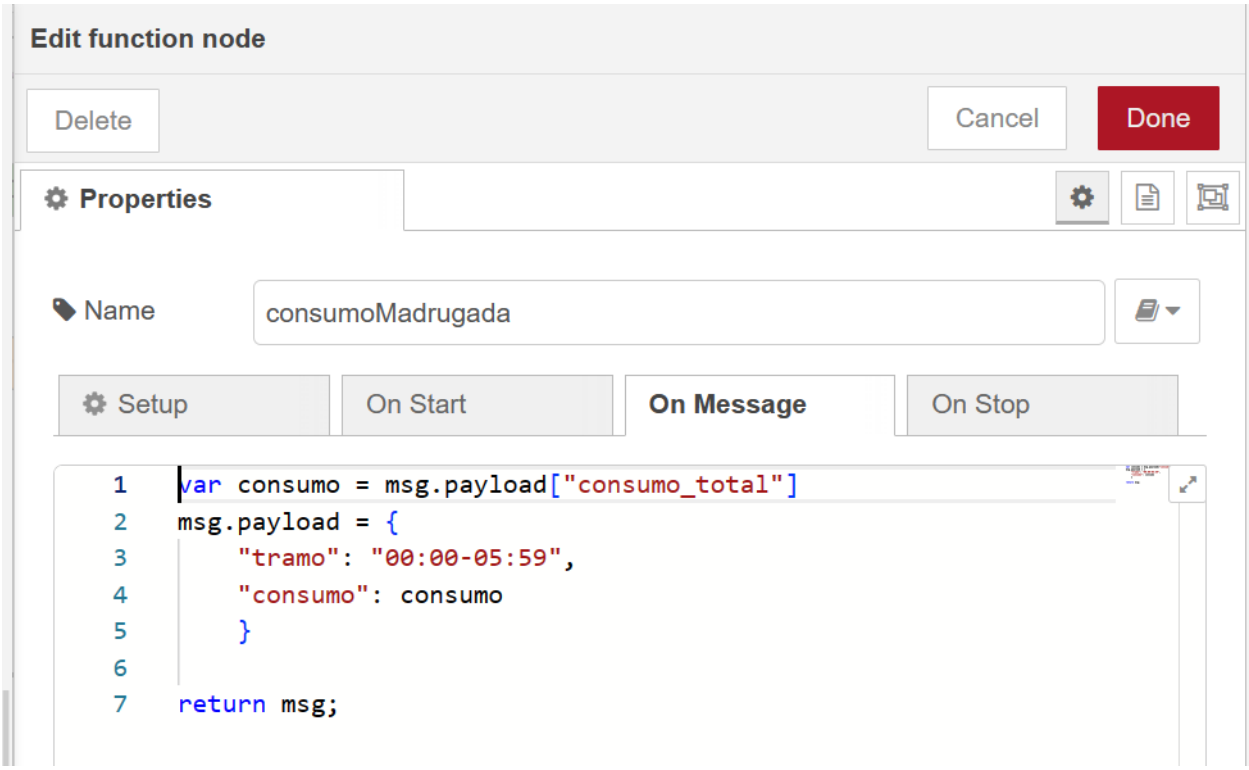


Figura 3-44. Código de la función consumoMadrugada

Para facilitar el manejo de los cuatro datos de consumo, en la figura 3-43 se puede ver que se hace uso de un nodo **join 4** para juntar los mensajes de los cuatro nodos anteriores en una lista. Gracias a esto, en la función **prepararTramos** podemos manipular estos datos de consumo para que puedan ser representados en una gráfica de barras. Para ello, se hace uso de un nodo de tipo gráfica similar al que se utilizó para representar la potencia a lo largo del tiempo, con la diferencia de que ahora se ha configurado para que se represente como gráfica de barras. Como resultado, en la interfaz de usuario podemos ver el consumo que se ha producido en el día en cuatro tramos horarios, como puede verse en la figura 3-45:

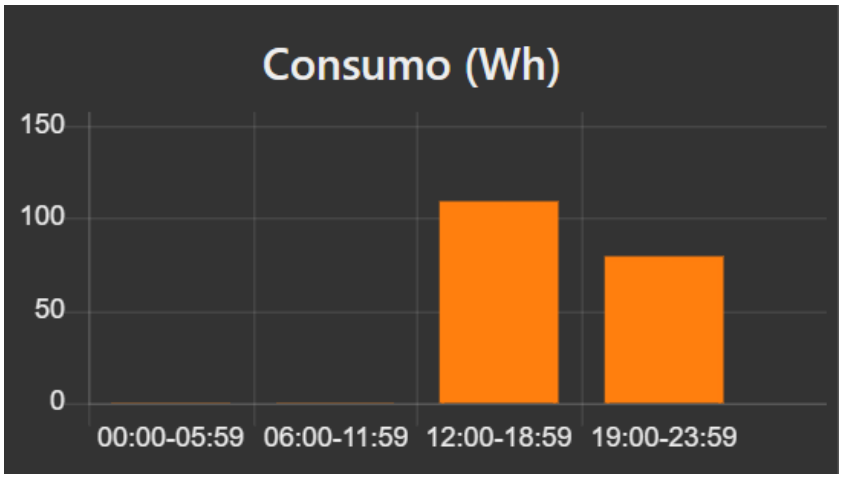


Figura 3-45. Representación gráfica del consumo energético por tramos horarios

3.8.3 Consumo en cada hora del día

De forma similar, podemos obtener el consumo desglosado en cada una de las 24 horas del día. Sin embargo, para simplificar la programación en NodeRED, se ha creado una ruta en el servicio REST de consumo y costes llamada **/shelly-plug/consumo/consumo-por-hora**. Cuando se hace una petición GET, esta ruta devuelve una lista con 24 valores de consumo para cada una de las horas del día, empezando desde las 00:00-00:59 hasta las 23:00-23:59. De esta forma, podemos consultar el consumo para cada hora del día de una forma mucho más eficiente, en vez de tener que realizar 24 peticiones GET a la ruta **/shelly-plug/consumo**. El flujo NodeRED que realiza esta consulta y muestra los resultados en una gráfica de barras se puede ver en la figura 3-46:

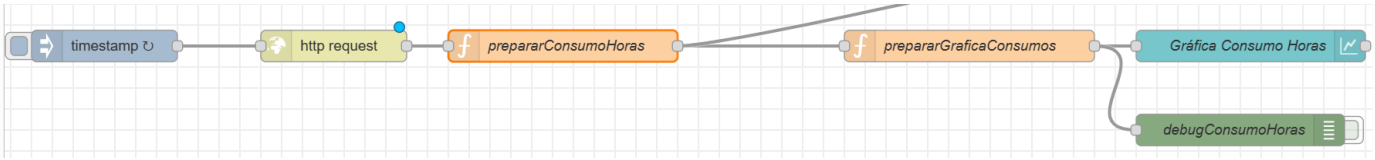


Figura 3-46. Flujo NodeRED para la visualización del consumo en cada hora del día

En este caso, el código JavaScript se ha separado en dos funciones distintas: **prepararConsumoHoras** y **prepararGraficaConsumos**. Esto se debe a que los datos de consumo por horas serán necesarios para hacer el cálculo de los costes de esos consumos. Sin embargo, para poder representar estos datos en una gráfica de barras primero hay que pasarlos a un formato distinto, lo cual justifica la separación del código en dos funciones que tienen salidas distintas. En la figura 3-47 se muestra la gráfica de barras (en este caso, horizontal) con los consumos producidos en cada hora del día:

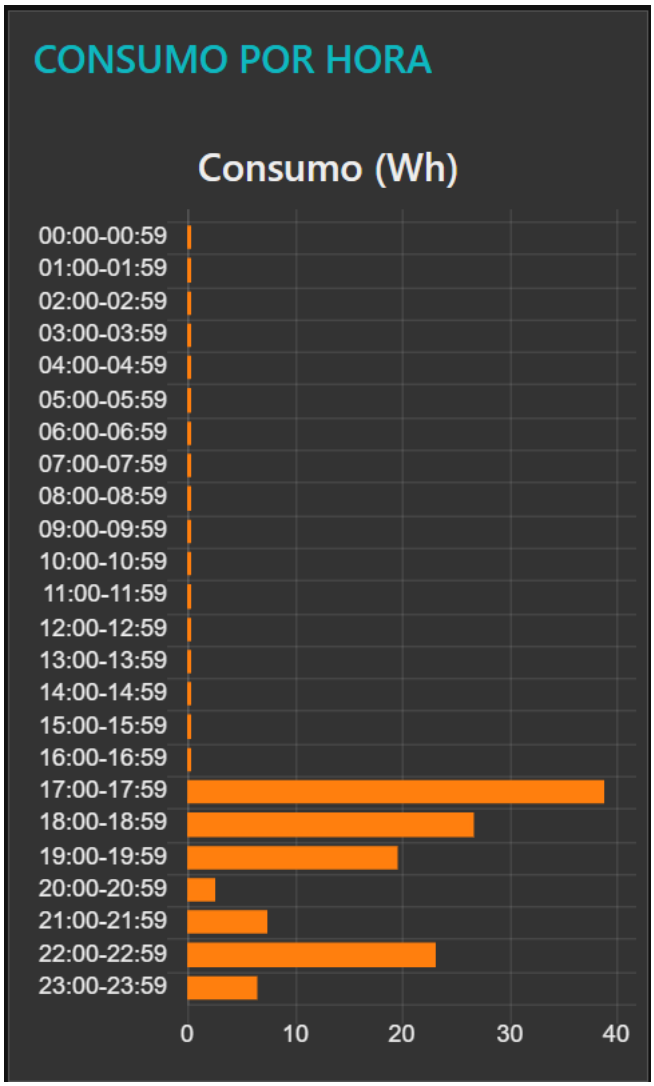


Figura 3-47. Representación gráfica del consumo en cada hora del día

3.9 Consulta de precios de la electricidad

Para hacer una estimación de los costes del consumo eléctrico, debemos conocer cuáles son los precios de la electricidad. Si bien el precio depende de la tarifa que el usuario tenga contratado con la empresa que le suministra electricidad, podemos tener como referencia la tarifa PVPC [6].

De forma general, se suelen contratar tarifas denominadas de **mercado libre**. Este tipo de tarifas dependen de la empresa comercializadora que las ofrezca, y pueden tener un precio fijo o variable según el horario, posibilidad de tener el suministro de electricidad y de gas con la misma compañía, descuentos, contratos con permanencia, etc. Sin embargo, otra tarifa muy común es la tarifa **regulada** o también llamada **PVPC**. La tarifa PVPC es una tarifa regulada por el Gobierno de España que establece los precios de la electricidad para todo el territorio nacional. Se trata de un precio dinámico que va cambiando en cada hora del día y está indexado al mercado mayorista de electricidad. Las características principales de la tarifa PVPC son las siguientes [36] [37]:

- Se pueden acoger a la tarifa PVPC personas físicas o microempresas, con tensiones no superiores a 1 kV y con potencia contratada menor o igual a 10 kW.
- Se trata de un precio variable que va cambiando para cada hora del día y que depende de la oferta y demanda en el mercado eléctrico.
- La tarifa PVPC solo puede ser comercializada por las denominadas comercializadoras de referencia [38]. Algunas de estas comercializadoras son Energía XXI (Endesa), Curenergía (Iberdrola) o

Comercializadora Regulada (Naturgy) [39].

- No tiene permanencia
- Para acceder al Bono Social ofrecido por el Gobierno de España es necesario tener contratada la tarifa PVPC [40].
- De acuerdo a lo establecido en el Real Decreto 216/2014 [38], de 28 de marzo, por el que se establece la metodología de cálculo de los precios voluntarios para el pequeño consumidor de energía eléctrica y su régimen jurídico de contratación, la empresa Red Eléctrica de España, operadora del sistema eléctrico español [41], publica cada día a las 20.15 horas los precios horarios para el día siguiente [6].

En referencia al último punto, la empresa Red Eléctrica de España es la encargada de publicar cada día la información de precios de la tarifa PVPC. Dichos precios pueden consultarse en la página de precios PVPC proporcionada por la propia empresa, como puede verse en la figura 3-48 [42]:

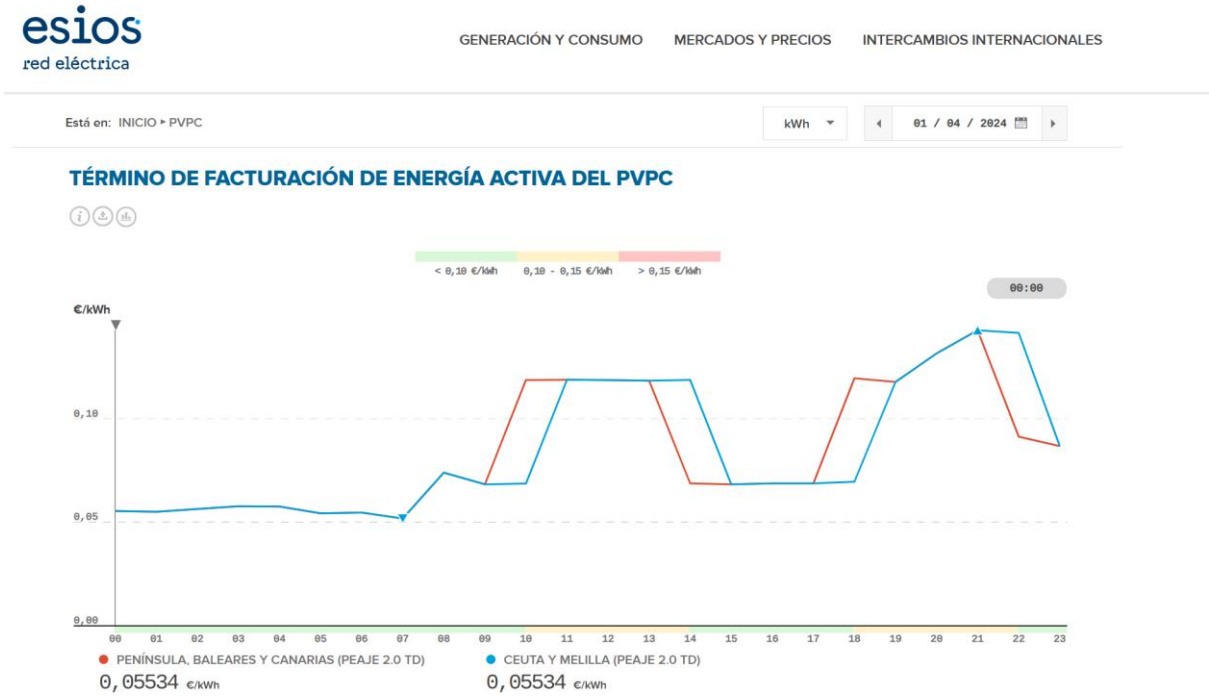


Figura 3-48. Consulta de precios de la tarifa PVPC en la página web de Red Eléctrica

Además, Red Eléctrica de España ofrece una API REST llamada REData API [5] que permite la consulta de una amplia gama de datos relacionados con la red eléctrica española. Entre la información que se puede consultar destacan demanda en tiempo real, estructura de la generación eléctrica desglosada por tecnologías (solar, eólica, nuclear, etc), emisiones de CO2, o lo que más nos concierne, los precios de la tarifa PVPC. En la figura 3-49 se muestra un ejemplo de consulta a dicha API REST para obtener los precios de la tarifa PVPC para las 24 horas del día. Los precios proporcionados por el servicio se dan en €/MWh:

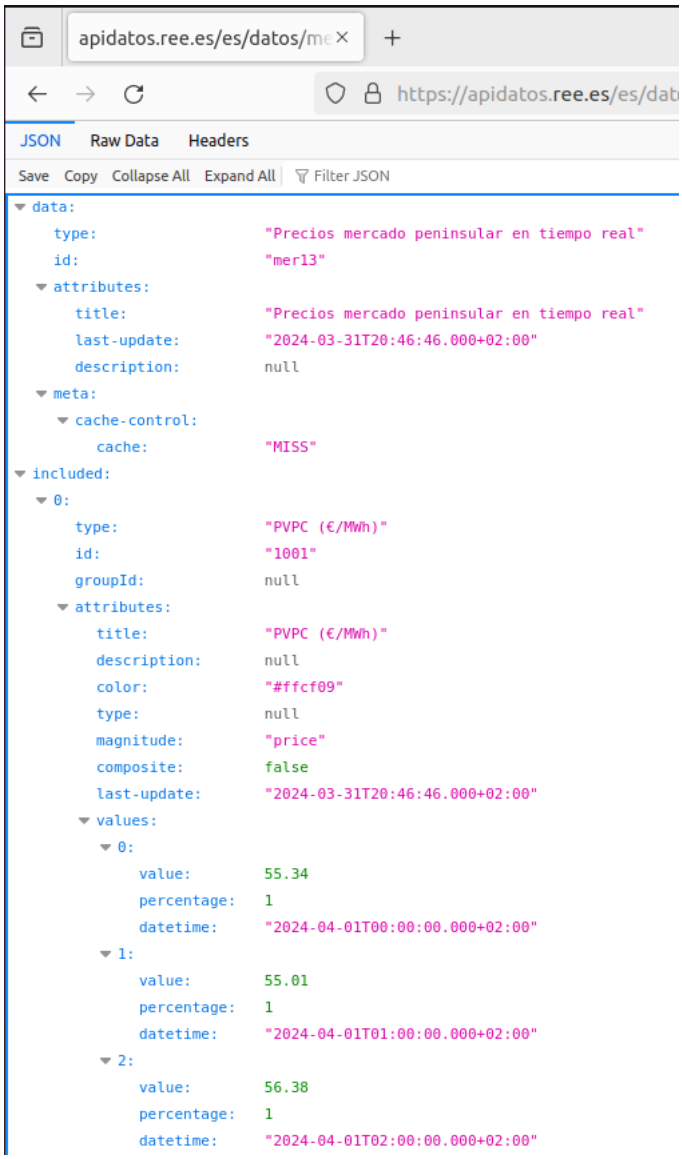


Figura 3-49. Consulta de precios de la tarifa PVPC en REData API

La URL utilizada en dicha consulta es la siguiente: https://apidatos.ree.es/es/datos/mercados/precios-mercados-tiempo-real?start_date=2024-4-1T00:00&end_date=2024-4-1T23:59&time_trunc=hour&geo_trunc=electric_system&geo_limit=peninsular&geo_ids=8741 .

Para empezar, el recurso que queremos consultar se corresponde con la ruta **/es/datos/mercados/precios-mercados-tiempo-real**, que es la que nos proporciona la información de precios de la tarifa PVPC en tiempo real. Posteriormente, en el *query string* se deben proporcionar una serie de parámetros mostrados en la tabla 3-3:

Parámetro	Significado
start_date	Fecha y hora de inicio en formato ISO 8601
end_date	Fecha y hora de fin en formato ISO 8601
time_trunc	Forma temporal de agrupar los datos. Para consultar la información de precios por hora su valor debe ser hour.

geo_trunc	Alcance geográfico de los datos consultados. Por ahora el único valor permitido es electric_system
geo_limit	Sistema eléctrico del cual se quieren obtener los datos
geo_ids	Identificador para obtener los datos de una región o comunidad autónoma específica. El valor 8741 se corresponde con la Península Ibérica

Tabla 3-3. Parámetros para la consulta de precios PVPC en REData API

De forma periódica y dinámica, podemos usar NodeRED para que haga consultas a la API REST de Red Eléctrica y así obtener los precios de la tarifa PVPC del día actual modificando los parámetros de fechas mostrados en la tabla 3-3. Las figuras 3-50 y 3-51 muestran el flujo NodeRED encargado de realizar la consulta al servicio REST de Red Eléctrica y mostrar los precios del día actual en una gráfica de barras:

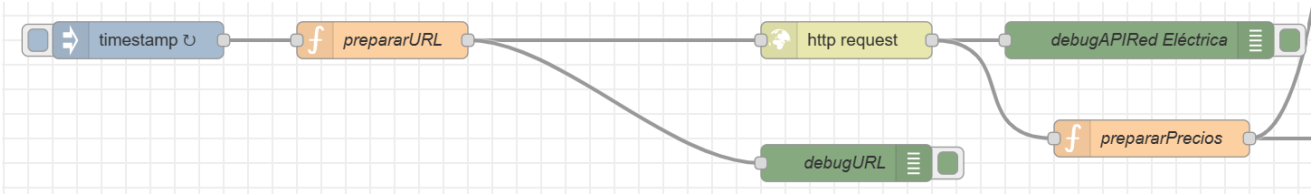


Figura 3-50. Flujo NodeRED encargado de la consulta de precios PVPC en REData API. Parte 1

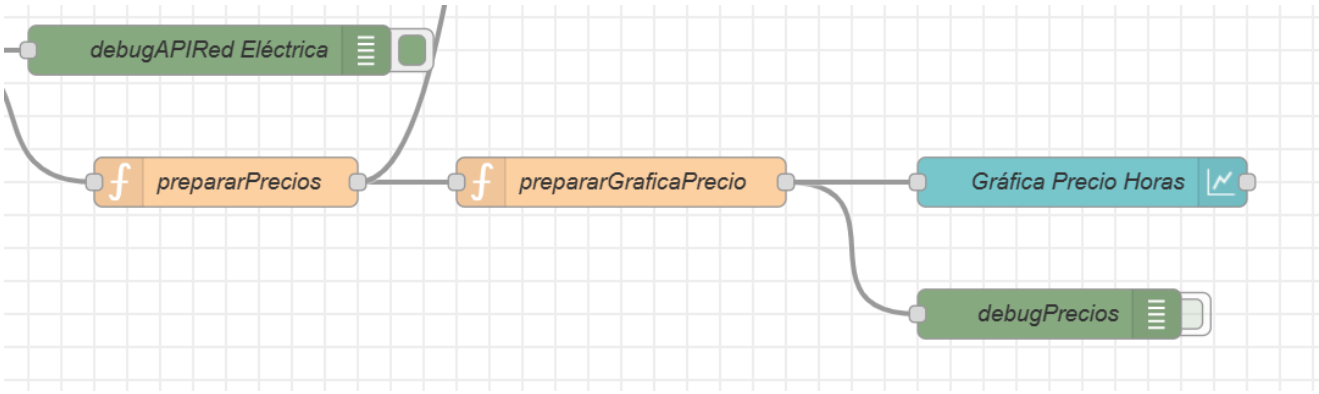


Figura 3-51. Flujo NodeRED encargado de la consulta de precios PVPC en REData API. Parte 2

En la función **prepararURL** construimos de forma dinámica la URL para consultar la información de precios PVPC para las 24 horas del día actual, como puede verse en la figura 3-52:

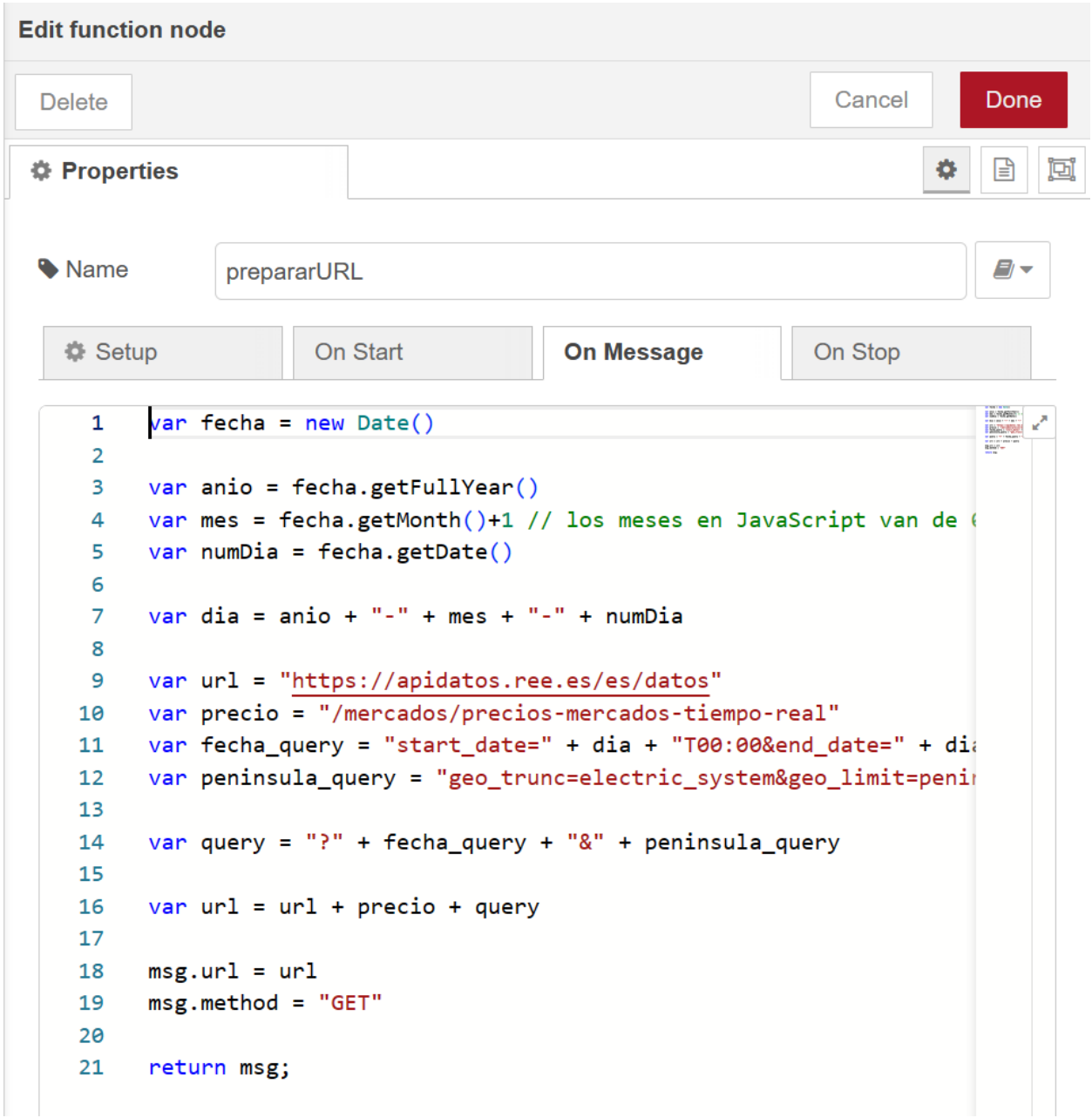


Figura 3-52. Código de la función prepararURL

Como podía verse en los resultados de la consulta mostrados en la figura 3-49, los atributos del objeto JSON de la respuesta que nos proporcionan la información de precios son: **datos.included[0].attributes.values[i]**, siendo el valor i-ésimo el correspondiente al del tramo de las 00:00-00:59, 01:00-01:59, … hasta las 23:00-23:59. La función **prepararPrecios** mostrada en la figura 3-53 se encarga de construir una lista más sencilla con los tramos y los precios de cada tramo. En este caso, el código JavaScript vuelve a separarse en dos funciones (**prepararPrecios** y **prepararGráficaPrecio**) ya que la salida de la función prepararPrecios también se usa para calcular el coste del consumo eléctrico, al igual que ocurría con los datos de consumo por horas.

Edit function node

Delete Cancel Done

Properties

Name prepararPrecios

Setup On Start On Message On Stop

```
1   var datos_precios = msg.payload
2
3   var resultado = []
4
5   var precios = []
6   var tramos = []
7
8   for (var i = 0; i < 10; i++) {
9       precios.push(datos_precios.included[0].attributes.values[i]
10
11      tramos.push("0" + i + ":00-0" + i + ":59")
12  }
13
14  for (var i = 10; i < 24; i++) {
15      precios.push(datos_precios.included[0].attributes.values[i]
16
17      tramos.push(i + ":00-" + i + ":59")
18  }
19
20  msg.payload = [precios, tramos]
21
22  return msg;
```

Figura 3-53. Código de la función prepararPrecios

Lo que obtenemos en la interfaz de usuario es una gráfica de barras horizontal con la información de precios de la tarifa PVPC expresados en €/MWh para cada hora del día, como puede verse en la figura 3-54:

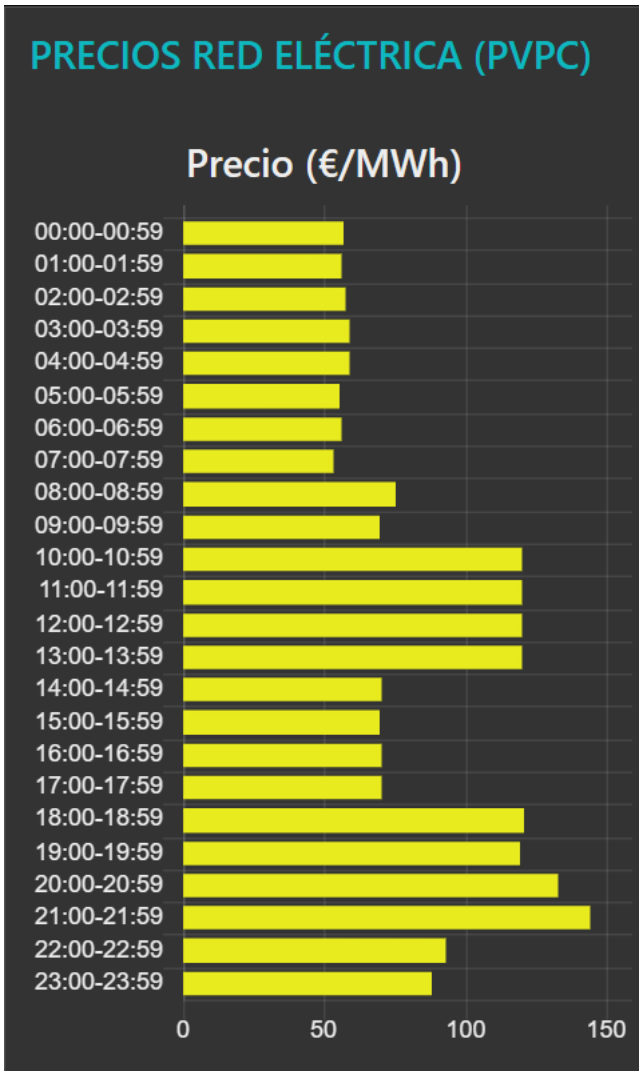


Figura 3-54. Representación gráfica de los precios de la tarifa PVPC para cada hora del día

Las interacciones entre NodeRED y el servicio REST de Red Eléctrica de España durante el proceso de consulta de precios de la electricidad se resumen en el diagrama de paso de mensajes mostrado en la figura 3-55:

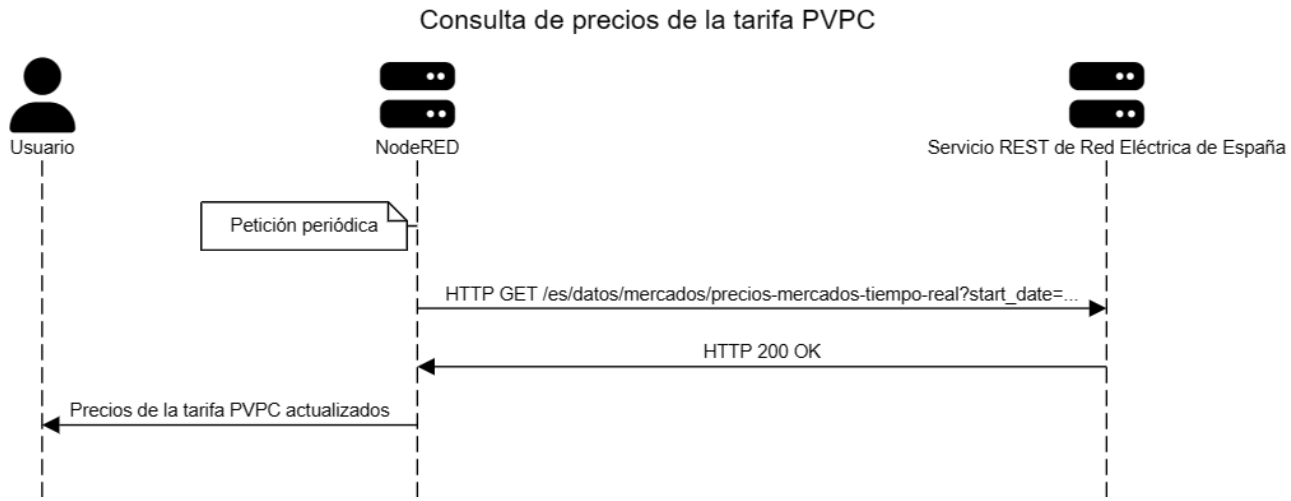


Figura 3-55. Diagrama de paso de mensajes para la consulta de precios de la tarifa PVPC

3.10 Cálculo de costes del consumo

Una vez conocidos los precios de la electricidad y los consumos que ha ido teniendo el usuario, podemos obtener los costes del consumo eléctrico. Para ello, simplemente tendríamos que multiplicar el consumo eléctrico (en Wh) por el precio de la electricidad (en €/Wh). Como se comentó en la Sección 3.9 (Consulta de precios de la electricidad), la tarifa PVPC establece un precio distinto para la electricidad según la hora del día, por lo tanto, debemos tener en cuenta este factor a la hora de calcular los costes energéticos.

Las figuras 3-56 y 3-57 muestran el flujo NodeRED encargado de calcular los costes del consumo eléctrico a partir de los datos de consumos por horas y de precios de la tarifa PVPC. Estos datos de consumo y de precios provienen de los flujos de las figuras 3-46 y 3-51, respectivamente. Por lo tanto, este flujo comienza su procesamiento una vez que se han obtenido los consumos por hora y se ha obtenido la información de precios desde el servicio REST de Red Eléctrica.

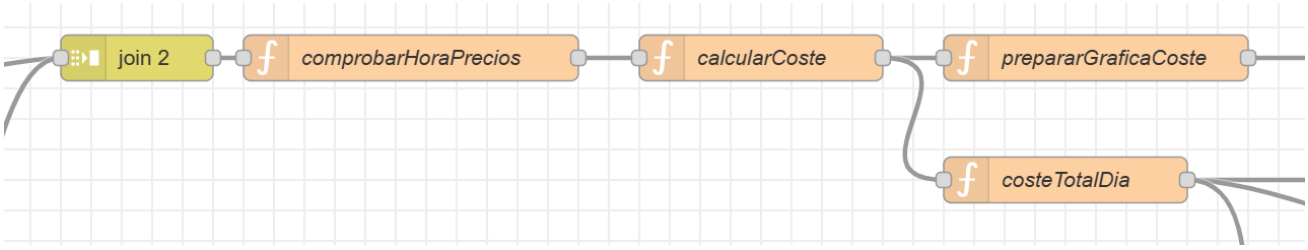


Figura 3-56. Flujo NodeRED encargado del cálculo de costes del consumo. Parte 1

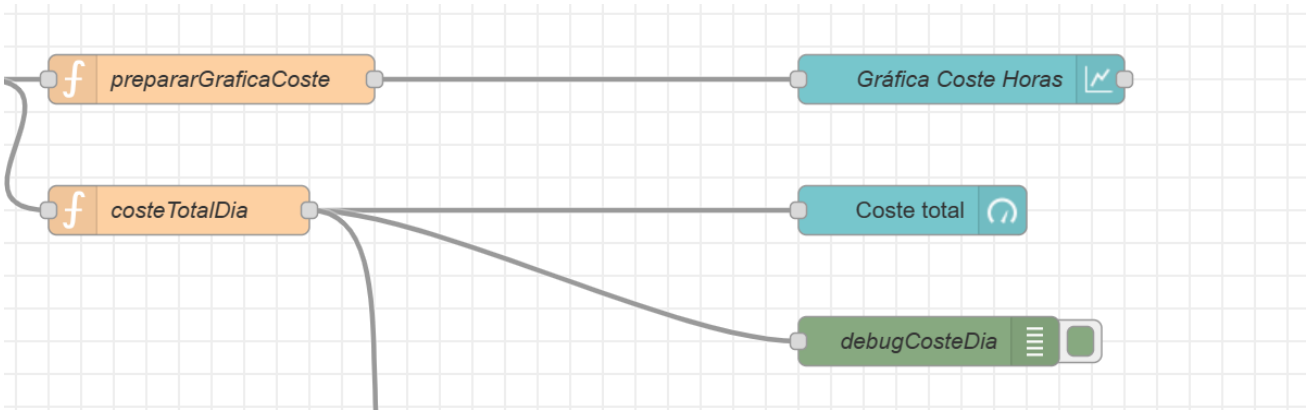


Figura 3-57. Flujo NodeRED encargado del cálculo de costes del consumo. Parte 2

La función **comprobarHoraPrecios** sirve para evitar que pasen mensajes por el flujo si la hora actual está entre las 23:55 y las 00:05 del día siguiente. De esta forma, dejamos un tiempo prudencial para que el servicio REST de Red Eléctrica actualice los nuevos precios de la electricidad, y así evitamos hacer cálculos con los precios del día anterior.

La función **calcularCoste** se encarga de calcular el coste de la energía consumida para cada una de las 24 horas del día. Como el precio es distinto según la hora en la que se haya consumido, multiplicamos los consumos de cada hora con su respectivo precio, como puede verse en la figura 3-58. Al estar los precios en €/MWh, se hace una conversión para pasarlos a €/Wh:

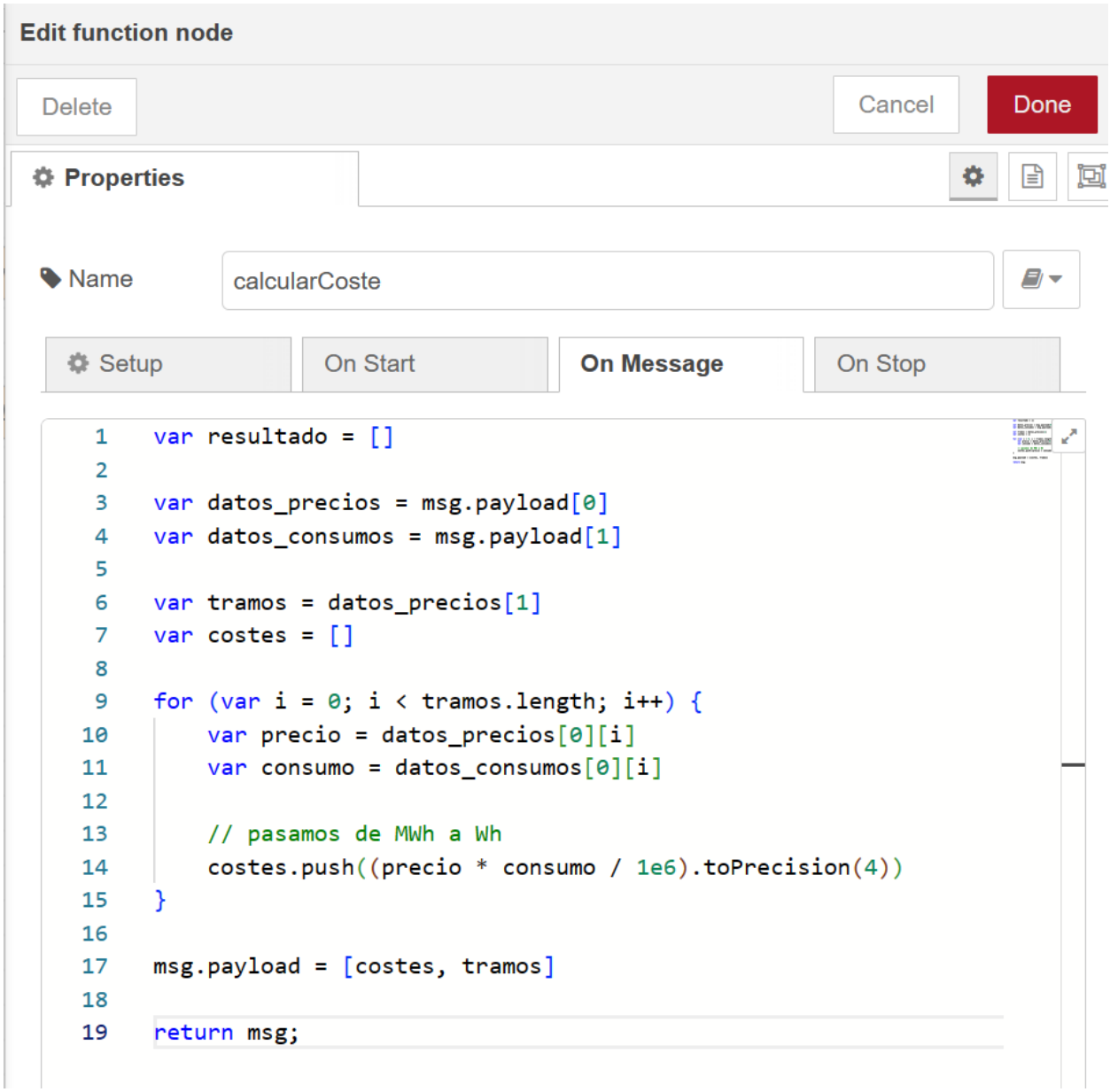


Figura 3-58. Código de la función calcularCoste

Una vez que se han calculado los costes del consumo, mediante la función **prepararGráficaCoste** organizamos los datos de costes por hora de tal forma que puedan ser representados en un diagrama de barras horizontal. Además, con la función **costeTotalDia** realizamos la suma de todos los costes para obtener el coste total del día. Tanto el diagrama de barras con el coste de cada hora, como el coste total del día se muestran en la figura 3-59:

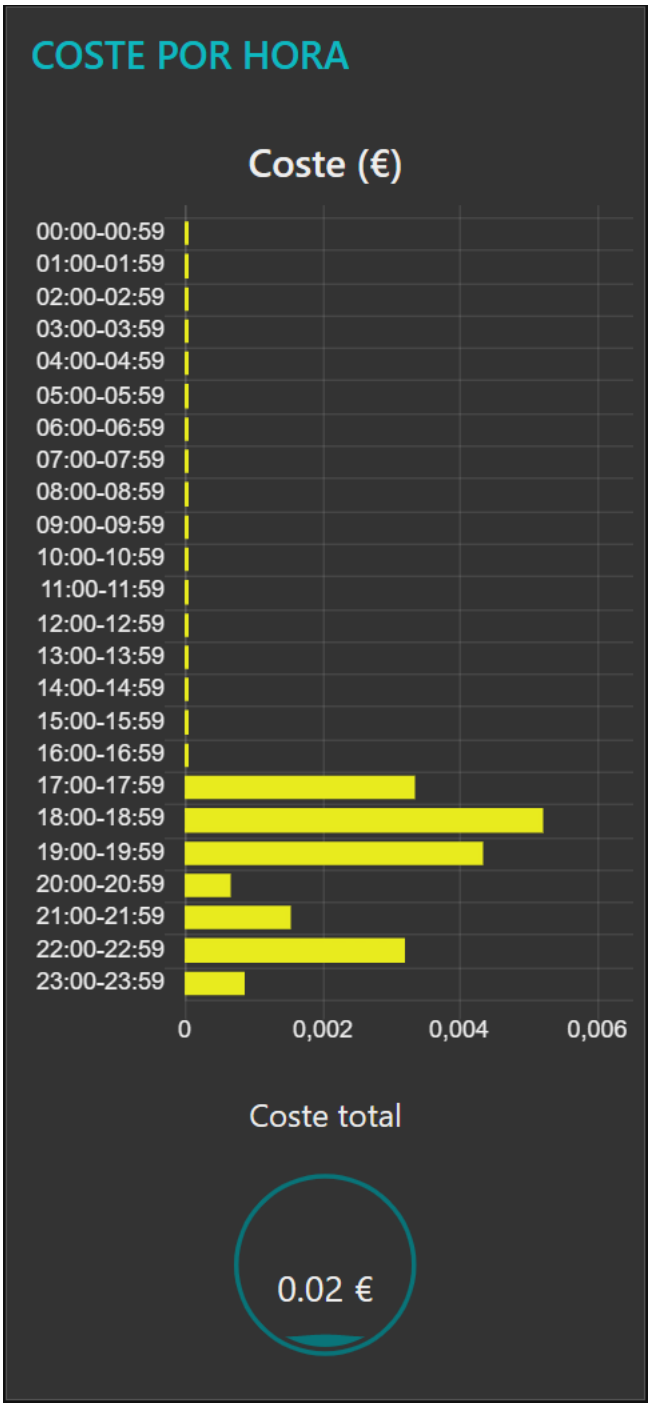


Figura 3-59. Representación gráfica del coste por hora y del coste total del día

3.11 Históricos de consumo y costes

Gracias a los datos de consumo y costes que se han ido generando, puede resultar de interés generar registros históricos para que el usuario pueda llevar un seguimiento del consumo que ha ido teniendo y de los costes que le han supuesto esos consumos. Por simplicidad y para evitar saturar la base de datos con demasiada información, se almacenarán los consumos y costes totales del día. Es decir, cada día que pase habrá un registro nuevo en el histórico de consumo y otro en el histórico de costes.

3.11.1 Almacenamiento de datos de consumo y costes

Como se explicará en el Capítulo 4 (Servicio REST de consumo y costes), cada vez que se almacena una nueva medida de potencia, automáticamente se recalcula el consumo total del día con esa nueva información y se almacena en la base de datos. Sin embargo, en el caso del coste es necesario conocer los precios. Como decisión de diseño se ha decidido que el cálculo de los costes se realice desde NodeRED, y que posteriormente se almacene esa información mediante una ruta del servicio REST de consumo y costes. El flujo encargado de esta última tarea se muestra en la figura 3-60:

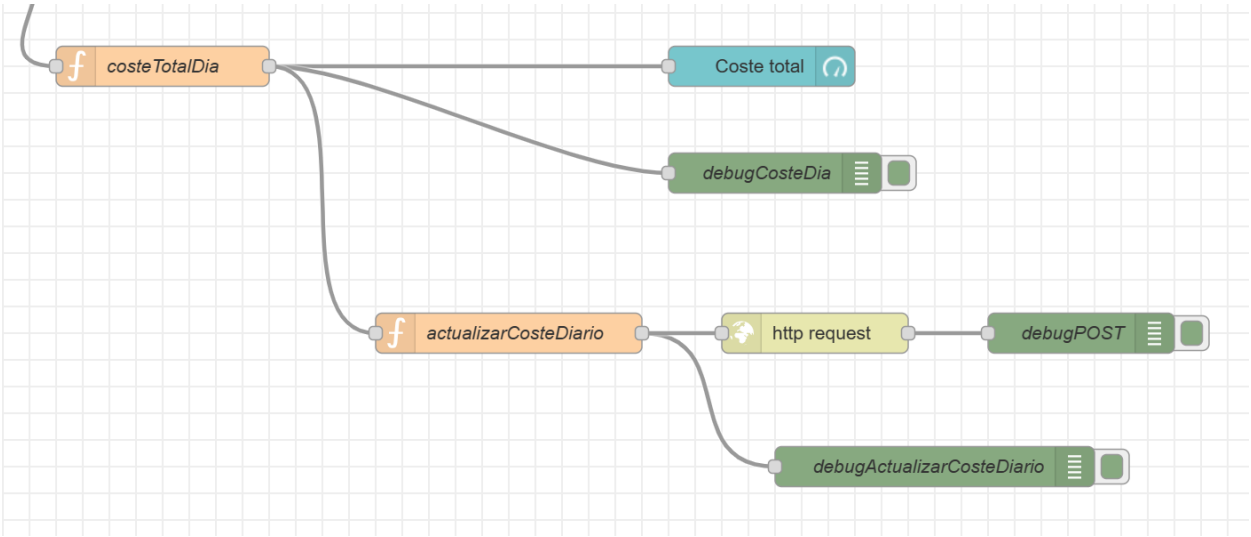


Figura 3-60. Flujo NodeRED encargado de almacenar los datos de coste diario

A partir del coste total del día proporcionado por la función **costeTotalDia,** la función **actualizarCosteDiario** construye un mensaje JSON con la fecha del día actual y el coste total. Este mensaje JSON se le pasará al nodo que se encarga de hacer la petición HTTP POST a la ruta **/shelly-plug/costes** del servicio REST de consumo y costes, para que almacene esta información en la base de datos. En el caso de que ya exista un valor de coste para ese día, en vez de crearse un nuevo registro se actualizará su valor en la base de datos.

3.11.2 Representación de los históricos

El servicio REST implementado en este proyecto también nos proporciona rutas para obtener los históricos de consumos y de costes almacenados en la base de datos. En el caso del histórico de consumo, el flujo mostrado en la figura 3-61 se encarga de obtener este histórico y representarlo en la interfaz de usuario mediante una gráfica de barras:

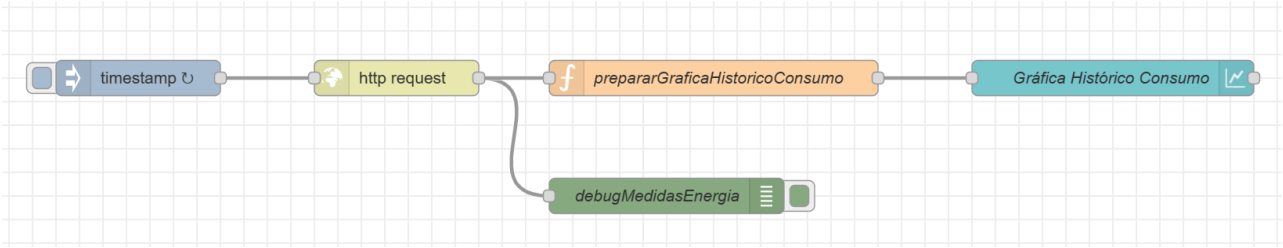


Figura 3-61. Flujo NodeRED encargado de obtener el histórico de consumo

En este flujo se hace una petición HTTP GET a la ruta **/shelly-plug/consumo/medidas** para obtener las medidas de consumo diario almacenadas en la base de datos. A continuación, se utiliza la función **prepararGraficaHistoricoConsumo** para preparar los datos para que puedan ser representados en la gráfica

de barras que puede verse en la figura 3-62:

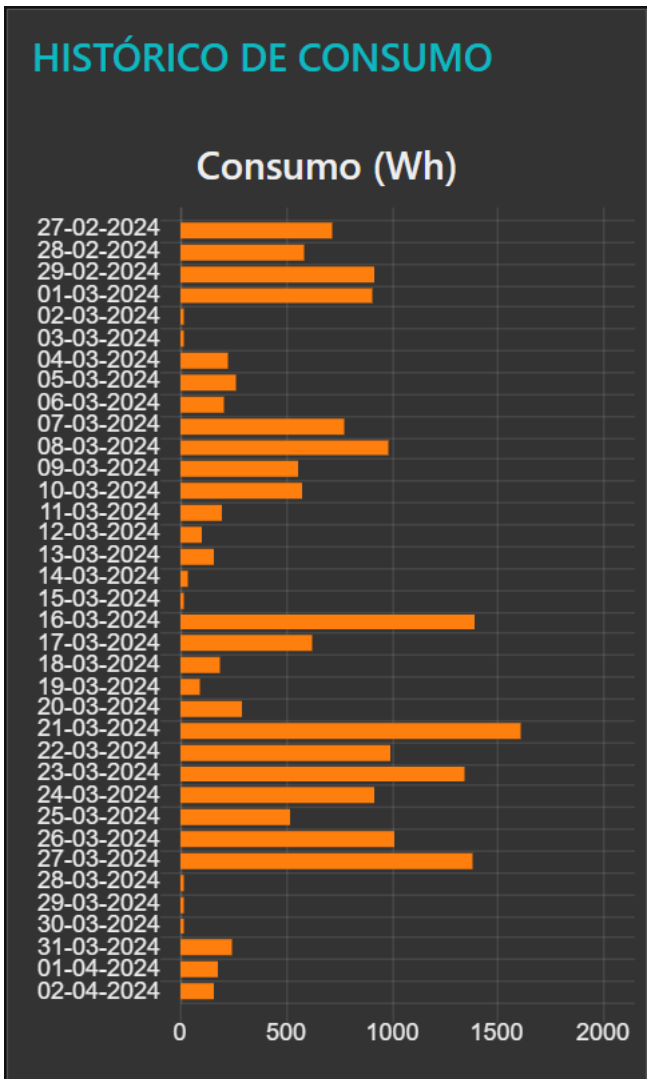


Figura 3-62. Representación gráfica del histórico de consumo

De forma análoga, mediante peticiones HTTP GET a la ruta **/shelly-plug/costes**, el flujo de la figura 3-63 nos permite obtener el histórico de costes y representarlo en la gráfica de barras de la figura 3-64:

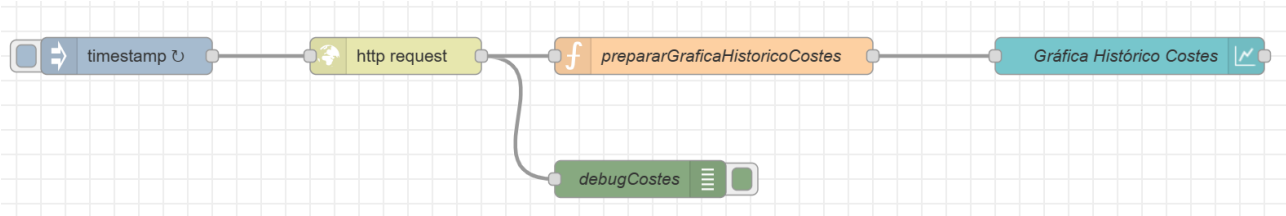


Figura 3-63. Flujo NodeRED encargado de obtener el histórico de costes

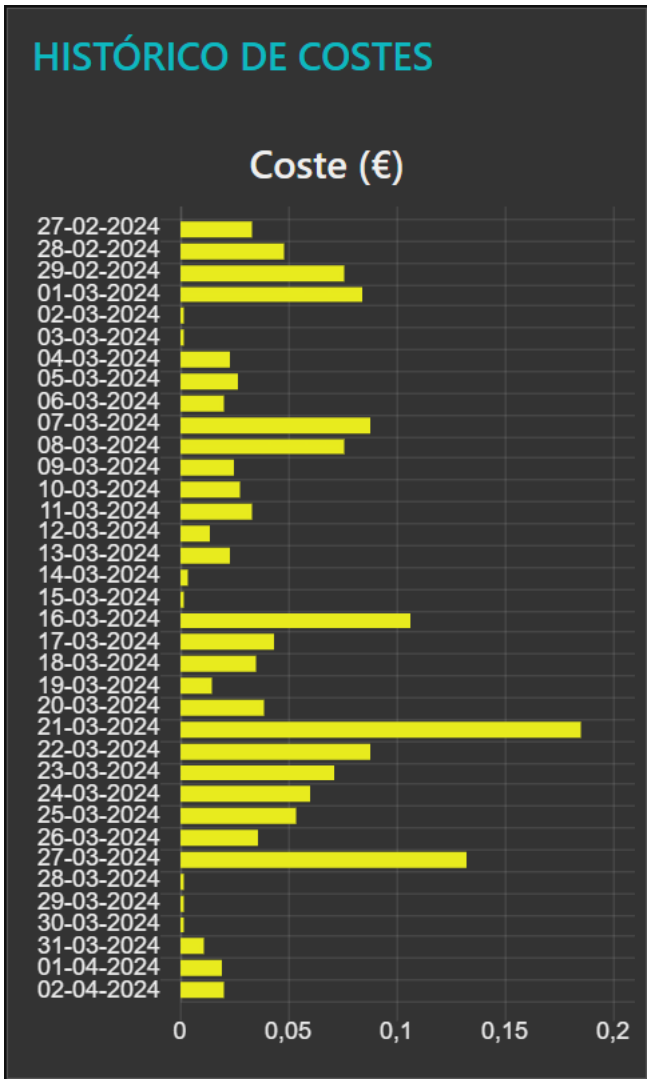


Figura 3-64. Representación gráfica del histórico de costes

De forma resumida, los mensajes que se intercambian para la consulta de los históricos de consumo y costes pueden verse en la figura 3-65:

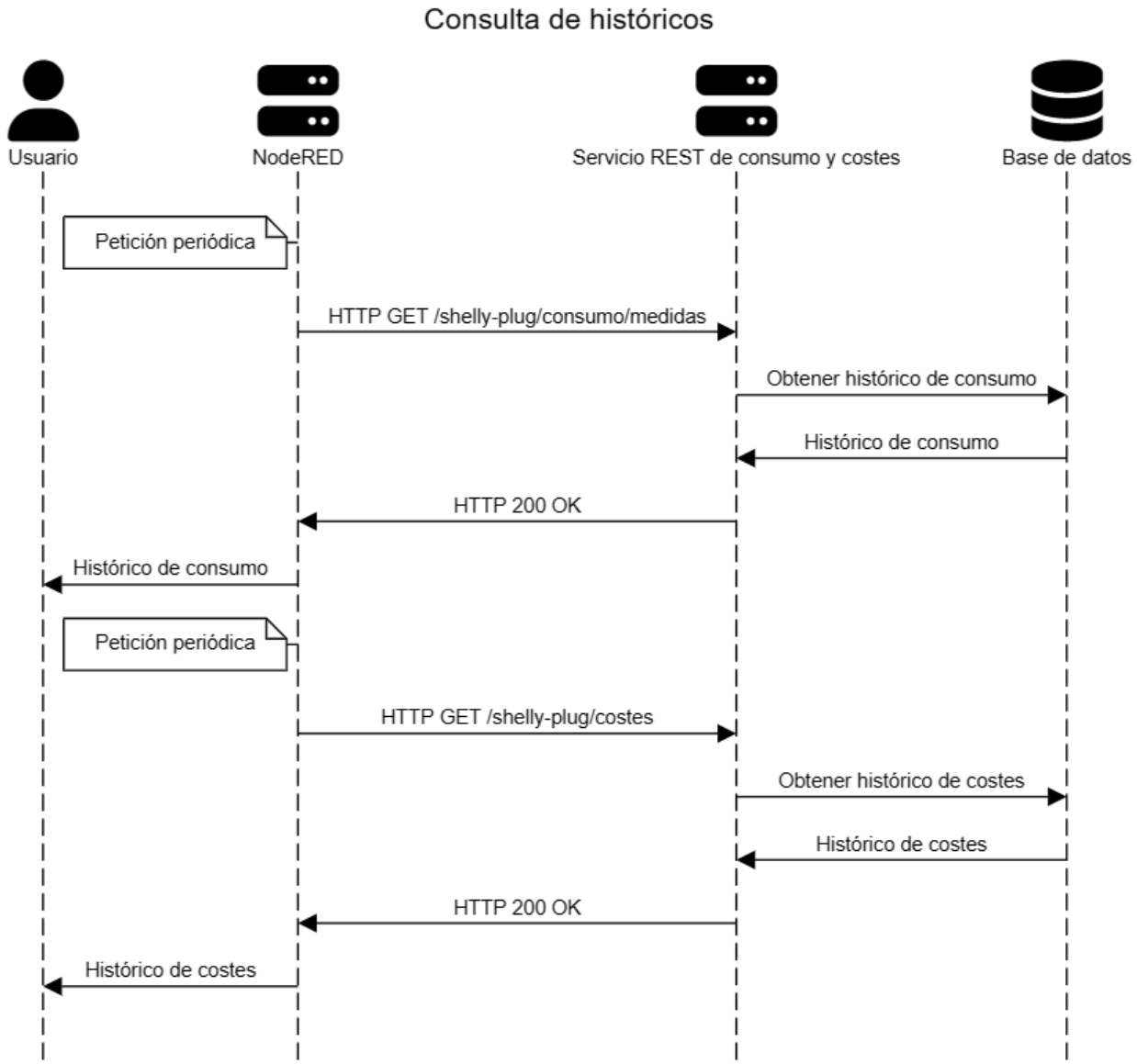


Figura 3-65. Diagrama de paso de mensajes para la consulta de históricos

3.12 Gestión de datos almacenados

En la Sección 3.5 ya se comentó que las medidas de potencia son borradas de la base de datos tras finalizar el día, puesto que dejan de tener utilidad una vez que se ha calculado y almacenado el consumo y costes totales de ese día. No obstante, los flujos de la figura 3-66 sirven para que el usuario pueda borrar manualmente tanto las medidas de potencia como los históricos de consumo y costes almacenados en la base de datos:

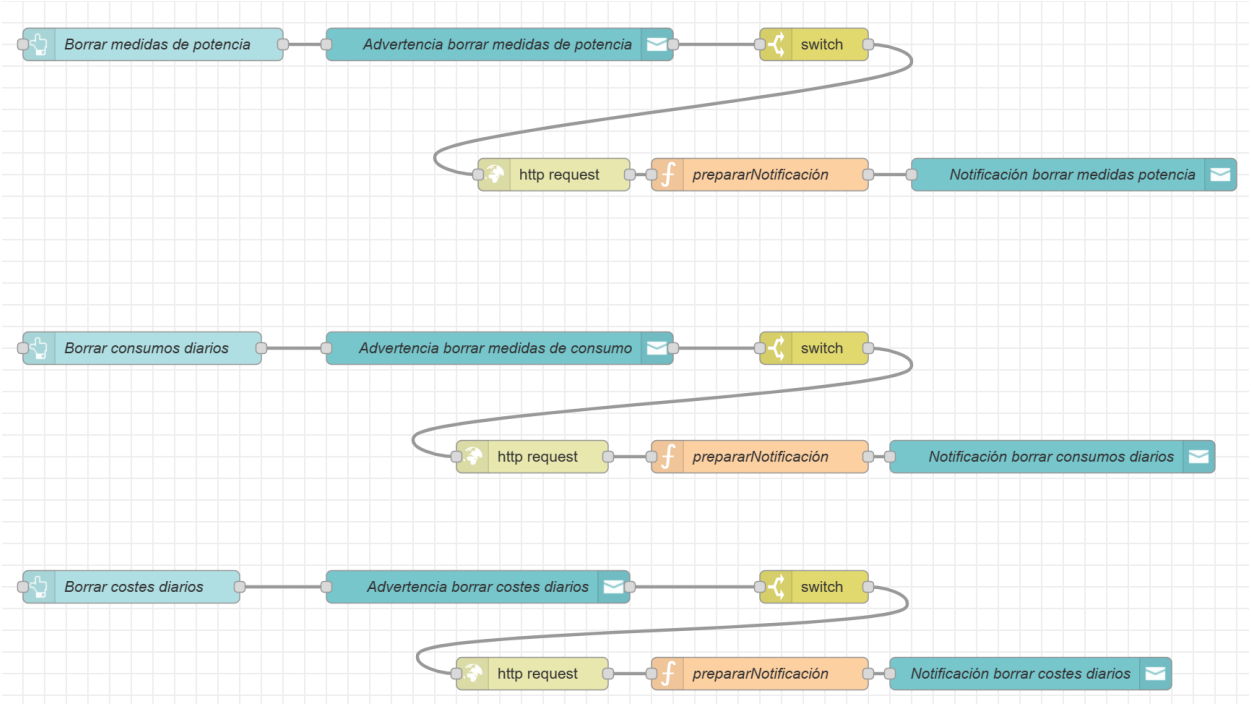


Figura 3-66. Flujos NodeRED para la gestión de datos del sistema domótico

Estos flujos crean los botones mostrados en la figura 3-67. Al pulsarlos, estos botones provocarán el envío de peticiones HTTP DELETE a las siguientes rutas:

- Borrar medidas de potencia: **/shelly-plug/potencia/medidas**
- Borrar consumos diarios: **/shelly-plug/consumo/medidas**
- Borrar costes diarios: **/shelly-plug/costes**

Figura 3-67. Botones para el borrado de datos de la base de datos

De forma preventiva, antes de mandar la petición de borrado, se le muestra una notificación al usuario para confirmar que realmente desea borrar los datos correspondientes de la base de datos, como puede verse en la figura 3-68:

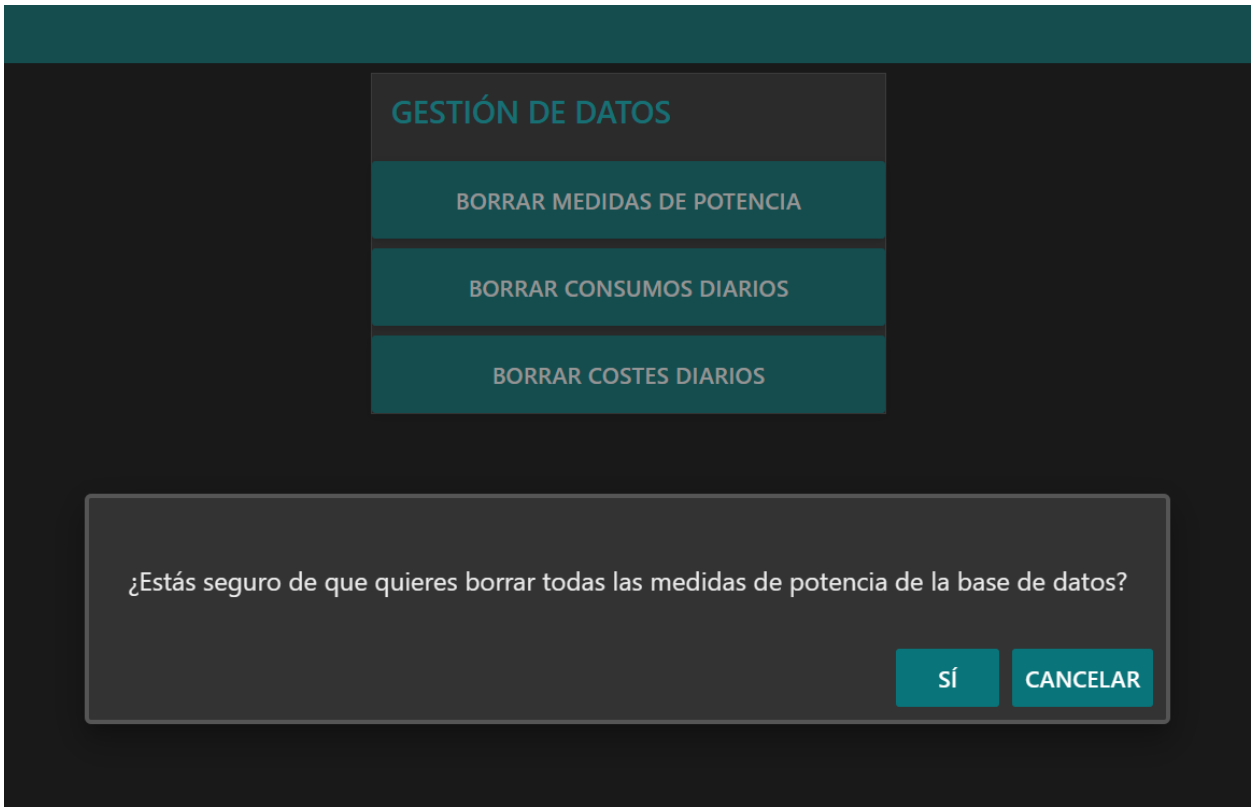


Figura 3-68. Notificación de confirmación de borrado de datos

Tras el borrado de datos, al usuario se le informa de que los datos han sido borrados de forma satisfactoria mediante la notificación mostrada en la figura 3-69:

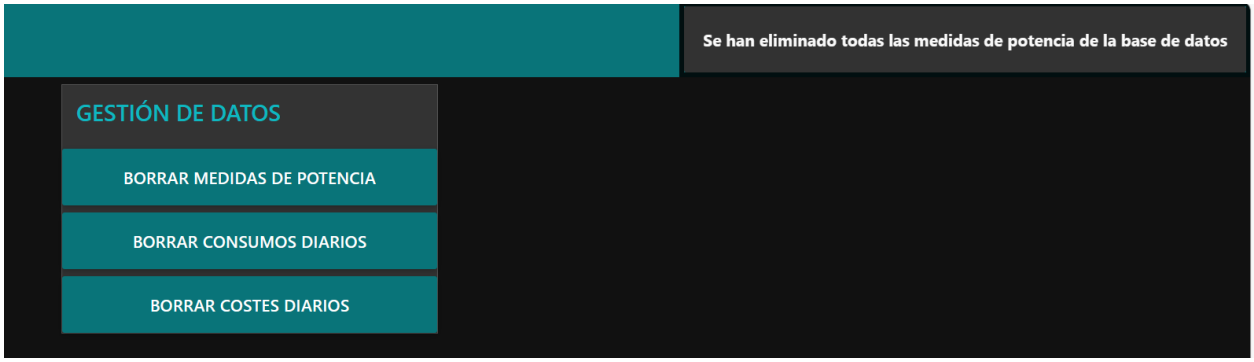


Figura 3-69. Notificación para informar de que los datos han sido borrados con éxito

3.13 Interfaz de usuario

3.13.1 Creación de la interfaz

Para la creación de la interfaz de usuario, se ha instalado el módulo **node-red-dashboard** [43]. Este módulo proporciona una serie de nodos de interfaz gráfica para la creación de botones, representación de series temporales, gráficas de barras, contadores, interruptores, envío de notificaciones, etc. Para instalar este módulo basta con ir a las opciones de NodeRED, seleccionar la opción **Manage Palette**, buscar el módulo e instalarlo. Una vez instalado, los nodos de este módulo estarán disponibles en el editor, como puede verse en las figuras 3-70 y 3-71:

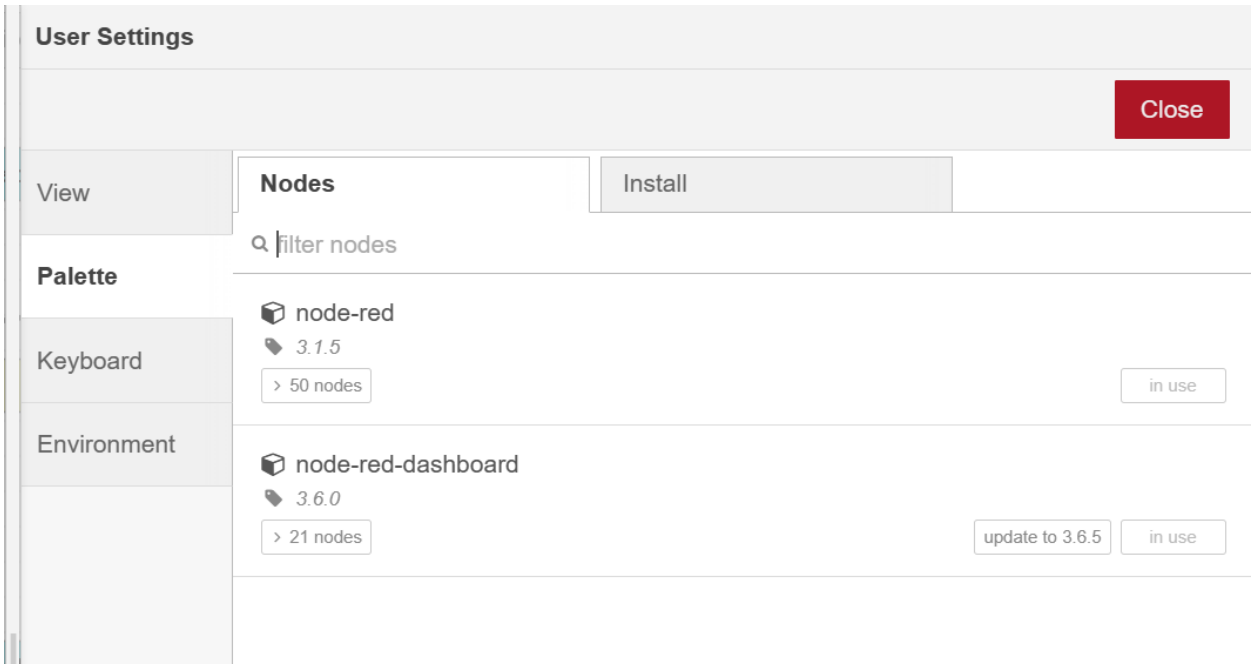


Figura 3-70. Opción Manage Palette de NodeRED

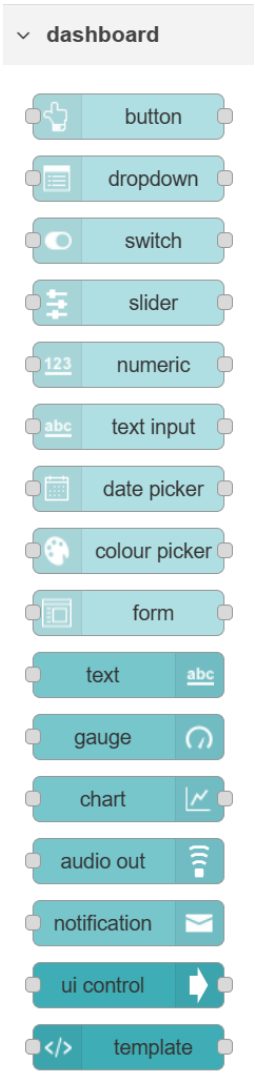


Figura 3-71. Nodos del módulo node-red-dashboard

Estos nodos se han utilizado para poder visualizar datos de forma muy sencilla y directa desde los propios flujos de NodeRED, evitando tener que desarrollar una interfaz externa.

El módulo de dashboard permite organizar las páginas de la interfaz en lo que se denominan **groups** (grupos). Cada uno de los nodos de interfaz gráfica se pueden asociar a un grupo, de tal forma que luego se pueda editar la disposición de los elementos en la interfaz de usuario, como puede verse en la figura 3-72:

Figura 3-72. Herramienta para la disposición de elementos en la interfaz de usuario

3.13.2 Página principal y navegación por la interfaz

Al acceder a la interfaz de usuario, la página principal que se muestra es la de **monitorización de consumo**, la cual se explica en la Sección 3.13.3 (Monitorización). El aspecto de esta página principal puede verse en la figura 3-73:

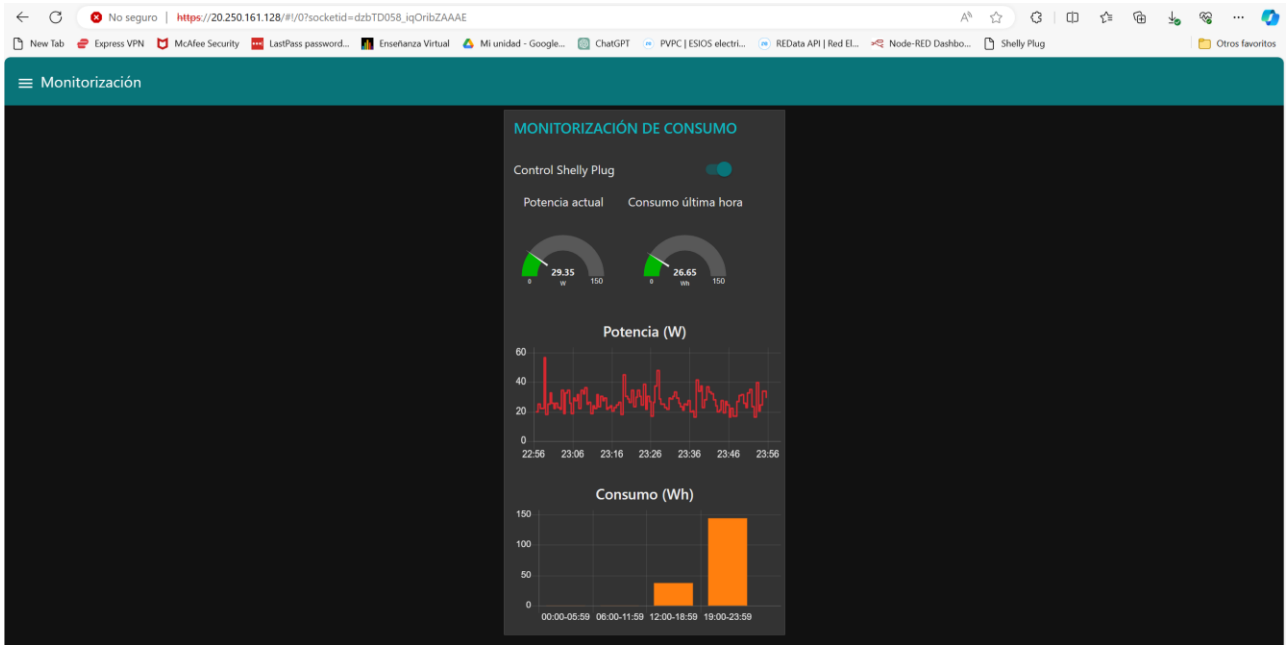


Figura 3-73. Página principal de la interfaz de usuario

Para navegar por las diferentes páginas de la interfaz, en la esquina superior izquierda hay un botón con tres rayas que permite seleccionar la página que quiere verse en la interfaz, como puede verse en la figura 3-74:

Figura 3-74. Navegación por la interfaz de usuario

Las páginas que forman la interfaz de usuario y que se explicarán a continuación son las siguientes:

- Monitorización (página principal)
- Consumo diario
- Histórico consumo
- Precio electricidad
- Coste diario
- Histórico costes
- Gestión datos

3.13.3 Monitorización

La página de monitorización de consumo puede verse en la figura 3-75:

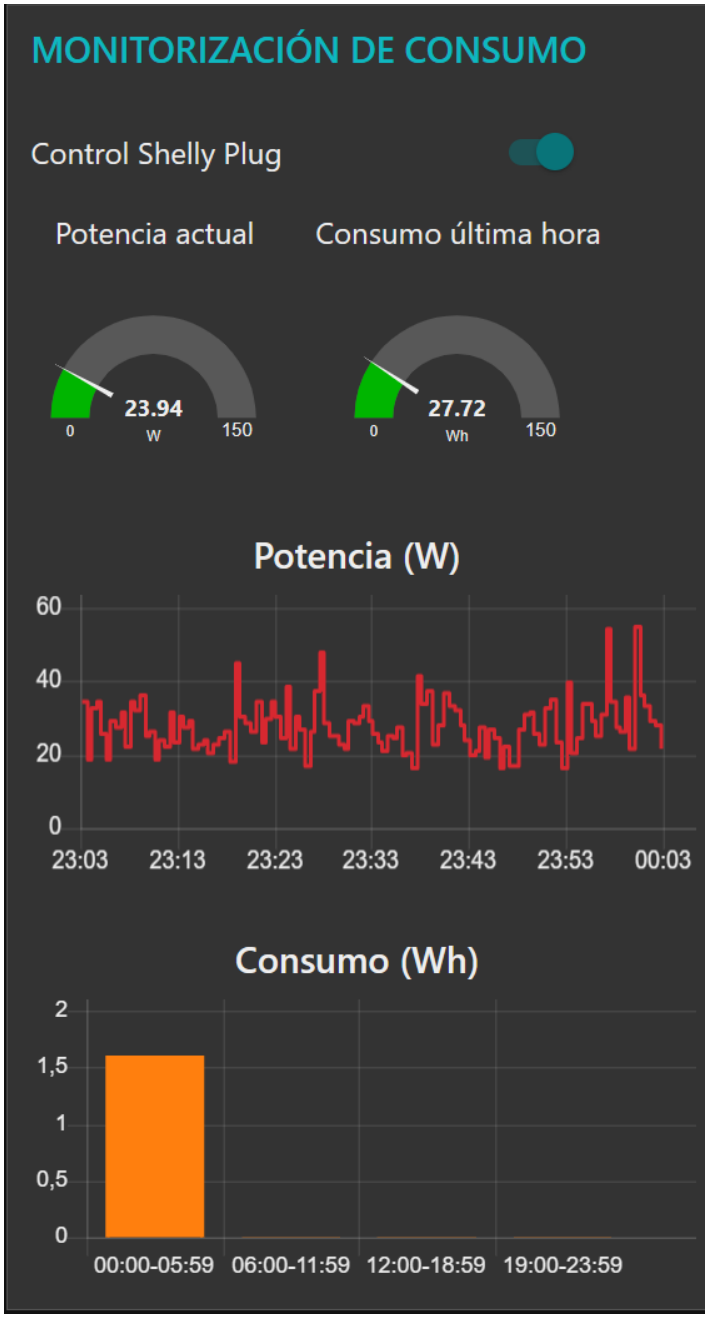


Figura 3-75. Monitorización de consumo

Esta página permite el control remoto del apagado y encendido del enchufe, proporciona información sobre la potencia actual medida por el enchufe, la potencia consumida a lo largo de la última hora, el consumo que se ha producido en la última hora, y el consumo del día desglosado en cuatro franjas horarias.

3.13.4 Consumo diario

La página de consumo diario se muestra en la figura 3-76:

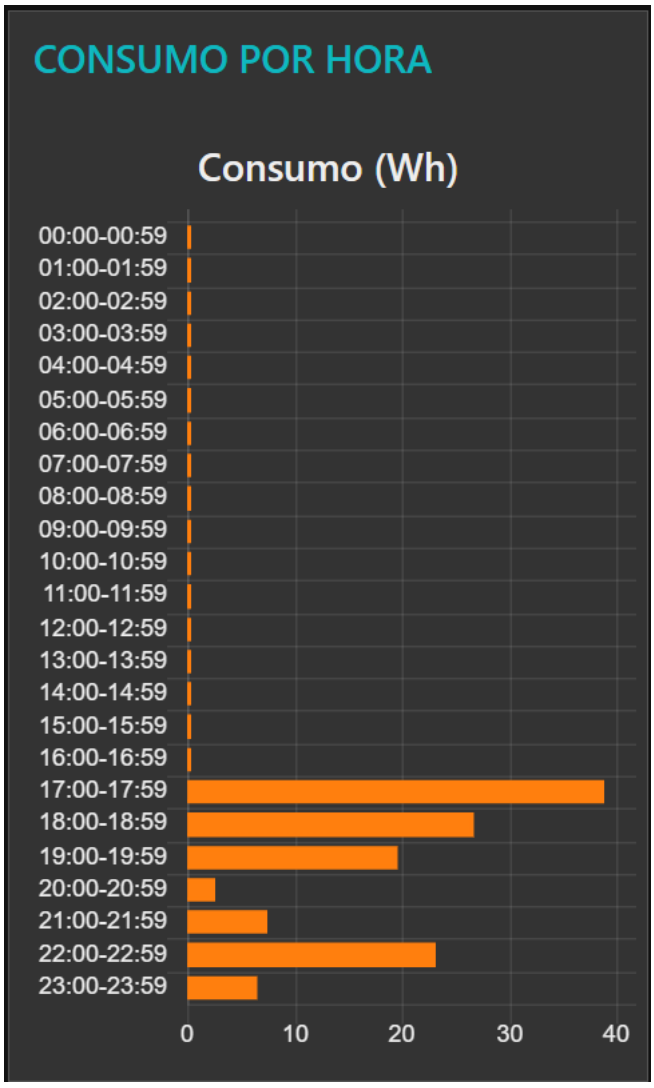


Figura 3-76. Consumo diario

En ella se muestra el consumo que se ha producido en cada una de las 24 horas del día.

3.13.5 Histórico consumo

La página con el histórico de consumo puede verse en la figura 3-77:

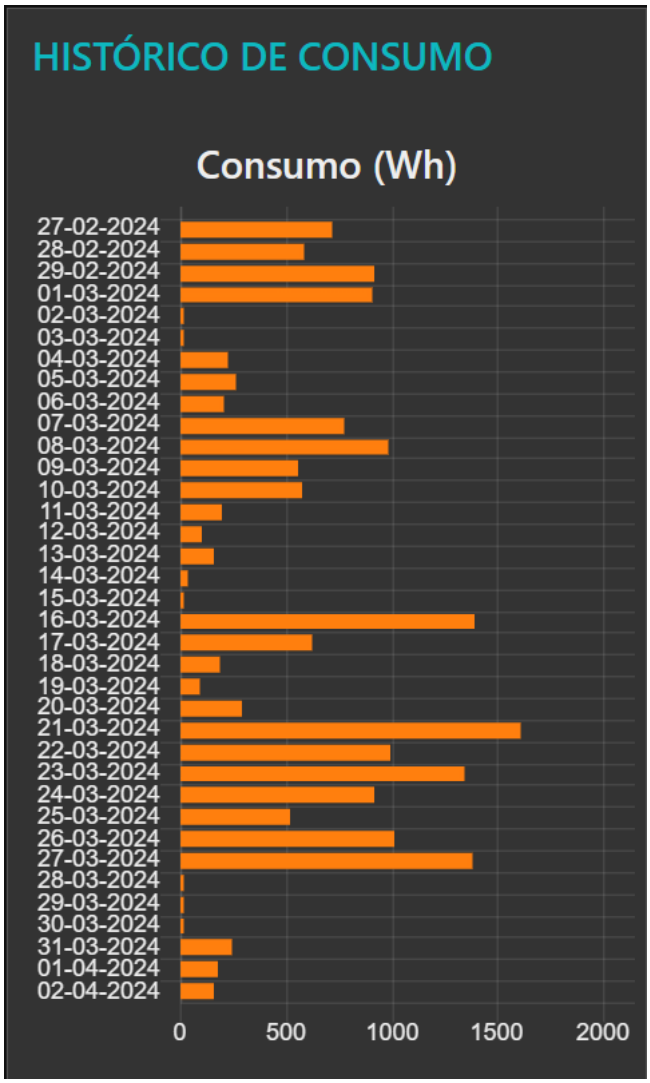


Figura 3-77. Histórico de consumo

En esta página se muestran los consumos totales diarios que hay registrados en la base de datos.

3.13.6 Precio electricidad

La página con la información de precios de la electricidad se muestra en la figura 3-78:

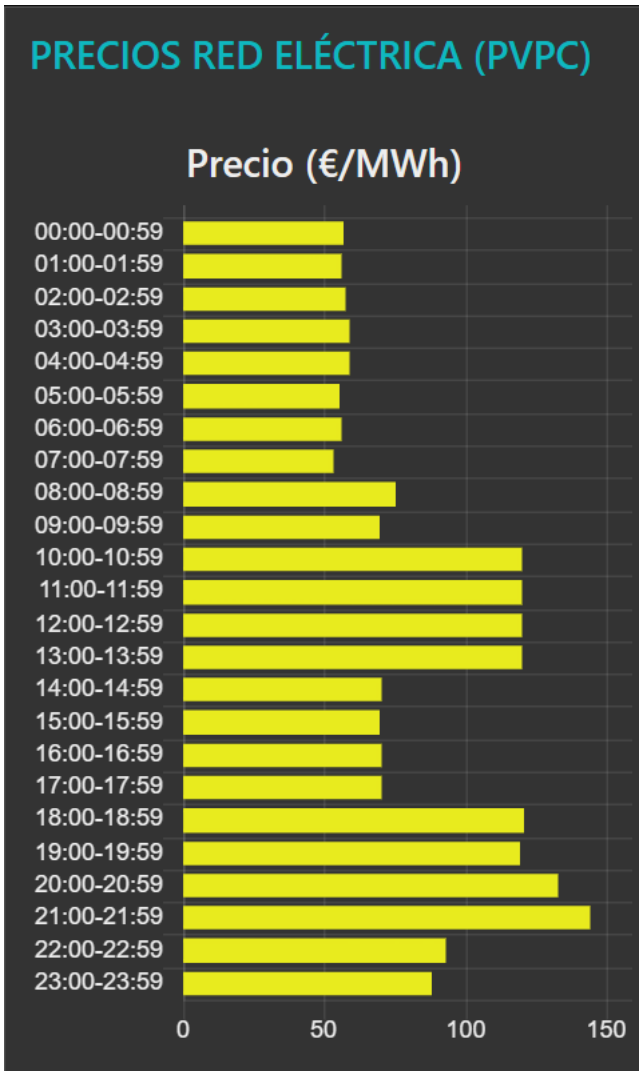


Figura 3-78. Precios Red Eléctrica (PVPC)

En esta página se muestran los precios de la electricidad de la tarifa PVPC para cada hora del día.

3.13.7 Coste diario

La página de coste diario se muestra en la figura 3-79:

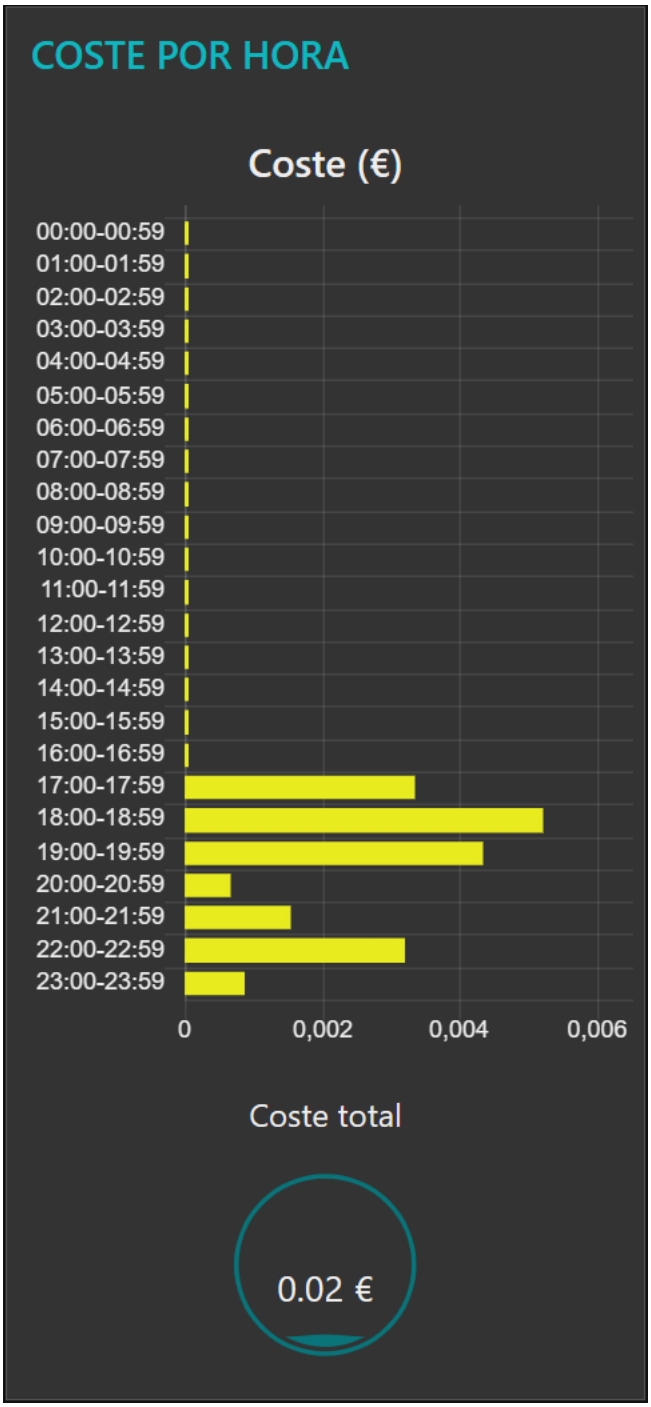


Figura 3-79. Coste por hora

En ella se muestran los costes del consumo eléctrico para cada hora del día y el coste total.

3.13.8 Histórico costes

La figura 3-80 muestra la página con el histórico de costes:

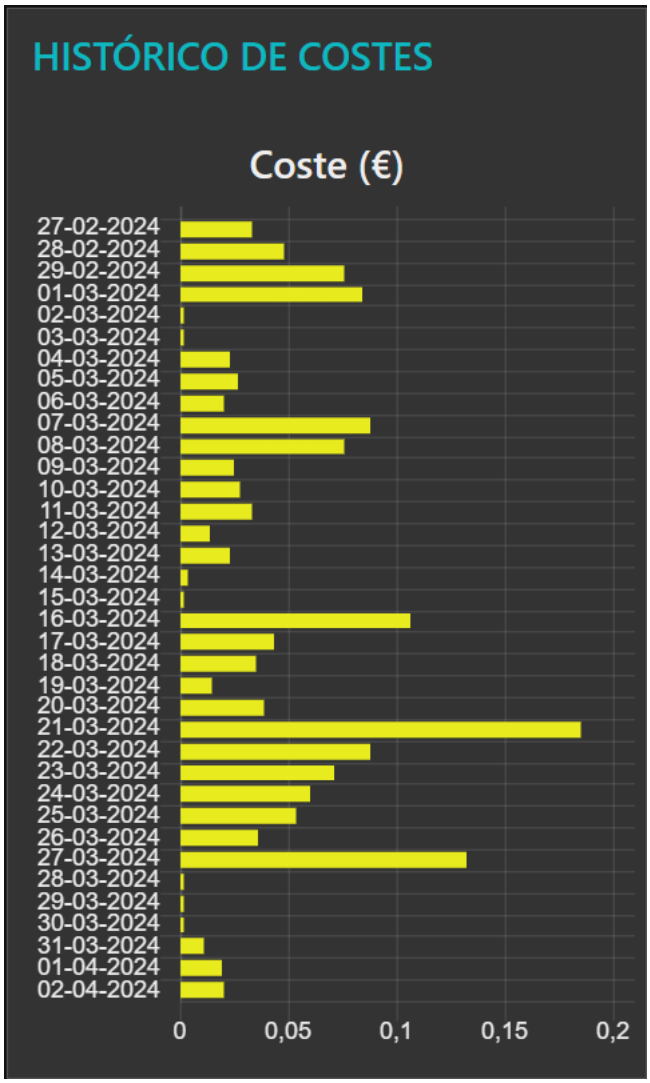


Figura 3-80. Histórico de costes

En esta página se muestran los costes totales diarios que hay registrados en la base de datos.

3.13.9 Gestión datos

En la figura 3-81 puede verse la página de gestión de datos:

Figura 3-81. Gestión de datos

Los botones de esta página permiten el borrado de medidas de potencia, consumos diarios, y costes diarios de la base de datos. La interacción con estos botones se muestra en la Sección 3.12 (Gestión de datos almacenados).

4 SERVICIO REST DE CONSUMO Y COSTES

Cada guerra es una destrucción del espíritu humano.

- Henry Miller -

De forma conjunta a la implementación de los aspectos de control y monitorización de consumo eléctrico realizados mediante NodeRED, se ha desarrollado un servicio REST para facilitar el almacenamiento de las medidas de potencia proporcionadas por el enchufe inteligente. A partir de estos datos de potencia, este servicio REST es capaz de realizar una estimación del consumo eléctrico que se ha producido en un rango horario específico. Además, conforme se van almacenando medidas de potencia, el servicio REST va creando y actualizando registros de consumo de forma automática, lo cual facilita la creación de un histórico de consumo diario en la base de datos. Por otro lado, este servicio también permite almacenar los costes asociados al consumo energético, creando así un histórico de costes.

De forma automática, desde NodeRED se hacen peticiones HTTP a las rutas de este servicio REST para monitorizar la potencia consumida en el enchufe, hacer un seguimiento del consumo eléctrico, y generar históricos de consumos y costes, así como recuperar esta información de la base de datos para poder visualizarla en la interfaz de usuario.

Para la creación de este servicio, se ha desarrollado una aplicación Flask en la que se han implementado todas las funcionalidades comentadas anteriormente. Esta aplicación se ha encapsulado en una imagen Docker personalizada para poder desplegar el servicio junto al resto de contenedores del sistema utilizando Docker-Compose.

4.1 Despliegue y configuración de la base de datos

Para el almacenamiento y gestión de los datos generados por el sistema domótico, el servicio REST de consumo y costes se apoya en el uso de una base de datos PostgreSQL.

4.1.1 Despliegue de la base de datos

En la página de la imagen oficial de PostgreSQL [44] podemos encontrar una guía detallada de cómo crear contenedores con esta imagen utilizando comandos Docker o mediante Docker-Compose. En la figura 4-1 se muestra la definición del contenedor PostgreSQL realizada mediante Docker-Compose:

```
54  base_datos:
55
56      container_name: postgresql
57
58      image: postgres:latest
59
60      environment:
61          - POSTGRES_USER=sergio
62          - POSTGRES_PASSWORD=sergio
63          - POSTGRES_DB=monitor_consumo
64          - PGDATA=/var/lib/postgresql/data/pgdata
65          - TZ=Europe/Madrid
66
67      ports:
68          - "5432:5432"
69
70      restart: always
71
72      volumes:
73          - ./postgresql/data:/var/lib/postgresql/data
74
```

Figura 4-1. Definición del contenedor PostgreSQL en Docker-Compose

La definición de este contenedor consiste en lo siguiente:

- Crear un servicio en Docker-Compose llamado **base_datos**.
- Crear un contenedor llamado **postgresql**, asociado al servicio **base_datos**.
- Crear dicho contenedor a partir de la última versión de la imagen de PostgreSQL, indicado por la etiqueta latest.
- Configurar las siguientes variables de entorno:
    - **POSTGRES_USER**: Usuario que queremos crear en PostgreSQL.
    - **POSTGRES_PASSWORD**: Contraseña del usuario que queremos crear.
    - **POSTGRES_DB**: Nombre de la base de datos que se creará automáticamente al crear el contenedor, en este caso, **monitor_consumo.**
    - **PG_DATA**: Directorio del contenedor en el que PostgreSQL almacenará los datos, que en este caso es **/var/lib/postgresql/data/pgdata**.
    - **TZ**: Zona horaria del contenedor, que en este caso es la de Madrid.
- Abrir el puerto 5432 de la máquina anfitriona y redirigirlo al puerto 5432 del contenedor, que es el puerto por defecto utilizado por PostgreSQL.
- Reiniciar automáticamente el contenedor en caso de que se detenga, garantizando que el servicio se esté ejecutando en todo momento.
- Compartir el directorio **./postgresql/data** del *host* (partiendo del directorio de trabajo donde está el fichero docker-compose.yaml) con el directorio **/var/lib/postgresql/data** del contenedor. De esta forma podemos persistir la información de la base de datos en la máquina anfitriona, aunque borremos el contenedor.

4.1.2 Creación de las tablas en la base de datos

Una vez que hayamos desplegado la base de datos utilizando Docker-Compose, podemos crear las tablas en la base de datos mediante un script Bash llamado **base_datos.sh**. El contenido de dicho script se muestra en la figura 4-2:

```
#!/bin/bash

PGPASSWORD=sergio psql -h 127.0.0.1 -p 5432 -U sergio monitor_consumo < crear_tablas.sql

```

Figura 4-2. Contenido del fichero base_datos.sh

En este script nos conectamos a la base de datos **monitor_consumo** y ejecutamos las sentencias SQL del fichero **crear_tablas.sql**. En la figura 4-3 pueden verse las sentencias SQL incluidas en este fichero:

```
--- Fichero SQL para crear las tablas de la base de datos

--- Creamos la tabla 'potencia_shelly'

DROP TABLE IF EXISTS potencia_shelly;

CREATE TABLE potencia_shelly(
    id_medida SERIAL PRIMARY KEY,
    fecha_medida TIMESTAMP NOT NULL UNIQUE,
    potencia FLOAT NOT NULL
);

--- Creamos la tabla 'consumo_shelly'

DROP TABLE IF EXISTS consumo_shelly;

CREATE TABLE consumo_shelly(
    id_medida SERIAL PRIMARY KEY,
    fecha_medida DATE NOT NULL UNIQUE,
    consumo FLOAT NOT NULL
);

--- Creamos la tabla 'coste_shelly'

DROP TABLE IF EXISTS coste_shelly;

CREATE TABLE coste_shelly(
    id_coste SERIAL PRIMARY KEY,
    fecha_coste DATE NOT NULL UNIQUE,
    coste FLOAT NOT NULL
);

```

Figura 4-3. Contenido del fichero crear_tablas.sql

Mediante este fichero SQL creamos las siguientes tablas:

- **potencia_shelly**: En esta tabla es donde almacenamos las medidas de potencia generadas por el Shelly Plug. Cada medida de potencia tiene un identificador único (**id_medida**), la fecha y hora en la que se tomó esa medida (**fecha_medida**), y el valor de potencia en vatios (**potencia**).
- **consumo_shelly**: Esta tabla sirve para almacenar el histórico de consumo. Cada medida de consumo tiene un identificador único (**id_medida**), la fecha en la que se obtuvo esa medida (**fecha_medida**), y el valor de consumo en vatios-hora (**consumo**).
- **coste_shelly**: Es la tabla en la que almacenamos el histórico de costes. Cada registro de coste tiene un identificador único (**id_coste**), la fecha en la que se calculó ese coste (**fecha_coste**), y el valor del coste en euros (**coste**).

4.2 Rutas del servicio REST

En esta sección es donde presentamos las rutas del servicio REST de consumo y costes, es decir, aquellos puntos de acceso que nos permiten interactuar con los recursos ofrecidos por el servicio. Para ello, haremos

uso de la interfaz Swagger que se ha generado para la documentación del servicio REST (ver Sección 4.5 Documentación OpenAPI mediante Flasgger). Gracias a esta interfaz, podemos hacer peticiones HTTP a las rutas del servicio REST y visualizar las respuestas del servidor de forma sencilla. En la figura 4-4 se muestra una imagen de la documentación OpenAPI del servicio REST y en la tabla 4-1 se hace una descripción de cada ruta y los métodos HTTP que soportan:

Figura 4-4. Especificación OpenAPI del servicio REST de consumo y costes

Ruta	Métodos HTTP	Descripción
/ping	GET	Ruta para probar el funcionamiento del servicio. El servidor responderá con el mensaje “pong!” como confirmación de que el servicio está operativo.
/shelly-plug/potencia/medidas	GET, POST, DELETE	Ruta que permite interactuar con las medidas de potencia (en vatios) del Shelly Plug según el método HTTP utilizado: • GET: Permite obtener las medidas de potencia del Shelly Plug en un intervalo horario especificado en el *query string* de la petición. En caso de no especificarse, se obtienen las medidas de la última hora. • POST: Permite añadir una nueva

		medida de potencia en la base de datos. • DELETE: Permite eliminar todas las medidas de potencia de la base de datos o una medida específica según su identificador.
/shelly-plug/consumo	GET	Ruta para obtener el consumo energético (en vatios-hora) en un intervalo horario especificado en el *query string* de la petición. En caso de no especificarse, se obtiene el consumo de la última hora. El cálculo del consumo se realiza a partir de las medidas de potencia almacenadas en la base de datos.
/shelly-plug/consumo/consumo-por-hora	GET	Ruta para obtener el consumo energético (en vatios-hora) en cada una de las 24 horas del día.
/shelly-plug/consumo/medidas	GET, DELETE	Ruta para interactuar con el histórico de medidas de consumo energético (en vatios-hora) según el método HTTP utilizado: • GET: Permite obtener las medidas de consumo almacenadas en la base de datos. • DELETE: Permite eliminar todas las medidas de consumo de la base de datos.
/shelly-plug/costes	GET, POST, DELETE	Ruta para interactuar con el histórico de costes de consumo energético (en euros) según el método HTTP utilizado: • GET: Permite obtener todos los registros de costes de consumo almacenados en la base de datos. • POST: Permite almacenar un nuevo registro de coste energético o actualizar uno ya existente. • DELETE: Permite eliminar todos los registros de coste energético de la base de datos.

Tabla 4-1. Rutas del servicio REST de consumo y costes

Como se comentó en el Capítulo 3 (Control y monitorización mediante NodeRED), NodeRED realiza peticiones HTTP a las rutas de este servicio REST para monitorizar la potencia instantánea consumida en el enchufe, hacer un seguimiento del consumo eléctrico, y generar históricos de consumos y costes. Como puede verse en la especificación OpenAPI del servicio mostrada en la figura 4-4, las rutas están organizadas según el grupo de recursos con el que se interactúa: **test**, **potencia**, **consumo** y **coste**. La implementación de las funciones que manejan estas rutas se encuentra en el fichero **rutas.py** de la aplicación Flask. En las siguientes secciones se presentarán ejemplos de uso de estas rutas.

4.2.1 Test

La única ruta de este grupo es **/ping**, la cual simplemente sirve para probar el funcionamiento del servicio REST. Si realizamos una petición GET a esta ruta, el servidor nos responderá con un objeto JSON que contiene el mensaje "pong!", como puede verse en la figura 4-5:

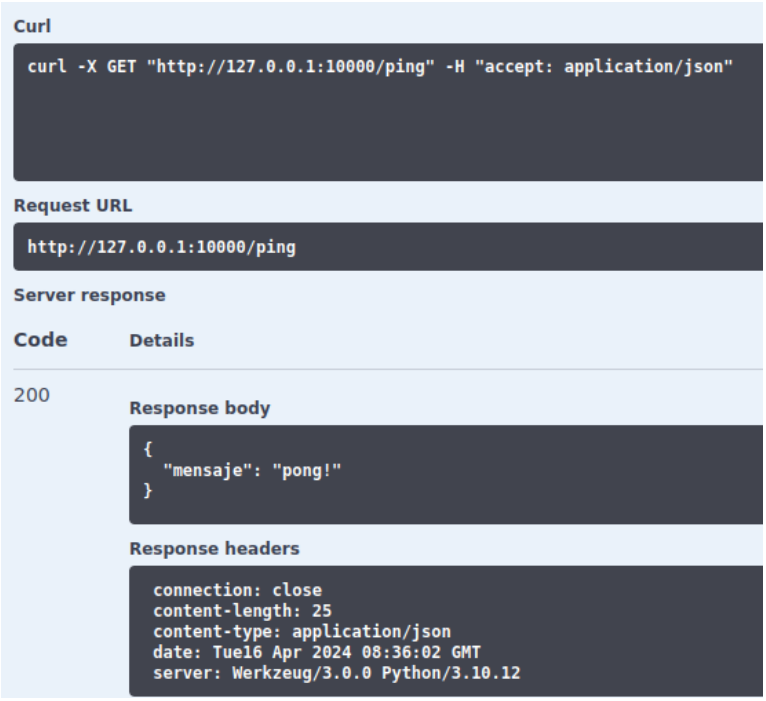


Figura 4-5. Petición GET a /ping

A modo de ejemplo, en la figura 4-6 se muestra un extracto del fichero **rutas.py** con el código de la función que maneja las peticiones GET a esta ruta. Cada combinación de rutas y métodos HTTP mostradas en la figura 4-4 tiene asociada una función de la aplicación Flask que maneja esas peticiones y devuelve la respuesta correspondiente.

```
12  # Ruta para probar el funcionamiento del servicio
13  @bp.route("/ping", methods=["GET"])
14  def ping():
15      """
16      Ruta para probar el funcionamiento del servicio.
17      ---
18      tags:
19        - Test
20      responses:
21        200:
22          description: El servicio REST es accesible.
23          content:
24            application/json
25      """
26
27      return jsonify({"mensaje":"pong!"})
```

Figura 4-6. Código de la función ping en rutas.py

4.2.2 Potencia

Este grupo de recursos tiene una única ruta: **/shelly-plug/potencia/medidas**, con la que se puede interactuar con los métodos GET, POST y DELETE.

Si realizamos una petición GET a esta ruta sin ningún parámetro en el *query string*, obtendremos una lista con las medidas de potencia (en vatios) registradas en la última hora, como puede verse en la figura 4-7:

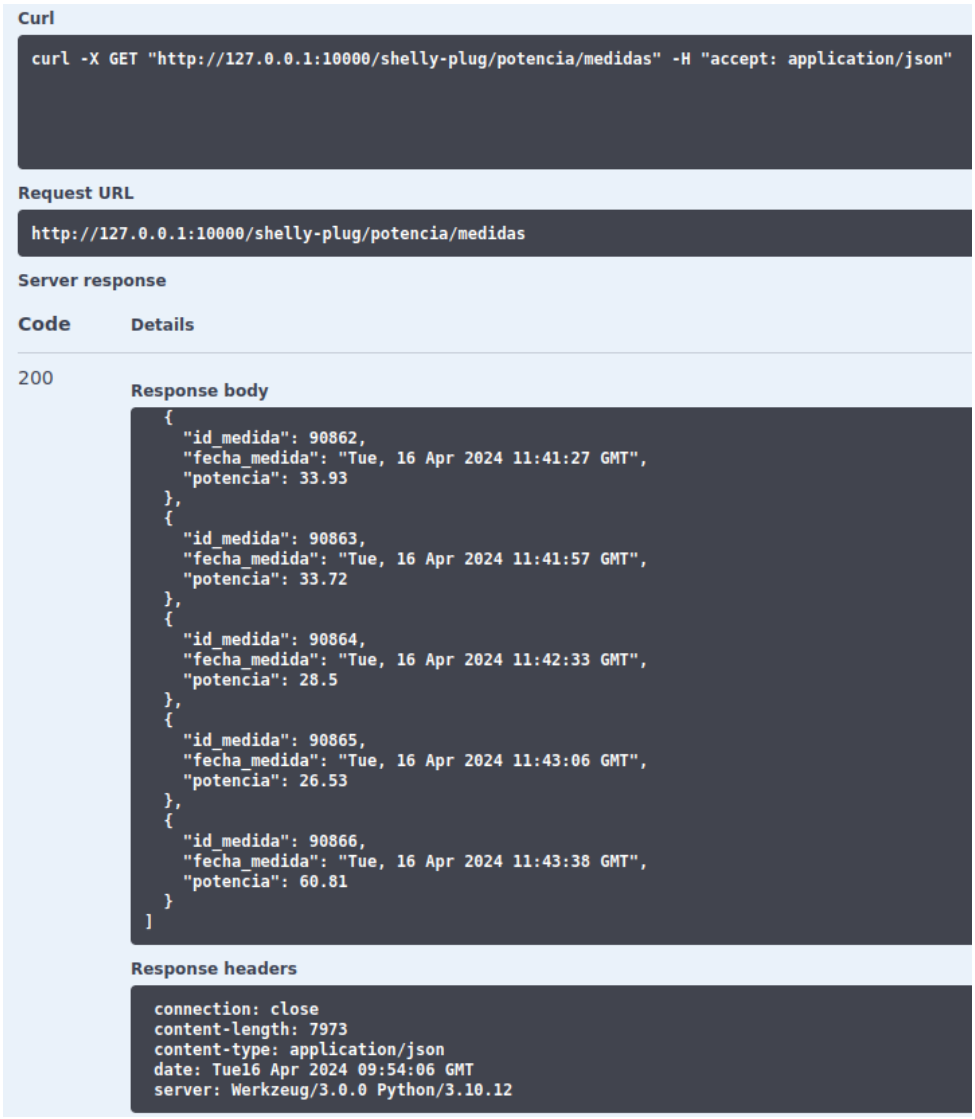


Figura 4-7. Petición GET a /shelly-plug/potencia/medidas (sin parámetros)

Sin embargo, en el *query string* podemos especificar que queremos obtener las medidas de potencia que se registraron desde una hora inicio a una hora fin. Por ejemplo, en la figura 4-8 puede verse como se obtienen las medidas registradas desde las 11:00 hasta las 11:02:

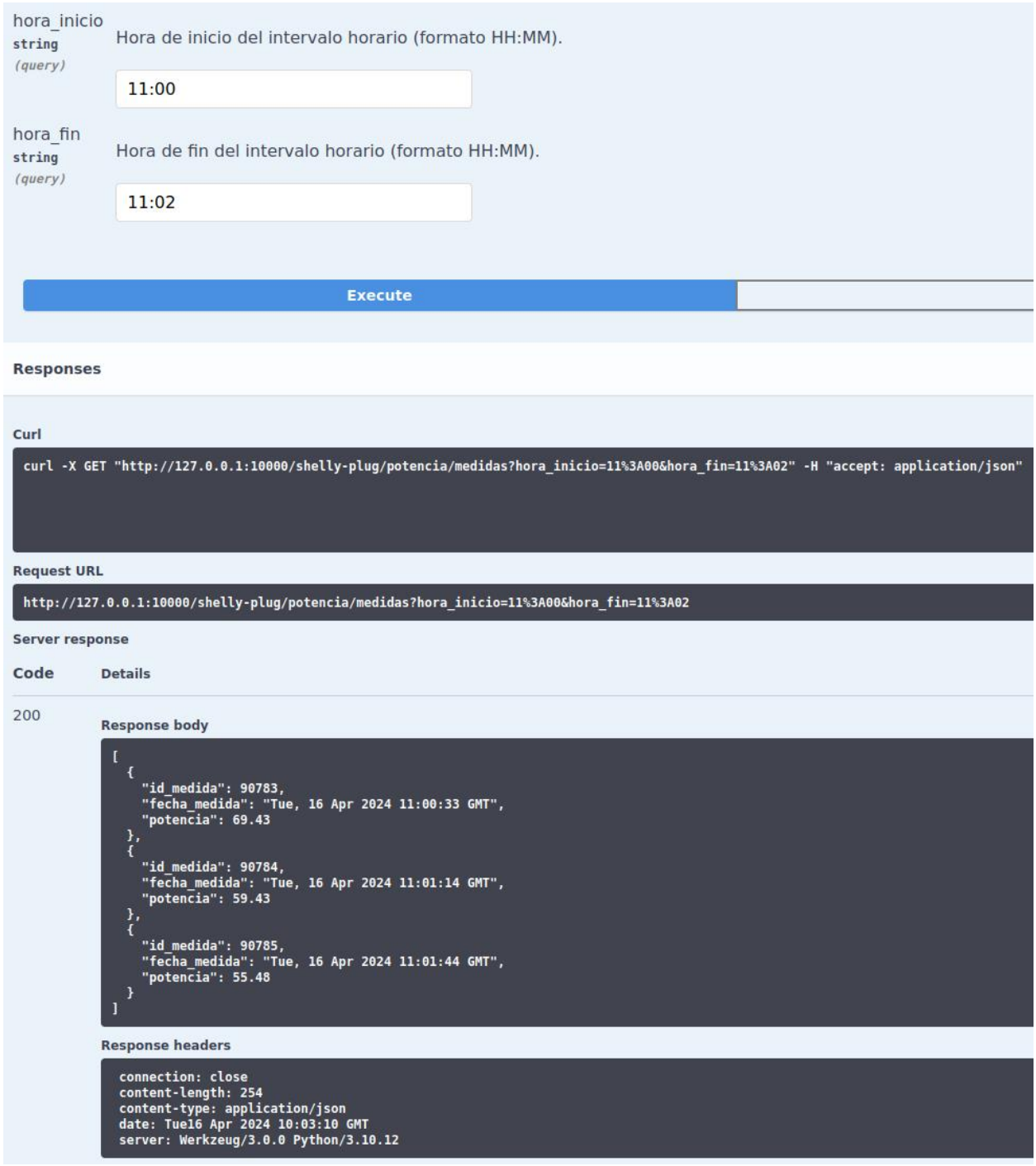


Figura 4-8. Petición GET a /shelly-plug/potencia/medidas (con parámetros)

Mediante peticiones POST, podemos insertar nuevas medidas de potencia en la base de datos a partir de objetos JSON que contienen la información de la medida, como puede verse en las figuras 4-9 y 4-10:

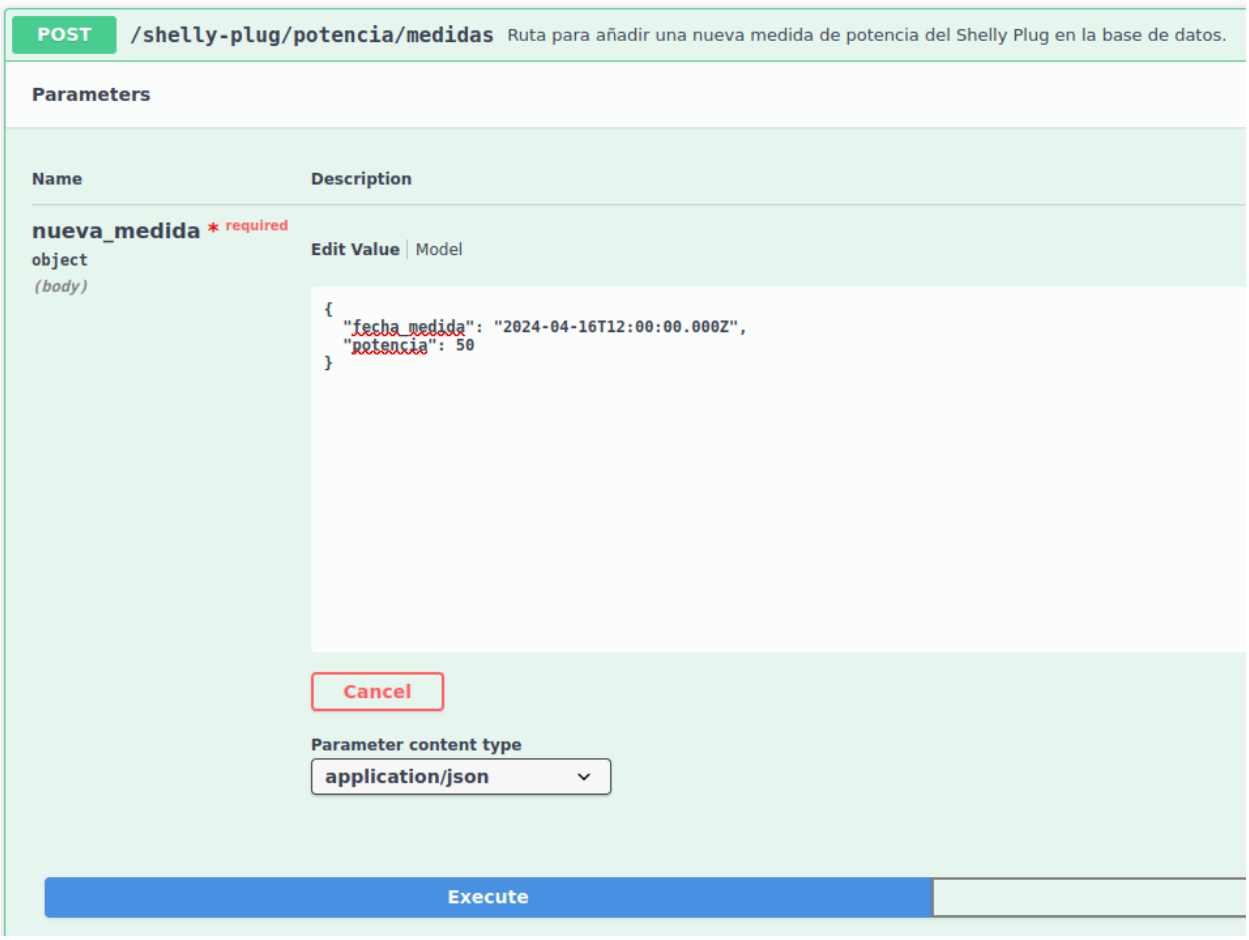


Figura 4-9. Cuerpo de la petición POST a /shelly-plug/potencia/medidas

Figura 4-10. Petición POST a /shelly-plug/potencia/medidas

Finalmente, mediante peticiones DELETE podemos borrar una medida específica de la base de datos si también especificamos su identificador, como se muestra en la figura 4-11, o si no lo indicamos, podemos borrar todas las medidas de la base de datos, como puede verse en la figura 4-12:

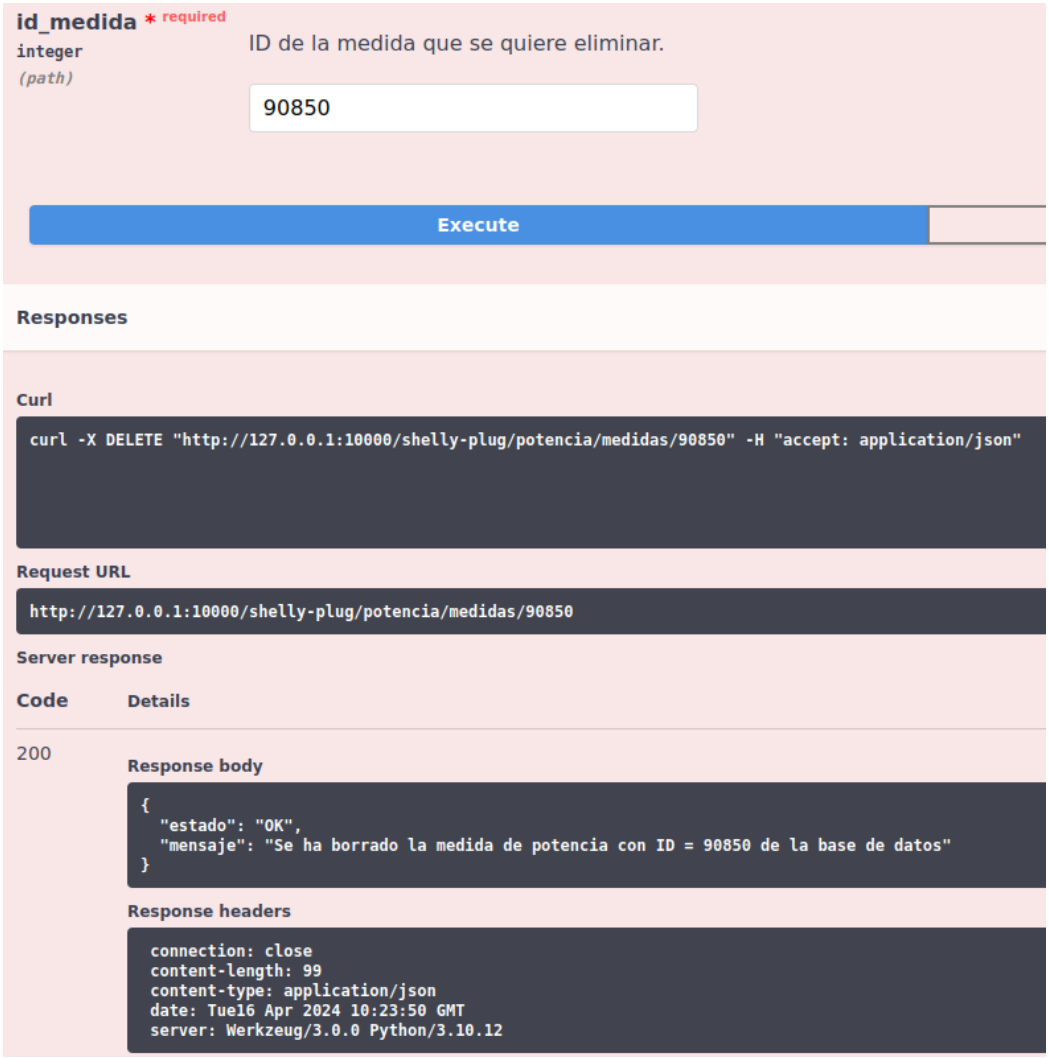


Figura 4-11. Petición DELETE a /shelly-plug/potencia/medidas con identificador

Figura 4-12. Petición DELETE a /shelly-plug/potencia/medidas sin identificador

4.2.3 Consumo

Este grupo de recursos contiene tres rutas:

- **/shelly-plug/consumo**
- **/shelly-plug/consumo/consumo-por-hora**
- **/shelly-plug/consumo/medidas**

Si hacemos una petición GET a **/shelly-plug/consumo** sin ningún parámetro en el *query string*, obtendremos el consumo (en vatios-hora) de la última hora, como puede verse en la figura 4-13:

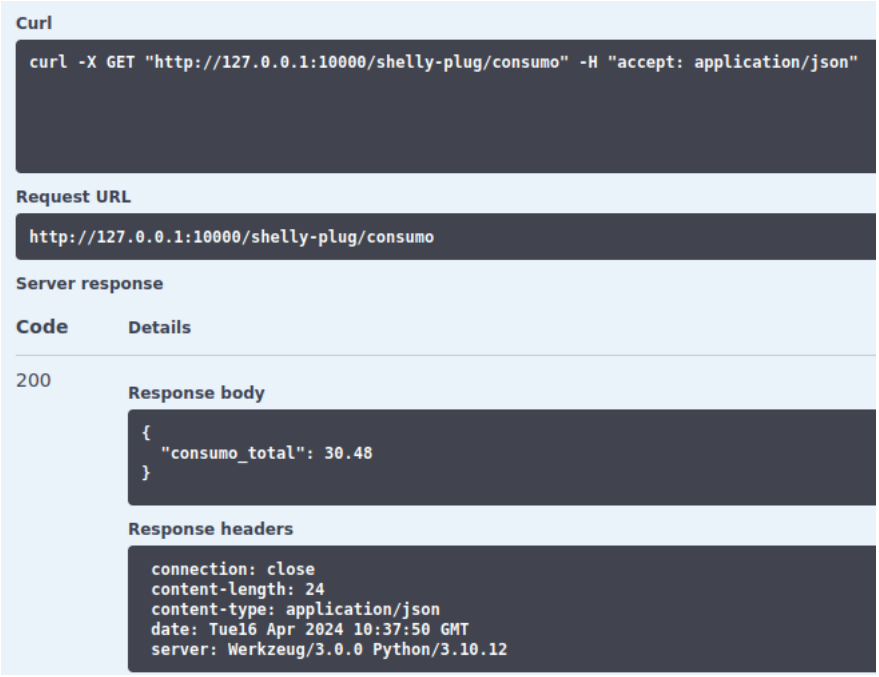


Figura 4-13. Petición GET a /shelly-plug/consumo (sin parámetros)

En cambio, si en el *query string* incluimos una hora de inicio y de fin, obtendremos el consumo que se ha producido en ese intervalo de tiempo. Por ejemplo, en la figura 4-14 puede verse como se obtiene el consumo que se ha producido entre las 12:00 y las 12:20:

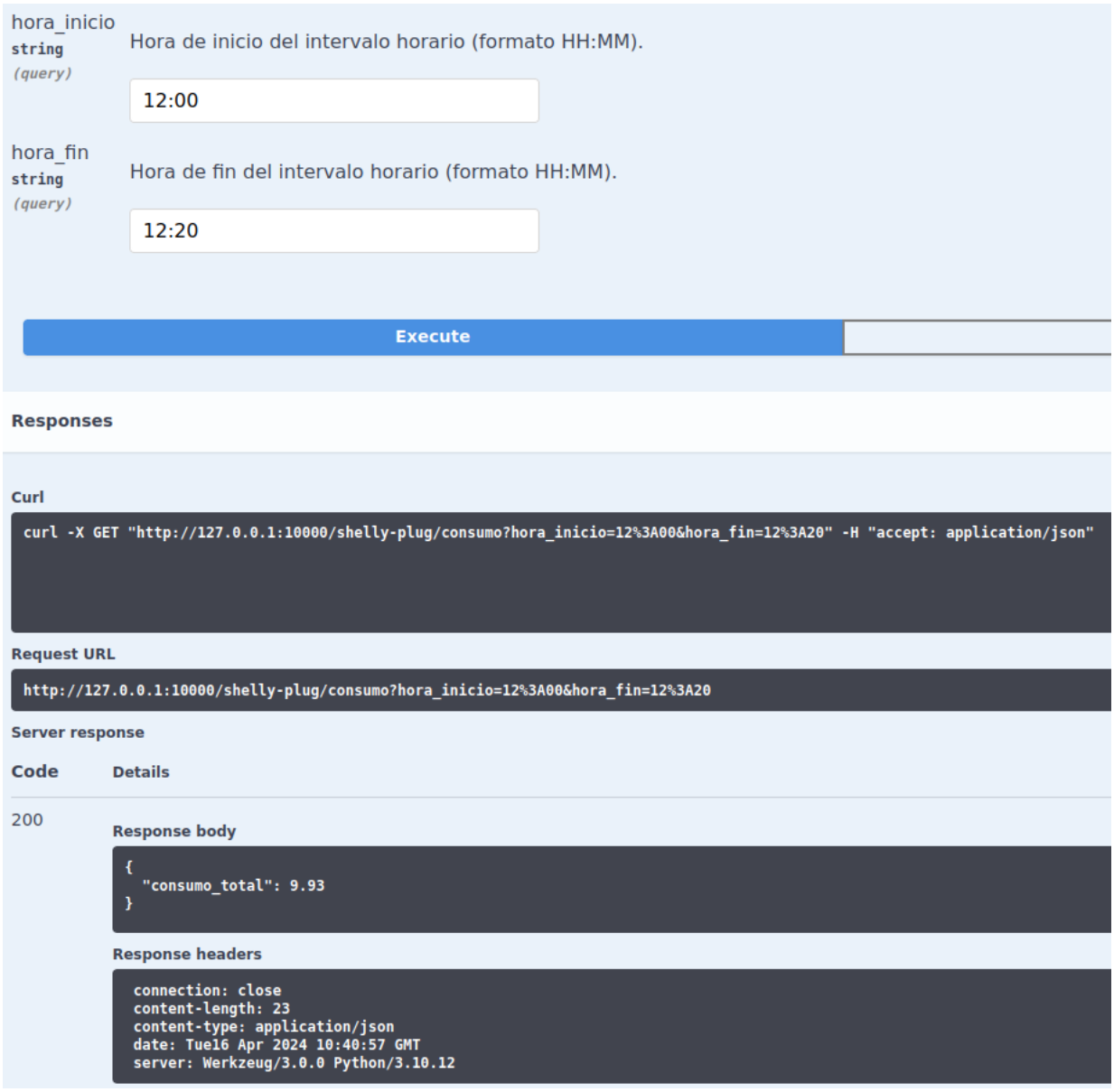


Figura 4-14: Petición GET a /shelly-plug/consumo (con parámetros)

Si hacemos una petición GET a la ruta **/shelly-plug/consumo/consumo-por-hora**, obtendremos el consumo eléctrico que se ha producido en cada una de las 24 horas del día, como se muestra en la figura 4-15:

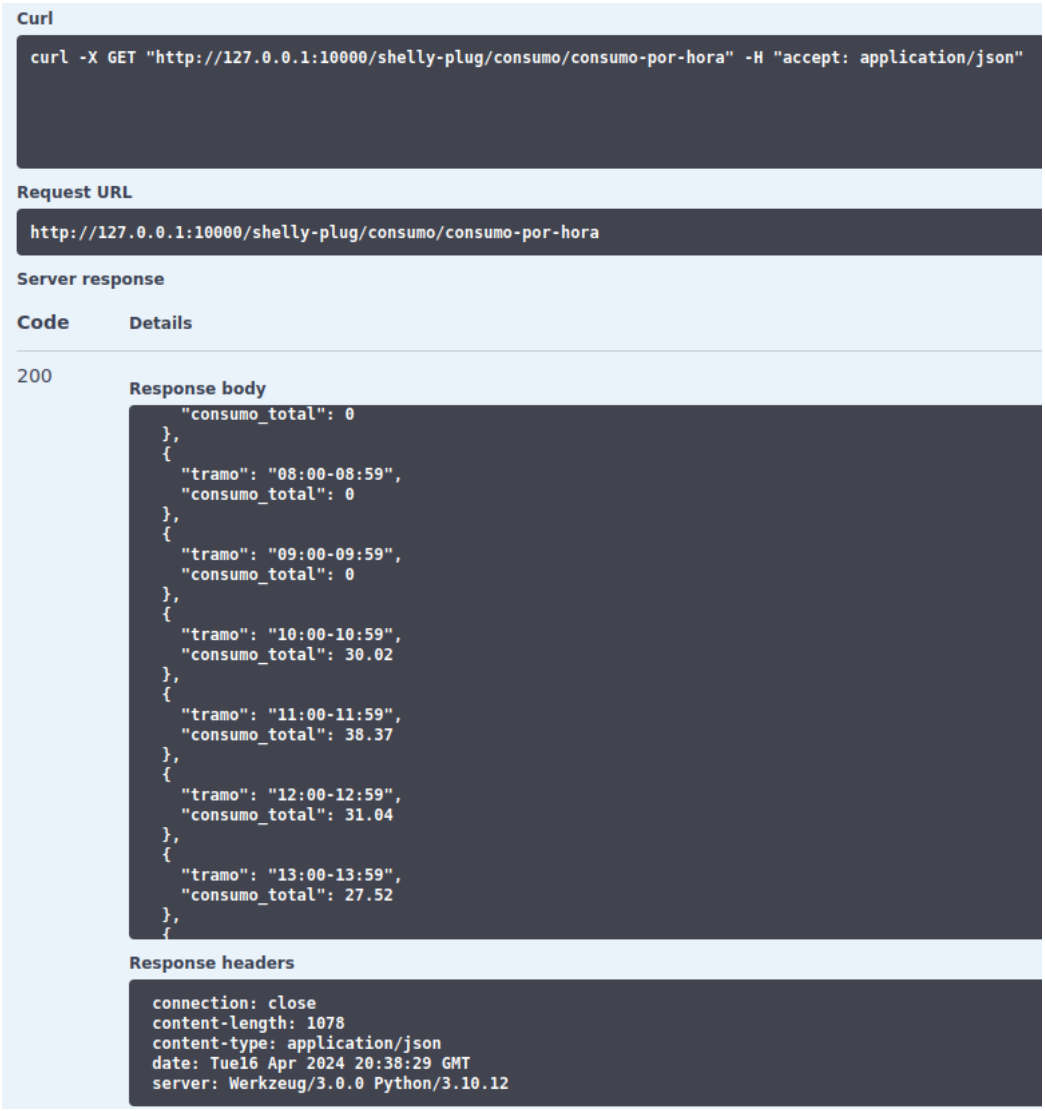


Figura 4-15. Petición GET a /shelly-plug/consumo/consumo-por-hora

Por otro lado, mediante peticiones GET a la ruta **/shelly-plug/consumo/medidas** podemos obtener el histórico de medidas de consumo de la base de datos, como se muestra en la figura 4-16:

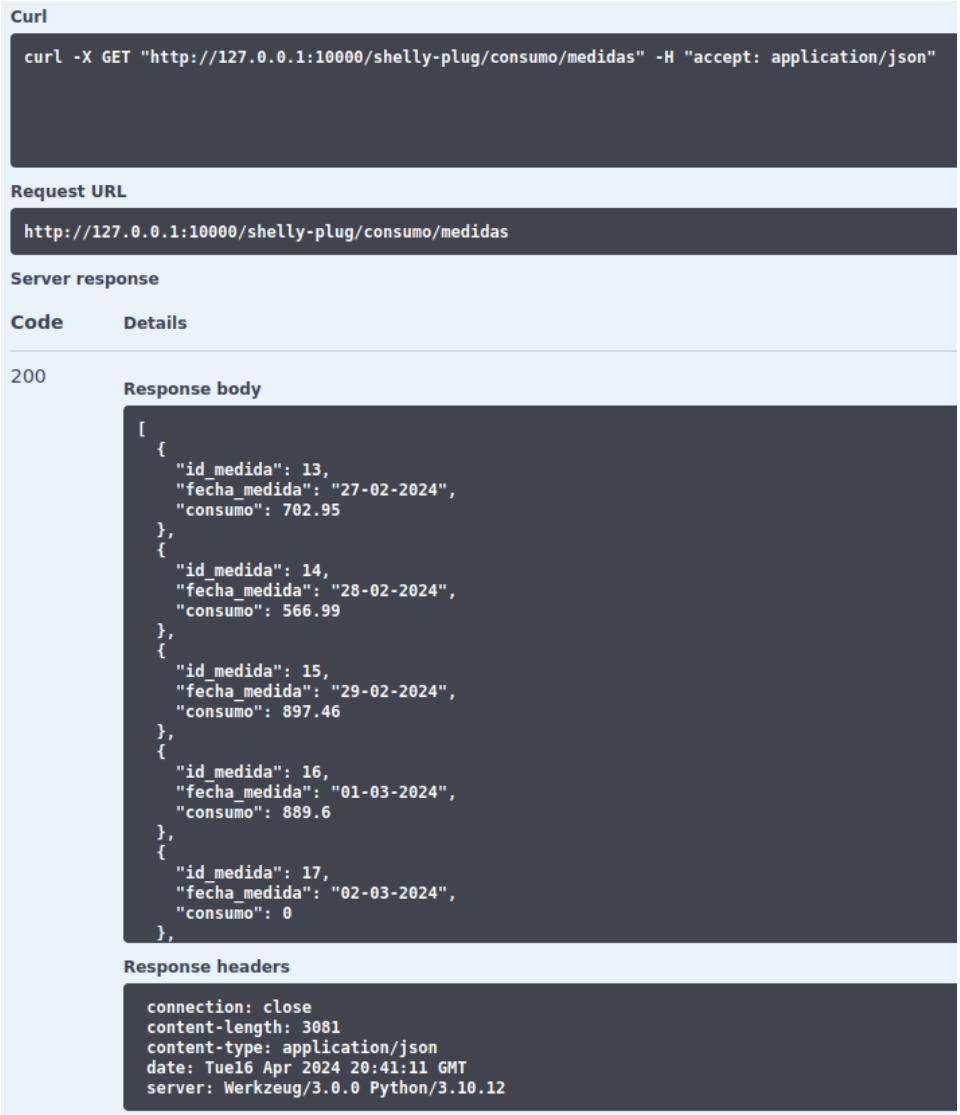


Figura 4-16. Petición GET a /shelly-plug/consumo/medidas

Por último, si hacemos una petición DELETE a esta misma ruta, borraremos todo el histórico de medidas de consumo, como se muestra en la figura 4-17:

Figura 4-17. Petición DELETE a /shelly-plug/consumo/medidas

4.2.4 Coste

Este grupo de recursos solo contiene la ruta **/shelly-plug/costes**.

Si hacemos una petición GET a esta ruta, obtendremos el histórico de costes (en euros) del consumo, como se puede observar en la figura 4-18:

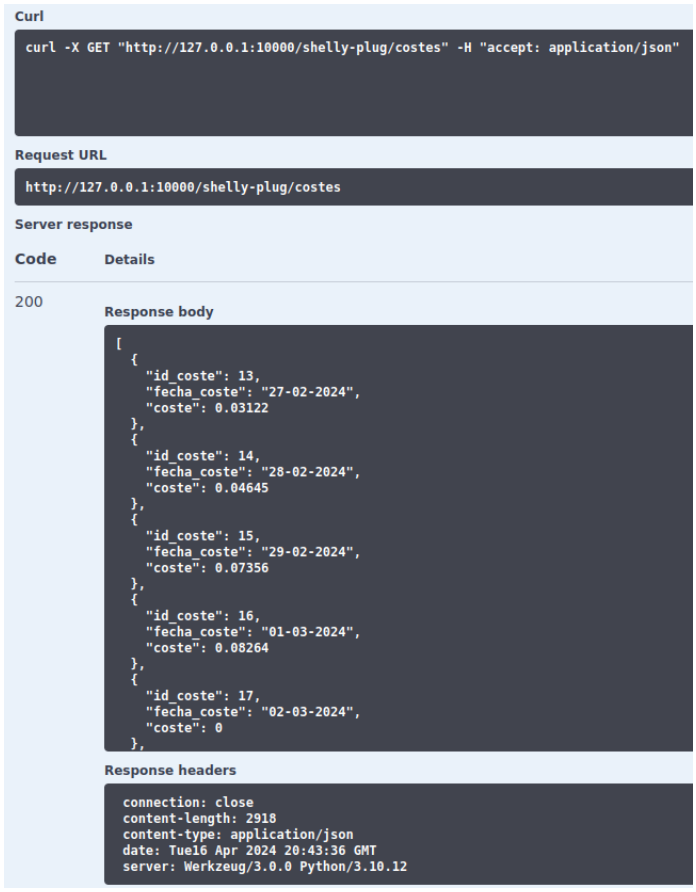


Figura 4-18. Petición GET a /shelly-plug/costes

En cambio, mediante peticiones POST, podemos insertar un nuevo registro de coste o actualizar uno ya existente a partir de objetos JSON que contienen la información de dicho coste, como puede verse en las figuras 4-19 y 4-20:

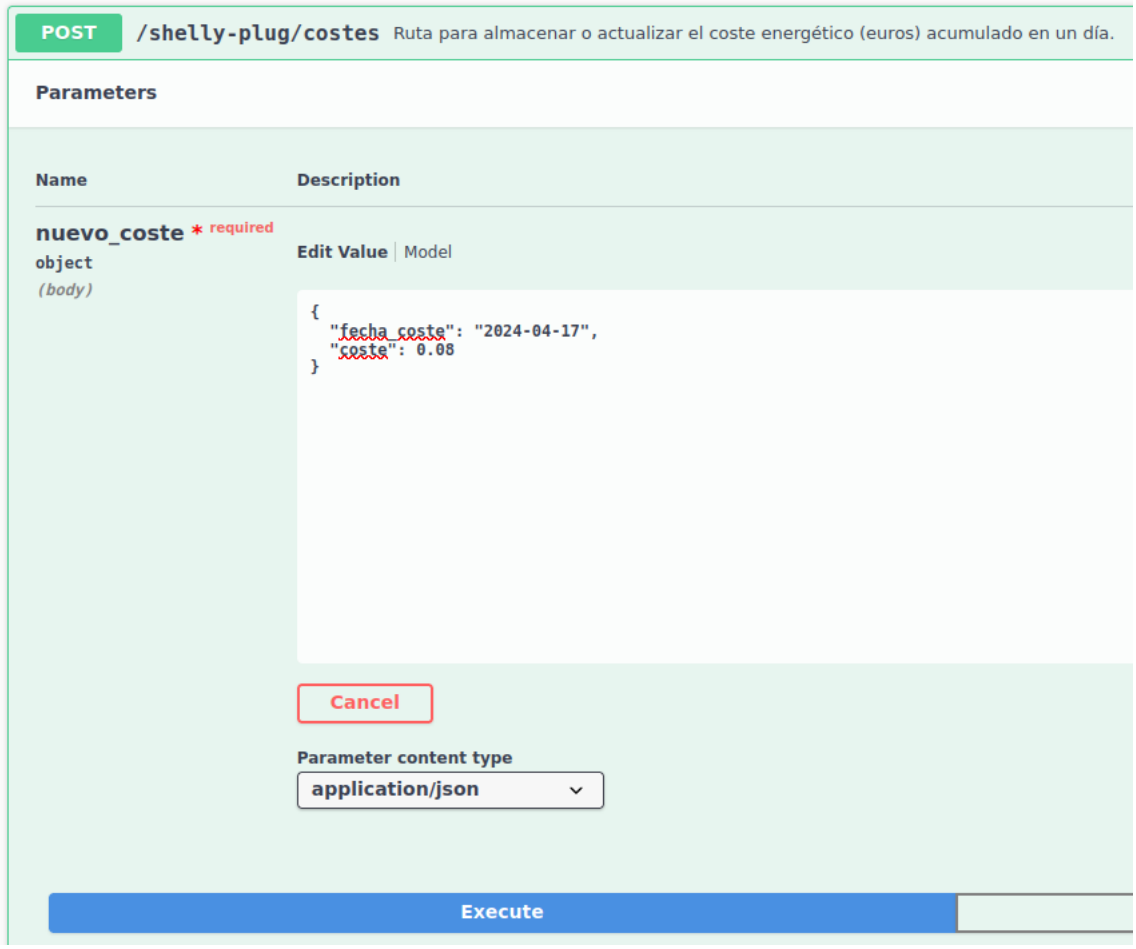


Figura 4-19. Cuerpo de la petición POST a /shelly-plug/costes

Figura 4-20. Petición POST a /shelly-plug/costes

Finalmente, mediante peticiones DELETE podemos borrar todo el histórico de costes de la base de datos, como se muestra en la figura 4-21:

Figura 4-21. Petición DELETE a /shelly-plug/costes

4.3 Cálculo del consumo eléctrico

En esta sección es donde se explica la metodología que se ha seguido para calcular el consumo eléctrico a partir de las medidas de potencia generadas por el Shelly Plug. Para poder calcular el consumo, primero se ha definido una función dentro del fichero **utilidades.py** llamada **calcula_consumo.** Además, este fichero contiene otras funciones auxiliares que nos permiten realizar conexiones con la base de datos y transformar objetos de la clase MedidaPotencia (ver Sección 4.4 Modelo de datos) a diccionarios, para facilitar su posterior conversión a objetos JSON. El código de la función calcula_consumo se muestra en la figura 4-22:

```
### Funciones ###

# Función para calcular el consumo energético (en Wh) a partir de medidas
# de potencia (proporcionadas en vatios)
def calcula_consumo(medidas):

    consumo_total = 0.0

    for i in range(len(medidas) - 1): # Sacamos la última medida del cálculo ya
                                      # que no podemos determinar el intervalo
                                      # de tiempo hasta la siguiente medida

        # Calculamos el tiempo entre medidas en segundos
        t_entre_medidas = (medidas[i + 1]["fecha_medida"] - medidas[i]["fecha_medida"]).total_seconds()

        # Si ha pasado mas de 1 minuto entre la medida actual y la
        # siguiente, no la consideramos para el calculo de la energia consumida
        if t_entre_medidas > 60:
            pass

        # Calculamos el consumo energético total en ese intervalo de tiempo (E = P * t) y lo
        # sumamos al consumo total (medido en Julios)
        consumo_total += medidas[i]["potencia"] * t_entre_medidas

    # Pasamos de Julios (W * s) a Wh
    consumo_total = round(consumo_total / 3600, 2)

    return consumo_total
```

Figura 4-22. Código de la función calcula_consumo en utilidades.py

Esta función recibe una lista de diccionarios en la que cada uno de ellos es una medida de potencia. Cada medida de potencia tiene un identificador único, la fecha y hora en la que se tomó la medida, y el valor de potencia que se registró (en vatios).

De forma general, el consumo eléctrico (o energía) se calcula como la integral de la potencia eléctrica instantánea en un intervalo de tiempo. Sin embargo, a partir de los mensajes de potencia provenientes del Shelly Plug, estamos muestreando valores de potencia cada 30 segundos. Si consideramos que la potencia instantánea no cambia de forma significativa entre dos medidas de potencia consecutivas, podemos aproximar la integral a un sumatorio de valores de potencia multiplicados por el intervalo de tiempo hasta la siguiente medida. La fórmula empleada para este cálculo se muestra en la figura 4-23:

$$E = \int_{t_1}^{t_2} p(t)\, dt \approx \sum_{i=1}^{N-1} \hat{p}_i * \Delta t_i$$

Figura 4-23. Fórmula para la estimación del consumo eléctrico

Donde:

- E es el consumo eléctrico.
- $\hat{p}_i$ es el valor de potencia de la medida i.
- Δt_i es el intervalo de tiempo entre las medidas i e $i + 1$.
- N es el número total de medidas de potencia.

En esta fórmula, la última medida de potencia queda fuera del cálculo porque no es posible determinar cuál será el intervalo de tiempo hasta la próxima medida. Además, cuando este intervalo de tiempo es superior al minuto, la medida tampoco se incluye en el cálculo. El motivo de esto es que intervalos de tiempo muy superiores al minuto pueden deberse a pérdidas de conexión con el Shelly Plug, a la parada y posterior rearranque del sistema domótico, o a la desconexión del Shelly Plug de la toma de corriente. Por lo tanto, incluir esa medida de potencia en el cálculo podría hacer que perdamos precisión en la estimación del consumo, ya que conforme mayor sea ese tiempo, es menos probable que la potencia se haya mantenido constante.

La función mostrada en la figura 4-22 implementa esta fórmula para calcular el consumo eléctrico a partir de las medidas de potencia proporcionadas como parámetro. Además, se realizan las conversiones de unidades necesarias para devolver el resultado de consumo eléctrico en vatios-hora.

Como ejemplo de uso de esta función, en la figura 4-24 se muestra un extracto de la función **obtener_consumo** dentro del fichero **rutas.py** que procesa las peticiones GET a la ruta /shelly-plug/consumo cuando queremos obtener el consumo eléctrico que se ha producido entre una hora inicio y una hora fin:

```
    respuesta = {}

    try:
        # Obtenemos los parámetros de la URI
        hora_inicio = request.args.get("hora_inicio")
        hora_fin = request.args.get("hora_fin")

        if hora_inicio and hora_fin:
            # Si se proporcionan ambas horas, calculamos el consumo en el intervalo horario

            # Convertimos las horas a objetos datetime
            hora_inicio_dt = datetime.strptime(hora_inicio, "%H:%M")
            hora_fin_dt = datetime.strptime(hora_fin, "%H:%M")

            # Obtenemos la fecha del día actual
            fecha_actual = datetime.now().date()

            # Combinamos la fecha actual con las horas proporcionadas
            fecha_inicio = datetime.combine(fecha_actual, hora_inicio_dt.time())
            fecha_fin = datetime.combine(fecha_actual, hora_fin_dt.time())

            # Nos conectamos a la base de datos
            session = conectar_bd(HOST, DATABASE, USER, PASSWORD, PORT)
            app.logger.info("Conexión con la base de datos")

            # Consultamos las medidas de potencia realizadas en el intervalo horario especificado
            medidas = session.query(MedidaPotencia).order_by(MedidaPotencia.fecha_medida).filter(
                MedidaPotencia.fecha_medida >= fecha_inicio, MedidaPotencia.fecha_medida <= fecha_fin).all()
            app.logger.info(f"Se han consultado las de medidas de potencia realizadas entre las {hora_inicio}-{hora_fin}")

            # Cerramos la sesión en la base de datos
            session.close()

            # Lista para almacenar las medidas como diccionarios
            medidas_dict = []

            for medida in medidas:
                medidas_dict.append(medida_to_dict(medida))

            # Calculamos el consumo energético total a partir de las medidas de potencia
            consumo_total = calcula_consumo(medidas_dict)
            app.logger.info("Se ha calculado el consumo energético total")

            respuesta = {"consumo_total": consumo_total}

```

Figura 4-24. Extracto de la función obtener_consumo en rutas.py

En este fragmento de código, obtenemos las horas de inicio y fin a partir de los parámetros del *query string* de la petición HTTP. Adicionalmente, necesitaremos construir objetos de tipo fecha (datetime) para poder obtener de la base de datos aquellas medidas que estén en el intervalo horario que queremos. Para ello, combinamos la fecha del día actual con las horas proporcionadas. Posteriormente, nos conectamos a la base de datos y realizamos la consulta de las medidas de potencia deseadas. Luego, creamos una lista de medidas de potencia (convirtiendo previamente los objetos de la clase MedidaPotencia a diccionarios), y calculamos el consumo energético llamando a la función calcula_consumo.

Además, a diferencia del histórico de costes, que se generaba a partir de peticiones POST a la ruta /shelly-plug/costes, como decisión de diseño se ha decidido que el histórico de consumo lo actualice automáticamente el servicio REST cada vez que se inserta una nueva medida de potencia en la base de datos. Podemos ver esto ilustrado en las figuras 4-25 y 4-26, que muestran parte del código de la función **nueva_medida_potencia** dentro del fichero **rutas.py**, que maneja las peticiones POST que se hacen a la ruta **/shelly-plug/potencia/medidas**:

```python
    respuesta = {}

    try:

        error = False

        # Obtenemos los datos de la medida en formato JSON
        nueva_medida = request.get_json()

        # Comprobamos que la medida tiene todos los atributos necesarios (menos el ID)
        lista_atributos = MedidaPotencia.__table__.columns.keys()[1:]
        for atributo in lista_atributos:
            if atributo not in nueva_medida:
                error = True
                app.logger.error(f"Falta el siguiente atributo en la medida de potencia: {atributo}")
                respuesta = {"estado":"ERROR","mensaje":f"Falta el siguiente atributo en la medida de potencia: {atributo}"}
                break

        if not error:

            # Nos conectamos a la base de datos
            session = conectar_bd(HOST, DATABASE, USER, PASSWORD, PORT)
            app.logger.info("Conexión con la base de datos")

            # Creamos un objeto ORM de la clase MedidaPotencia

            medida_potencia = MedidaPotencia(
                fecha_medida=nueva_medida["fecha_medida"],
                potencia=nueva_medida["potencia"]
            )

            # Añadimos la medida de potencia en la base de datos
            session.add(medida_potencia)
            session.commit()
            app.logger.info("Se ha añadido una nueva medida de potencia en la base de datos")

            # Actualizamos el consumo energético en la base de datos
            # Para ello, primero obtenemos la fecha del día de hoy
            fecha_hoy = datetime.now().date()
            app.logger.info("Actualización del consumo energético diario")

            # Obtenemos la hora inicial y final del día como objetos datetime
            hora_inicio_dt = datetime.strptime("00:00", "%H:%M")
            hora_fin_dt = datetime.strptime("23:59", "%H:%M")

            # Combinamos la fecha actual con la hora inicial y final del día
            fecha_inicio = datetime.combine(fecha_hoy, hora_inicio_dt.time())
            fecha_fin = datetime.combine(fecha_hoy, hora_fin_dt.time())

            # Consultamos las medidas de potencia realizadas en el dia de hoy (00:00-23:59)
            medidas_potencia = session.query(MedidaPotencia).order_by(MedidaPotencia.fecha_medida).filter(
                MedidaPotencia.fecha_medida >= fecha_inicio, MedidaPotencia.fecha_medida <= fecha_fin).all()
            app.logger.info("Se han consultado las de medidas de potencia realizadas en el día de hoy (00:00-23:59)")

            # Lista para almacenar las medidas de potencia como diccionarios
            medidas_pot_dict = []
```

Figura 4-25. Extracto de la función nueva_medida_potencia en rutas.py. Parte 1

```
            # Consultamos las medidas de potencia realizadas en el dia de hoy (00:00-23:59)
            medidas_potencia = session.query(MedidaPotencia).order_by(MedidaPotencia.fecha_medida).filter(
                MedidaPotencia.fecha_medida >= fecha_inicio, MedidaPotencia.fecha_medida <= fecha_fin).all()
            app.logger.info("Se han consultado las de medidas de potencia realizadas en el día de hoy (00:00-23:59)")

            # Lista para almacenar las medidas de potencia como diccionarios
            medidas_pot_dict = []

            for medida_potencia in medidas_potencia:
                medidas_pot_dict.append(medida_to_dict(medida_potencia))

            # Calculamos el consumo energético total a partir de las medidas de potencia
            consumo_total = calcula_consumo(medidas_pot_dict)
            app.logger.info("Se ha calculado el consumo energético total")

            # Consultamos si ya existe un registro para el consumo en el dia de hoy
            consumo_hoy = session.query(MedidaConsumo).filter(MedidaConsumo.fecha_medida == fecha_hoy).first()
            app.logger.info("Comprobación de la existencia de un registro de consumo en el dia de hoy")

            if consumo_hoy:
                # Si ya existe, lo actualizamos
                consumo_hoy.consumo = consumo_total
                app.logger.info("Se ha actualizado el consumo diario")
            else:
                # Si no existe, lo creamos
                medida_consumo = MedidaConsumo(
                    fecha_medida=fecha_hoy,
                    consumo=consumo_total
                )
                session.add(medida_consumo)
                app.logger.info("Se ha creado un nuevo registro de consumo")

            # Confirmamos los cambios haciendo un commit
            session.commit()

            # Cerramos la sesión en la base de datos
            session.close()

            respuesta = {"estado": "OK", "mensaje": "Se ha aniadido la medida de potencia a la base de datos"}

    except SQLAlchemyError as e:
        app.logger.error(f"Se ha producido un error en una operación con la base de datos: {e}", exc_info=True)
        respuesta = {"estado":"ERROR", "mensaje":"Se ha producido un error en una operacion con la base de datos"}

        # En caso de error realizamos un rollback para deshacer los cambios
        session.rollback()
        session.close()

    return jsonify(respuesta)

```

Figura 4-26. Extracto de la función nueva_medida_potencia en rutas.py. Parte 2

En esta función, primero verificamos que el objeto JSON proporcionado en la petición tiene todos los atributos necesarios de una medida de potencia (fecha de la medida y valor de potencia). Luego, nos conectamos a la base de datos, creamos un objeto de la clase MedidaPotencia, y lo añadimos a la base de datos. A continuación, actualizamos el consumo del día de hoy. Para ello, obtenemos de la base de datos todas las medidas de potencia realizadas en el día actual, calculamos el consumo energético, y actualizamos el valor de consumo en la base de datos. Si no existía un registro de consumo para el día de hoy, creamos un objeto de la clase MedidaConsumo (ver Sección 4.4 Modelo de datos) y lo persistimos en la base de datos. De esta forma, el consumo del día actual se irá actualizando conforme se vayan obteniendo medidas de potencia, y cuando llegue el día siguiente, se creará un nuevo registro de consumo en la base de datos.

4.4 Modelo de datos

Como se comentó en la Sección 2.2.6 (SQLAlchemy), la librería SQLAlchemy nos permite interactuar con bases de datos relacionales mediante programación orientada a objetos, sin necesidad de utilizar directamente sentencias SQL. El fichero **modelos.py** contiene la definición de las clases ORM que nos permiten interactuar con las tablas explicadas en la Sección 4.1.2 (Creación de las tablas en la base de datos) siguiendo este enfoque. El contenido de este fichero se muestra en la figura 4-27:

```python
from sqlalchemy import Column, Integer, String, Date, DateTime, Float
from sqlalchemy.orm import declarative_base

### Definición del modelo de la base de datos ###

Base = declarative_base()

# Clase ORM (Object Relational Mapping) para las medidas de potencia
class MedidaPotencia(Base):

    # Nombre de la tabla en la base de datos
    __tablename__ = "potencia_shelly"

    # Columnas en la tabla
    id_medida = Column(Integer, primary_key=True)
    fecha_medida = Column(DateTime, nullable=False, unique=True)
    potencia = Column(Float, nullable=False)

# Clase ORM para las medidas de consumo energético
class MedidaConsumo(Base):

    # Nombre de la tabla en la base de datos
    __tablename__ = "consumo_shelly"

    # Columnas en la tabla
    id_medida = Column(Integer, primary_key=True)
    fecha_medida = Column(Date, nullable=False, unique=True)
    consumo = Column(Float, nullable=False)

# Clase ORM para los costes del consumo energético
class Coste(Base):

    # Nombre de la tabla en la base de datos
    __tablename__ = "coste_shelly"

    # Columnas en la tabla
    id_coste = Column(Integer, primary_key=True)
    fecha_coste = Column(Date, nullable=False, unique=True)
    coste = Column(Float, nullable=False)

```

Figura 4-27. Contenido del fichero modelos.py

En este fichero se definen tres clases ORM:

- **MedidaPotencia**: Relacionada con la tabla **potencia_shelly**.
- **MedidaConsumo**: Relacionada con la tabla **consumo_shelly**.
- **Coste**: Relacionada con la tabla **coste_shelly**.

4.5 Documentación OpenAPI mediante Flasgger

Para crear la documentación del servicio REST, se ha hecho uso de Flasgger [15], una extensión de Flask que permite realizar especificaciones OpenAPI interactivas de forma automática. En la Sección 4.2 (Rutas del servicio REST) ya se presentaron las rutas implementadas en este servicio, y se mostraron ejemplos de uso de dichas rutas gracias a la interfaz Swagger que nos proporciona Flasgger. Esta interfaz nos permite interactuar directamente con el servicio REST sin necesidad de realizar las peticiones HTTP GET, POST, PUT y DELETE utilizando las herramientas avanzadas del navegador o usando herramientas externas como Curl o Postman.

El fichero **__init__.py**, que sirve para indicar que el directorio **app** con la aplicación Flask es un paquete Python, define una función llamada **create_app** que nos permite crear instancias de la aplicación Flask desarrollada. Además, en esta función es donde realizamos la configuración de Flassger, especificando la versión de Swagger que queremos utilizar, y la descripción y versión del servicio REST desarrollado. En la figura 4-28 se muestra el contenido del fichero __init__.py:

```python
from flask import Flask
from flasgger import Swagger
from . import rutas

# Función para crear una instancia de la aplicación Flask
def create_app():
    app = Flask(__name__)

    # Configuración de la aplicación Flask

    # Evitamos que 'jsonify' ordene los campos por orden alfabético
    app.json.sort_keys = False

    # Registramos las rutas del servicio REST
    app.register_blueprint(rutas.bp)

    # Configuración de Swagger
    swagger_config = {
        "swagger_version": "2.0",
        "headers": [],
        "specs": [
            {
                "endpoint": 'apispec',
                "route": '/apispec_1.json',
            }
        ],
        "static_url_path": "/flasgger_static",
        "swagger_ui": True,
        "info": {
            "version": "1.0.0",
            "title": "API del servicio REST de consumo y costes",
            "description": "Especificación OpenAPI del servicio REST de consumo y costes utilizado en el sistema domótico."
        }
    }

    # Inicializamos Swagger en la aplicación
    Swagger(app, config=swagger_config)

    return app
```

Figura 4-28. Contenido del fichero __init__.py

Para realizar la documentación de la API utilizando Flassger, en las funciones de la aplicación Flask que manejan las rutas del servicio REST, debemos escribir comentarios en formato YAML con la información que queramos especificar sobre la ruta. En la figura 4-29 se muestran los comentarios en formato YAML utilizados para documentar la ruta **/shelly-plug/potencia/medidas** atendiendo a peticiones GET:

```python
# Ruta para obtener las medidas de potencia del Shelly Plug
@bp.route("/shelly-plug/potencia/medidas", methods=["GET"])
def obtener_medidas():
    """
    Ruta para obtener las medidas de potencia (vatios) del Shelly Plug en un intervalo horario específico o en la última hora si no se especifica.
    ---
    parameters:
      - name: hora_inicio
        in: query
        type: string
        required: false
        description: Hora de inicio del intervalo horario (formato HH:MM).
      - name: hora_fin
        in: query
        type: string
        required: false
        description: Hora de fin del intervalo horario (formato HH:MM).
    tags:
      - Potencia
    responses:
      200:
        description: Lista de medidas de potencia (en vatios) del Shelly Plug.
        content:
          application/json
    """

    respuesta = {}
    medidas = []

    try:
        # Obtenemos los parámetros de la URI
        hora_inicio = request.args.get("hora_inicio")
        hora_fin = request.args.get("hora_fin")
```

Figura 4-29. Ejemplos de comentarios Flassger para documentar las rutas del servicio REST

En estos comentarios podemos hacer una descripción de la funcionalidad que nos ofrece la ruta, los parámetros que recibe, establecer etiquetas para organizar las rutas en distintos grupos, o especificar las respuestas del

servidor. Tras documentar todas las rutas implementadas, Flassger generará de forma automática la documentación OpenAPI al lanzar la aplicación Flask. Para visualizar esta documentación, tenemos que acceder a la ruta **/apidocs** del servicio REST, como puede verse en la figura 4-30:

Figura 4-30. Acceso a la interfaz Swagger del servicio REST

Esta interfaz Swagger también permite interactuar con cada una de las rutas documentadas mediante el botón "**Try it out**". Además, si en los comentarios de las funciones de la aplicación Flask hemos especificado los parámetros de la ruta y su formato, podemos hacer peticiones más complejas que nos permiten probar el servicio REST de forma más completa, como puede verse en la figura 4-31:

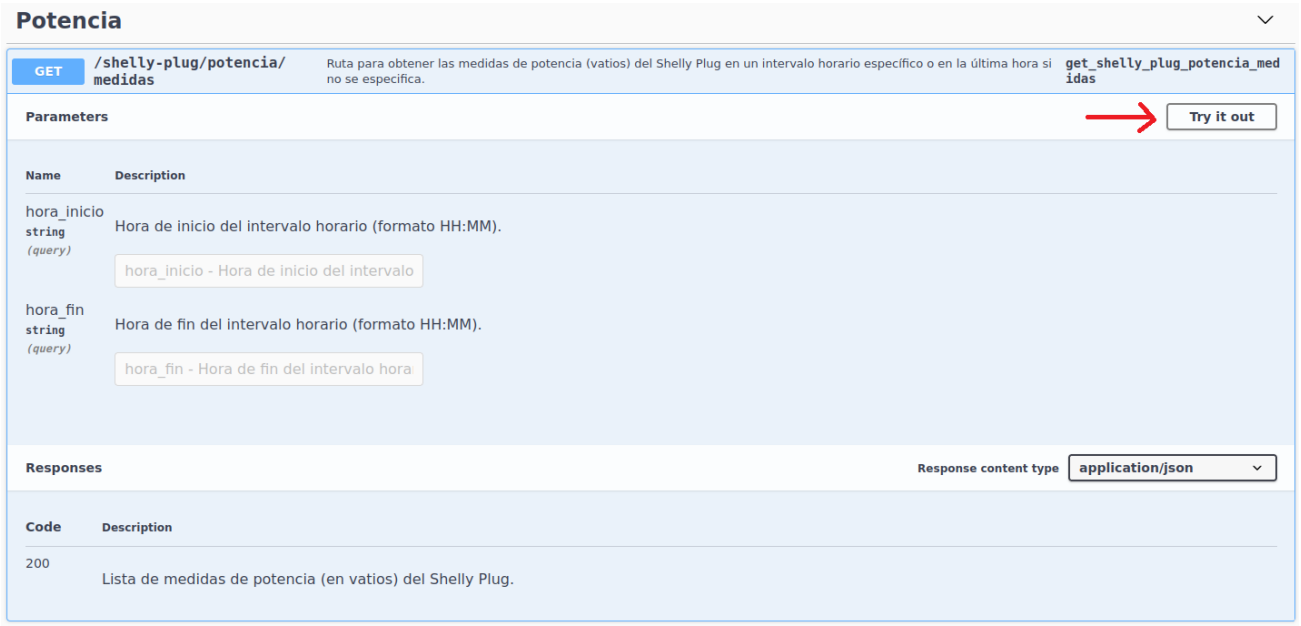


Figura 4-31. Interacción con las rutas del servicio REST mediante Swagger

4.6 Pruebas de integración del servicio REST

Para las pruebas de integración del servicio REST se ha utilizado Pytest [18], un *framework* de Python que nos permite realizar pruebas unitarias, de integración y funcionales de una aplicación Python. Para ello, se ha desarrollado un script Bash llamado **pruebas.sh**, que se encarga de crear una base de datos de pruebas en el contenedor PostgreSQL, crear las tablas y poblarlas de datos mediante el fichero **tablas_tests.sql**, ejecutar las pruebas con Pytest, y borrar la base de datos de pruebas. El contenido del fichero pruebas.sh se muestra en la figura 4-32:

```
#!/bin/bash

# Arrancamos el contenedor postgresql
docker compose up -d base_datos
echo "Esperamos unos segundos para que arranque la base de datos..."
sleep 3

# Creamos una base de datos de pruebas
PGPASSWORD=sergio createdb -h 127.0.0.1 -p 5432 -U sergio -O sergio pruebas

# Nos conectamos a la base de datos de pruebas y creamos las tablas
PGPASSWORD=sergio psql -h 127.0.0.1 -p 5432 -U sergio pruebas < ./python/tests/tablas_tests.sql

# Ejecutamos las pruebas con pytest
cd python
export PYTHONPATH=$(pwd)
pytest -v

# Borramos la base de datos de pruebas y paramos el contenedor postgresql
PGPASSWORD=sergio dropdb -h 127.0.0.1 -p 5432 -U sergio pruebas
docker compose stop base_datos

```

Figura 4-32. Contenido del fichero pruebas.sh

El contenido del fichero **tablas_tests.sql** es el mismo que el del fichero crear_tablas.sql de la figura 4-3, pero además incluimos sentencias para insertar datos de ejemplo para poder realizar las pruebas del servicio REST, como puede verse en la figura 4-33:

```
-- Añadimos datos de ejemplo

INSERT INTO potencia_shelly (fecha_medida, potencia) VALUES
    ('2024-01-06 15:15:19', 150.4),
    ('2024-01-06 15:15:49', 102.2),
    ('2024-01-06 15:16:19', 183.6),
    ('2024-01-06 15:16:49', 72.7),
    ('2024-01-06 15:17:19', 191.1),
    ('2024-01-06 15:17:49', 88.2),
    ('2024-01-06 15:18:19', 164.3);

INSERT INTO consumo_shelly (fecha_medida, consumo) VALUES
    ('12-31-2023', 960.2),
    ('01-01-2024', 879.3),
    ('01-02-2024', 1014.6),
    ('01-03-2024', 653.0),
    ('01-04-2024', 971.7),
    ('01-05-2024', 1121.5),
    ('01-06-2024', 548.2);

INSERT INTO coste_shelly (fecha_coste, coste) VALUES
    ('12-31-2023', 0.14403),
    ('01-01-2024', 0.131895),
    ('01-02-2024', 0.15219),
    ('01-03-2024', 0.09795),
    ('01-04-2024', 0.145755),
    ('01-05-2024', 0.168225),
    ('01-06-2024', 0.08223);

```

Figura 4-33. Inserción de datos de ejemplo en tablas_tests.sql

Una vez que la base de datos de pruebas está preparada, podemos comenzar con las pruebas del servicio REST. Para poder ejecutar las pruebas con Pytest, debemos escribir ficheros con el código de pruebas y nombrarlos con el prefijo "**test_**", para que Pytest pueda buscar estos ficheros y ejecutarlos automáticamente. Para las pruebas del servicio REST se ha desarrollado el fichero **test_rutas.py**, en el que realizamos pruebas en cada una de las rutas del servicio. Previamente, en este código de pruebas debemos crear la aplicación Flask de nuestro servicio, cambiar la configuración de conexión a la base de datos para que las pruebas se realicen en la base de datos de pruebas, y crear un cliente de pruebas, como puede verse en la figura 4-34:

```
from flask import Flask
from app import rutas
import pytest
from datetime import datetime

# Configuramos que la base de datos en los tests sea la de pruebas
rutas.DATABASE = "pruebas"

# Configuramos la dirección IP en la que está la base de datos
rutas.HOST = "127.0.0.1"

@pytest.fixture
def app():
    """
    Fixture para crear una aplicación Flask.
    """

    app = Flask(__name__)
    app.register_blueprint(rutas.bp)
    yield app

@pytest.fixture
def client(app):
    """
    Fixture para crear un cliente de prueba.
    """
    return app.test_client()

```

Figura 4-34. Configuración de la conexión a la base de datos de pruebas en test_rutas.py

En las pruebas desarrolladas utilizamos sentencias de tipo **assert** para verificar que las respuestas proporcionadas por el servicio REST son correctas, tanto en formato como en los valores que se deberían obtener. Como ejemplo, en la figura 4-35 se muestra la función de pruebas que se ha utilizado para probar la ruta **/shelly-plug/potencia/medidas** con el método GET:

```python
def test_obtener_medidas(client):
    """
    Test para la ruta /shelly-plug/potencia/medidas (GET).
    """

    # Intentamos obtener medidas de potencia en un intervalo horario
    respuesta = client.get("/shelly-plug/potencia/medidas?hora_inicio=12:00&hora_fin=13:00")
    assert respuesta.status_code == 200

    medidas = respuesta.get_json()

    # Comprobamos el formato de la respuesta y que no falte ningún atributo
    assert isinstance(medidas, list)
    assert all(isinstance(medida, dict) for medida in medidas)
    assert all("id_medida" in medida and "fecha_medida" in medida and "potencia" in medida for medida in medidas)

    for medida in medidas:
        id_medida = medida["id_medida"]
        assert isinstance(id_medida, int)

        fecha_medida = medida["fecha_medida"]
        assert isinstance(fecha_medida, str)
        assert datetime.strptime(fecha_medida, "%Y-%m-%d %H:%M:%S.%f")

        potencia = medida["potencia"]
        assert isinstance(potencia, float)

    # Intentamos obtener las medidas de potencia en la última hora

    respuesta = client.get("/shelly-plug/potencia/medidas")
    assert respuesta.status_code == 200

    medidas = respuesta.get_json()

    # Comprobamos el formato de la respuesta y que no falte ningún atributo
    assert isinstance(medidas, list)
    assert all(isinstance(medida, dict) for medida in medidas)
    assert all("id_medida" in medida and "fecha_medida" in medida and "potencia" in medida for medida in medidas)

    for medida in medidas:
        id_medida = medida["id_medida"]
        assert isinstance(id_medida, int)

        fecha_medida = medida["fecha_medida"]
        assert isinstance(fecha_medida, str)
        assert datetime.strptime(fecha_medida, "%Y-%m-%d %H:%M:%S.%f")

        potencia = medida["potencia"]
        assert isinstance(potencia, float)

    # Intentamos obtener las medidas en un formato de hora erróneo
    respuesta = client.get("/shelly-plug/potencia/medidas?hora_inicio=12_00&hora_fin=13_00")
    assert respuesta.status_code == 200

    datos = respuesta.get_json()
    assert datos == {"estado": "ERROR", "mensaje": "Parámetros de hora incorrectos"}

```

Figura 4-35. Función de pruebas de la ruta /shelly-plug/potencia/medidas (GET) en test_rutas.py

Una vez que tengamos el código de pruebas de nuestra aplicación, podemos ejecutar las pruebas con Pytest gracias al script **pruebas.sh** mostrado en la figura 4-32. Como resultado de la ejecución, Pytest nos mostrará un informe con los resultados de las pruebas, como puede verse en las figuras 4-36 y 4-37:

```
sergio@ubuntu:~/TFG/docker$ ./pruebas.sh
[+] Running 1/0
 ✔ Container postgresql  Running                                    0.0s
Esperamos unos segundos para que arranque la base de datos...
NOTICE:  table "potencia_shelly" does not exist, skipping
DROP TABLE
CREATE TABLE
NOTICE:  table "consumo_shelly" does not exist, skipping
DROP TABLE
CREATE TABLE
NOTICE:  table "coste_shelly" does not exist, skipping
DROP TABLE
CREATE TABLE
INSERT 0 7
INSERT 0 7
INSERT 0 7
============================= test session starts ==============================
platform linux -- Python 3.10.12, pytest-8.0.1, pluggy-1.4.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /home/sergio/TFG/docker/python
collected 12 items

tests/test_rutas.py::test_ping PASSED                                [  8%]
tests/test_rutas.py::test_obtener_medidas PASSED                     [ 16%]
```

Figura 4-36. Informe con los resultados de las pruebas del servicio REST. Parte 1

```
============================= test session starts ==============================
platform linux -- Python 3.10.12, pytest-8.0.1, pluggy-1.4.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /home/sergio/TFG/docker/python
collected 12 items

tests/test_rutas.py::test_ping PASSED                                [  8%]
tests/test_rutas.py::test_obtener_medidas PASSED                     [ 16%]
tests/test_rutas.py::test_nueva_medida_potencia PASSED               [ 25%]
tests/test_rutas.py::test_obtener_consumo PASSED                     [ 33%]
tests/test_rutas.py::test_obtener_consumo_por_hora PASSED            [ 41%]
tests/test_rutas.py::test_obtener_medidas_consumo PASSED             [ 50%]
tests/test_rutas.py::test_obtener_costes PASSED                      [ 58%]
tests/test_rutas.py::test_nuevo_coste PASSED                         [ 66%]
tests/test_rutas.py::test_borrar_medida_potencia PASSED              [ 75%]
tests/test_rutas.py::test_borrar_medidas_potencia PASSED             [ 83%]
tests/test_rutas.py::test_borrar_medidas_consumo PASSED              [ 91%]
tests/test_rutas.py::test_borrar_costes PASSED                       [100%]

============================== 12 passed in 1.82s ==============================
[+] Stopping 1/1
 ✔ Container postgresql  Stopped                                    0.5s
sergio@ubuntu:~/TFG/docker$
```

Figura 4-37. Informe con los resultados de las pruebas del servicio REST. Parte 2

4.7 Despliegue y configuración del servicio REST

Como se comentó previamente, este servicio REST se ha desarrollado como una aplicación Flask que se ha encapsulado en una imagen Docker personalizada. Dicha imagen se ha creado a partir de la imagen oficial de Ubuntu 22.04.

4.7.1 Creación de la imagen Docker del servicio REST

Para crear una nueva imagen Docker, debemos crear un fichero Dockerfile con las instrucciones que Docker necesita para crear esta imagen. En la figura 4-38 se muestra el Dockerfile que se ha usado para crear la imagen del servicio REST de consumo y costes:

```
  GNU nano 6.2                                  Dockerfile
# Creamos nuestra imagen a partir de la imagen de Ubuntu 22.04
FROM ubuntu:22.04

# Actualizamos e instalamos python3 y pip3
RUN apt update && apt install -y python3 && apt install -y python3-pip

# Configuramos la zona horaria del contenedor en una variable de entorno
# e instalamos el paquete tzdata para que se configure correctamente
ENV TZ=Europe/Madrid
RUN DEBIAN_FRONTEND=noninteractive apt install -y tzdata

# Creamos el directorio de trabajo
RUN mkdir /usr/src/python
RUN mkdir /usr/src/python/app
WORKDIR   /usr/src/python

# Copiamos el fichero con las librerias de python necesarias y las instalamos
COPY requirements.txt .
RUN pip3 install -r requirements.txt

# Copiamos los ficheros de la aplicación Flask
COPY run.py .
COPY app ./app

# Ejecutamos la aplicación
CMD [ "python3", "run.py"]
```

Figura 4-38. Dockerfile del servicio REST de consumo y costes

En este fichero Dockerfile hemos especificado las siguientes instrucciones:

- Crear la nueva imagen a partir de la imagen de **Ubuntu 22.04**.
- Actualizar los repositorios de paquetes del sistema e instalar **Python 3** y su gestor de paquetes **Pip3**.
- Configurar la variable de entorno **TZ** a la zona horaria de Madrid e instalar el paquete **tzdata** para poder establecer la zona horaria del sistema, que por defecto no viene instalado en la imagen de Ubuntu.
- Crear la estructura de directorios de la aplicación Flask y establecer el directorio de trabajo en **/usr/src/python**.
- Copiar el fichero **requirements.txt** en la imagen, que contiene las dependencias utilizadas por la aplicación Flask e instalarlas utilizando Pip3.
- Copiar los ficheros de la aplicación Flask en la imagen.
- Una vez que haya creado un contenedor a partir de la imagen y se haya puesto en ejecución, establecer que se ejecute el comando "**python3 run.py**" para arrancar la aplicación Flask.

Para especificar las dependencias de una aplicación desarrollada en Python, se suele hacer uso de un fichero **requirements.txt**. Para ello, podemos utilizar el comando "**pip3 freeze"** para obtener una lista con las librerías que tenemos instaladas en Python y sus versiones, y a partir de ahí seleccionar las librerías utilizadas por nuestra aplicación y escribirlas en dicho fichero. En la figura 4-39 se muestra el contenido del fichero requirements.txt con las dependencias de la aplicación Flask que se ha desarrollado:

```
blinker==1.6.2
click==8.1.7
Flask==3.0.0
itsdangerous==2.1.2
Jinja2==3.1.2
MarkupSafe==2.1.3
Werkzeug==3.0.0
psycopg2-binary==2.9.8
greenlet==3.0.2
SQLAlchemy==2.0.23
typing_extensions==4.9.0
flasgger==0.9.7.1
```

Figura 4-39: Contenido del fichero requirements.txt con las dependencias de la aplicación Flask

Por otro lado, en el fichero **run.py** creamos una instancia de la aplicación Flask desarrollada, que se encuentra dentro del paquete **app**, y ejecutamos la aplicación escuchando peticiones en el puerto 10000 y en todas las interfaces de la máquina. En la figura 4-40 se muestra el contenido del fichero run.py:

```
from app import create_app

# Creamos una instancia de la aplicación Flask
app = create_app()

if __name__ == "__main__":
    # Ejecutamos la aplicación en el puerto 10000
    app.run(host="0.0.0.0", port=10000, debug=False)

```

Figura 4-40. Contenido del fichero run.py

4.7.2 Despliegue del servicio REST

Al igual que con el resto de componentes software del sistema, se ha hecho uso de Docker-Compose para facilitar el despliegue y gestión del contenedor que ejecuta el servicio REST de consumo y costes. En la figura 4-41 se muestra un extracto del fichero docker-compose.yaml con la definición de dicho contenedor:

```
  servicio_rest:

    build: ./python

    container_name: python_rest

    image: python_rest

    ports:
      - "10000:10000"

    restart: always

    environment:
      - TZ=Europe/Madrid

```

Figura 4-41. Definición del contenedor con el servicio REST de consumo y costes en Docker-Compose

En la definición de este contenedor se especifica lo siguiente:

- Crear un servicio en Docker-Compose llamado **servicio_rest**.
- Construir la imagen del contenedor utilizando el fichero Dockerfile ubicado en el directorio **./python** (partiendo del directorio de trabajo donde está el fichero docker-compose.yaml). El contenido de dicho Dockerfile se muestra en la Sección 4.7.1 (Creación de la imagen Docker del servicio REST).
- Crear un contenedor llamado **python_rest**, asociado al servicio **servicio_rest**.
- Abrir el puerto 10000 de la máquina anfitriona y redirigirlo al puerto 10000 del contenedor, que es el puerto utilizado por la aplicación Flask del servicio REST.
- Reiniciar automáticamente el contenedor en caso de que se detenga, garantizando que el servicio se esté ejecutando en todo momento.
- Establecer la variable de entorno **TZ** (*Time Zone*) a la zona horaria de Madrid.

4.7.3 Configuración del servicio REST

El servicio REST implementado en este proyecto utiliza un fichero de configuración llamado **config.py** que permite configurar las variables utilizadas para la conexión con la base de datos. El contenido de este fichero se muestra en la figura 4-42:

```
1  # Constantes para la configuración de la conexión con la base de datos
2
3  HOST = "postgresql"
4  DATABASE = "monitor_consumo"
5  USER = "sergio"
6  PASSWORD = "sergio"
7  PORT = "5432"
8
```

Figura 4-42. Contenido del fichero de configuración config.py

Las variables definidas en este fichero son las siguientes:

- **HOST**: Permite configurar la dirección del *host* donde se encuentra la base de datos. Como en este sistema se está haciendo uso de un contenedor con una base de datos PostgreSQL, podemos hacer uso del servicio DNS interno de Docker-Compose y utilizar el nombre del contenedor, que en este caso es **postgresql**. De esta forma, facilitaremos el despliegue del servicio en entornos distintos al de desarrollo, como por ejemplo, en una máquina virtual en la nube.
- **DATABASE:** Nombre de la base de datos a la que queremos conectarnos, que en este caso es **monitor_consumo**.
- **USER:** Nombre de usuario en la base de datos.
- **PASSWORD:** Contraseña del usuario especificado.
- **PORT:** Puerto utilizado por la base de datos, que en este caso es el **5432**.

5 DESPLIEGUE DEL SISTEMA EN MICROSOFT AZURE

Las grandes oportunidades nacen de haber sabido aprovechar las pequeñas.

- Bill Gates -

En este capítulo es donde explicamos el proceso de despliegue del sistema domótico en una máquina virtual en la nube de Microsoft Azure, así como aspectos de operación del sistema realizados mediante Docker-Compose. Entre otras opciones disponibles, como Amazon Web Services o Google Cloud Platform, se ha elegido Azure por las facilidades que ofrece en el despliegue y gestión de máquinas virtuales, y por los acuerdos alcanzados entre la Universidad de Sevilla y Microsoft, que permiten a los estudiantes utilizar estos servicios de forma gratuita gracias a la licencia Azure for Students.

5.1 Creación de la máquina virtual en Azure

Una vez que hayamos activado nuestra cuenta de estudiante [45], en el portal de Azure [46] tendremos disponible un panel de control que nos permitirá acceder a los distintos servicios ofrecidos por la plataforma, así como información acerca de nuestros recursos, suscripciones, facturación, monitorización, seguridad, etc. En la figura 5-1 se muestra una imagen del panel de control del portal de Azure:

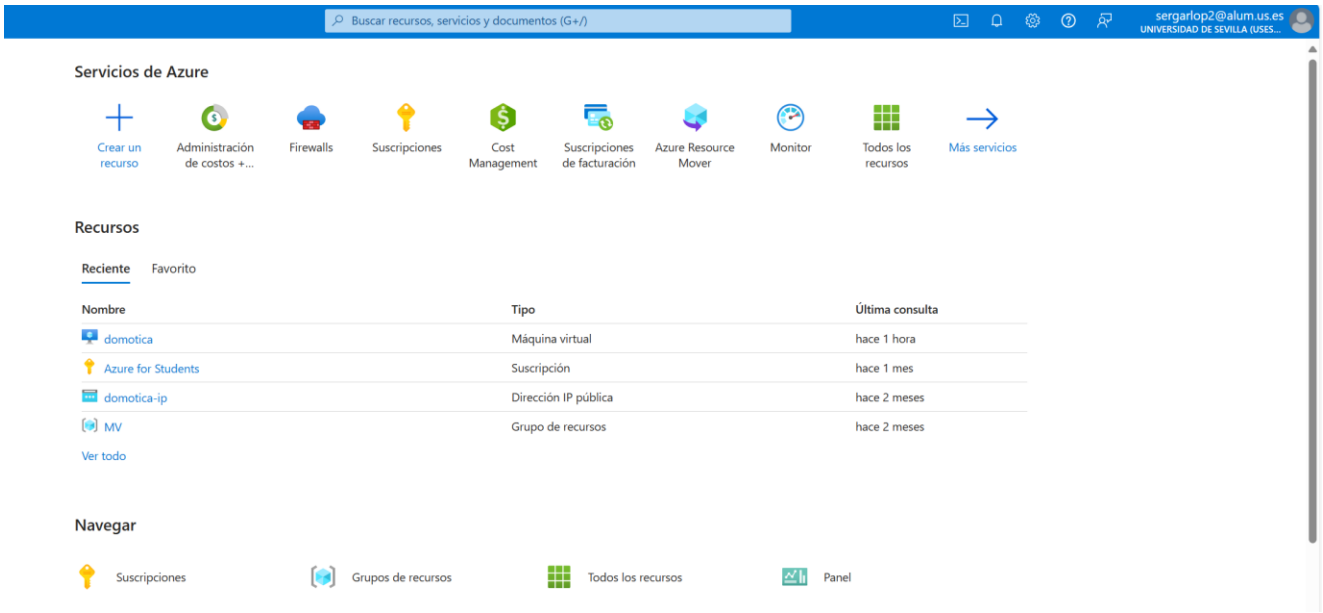


Figura 5-1. Panel de control del portal de Azure

Para crear un nuevo recurso basta con seleccionar la opción “Crear un recurso” y elegir el tipo de servicio de que deseamos, como puede verse en la figura 5-2. Entre las opciones disponibles está la creación de máquinas virtuales, aplicaciones web, servidores Windows o Linux, bases de datos, herramientas de inteligencia artificial, contenedores, etc. En nuestro caso, seleccionaremos la opción de crear una máquina virtual.

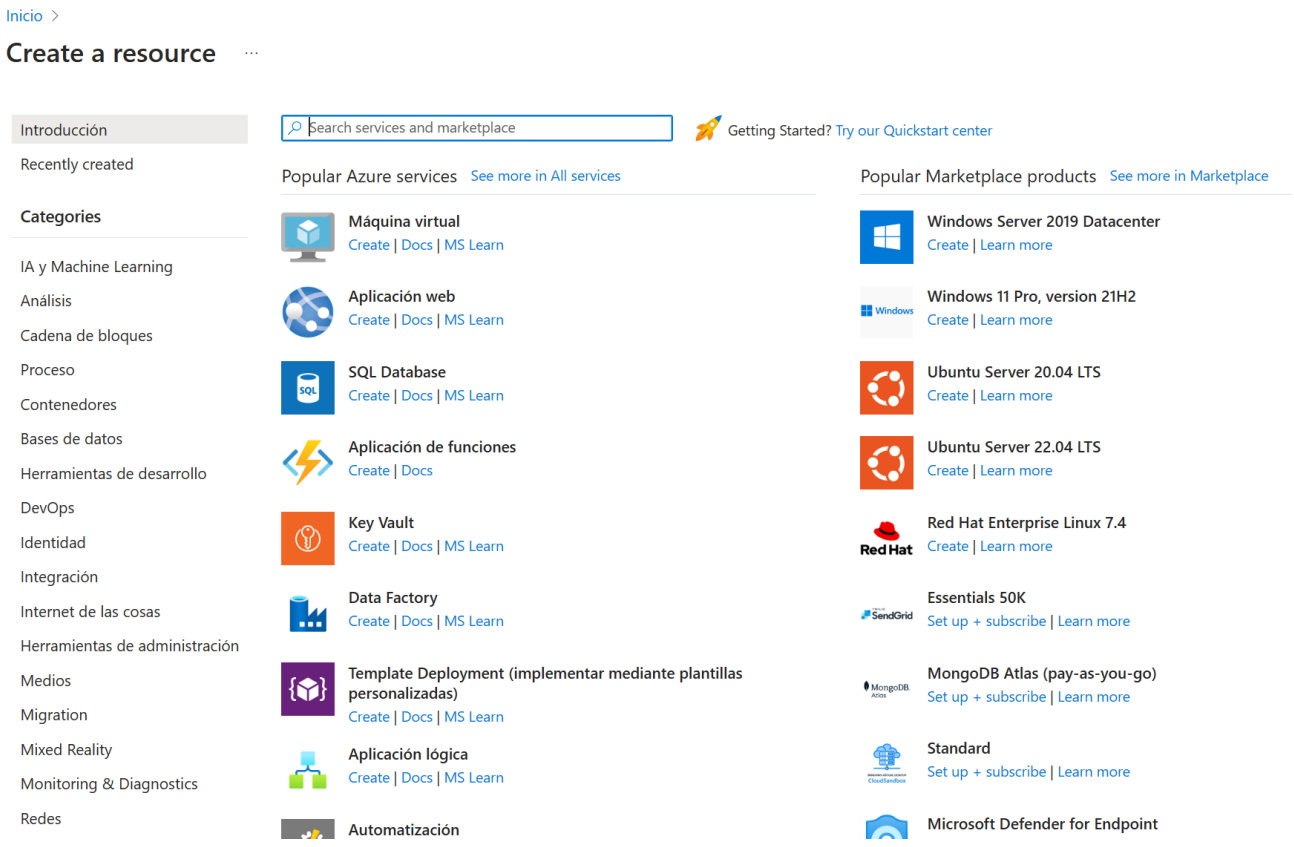


Figura 5-2. Servicios ofrecidos por Azure

Tras seleccionar dicha opción, se nos mostrarán las opciones de creación de la máquina virtual, tal como se muestra en la figura 5-3. En estas opciones, debemos especificar el tipo de suscripción que tenemos (en nuestro caso, Azure for Students), el nombre que le daremos a la máquina virtual, la región en la que queremos desplegarla, el sistema operativo, el número de CPUs, la cantidad de memoria, y el almacenamiento, entre otras configuraciones. Además, al configurar el usuario administrador, podemos generar un par de claves SSH para conectarnos de forma remota a la máquina virtual sin necesidad de configurar una contraseña.

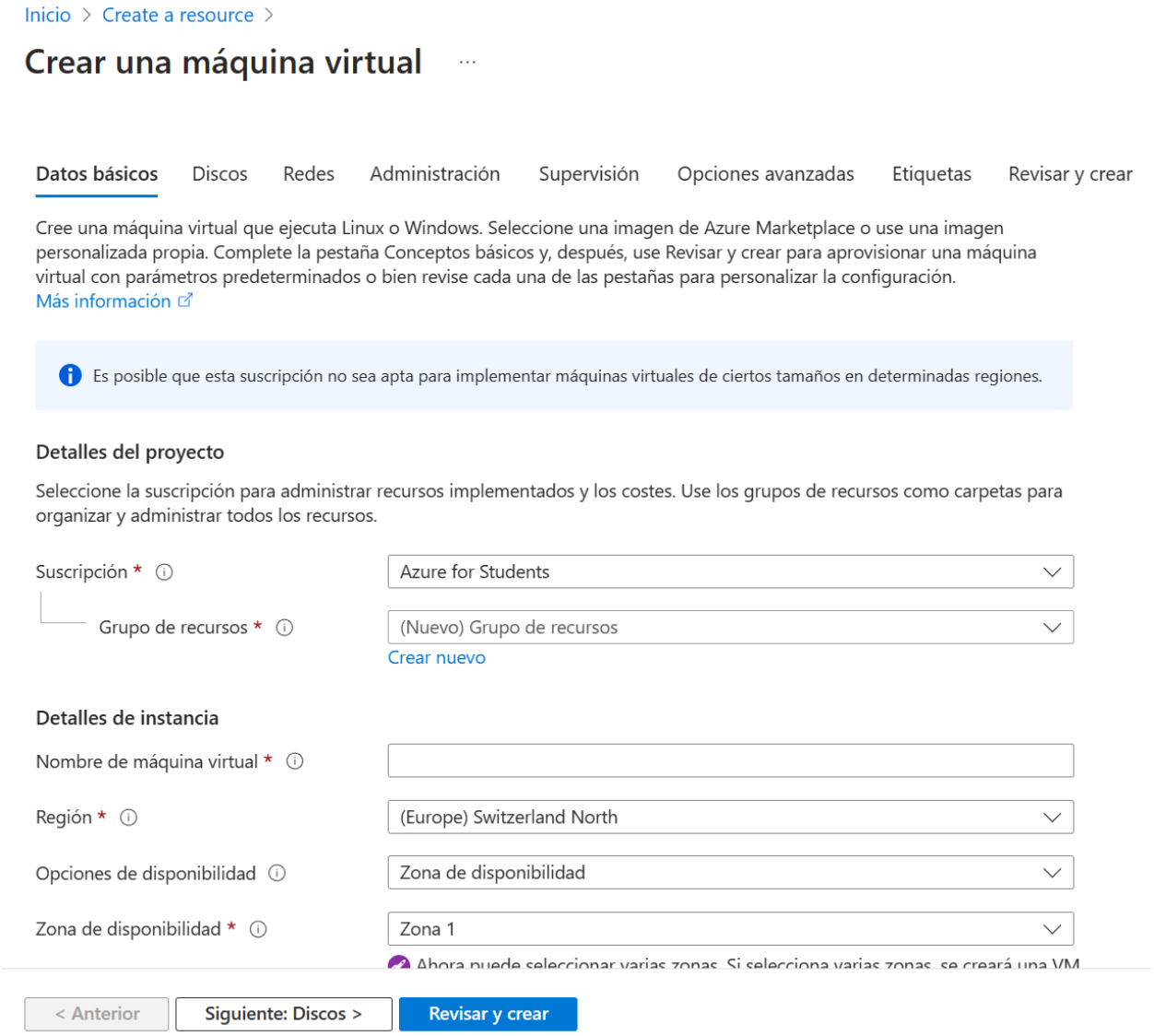


Figura 5-3. Opciones de creación de una máquina virtual en Azure

Una vez creada, en los recursos del portal de Azure mostrados en la figura 5-1 aparecerá nuestra máquina virtual. Al seleccionarla, se abrirá una nueva ventana en la que podremos ver información sobre la máquina virtual y realizar acciones sobre ella, como iniciarla, detenerla o reiniciarla a través de los botones disponibles, como podemos ver en la figura 5-4. Además, se nos ofrece una amplia gama de opciones que nos permiten monitorizar el uso de recursos de la máquina virtual, conocer detalles de facturación por horas de uso, realizar *backups*, y configurar las reglas de entrada y salida del *firewall*, entre otras funcionalidades.

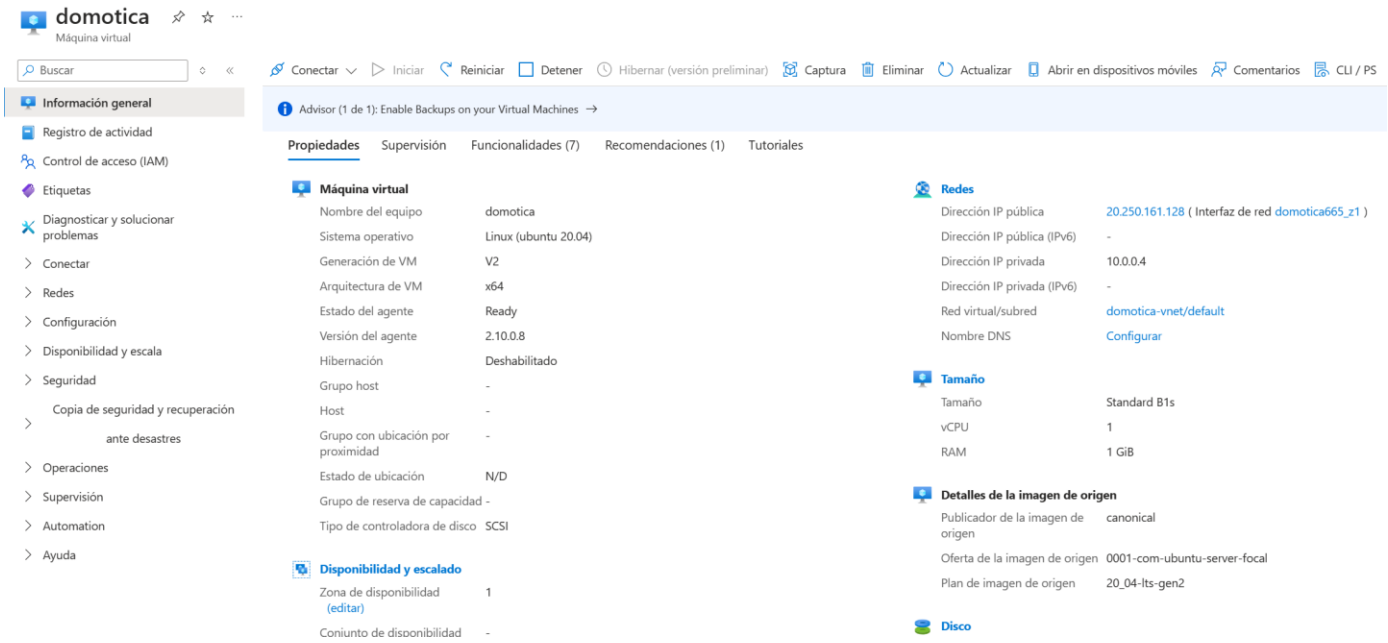


Figura 5-4. Detalles de la máquina virtual creada en Azure

Para este proyecto, se ha creado una máquina virtual llamada **domotica** con sistema operativo Ubuntu 20.04, una CPU virtual, 1 GB de RAM, 30 GB de almacenamiento, y se le ha asignado la IP pública 20.250.161.128.

Además, para poder utilizar la interfaz de usuario del sistema domótico y que el enchufe inteligente pueda publicar las medidas de potencia en el broker MQTT, debemos añadir nuevas reglas de entrada en el *firewall* de la máquina virtual. Para ello, accedemos a la sección "Redes" y añadimos las reglas mostradas en la figura 5-5. Estas reglas permiten conexiones entrantes por el puerto 22 para poder conectarnos a la máquina mediante SSH, así como por los puertos 443 y 1883 para aceptar tráfico HTTPS y MQTT, respectivamente.

Figura 5-5. Reglas de entrada del firewall de la máquina virtual

5.2 Despliegue y operación del sistema domótico

Una vez que la máquina virtual esté operativa, para desplegar el sistema domótico podemos conectarnos mediante SSH y clonar el repositorio GitHub del proyecto, que se encuentra disponible en el siguiente enlace:

https://github.com/sergarlop2/TFG

Tras clonar el repositorio, en el directorio **TFG/docker** encontraremos el fichero **docker-compose.yaml**, que nos permitirá crear y gestionar los contenedores del sistema domótico utilizando Docker-Compose, como podemos ver en la figura 5-6:

```
sergio@domotica:~$ ls
TFG
sergio@domotica:~$ cd TFG/docker/
sergio@domotica:~/TFG/docker$ ls
base_datos.sh  crear_tablas.sql  docker-compose.yaml  mosquitto  nodered  postgresql  pruebas.sh  python
sergio@domotica:~/TFG/docker$
```

Figura 5-6. Ficheros del directorio TFG/docker

Para arrancar el sistema domótico nos situamos en el directorio en el que está el fichero docker-compose.yaml y ejecutamos el comando "**docker compose up**". De forma alternativa, podemos ejecutar el comando con la opción "**-d**" para ejecutar los contenedores en segundo plano, como podemos ver en la figura 5-7:

```
sergio@domotica:~/TFG/docker$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
sergio@domotica:~/TFG/docker$ docker compose up -d
[+] Running 4/4
 ✔Container nodered      Started                                                                    0.0s
 ✔Container mosquitto    Started                                                                    0.0s
 ✔Container python_rest  Started                                                                    0.0s
 ✔Container postgresql   Started                                                                    0.0s
sergio@domotica:~/TFG/docker$ docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED       STATUS                            PO
RTS                                                                                  NAMES
47e2c550de23   python_rest                    "python3 run.py"         2 weeks ago   Up 5 seconds                      0.
0.0.0:10000->10000/tcp, :::10000->10000/tcp                                          python_rest
c5656a2cecb8   eclipse-mosquitto:latest       "/docker-entrypoint.…"   8 weeks ago   Up 5 seconds                      0.
0.0.0:1883->1883/tcp, :::1883->1883/tcp, 0.0.0.0:9001->9001/tcp, :::9001->9001/tcp   mosquitto
f15478990750   postgres:latest                "docker-entrypoint.s…"   8 weeks ago   Up 5 seconds                      0.
0.0.0:5432->5432/tcp, :::5432->5432/tcp                                              postgresql
5af906240fc3   nodered/node-red:3.1-debian    "./entrypoint.sh"        8 weeks ago   Up 5 seconds (health: starting)   0.
0.0.0:443->1880/tcp, :::443->1880/tcp                                                nodered
sergio@domotica:~/TFG/docker$
```

Figura 5-7. Arranque del sistema domótico utilizando Docker-Compose

Si es la primera vez que arrancamos el sistema domótico, debemos ejecutar el script **base_datos.sh** explicado en la Sección 4.1.2 (Creación de las tablas en la base de datos) para crear las tablas en la base de datos del contenedor PostgreSQL. En posteriores rearranques esto no será necesario, ya que la información de la base de datos del contenedor se persiste en el sistema de ficheros de la máquina anfitriona, como se explicó en la Sección 4.1.1 (Despliegue de la base de datos). Por otro lado, no será necesario modificar la configuración del servicio REST ni la de los nodos de NodeRED, ya que a pesar de que el sistema está desplegado en un nuevo entorno en el que las direcciones IP de los contenedores o de la máquina anfitriona pueden ser diferentes, al utilizar el servicio DNS interno de Docker Compose no tenemos que hacer ningún cambio, y podemos reutilizar la configuración usada en el entorno de desarrollo.

Tras poner en funcionamiento el sistema domótico, al navegar a la dirección https://20.250.161.128/, tendremos acceso a la interfaz de usuario de NodeRED de forma totalmente remota, como podemos ver en la figura 5-8:

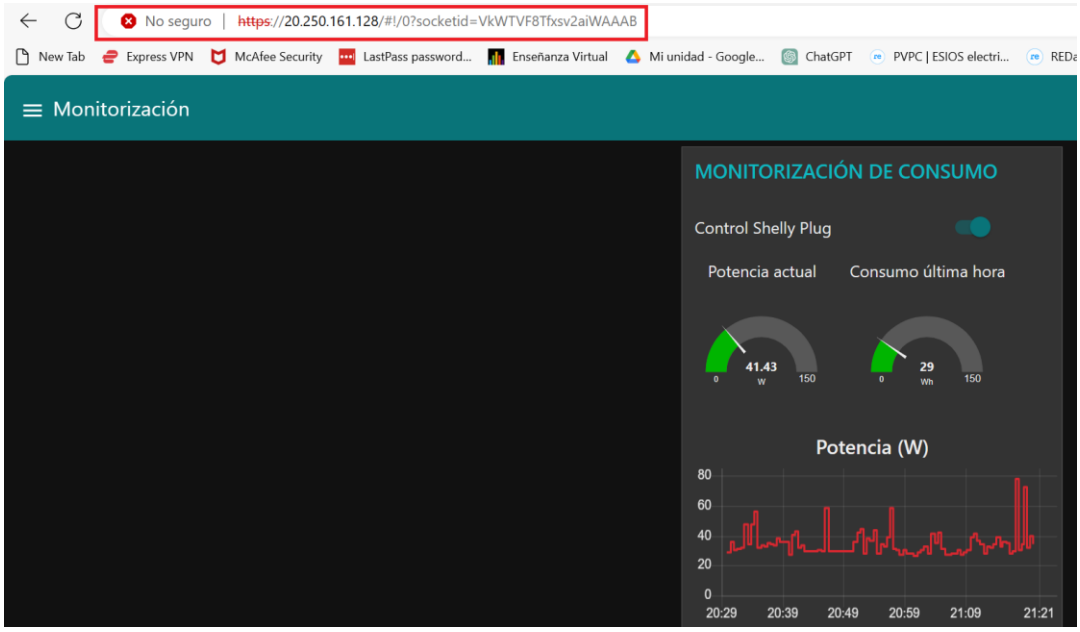


Figura 5-8. Acceso a la interfaz de usuario de NodeRED de forma remota

Además, Docker-Compose proporciona una gran cantidad de comandos que pueden ser de gran utilidad a la hora de operar y monitorizar los contenedores del sistema. Por ejemplo, en la figura 5-9 se muestra el uso del comando "**docker compose logs**", que nos permite ver los logs generados por cada uno de los contenedores que tenemos desplegados:

```
sergio@domotica:~/TFG/docker$ docker compose logs --tail=10 servicio_rest
python_rest  | 172.18.0.5 - - [01/May/2024 21:09:05] "GET /shelly-plug/consumo HTTP/1.1" 200 -
python_rest  | 172.18.0.5 - - [01/May/2024 21:09:06] "GET /shelly-plug/consumo?hora_inicio=19:00&hora_fin=23:59 HTTP/1.1
" 200 -
python_rest  | 172.18.0.5 - - [01/May/2024 21:09:06] "GET /shelly-plug/consumo?hora_inicio=06:00&hora_fin=11:59 HTTP/1.1
" 200 -
python_rest  | 172.18.0.5 - - [01/May/2024 21:09:06] "GET /shelly-plug/costes HTTP/1.1" 200 -
python_rest  | 172.18.0.5 - - [01/May/2024 21:09:06] "GET /shelly-plug/consumo/medidas HTTP/1.1" 200 -
python_rest  | 172.18.0.5 - - [01/May/2024 21:09:06] "GET /shelly-plug/consumo?hora_inicio=00:00&hora_fin=05:59 HTTP/1.1
" 200 -
python_rest  | 172.18.0.5 - - [01/May/2024 21:09:06] "GET /shelly-plug/consumo?hora_inicio=12:00&hora_fin=18:59 HTTP/1.1
" 200 -
python_rest  | 172.18.0.5 - - [01/May/2024 21:09:06] "GET /shelly-plug/consumo/consumo-por-hora HTTP/1.1" 200 -
python_rest  | 172.18.0.5 - - [01/May/2024 21:09:06] "POST /shelly-plug/costes HTTP/1.1" 200 -
python_rest  | 172.18.0.5 - - [01/May/2024 21:09:12] "POST /shelly-plug/potencia/medidas HTTP/1.1" 200 -
sergio@domotica:~/TFG/docker$
```

Figura 5-9. Consulta de los logs de los contenedores utilizando Docker Compose

Finalmente, para reiniciar el sistema domótico podemos usar el comando "**docker compose restart**", y para detenerlo podemos usar "**docker compose stop**", como podemos ver en la figura 5-10:

```
sergio@domotica:~/TFG/docker$ docker compose restart
[+] Restarting 4/4
 ✓Container nodered      Started                                    4.1s
 ✓Container mosquitto    Started                                    4.1s
 ✓Container python_rest  Started                                   11.9s
 ✓Container postgresql   Started                                    4.1s
sergio@domotica:~/TFG/docker$ docker compose stop
[+] Stopping 4/4
 ✓Container nodered      Stopped                                    2.3s
 ✓Container mosquitto    Stopped                                    1.0s
 ✓Container python_rest  Stopped                                   10.2s
 ✓Container postgresql   Stopped                                    1.1s
sergio@domotica:~/TFG/docker$
```

Figura 5-10. Reinicio y parada del sistema domótico utilizando Docker Compose

6 CONCLUSIONES Y LÍNEAS FUTURAS

El mundo que hemos creado es un proceso de nuestro pensamiento. No se puede cambiar sin cambiar nuestra forma de pensar.

- Albert Einstein -

Como capítulo final de esta memoria, haremos una reflexión acerca de los objetivos que se han alcanzado tras la finalización del proyecto, destacando las lecciones aprendidas durante todo el proceso de investigación, diseño y desarrollo, así como las dificultades que se han ido encontrando y cómo se han superado. Además, en base a los resultados y las experiencias obtenidas, exploraremos una serie de líneas futuras que permitirán mejorar y ampliar el sistema domótico presentado en este Trabajo Fin de Grado.

6.1 Conclusiones y lecciones aprendidas

Tras la realización de este proyecto, podemos afirmar que se han alcanzado todos los objetivos que se propusieron al comienzo de este. Como resultado, se ha conseguido desarrollar un sistema domótico de control y monitorización del consumo eléctrico que cumple con una serie de funcionalidades atractivas para aquellas personas que quieran tener un mayor control acerca de su consumo y de los costes correspondientes. Considero que no solo se han conseguido los objetivos marcados desde el punto de vista académico, sino que se ha construido un sistema que aporta valor a todos aquellos consumidores que tienen contratada la tarifa regulada PVPC, y que a partir de este proyecto podría desarrollarse un producto que fuera competitivo en el mercado.

La elección de NodeRED como nodo central de la arquitectura ha sido uno de los mayores éxitos de este proyecto, no solo por su capacidad de automatización de procesos e interconexión con distintas entidades, sino por el amplio grado de aplicación que tiene esta herramienta, que cada vez se va utilizando más en todo tipo de proyectos relacionados con el IoT y la domótica. Para mí ha supuesto un reto aprender esta nueva tecnología, puesto que está basada en un paradigma de programación totalmente distinto al que estoy acostumbrado, pero considero que todo este esfuerzo ha merecido la pena, porque probablemente me será de gran utilidad en el mundo laboral. Por otro lado, considero que otra buena decisión fue la de realizar la interfaz de usuario desde el propio NodeRED. Su rapidez de creación, así como la gran cantidad de elementos gráficos que ofrece para la visualización de datos, y su fácil integración en los flujos de NodeRED, me han permitido crear una interfaz intuitiva, estética y muy apropiada para el tipo de problema que se está resolviendo. De otra manera, por ejemplo realizando una aplicación móvil, habría tardado mucho más en crear la interfaz, y probablemente me habrían surgido una gran cantidad de problemas a la hora de intentar integrarla con el resto del sistema.

La decisión de diseño de crear un servicio REST para el almacenamiento y gestión de los datos producidos por el sistema domótico fue otro gran acierto. En un principio consideré que desde el propio NodeRED podría ser

capaz de realizar todas estas tareas de persistencia, puesto que existen módulos en NodeRED que permiten la conexión e interacción con bases de datos. Sin embargo, una de las virtudes de NodeRED, que es la simplicidad de su programación mediante la configuración y conexión de elementos gráficos, es a su vez un defecto a la hora de implementar estas funcionalidades avanzadas, puesto que la complejidad y dificultad de depuración crece de forma exponencial. NodeRED resulta de gran utilidad a la hora de manejar flujos de datos e interconectar sistemas, pero no es la herramienta más adecuada para implementar funcionalidades más complejas, teniendo que recurrir por lo tanto a un lenguaje de programación tradicional. No obstante, esto ha supuesto una oportunidad para profundizar mis conocimientos sobre la implementación de servicios REST, y de forma más concreta, para aprender más en el desarrollo de proyectos Flask. Enfrentarme a la implementación de un servicio de este tipo de complejidad me ha permitido aprender mucho, no solo desde el punto de vista de la codificación, sino también en el diseño, documentación y realización de pruebas.

Por otro lado, considero que la interacción del sistema domótico con otros sistemas externos, que en este caso es el servicio REST de Red Eléctrica, aporta un gran valor al proyecto. Durante el proceso encontré una buena cantidad de dificultades a la hora de manejar su API, realizar peticiones y procesar las respuestas, pero la inclusión de toda la información de precios y costes de la electricidad en el sistema final añade un gran atractivo al proyecto, justificando todo ese esfuerzo.

Otra de las virtudes del proyecto ha sido haber planteado la arquitectura del sistema en microservicios y haber utilizado contenedores Docker desde el comienzo de la implementación. Con este enfoque, se han identificado cuáles debían ser los componentes fundamentales del sistema, y se ha realizado la correspondiente separación de responsabilidades, pensando además en aspectos de portabilidad durante el proceso de implementación. Este factor, junto a las ventajas de virtualización de Docker, ha simplificado el despliegue del sistema final en la nube de Azure, minimizando al máximo las diferencias entre el entorno de desarrollo y el de producción. De igual manera, el uso de Docker-Compose ha facilitado enormemente la operación coordinada de los servicios del sistema domótico. Además, haber desplegado el sistema final en la nube en vez de haberlo mantenido en la red local del hogar no solo permite el acceso remoto a la interfaz de usuario desde cualquier ubicación, sino que también abre la posibilidad de instalar el enchufe en diversas localizaciones. Este despliegue también me ha servido para aprender más acerca de la nube, aplicando dichos conocimientos para el caso particular de Azure.

Finalmente, este proyecto me ha permitido adquirir nuevos conocimientos en los campos de la domótica y el IoT, desde el uso de protocolos como MQTT para interconectar dispositivos y sistemas, hasta aspectos de automatización, control, monitorización y gestión de los datos producidos en este tipo de entornos.

6.2 Líneas futuras

En base a los resultados y experiencias obtenidos del proyecto, a continuación, enumeramos los distintos aspectos que podrían mejorarse, así como nuevas funcionalidades o ampliaciones del sistema domótico:

- **Posibilidad de añadir nuevos enchufes:** Actualmente, solo se realiza la monitorización y control del consumo medido en un único enchufe. No obstante, para ampliar el sistema, se podría considerar la inclusión de nuevos enchufes. Para ello, sería necesario crear un formulario que registre tanto el identificador del enchufe como los tópicos MQTT utilizados por el dispositivo, permitiendo así realizar un control y seguimiento de los enchufes de forma independiente. Además, también se podría registrar cualquier otra información relevante, como localización, modelo, potencia máxima admitida, fecha de instalación, etc.
- **Ampliación a nuevos dispositivos domóticos:** En concordancia con el punto anterior, el sistema podría expandirse no solo a enchufes inteligentes, sino a cualquier tipo de dispositivo domótico que permita su control a través de MQTT o que proporcione información sobre su estado y funcionamiento a través de este protocolo.
- **Mejora de la seguridad en las comunicaciones MQTT:** El enchufe Shelly Plug S utilizado en el sistema no permite utilizar ningún mecanismo de cifrado en las comunicaciones realizadas mediante MQTT. Por lo tanto, en el broker Mosquitto no se ha podido configurar el uso del protocolo TLS, lo

cual es una deficiencia de seguridad que se debe a estas limitaciones en el hardware. Sin embargo, es probable que los nuevos modelos del Shelly Plug incluyan soporte para TLS. Además, podría considerarse la configuración del control de acceso al broker mediante usuario y contraseña, pero nuevamente este es un aspecto que depende de que el hardware admita estas funcionalidades.

- **Mejora de la seguridad en NodeRED:** Tal y como está planteado el sistema, buena parte de su seguridad depende de los mecanismos ofrecidos por NodeRED, que en algunos casos pueden ser limitados. A pesar de que se ha configurado el uso de HTTPS y de que el acceso al editor de NodeRED está protegido mediante usuario y contraseña, podría añadirse autenticación en la interfaz de usuario. Para ello, lo más conveniente sería utilizar algún tipo de proxy inverso, como por ejemplo Nginx, que se encargue de la autenticación y pueda implementar mecanismos de seguridad más avanzados que los de NodeRED.
- **Control automático del enchufe en función de los precios de la electricidad:** Gracias a la información de precios de la electricidad obtenida del servicio REST de Red Eléctrica, podría programarse un algoritmo en NodeRED que encendiera y apagara el enchufe de forma automática en función de unos umbrales de coste configurados por el usuario.
- **Añadir reglas de control personalizadas:** Continuando con la idea del punto anterior, podría implementarse algún mecanismo para añadir reglas de control personalizadas, por ejemplo, para encender el enchufe en determinadas horas del día y apagarlo en las horas en las que no se requiera el consumo de energía.
- **Añadir filtros en la interfaz para mejorar la visualización de datos:** Actualmente, en los históricos de consumo y costes solo es posible visualizar los datos organizados en días, obteniendo además todos los registros del histórico en cada consulta. Por lo tanto, en la interfaz de usuario se podría implementar una función de filtrado que permita, por ejemplo, obtener los datos de un mes específico, y desglosar los datos en días, semanas o meses, según lo indique el usuario.
- **Análisis de los datos registrados por el sistema:** A partir de los datos registrados por el sistema, podrían generarse estadísticas tales como consumo medio, mínimo y máximo, análisis de periodicidad, correlación de los datos, detección de anomalías, etc. Además, mediante técnicas de inteligencia artificial, se podrían detectar tendencias y patrones de consumo, e incluso realizar predicciones sobre el consumo futuro de los usuarios.
- **Mejor gestión de los datos almacenados:** Se podrían añadir nuevas opciones de borrado y actualización de los datos almacenados en el sistema. Por ejemplo, actualizar y borrar registros de días específicos, borrar datos a partir de una cierta fecha de antigüedad, o borrar datos de rangos de fechas personalizados.
- **Uso de DNS para la máquina virtual:** Se podría asociar un nombre de dominio a la IP pública de la máquina virtual para facilitar la navegación a la interfaz de usuario de NodeRED. Para ello, se podrían utilizar servicios gratuitos de DNS dinámico, como DuckDNS, u optar por opciones de pago ofrecidas por el propio Azure o por registradores de dominios.
- **Uso de certificados digitales válidos en NodeRED:** Para poder utilizar HTTPS en la interfaz de usuario de NodeRED, se ha generado un certificado x509 autofirmado que no está reconocido por ninguna CA, y por lo tanto, al acceder a la interfaz mediante un navegador aparece un aviso de seguridad. Para solucionar esto, podríamos obtener un certificado de forma gratuita mediante plataformas como Let's Encrypt u obtenerlo de proveedores de certificados de pago.

REFERENCIAS

[1] M. Mena Roa, «Hogares inteligentes: un futuro prometedor,» [En línea]. Available: https://es.statista.com/grafico/27074/numero-de-dispositivos-domesticos-inteligentes-en-todo-el-mundo/.

[2] Mordor Intelligence, «Tamaño del mercado de dispositivos IoT y análisis de participación: tendencias y pronósticos de crecimiento (2024 - 2029),» [En línea]. Available: https://www.mordorintelligence.ar/industry-reports/iot-devices-market.

[3] Shelly, «Shelly Plug S - enchufe WiFi,» [En línea]. Available: https://shellyspain.com/shelly-plug-s-enchufe-wifi.html.

[4] NodeRED, «NodeRED, Low-code programming for event-driven applications,» [En línea]. Available: https://nodered.org/.

[5] Red Eléctrica de España, «REData API,» [En línea]. Available: https://www.ree.es/es/apidatos.

[6] «Precio voluntario para el pequeño consumidor (PVPC),» [En línea]. Available: https://www.ree.es/es/actividades/operacion-del-sistema-electrico/precio-voluntario-pequeno-consumidor-pvpc.

[7] Docker, «Docker,» [En línea]. Available: https://www.docker.com/.

[8] Docker, «Docker Compose overview,» [En línea]. Available: https://docs.docker.com/compose/.

[9] Microsoft, «Azure,» [En línea]. Available: https://azure.microsoft.com/es-es/.

[10] B. M. Lozano Jurado. [En línea]. Available: https://idus.us.es/handle/11441/149464.

[11] Omen, «Portátil gaming OMEN 16,» [En línea]. Available: https://www.omen.com/es/es/laptops/2023-omen-16-intel.html.

[12] MQTT, «MQTT: The Standard for IoT Messaging,» [En línea]. Available: https://mqtt.org/.

[13] Mosquitto, «Eclipse Mosquitto,» [En línea]. Available: https://mosquitto.org/.

[14] Flask, «Documentación de Flask,» [En línea]. Available: https://flask.palletsprojects.com/en/3.0.x/.

[15] Flassger, «GitHub Flassger,» [En línea]. Available: https://github.com/flasgger/flasgger.

[16] OpenAPI, «OpenAPI,» [En línea]. Available: https://www.openapis.org/.

[17] SQLAlchemy, «SQLAlchemy,» [En línea]. Available: https://www.sqlalchemy.org/.

[18] Pytest, «Pytest,» [En línea]. Available: https://docs.pytest.org/en/8.0.x/.

[19] PostgreSQL, «PostgreSQL,» [En línea]. Available: https://www.postgresql.org/.

[20] Canonical, «Ubuntu,» [En línea]. Available: https://ubuntu.com/.

[21] Oracle, «Virtualbox,» [En línea]. Available: https://www.virtualbox.org/.

[22] Wireshark, «Wireshark,» [En línea]. Available: https://www.wireshark.org/.

[23] Sublime Text, «Sublime Text,» [En línea]. Available: https://www.sublimetext.com/.

[24] HiveMQ, «Free Public MQTT Broker,» [En línea]. Available: https://www.hivemq.com/mqtt/public-mqtt-broker/.

[25] Docker Hub, «Docker Hub,» [En línea]. Available: https://hub.docker.com/.

[26] Docker Hub, «Eclipse Mosquitto Image,» [En línea]. Available: https://hub.docker.com/_/eclipse-mosquitto/.

[27] NodeRED, «Running under Docker,» [En línea]. Available: https://nodered.org/docs/getting-started/docker.

[28] NodeRED, «Configuration,» [En línea]. Available: https://nodered.org/docs/user-guide/runtime/configuration.

[29] NodeRED, «Securing NodeRED,» [En línea]. Available: https://nodered.org/docs/user-guide/runtime/securing-node-red.

[30] Wikipedia, «Bcrypt algorithm,» [En línea]. Available: https://en.wikipedia.org/wiki/Bcrypt.

[31] Microsoft, «X.509 certificates,» [En línea]. Available: https://learn.microsoft.com/en-us/azure/iot-hub/reference-x509-certificates.

[32] OpenSSL, «OpenSSL,» [En línea]. Available: https://www.openssl.org/.

[33] Shelly, «Shelly Plug/PlugS: MQTT,» [En línea]. Available: https://shelly-api-docs.shelly.cloud/gen1/#shelly-plug-plugs-mqtt.

[34] Mosquitto, «Mosquitto_sub man page,» [En línea]. Available: https://mosquitto.org/man/mosquitto_sub-1.html.

[35] Docker, «Networking in Compose,» [En línea]. Available: https://docs.docker.com/compose/networking/.

[36] Ministerio para la Transición Ecológica y el Reto Demográfico, «Precio Voluntario para el Pequeño Consumidor (PVPC),» [En línea]. Available: https://www.miteco.gob.es/es/energia/energia-electrica/electricidad/contratacion-suministro/precio-voluntario.html.

[37] OCU, «Tarifa de la luz PVPC o en el mercado libre: cómo saber cuál conviene más en febrero de 2024,» [En línea]. Available: https://www.ocu.org/vivienda-y-energia/gas-luz/informe/mercado-libre-o-regulado.

[38] Ministerio de Industria, Energía y Turismo, «Real Decreto 216/2014,» [En línea]. Available: https://www.boe.es/buscar/act.php?id=BOE-A-2014-3376.

[39] Comisión Nacional de los Mercados y la Competencia, «Listado de Comercializadores de Referencia con la información relativa al Bono Social,» [En línea]. Available: https://sede.cnmc.gob.es/listado/censo/10.

[40] Gobierno de España, «Bono Social de Electricidad,» [En línea]. Available: https://www.bonosocial.gob.es/#inicio.

[41] Red Eléctrica de España, «Operación del sistema eléctrico,» [En línea]. Available: https://www.ree.es/es/actividades/operacion-del-sistema-electrico.

[42] Red Eléctrica de España, «Término de facturación de energía activa del PVPC,» [En línea]. Available: https://www.esios.ree.es/es/pvpc.

[43] NodeRED, «node-red-dashboard,» [En línea]. Available: https://flows.nodered.org/node/node-red-dashboard.

[44] Docker Hub, «Postgres Image,» [En línea]. Available: https://hub.docker.com/_/postgres/.

[45] Microsoft, «Azure for Students,» [En línea]. Available: https://azure.microsoft.com/es-es/free/students/.

[46] Microsoft, «Portal de Azure,» [En línea]. Available: https://portal.azure.com/.

ANEXO A: SIMULACIÓN DEL ENCHUFE INTELIGENTE

En caso de querer probar el funcionamiento del sistema domótico sin disponer de un enchufe Shelly Plug S, se puede utilizar una herramienta de simulación que se ha desarrollado para modelar el comportamiento real del enchufe inteligente. Esta herramienta de simulación consiste en un script de Python llamado **sim_enchufe.py**, que se encuentra dentro del directorio **TFG/simulador** del proyecto. Para simular el comportamiento del enchufe, se hace uso de la librería Paho MQTT para crear un cliente MQTT que se suscribe y publica en los mismos tópicos que el enchufe real. El código del simulador del enchufe inteligente se muestra en las figuras A-1 y A-2:

```python
import paho.mqtt.client as mqtt
import random
import time

# Parametros de potencia máxima y mínima de la simulación
POTENCIA_MAX = 100.0 # vatios
POTENCIA_MIN = 50.0  # vatios

# Estado del enchufe. Si esta encendido, deja pasar la corriente
encendido = True

# Configuración del broker MQTT
BROKER = "127.0.0.1"
PUERTO = 1883
TOPICO_POTENCIA = "shellies/enchufe/relay/0/power"
TOPICO_COMANDOS = "shellies/enchufe/relay/0/command"
TOPICO_ESTADO   = "shellies/enchufe/relay/0"

# Retardo de tiempo entre medidas de potencia
RETARDO_PUB = 1 # segundos

# Funcion callback cuando se recibe un mensaje de un topico
# al que estamos suscritos
def on_message(client, userdata, msg):

    comando = str(msg.payload.decode())

    print(f"Mensaje recibido: [{msg.topic}] {comando}")

    if msg.topic == TOPICO_COMANDOS:

        global encendido

        if comando == "on" and not encendido:
            encendido = True
            print("Se ha encendido el enchufe")

        elif comando == "off" and encendido:
            encendido = False
            print("Se ha apagado el enchufe")

```

Figura A-1: Contenido del fichero sim_enchufe.py. Parte 1

```python
# Funcion callback cuando nos conectamos al servidor
def on_connect(client, userdata, flags, rc):

    print(f"Conectado al broker con codigo de estado: {rc}")
    #Al conectarnos, nos suscribimos al topico donde
    # queremos recibir mensajes
    client.subscribe(TOPICO_COMANDOS)

try:

    print("Iniciando la simulación del enchufe inteligente...")
    cliente = mqtt.Client()
    cliente.on_message = on_message
    cliente.on_connect = on_connect
    cliente.connect(BROKER, PUERTO)

    while True:

        cliente.loop_start()

        medida = 0.0

        #Si el enchufe esta encendido, la potencia medida podra ser distinta de 0
        if encendido:
            medida = round(random.uniform(POTENCIA_MIN, POTENCIA_MAX), 2)

        #Publicamos la potencia medida
        cliente.publish(TOPICO_POTENCIA, medida)

        time.sleep(RETARDO_PUB)

        print(f"[{TOPICO_POTENCIA}] {medida}")

        #Publicamos el estado del enchufe
        if encendido:
            cliente.publish(TOPICO_ESTADO, "on")
        else:
            cliente.publish(TOPICO_ESTADO, "off")

except KeyboardInterrupt:
    cliente.publish(TOPICO_POTENCIA, 0.0)
    cliente.publish(TOPICO_ESTADO, "off")
    time.sleep(0.5)
    print("\nFin de la simulación")

```

Figura A-2: Contenido del fichero sim_enchufe.py. Parte 2

Para comenzar la simulación, debemos ejecutar el comando "**python3 sim_enchufe.py**". Por defecto, en la simulación el enchufe estará en el estado "encendido", y comenzará a publicar medidas de potencia de la misma forma que haría el enchufe real, como podemos ver en la figura A-3:

```
sergio@ubuntu:~/TFG/simulador$ python3 sim_enchufe.py
Iniciando la simulación del enchufe inteligente...
Conectado al broker con codigo de estado: 0
[shellies/enchufe/relay/0/power] 86.05
[shellies/enchufe/relay/0/power] 80.98
[shellies/enchufe/relay/0/power] 75.01
[shellies/enchufe/relay/0/power] 85.57
[shellies/enchufe/relay/0/power] 62.13
[shellies/enchufe/relay/0/power] 81.36
[shellies/enchufe/relay/0/power] 96.93
^C
Fin de la simulación
sergio@ubuntu:~/TFG/simulador$
```

Figura A-3: Ejecución de la simulación del enchufe inteligente

Además, de forma periódica se irá publicando el estado del enchufe (encendido o apagado) en el broker MQTT, lo cual nos permitirá hacer el seguimiento del estado desde la interfaz de usuario de NodeRED de la misma forma que haríamos con el enchufe real. De igual manera, podemos seguir enviando comandos de apagado y encendido desde la interfaz de usuario. Al apagar el enchufe, esto se verá reflejado en la simulación haciendo que la potencia publicada por el enchufe sea de cero vatios, y al encenderse volverán a publicarse valores aleatorios de potencia, como se muestra en la figura A-4:

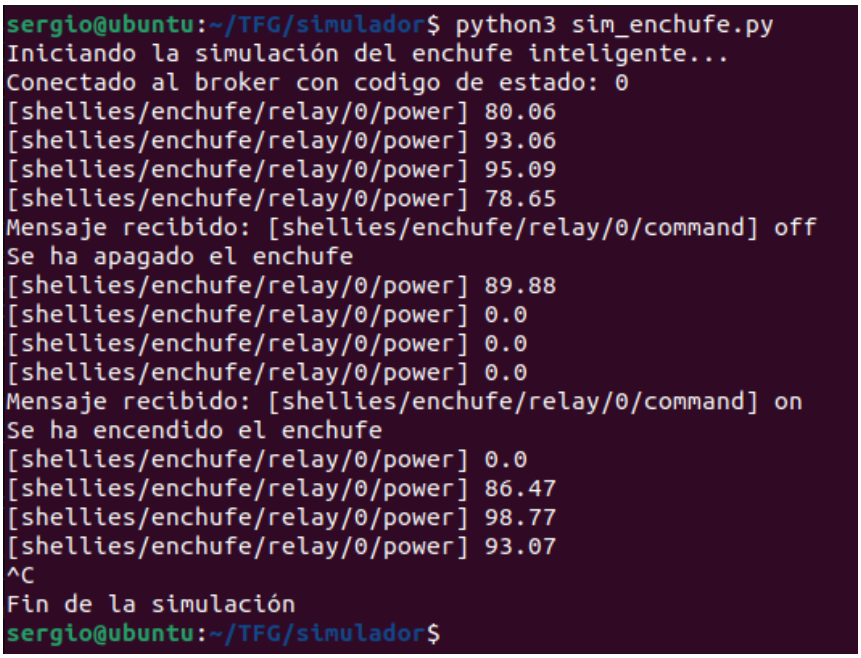
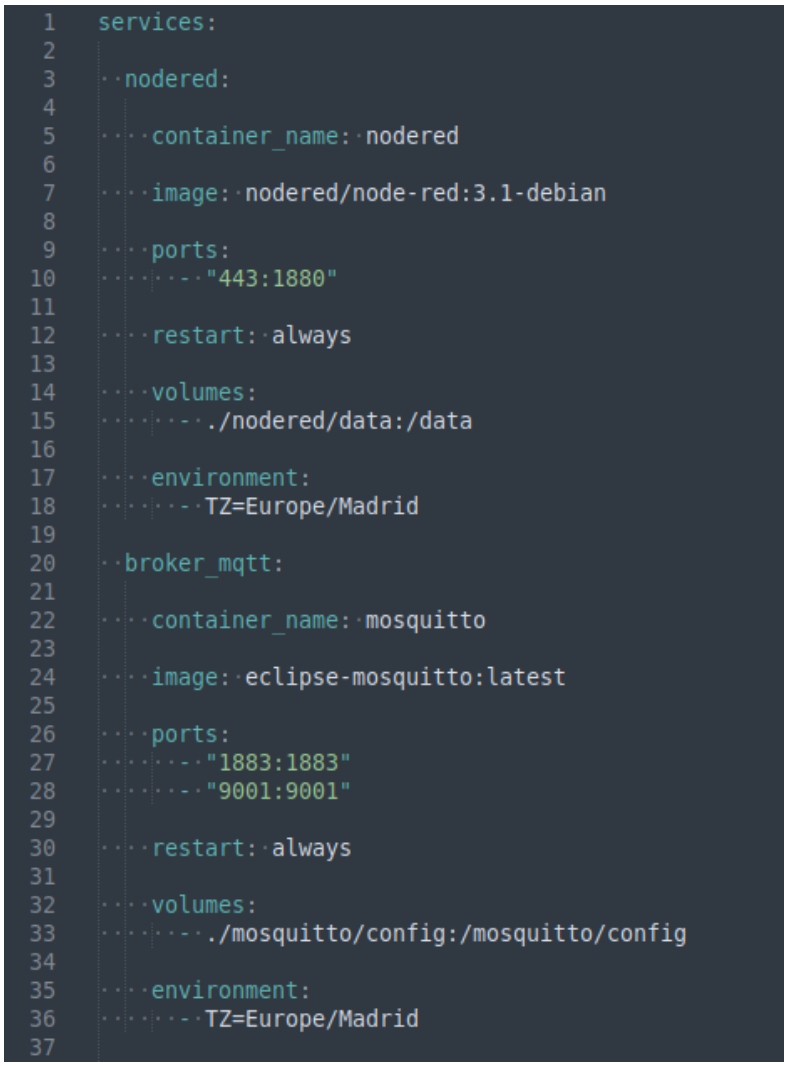


Figura A-4: Simulación del estado del enchufe mediante la recepción de comandos de control

ANEXO B: FICHERO DOCKER-COMPOSE.YAML

Docker-Compose es la herramienta que nos ha permitido definir, configurar y desplegar los servicios del sistema domótico en contenedores Docker a partir de un fichero de configuración **docker-compose.yaml**. A lo largo de la memoria, se han ido mostrando extractos de este fichero y se ha explicado la definición de cada uno de los contenedores que encapsulan los servicios del sistema. En las figuras B-1 y B-2 se muestra el contenido completo del fichero docker-compose.yaml utilizado en el proyecto:



```
services:

  nodered:

    container_name: nodered

    image: nodered/node-red:3.1-debian

    ports:
      - "443:1880"

    restart: always

    volumes:
      - ./nodered/data:/data

    environment:
      - TZ=Europe/Madrid

  broker_mqtt:

    container_name: mosquitto

    image: eclipse-mosquitto:latest

    ports:
      - "1883:1883"
      - "9001:9001"

    restart: always

    volumes:
      - ./mosquitto/config:/mosquitto/config

    environment:
      - TZ=Europe/Madrid

```

Figura B-1: Contenido del fichero docker-compose.yaml. Parte 1

```
37
38  servicio_rest:
39
40      build: ./python
41
42      container_name: python_rest
43
44      image: python_rest
45
46      ports:
47          - "10000:10000"
48
49      restart: always
50
51      environment:
52          - TZ=Europe/Madrid
53
54  base_datos:
55
56      container_name: postgresql
57
58      image: postgres:latest
59
60      environment:
61          - POSTGRES_USER=sergio
62          - POSTGRES_PASSWORD=sergio
63          - POSTGRES_DB=monitor_consumo
64          - PGDATA=/var/lib/postgresql/data/pgdata
65          - TZ=Europe/Madrid
66
67      ports:
68          - "5432:5432"
69
70      restart: always
71
72      volumes:
73          - ./postgresql/data:/var/lib/postgresql/data
74
```

Figura B-2: Contenido del fichero docker-compose.yaml. Parte 2