UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Ingeniería del Software

Desarrollo de aplicación IoT para monitorizar y optimizar el consumo energético
Development of an IoT application for monitoring and optimizing energy usage

Realizado por
Daniel de Lizaur García

Tutorizado por
Luis Manuel Llopis Torres

Departamento
LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE MÁLAGA

MÁLAGA, FEBRERO DE 2026

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

Desarrollo de aplicación IoT para monitorizar y optimizar el consumo energético

Development of an IoT application for monitoring and optimizing energy usage

Realizado por
Daniel de Lizaur García

Tutorizado por
Luis Manuel Llopis Torres

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, ENERO DE 2026

Fecha defensa: **Marzo de 2026**

Resumen

En el contexto de la necesidad de mejorar la eficiencia energética y reducir el coste de la electricidad en el ámbito doméstico, se ha desarrollado un sistema de Internet de las Cosas (IoT) dedicado a la recopilación y análisis del consumo de dispositivos eléctricos mediante enchufes inteligentes WiFi. Como objetivo principal, se ha planteado que los usuarios del sistema puedan obtener información detallada sobre el consumo de sus dispositivos, que podrá utilizar con el fin de tomar medidas para reducir su coste en energía.

Para ello, se ha desarrollado una aplicación móvil Android que ofrece visualizaciones en gráficas del consumo histórico de los dispositivos, así como datos del precio de la energía obtenidos de la Red Eléctrica Española, tanto históricos como futuros con hasta 24 horas de antelación. Con esta información, la aplicación calcula el coste energético de los dispositivos y utiliza un algoritmo de optimización para generar recomendaciones que ayuden al usuario a reducirlo.

Para la recopilación de datos, se ha empleado una arquitectura basada en el microcontrolador ESP32, utilizado para detectar los enchufes inteligentes cercanos, configurar la conexión WiFi automáticamente y enviar los datos a un servidor en la nube. El sistema está diseñado para admitir múltiples usuarios suponiendo que cada uno disponga de un ESP32, manteniendo los datos seguros mediante autenticación.

Como resultado, se ha obtenido un sistema funcional capaz de combinar adquisición, visualización y análisis de datos energéticos que cumple con el objetivo de ser una herramienta útil para reducir el consumo doméstico.

Palabras clave: Internet de las Cosas, Hogar Inteligente, Sistema Embebido, Eficiencia Energética, Desarrollo de Aplicaciones Móviles

Abstract

In response to the need to improve energy efficiency and reduce household energy costs, an Internet of Things (IoT) system was developed, aimed at the collection and analysis of the consumption of electrical devices using WiFi-compatible smart plugs. The main objective was to enable users to obtain detailed information about the consumption of their devices, which could be used to take actions aimed at reducing their energy costs.

To this end, an Android mobile application was developed to provide graphical visualizations of devices' historical consumption, as well as energy price data obtained from the Spanish electric grid, including both historical and future prices up to 24 hours in advance. Using the information, the application calculated the energy cost of devices and applied an optimization algorithm to generate recommendations to help users reduce it.

For data acquisition, the system employs an architecture based on the ESP32 microcontroller. It was used to detect nearby smart plugs, automatically configure the WiFi connection, and send data to a cloud server. The system was designed to support multiples users assuming each user had their own ESP32, and to keep data secure through authentication.

As a result, a functional system was obtained that combined the acquisition, visualization and analysis of energy data, thus providing a useful tool for reducing household energy costs.

Keywords: Internet of Things, Smart Home, Embedded Systems, Energy Efficiency, Mobile Application Development

Índice

Índice de Figuras

1

Introducción

1.1. Contexto y descripción del proyecto

En los últimos años se ha producido una rápida evolución en el sector del Internet de las Cosas (*IoT*), lo que ha provocado que el uso de dispositivos inteligentes en el entorno doméstico sea cada vez más popular y accesible, creándose así el concepto de *hogar inteligente*. Existe una gran variedad de sensores, electrodomésticos y sistemas de monitorización conectados a internet económicos y fáciles de utilizar, lo que ha facilitado su adopción por usuarios no especializados.

En este contexto, la monitorización del consumo energético se ha convertido en una de las aplicaciones más relevantes del IoT, al permitir obtener información sobre cómo y dónde se consume la energía. Actualmente existen en el mercado diversos dispositivos para medir el consumo eléctrico, como enchufes o contadores inteligentes. Estas herramientas ofrecen datos de consumo en tiempo real e históricos, que el usuario puede utilizar para concienciarse sobre su consumo y tomar medidas para intentar reducirlo. Sin embargo, en muchos casos se limitan a la visualización de datos, sin ayudar al usuario a interpretarlos y tomar decisiones.

La propuesta planteada en este proyecto es la recopilación de datos de consumo de dispositivos eléctricos mediante enchufes inteligentes, pero sin limitarse a su monitorización y visualización. El consumo se combina con datos del precio de la energía para ofrecer al usuario información completa tanto sobre el consumo de sus dispositivos como su coste, teniendo en cuenta las variaciones del precio entre los máximos y mínimos de demanda a lo largo del día. Con esta combinación, el sistema desarrollado proporciona al usuario recomendaciones concretas para reducir el coste energético.

Por otro lado, en el proyecto se ha considerado la importancia de que el sistema sea fácil de utilizar. Para conseguirlo, se ha desarrollado una aplicación móvil Android donde el usuario puede visualizar los datos de forma intuitiva. Además, se ha empleado un microcontrolador ESP32 para automatizar el proceso de conexión de los enchufes inteligentes a internet y la recopilación de los datos, eliminando el proceso de configuración que requieren este tipo de dispositivos. El microcontrolador incorpora además una pantalla que muestra el estado de la conexión y las medidas obtenidas.

Con la arquitectura descrita en el siguiente apartado, el sistema se podría plantear como una aplicación comercial donde los usuarios pueden adquirir dispositivos ESP32 preconfigurados para su uso con la aplicación móvil.

1.2. Arquitectura

Cada usuario de la aplicación dispondrá de un ESP32 que creará un punto de acceso WiFi al que se conectarán los enchufes inteligentes. Periódicamente hará peticiones a los enchufes y éstos enviarán sus datos de consumo. A su vez, el ESP32 se conectará a un router WiFi para obtener acceso a internet y enviar los datos recopilados al servidor, que aloja la base de datos. Por otro lado, el servidor realiza peticiones a la API de Red Eléctrica (*ESIOS*) diariamente para obtener el precio de la energía y almacenarlos también en la base de datos mediante una *función cloud*.
Finalmente, la aplicación móvil se conectará al servidor para acceder a todos los datos.

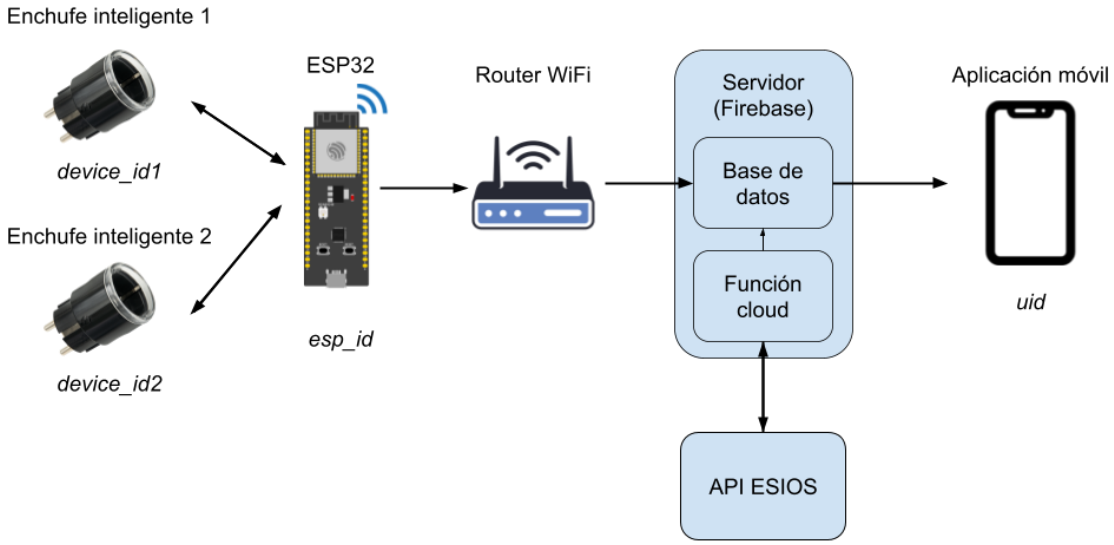

Fig. 1. Diagrama con los componentes del sistema para un usuario y sus IDs asociados

Además, cada usuario de la aplicación tiene un identificador único (uid) generado por el servicio de autenticación. Para asociar a usuarios de la aplicación con su ESP32, cada ESP32 tendrá asociado un identificador único (*esp_id*) que estará programado en su firmware.
Conociendo este identificador, el usuario podrá introducirlo en la aplicación para vincularlo, demostrando que ese ESP32 le pertenece y consiguiendo así acceso a sus datos. Los enchufes también disponen de un identificador único (*device_id*) dado por el fabricante. Cuando el ESP32 envía datos de consumo al servidor, los clasificará de acuerdo con los identificadores de los enchufes. Desde la aplicación, el usuario podrá asociarles un nombre más intuitivo.

1.3. Tecnologías empleadas

En este proyecto se han empleado las siguientes tecnologías:

1.3.1. ESP32

El ESP32 es un microcontrolador fabricado por la empresa china Espressif, ampliamente utilizado en proyectos de Internet de las Cosas (IoT) debido a su potencia, versatilidad, y coste reducido. Cuenta con un procesador de doble núcleo Xtensa LX6 de 32 bits y una frecuencia de reloj de 240 MHz. Permite conexiones inalámbricas mediante módulos Bluetooth y Wi-Fi 2.4 GHz integrados, y admite multitud de periféricos al ser compatible con los protocolos serie SPI, I2C, y UART, además de incorporar convertidores digital-analógico y analógico-digital (ADC/DAC) [1]. Entre todos los modelos de la familia ESP32, en el proyecto se utiliza concretamente el *ESP32-DevkitC*.

Fig. 2. ESP32-DevkitC

Para programar el firmware del ESP32 se puede utilizar el framework oficial *esp-idf* (*Espressif IoT Development Framework*) [2], que ofrece un control completo del hardware a bajo nivel, incluyendo gestión de tareas y concurrencia con *FreeRTOS* [3]. Sin embargo, este proyecto aprovecha la compatibilidad con el framework de Arduino [4] y con muchas de las librerías creadas para ese sistema, que ofrecen APIs de más alto nivel. De esta forma, se facilita el desarrollo del firmware. El lenguaje utilizado ha sido C++.

Para grabar el firmware en el microcontrolador, primero se compila el código a un archivo binario, que se transfiere por USB a la memoria flash del ESP32 mediante el protocolo serie UART [5]. El ESP32 también puede utilizar este protocolo para enviar cadenas de texto, lo que resulta útil para la depuración, además de permitir leer el estado de las variables y registros de la CPU. Espressif ofrece la herramienta *esptool* [6], un script Python que gestiona la compilación, grabación y depuración del firmware mediante la línea de comandos. Sin embargo, en este proyecto se utilizó *PlatformIO*, que se explica más adelante.

1.3.2. Firebase

Firebase es una plataforma de desarrollo en la nube creada por Google que ofrece múltiples servicios como hosting, base de datos, autenticación, almacenamiento y analítica de aplicaciones. Su principal ventaja es que permite desarrollar aplicaciones en la nube sin necesidad de implementar y desplegar un backend completo, ya que estos servicios gestionan automáticamente muchas de las responsabilidades que implementaría el backend [7].

Fig. 3. Logotipo de Firebase

En el proyecto se han utilizado los siguientes servicios de Firebase:

- **Firebase Authentication**: Gestiona el inicio de sesión de usuarios mediante correo electrónico y contraseña o proveedores externos como Google, Facebook o Apple [8].
- **Firebase Realtime Database**: Es una base de datos NoSQL basada en objetos JSON, diseñada para obtener fácilmente sincronización entre múltiples usuarios [9].
- **Cloud functions:** Permite definir funciones que se ejecutan en el backend automáticamente cuando ocurren eventos o peticiones. Se utiliza para implementar lógica de servidor que no debe ejecutarse en el cliente. Las funciones se ejecutan utilizando la infraestructura en la nube de *Google Cloud* [10] En este proyecto se utiliza para gestionar las llamadas a la API externa de Red Eléctrica.

Para conectar y trabajar con Firebase existen librerías para multitud de plataformas como Android, Flutter, Javascript, Apple, C++ y Unity. [11]

1.3.3. Flutter

Flutter [12] es un framework de código abierto creado por Google que permite el desarrollo de aplicaciones multiplataforma, soportando Web, Android, iOS, Windows, Linux y MacOS entre otros [13]. No utiliza componentes nativos de cada plataforma, en su lugar emplea un motor de renderizado llamado *Skia* [14] para dibujar los píxeles directamente. De esta forma, se garantiza un comportamiento y apariencia consistente en todas las plataformas. Para el desarrollo se utiliza el lenguaje *Dart* [15], creado también por Google como lenguaje multipropósito, pero es utilizado principalmente en Flutter. Es un lenguaje orientado a objetos similar a C# o Java.

La gran ventaja de Flutter es que permite usar un solo lenguaje (Dart) para todas las plataformas en lugar de tener que usar Javascript para web, Java para Android, o Swift para iOS.

Fig. 4. Logotipos de Flutter y Dart

Una de sus funcionalidades más interesantes es el *hot reloading*, que permite hacer cambios en el código y que éstos se reflejen en la aplicación inmediatamente, sin necesidad de reiniciarla ni recompilar todo el código [16].

De todas las plataformas que soporta, este proyecto se centrará en crear una aplicación móvil Android, aunque también se utilizó la plataforma web para pruebas. Para el desarrollo en Android es necesario instalar *Android Studio* [17] que incluye los emuladores necesarios para probar aplicaciones Flutter rápidamente.

En Flutter, las interfaces de usuario se modelan mediante componentes reutilizables denominados *Widgets*, que pueden a su vez contener otros *widgets* anidados, formando una estructura de árbol similar a los documentos HTML.

Flutter proporciona en su librería estándar multitud de *widgets* para poder incluir elementos de interfaz como texto, imágenes, listas, botones, etc. Otros *widgets* son puramente estructurales y sirven, por ejemplo, para para definir márgenes o el tamaño y posición de sus *widgets* anidados. Además de la librería estándar, Flutter ofrece multitud de paquetes creados por la comunidad, que se pueden instalar con el gestor de paquetes *pub [18]*.

El desarrollador puede definir sus propios *widgets* creando una clase que herede de *StatefulWidget* o *StatelessWidget* y sobreescribiendo el método *build*, que definirá los componentes del *widget* en forma de árbol. Los dos tipos principales de *widget son*:

- **StatelessWidget**: Representa *widgets* inmutables, cuya apariencia solo depende de datos dados en su constructor y no cambia durante la ejecución de la aplicación.
- **StatefulWidget:** Representa *widgets* cuya apariencia depende de un estado interno que puede cambiar durante la ejecución de la aplicación. Tendrá asociado un objeto de tipo *State*, con el que se puede modificar el estado interno con llamadas a *setState*, que notifican a Flutter para que vuelva a renderizar el *widget* si es necesario [19].

Este es un ejemplo de un widget que contiene un contador y un botón que lo incrementa. Este código se genera al crear una nueva aplicación Flutter, y sirve como ejemplo de los conceptos fundamentales de Flutter y Dart, incluyendo la creación de un *StatefulWidget* y su *State* asociado, llamadas a *setState* y la construcción del *widget* en el método *build*. Se utiliza un *StatefulWidget* para que el valor del contador pueda cambiar y se vuelva a renderizar el widget cuando sea necesario. Se pueden observar algunos de los *widgets* estructurales como *Center* y *Column*.

```dart
class MyHomePage extends StatefulWidget {
  const MyHomePage({super.key, required this.title});

  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text('You have pushed the button this many times:'),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headlineMedium,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: const Icon(Icons.add),
      ),
    );
```

Fig. 5. Apariencia del widget anterior dentro de una aplicación Flutter

Para navegar entre distintas pantallas se utiliza la clase *Navigator*, que emplea una estructura de pila, donde se hace una operación *push* para ir a una pantalla nueva y *pop* para volver a la pantalla anterior [20].

1.3.4. Visual Studio Code

Visual Studio Code o VSCode es un editor de código creado por Microsoft que permite multitud de extensiones. Las más relevantes para este proyecto han sido:

- PlatformIO [21]: Simplifica el desarrollo del firmware en microcontroladores gestionando la compilación, grabación y depuración del código por el puerto serie ofreciendo una interfaz gráfica que sustituye a la terminal. Incorpora también un gestor de librerías. Admite una gran cantidad de arquitecturas y microcontroladores, y en el caso del ESP32, permite usar tanto esp-idf como Arduino, además de gestionar internamente los comandos de *esptool*.
- Dart [22]: Añade soporte para el lenguaje Dart, permitiendo acceso a la documentación, autocompletado y ayudas para el formato y refactorización del código. Incluye un depurador que gestiona fácilmente el *hot reloading* y permite observar el estado del programa insertando puntos de ruptura.
- Flutter [23]: Permite el código Dart desde VSCode para ejecutarlo en la plataforma deseada, en este caso un emulador Android. Utiliza internamente la extensión *Dart* para aprovechar todas sus funcionalidades.

Fig. 6. Logotipo de Visual Studio Code

1.3.5. Git

Git es un sistema de control de versiones de código abierto creado originalmente por Linus Torvalds, que permite almacenar un historial completo del desarrollo, revertir a versiones anteriores y trabajar de manera modular utilizando ramas [24].

Fig. 7. Logotipo de Git

1.3.6. Github

Github es un repositorio Git remoto para proyectos, diseñado para alojar y compartir repositorios de código y facilitar la colaboración entre varios desarrolladores trabajando en un mismo proyecto [25].

Fig. 8. Logotipo de Github

1.3.7. Postman

Postman es una herramienta con versiones web y de escritorio utilizada para probar APIs fácilmente. Permite enviar solicitudes HTTP a un endpoint, configurando los parámetros de la URL, los encabezados y el cuerpo de la petición, pudiendo utilizar JSON u otros formatos. Una vez enviada la solicitud, se podrá analizar la respuesta del servidor y su funcionamiento [26].

Fig. 9. Logotipo de Postman

1.3.8. PlantUML

PlantUML es una herramienta de código abierto que permite generar distintos tipos de diagramas UML a partir de una descripción textual utilizando un lenguaje propio [27]. Puede producir diagramas de clases, objetos, actividad, estados y casos de uso entre otros, pero en este proyecto se ha empleado para modelar diagramas de secuencia. Es una herramienta escrita en Java, pero se puede utilizar de forma directa accediendo a su versión web [28].

1.4. Estado del arte

Existen ya desarrolladas otras aplicaciones o proyectos relacionados con el *IoT* y la eficiencia energética que incorporan muchas de las funcionalidades propuestas en este proyecto, entre las que se pueden destacar:

1.4.1. Shelly

Shelly es un fabricante que ofrece multitud productos inteligentes orientados a la domótica y el *IoT* como bombillas, enchufes, sensores de movimiento, termostatos, cerraduras y relés que permiten controlar todo tipo de dispositivos [29]. Todos los productos forman un ecosistema con el que se puede interactuar a través de una aplicación propia llamada *Shelly Smart Control* [30]. Destaca por ofrecer una API abierta, que permite integrar sus productos en aplicaciones fuera del ecosistema Shelly. Este proyecto se centra en los enchufes inteligentes, cuyo uso se explica con más detalle en el apartado 3.2.1 *Enchufes inteligentes*.

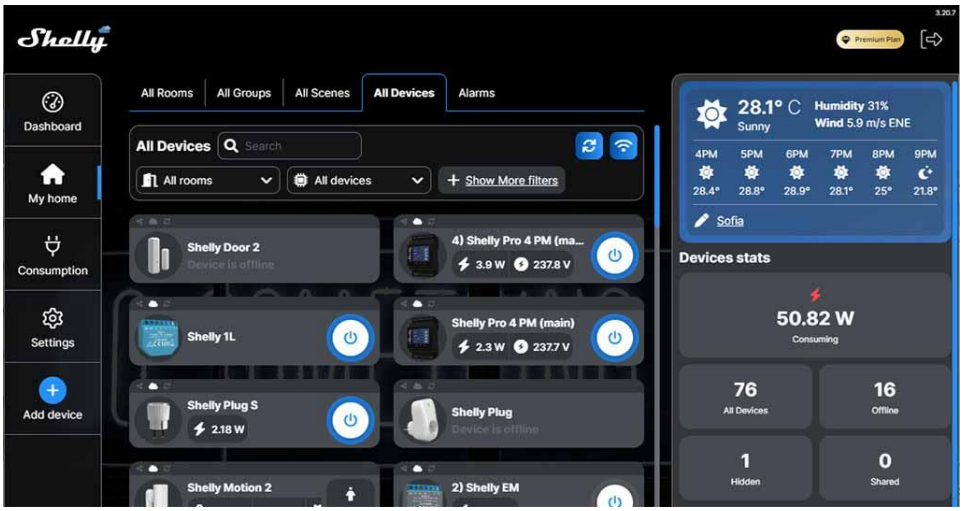

Fig. 10. Shelly Smart Control.

1.4.2. ESIOS

ESIOS es una web ofrecida por la Red Eléctrica de España que permite visualizar en gráficas una gran cantidad de datos históricos relevantes de la red, incluyendo generación, demanda y precios de la energía. Ofrece una API utilizada en el proyecto para obtener los datos del precio de la energía [31].

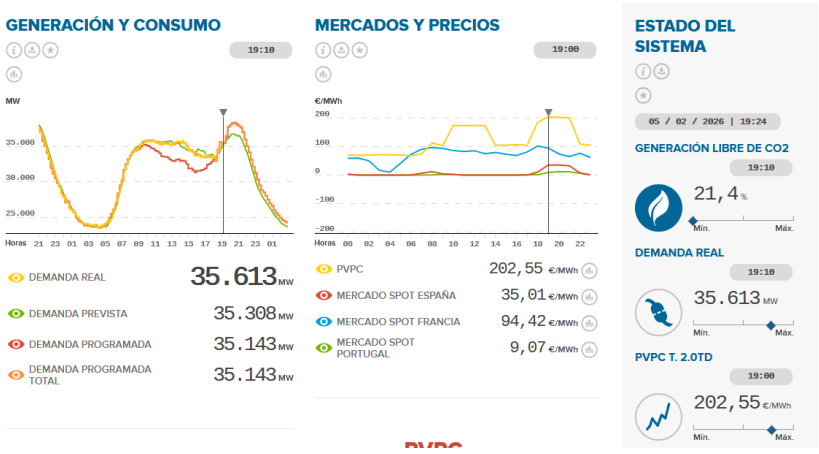

Fig. 11. Vista principal de la web de ESIOS

1.4.3. Hello Watt

Hello Watt es una empresa que ofrece servicios dedicados a comparar distintas tarifas de electricidad y gas con el objetivo de reducir el coste energético [32]. Ofrece una aplicación que permite analizar el consumo de una vivienda completa utilizando un contador inteligente. Permite además introducir datos de la vivienda que afectan a su eficiencia energética, como su superficie o tipo de aislamiento. Agregando esta información con datos proporcionados por la comercializadora de energía del usuario, compara la eficiencia con otras viviendas similares y genera soluciones para reducir el coste, incluyendo la búsqueda de otras tarifas de energía alternativas. A diferencia de Shelly Smart Control y la aplicación desarrollada en este proyecto, Hello Watt utiliza el consumo global de la vivienda y lo divide en categorías mediante datos estadísticos, en lugar de centrarse en dispositivos concretos [33].

Fig. 12. Aplicación de Hello Watt

1.4.4. Proyectos de ámbito académico

Además de aplicaciones comerciales, existen otros proyectos de ámbito académico centrados en desarrollar algoritmos dedicados a reducir el consumo y coste energético.

Entre ellos se puede destacar *(EM)³ Framework* [34], un sistema que recopila información no solo de consumo energético, sino también de temperatura, luminosidad y otros parámetros. Utiliza un algoritmo basado en detección de anomalías y eventos que resultan en un consumo excesivo denominados *micro-momentos*. En lugar de una arquitectura en la nube, utiliza *Edge-computing* combinado con un servidor local basado en *Home Assistant* [35]. El usuario recibe recomendaciones mediante notificaciones o la aplicación móvil de *Home Assistant*, Al disponer de datos de varios tipos, puede ofrecer recomendaciones específicas como abrir una ventana en lugar de utilizar aire acondicionado, o apagar una lámpara si hay luz suficiente.

1.5. Metodología de trabajo

Durante el desarrollo se ha seguido una metodología ágil basada en iteraciones. Para añadir nuevas funcionalidades al proyecto en cada iteración, en primer lugar, se han decidido los requisitos y comportamientos deseados en una fase de diseño para posteriormente ser implementados en una fase de desarrollo. A lo largo del proceso se han ido realizando las pruebas necesarias y generando la documentación de las funcionalidades en el código y la memoria.
El orden de trabajo de las iteraciones ha seguido aproximadamente el flujo que siguen los datos desde que se recopilan desde el ESP32 hasta que se visualizan en la aplicación, pasando por el servidor en la nube. El proceso se puede resumir en cuatro iteraciones:

- **Primera iteración**: Dedicada a gestionar la conexión WiFi entre los enchufes inteligentes y el ESP32. Como resultado, se ha conseguido conectar con los enchufes inteligentes automáticamente, obteniendo sus mediciones de consumo mediante peticiones HTTP.

- **Segunda iteración**: Dedicada a crear y configurar el servidor en la nube, asegurando que los ESP32 escriben sus datos de consumo recopilados en la base de datos.

- **Tercera iteración**: Dedicada a iniciar el desarrollo de la aplicación móvil, configurar la conexión al servidor y la autenticación de usuarios para poder obtener y visualizar los datos de consumo en gráficas.

- **Cuarta iteración**: Dedicada a completar la aplicación móvil añadiendo funcionalidades para visualizar los precios históricos y coste, así como implementar el sistema de recomendaciones de ahorro.

En la práctica, el proceso no ha resultado completamente lineal, debido a que en iteraciones posteriores se han definido nuevas funcionalidades que han requerido modificar los resultados de las iteraciones anteriores, especialmente en las dos primeras, que comprenden lo relacionado con el firmware del microcontrolador y la base de datos.

Durante el desarrollo se ha empleado *Git* como sistema de control de versiones, alojando el repositorio en *Github*.

2

Análisis y diseño

En este capítulo se analizará el proyecto desde el punto de vista de la ingeniería del software, aplicando técnicas como el análisis de requisitos, especificación de casos de uso y diagramas de secuencia.

2.1. Requisitos

2.1.1. Requisitos funcionales

2.1.1.1. ESP32

- **RF1: Detección de puntos de acceso de enchufes inteligentes**
  El sistema, una vez conectado, deberá escanear las redes WiFi cercanas e identificar cuáles pertenecen a enchufes inteligentes Shelly.
- **RF2: Configuración automática de enchufes inteligentes**
  El sistema deberá enviar sus credenciales de conexión a cada enchufe inteligente encontrado.
- **RF3: Obtención de credenciales WiFi mediante WPS**
  El sistema deberá comprobar si tiene almacenadas las credenciales de un router WiFi, y en caso contrario, habilitar el modo WPS para obtenerlas.
- **RF4: Conexión a router WiFi**
  El sistema deberá utilizar las credenciales almacenadas para establecer una conexión con el router y el servidor.
- **RF5: Obtención periódica de datos de consumo**
  El sistema deberá, con una frecuencia constante, solicitar mediciones de consumo a cada enchufe inteligente conectado y enviarlas al servidor.
- **RF6: Visualización del estado de conexión y mediciones realizadas**
  El sistema dispondrá de una pantalla donde se muestre el estado del proceso de conexión y sus posibles errores, y una vez completado, se muestre el valor de las mediciones periódicas.

2.1.1.2. Aplicación móvil

- **RF7: Inicio de sesión con cuenta de Google**
  El usuario deberá autenticarse en la aplicación utilizando una cuenta de Google.
- **RF8: Vinculación con ESP32 mediante ID**
  El usuario podrá introducir el identificador asociado a su ESP32 en la aplicación para demostrar que le pertenece y poder acceder a sus datos. La aplicación comprobará que ese identificador no esté ya asociado con otro usuario.
- **RF 9: Listado de dispositivos**
  El usuario dispondrá de un listado de dispositivos (enchufes) conectados a su ESP32, donde para cada uno se mostrará su identificador, nombre, consumo y tiempo desde la última medición.
- **RF 10: Asignación de nombres personalizados a dispositivos**
  El usuario podrá asignar un nombre personalizado a cada uno de sus dispositivos, que se podrá editar posteriormente.
- **RF 11: Visualización del consumo de un dispositivo**
  El usuario podrá, pulsando en un dispositivo del listado, acceder a una pantalla donde se mostrará en gráficas su consumo histórico, incluyendo la potencia instantánea en vatios y la energía acumulada en vatios-hora.
- **RF 12: Visualización del precio de la energía**
  El sistema mostrará en otra gráfica los precios históricos de la energía en €/kWh y las previsiones de precios con hasta 24 horas de antelación, obtenidos de la API de la Red Eléctrica.
- **RF 13: Selección del rango de fechas para la visualización.** El usuario podrá seleccionar un intervalo de tiempo personalizado para las gráficas, o bien usar los intervalos predefinidos: últimas 24 horas, última semana y último mes. También tendrá la opción de avanzar o retroceder un día, semana o mes dependiendo del intervalo seleccionado.
- **RF 14: Cálculo de coste energético en el rango seleccionado**
  El sistema calculará el coste total en € en el intervalo seleccionado teniendo en cuenta el consumo del dispositivo y los precios históricos y mostrará una gráfica con la evolución del coste a lo largo del tiempo.
- **RF 15: Generación de recomendaciones sobre el consumo energético**
  El sistema generará una vez al día una recomendación que ayude al usuario a reducir el consumo y coste energético de un dispositivo.
- **RF 16: Comparativa de precios pasados y futuros**
  El sistema incluirá en la recomendación una gráfica comparando los precios de la energía de las últimas 24 horas con las siguientes 24 horas.
- **RF 17: Configuración de objetivo de ahorro**
  El usuario podrá establecer un objetivo de ahorro económico en porcentaje.
- **RF 18: Optimización del consumo y coste**.
  El sistema, utilizando el consumo promedio de los últimos 7 días y la predicción de precios, generará una nueva gráfica de consumo ajustada al objetivo de ahorro.
- **RF 19: Cierre de sesión**
  El usuario podrá cerrar sesión en la aplicación.
- **RF 20: Desvinculación de ESP32**
  El usuario podrá desvincular su ESP32, que dejará de estar asociado a su cuenta.

2.1.2. Requisitos no funcionales

2.1.2.1. ESP32

- **RNF 1: Recuperación ante pérdidas de conexión**
  El sistema asegurará que, si uno de los dispositivos conectados pierde la conexión, intentará restablecer la conexión sin afectar al resto de dispositivos.
- **RNF 2: Eficiencia energética**
  El sistema mantendrá su consumo energético por debajo de 1.5 W.
  RNF 3: Número de dispositivos simultáneos
  El sistema admitirá hasta 4 enchufes inteligentes conectados simultáneamente.
- **RNF 4: Enchufes inteligentes compatibles**
  El sistema garantizará la compatibilidad con los enchufes inteligentes Shelly Plug Plus S.

2.1.2.2. Aplicación móvil y servidor

- **RNF 5: Seguridad de las credenciales**.
  El sistema no almacenará credenciales de usuarios de la aplicación como direcciones de correo electrónico y contraseñas. En su lugar, la autenticación se gestionará con un proveedor externo de confianza (Google OAuth).
- **RNF 6: Control de acceso a los datos**.
  El sistema asegurará que cada usuario solo pueda visualizar los datos del dispositivo asociado a su cuenta.
- **RNF 7: Copias de seguridad**
  El servidor creará copias de seguridad de la base de datos diariamente.
- **RNF 8: Facilidad de uso**
  La aplicación deberá ser intuitiva para usuarios sin conocimientos técnicos. Todas las funcionalidades serán accesibles en un máximo de 4 interacciones.
- **RNF 9: Sistemas operativos compatibles**
  La aplicación móvil será compatible con Android 14 o versiones posteriores.
  RNF 10: Disponibilidad del servidor
  El servidor deberá estar disponible para procesar peticiones correctamente al menos un 95% del tiempo.
- **RNF 11: Precisión de las mediciones**
  El sistema almacenará las mediciones de potencia instantánea con una precisión de 0.1 vatios, y la energía acumulada con una precisión de 0.001 vatios-hora.

2.2. Flujo de navegación de la aplicación.

En este diagrama se muestran las principales pantallas de la aplicación y las acciones para transitar entre ellas.

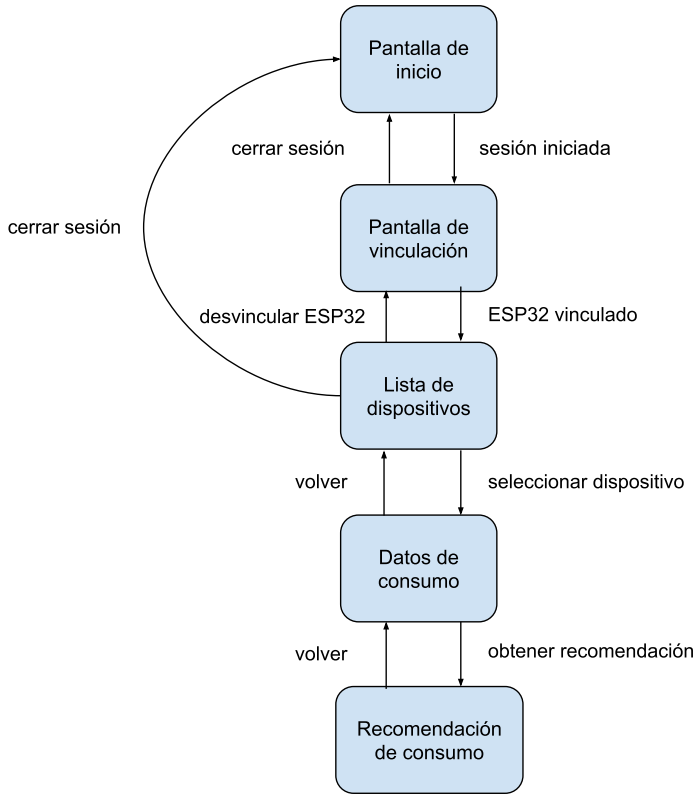

Fig. 13. Flujo de navegación de la aplicación

2.3. Casos de uso

Nombre	Iniciar sesión y vincular con ESP32
Requisitos asociados	RF 7, 8
Actor principal	Usuario
Precondiciones	El usuario ha iniciado la aplicación.
Postcondiciones	El usuario tiene el ESP32 asociado a su cuenta correctamente. El sistema muestra la pantalla con el listado de dispositivos.
Flujo principal	1. El sistema comprueba si el usuario está autenticado. 2. El sistema muestra el botón "iniciar sesión". 3. El usuario pulsa el botón "iniciar sesión". 4. El sistema redirige al usuario al gestor de cuentas del dispositivo. 5. El usuario añade o elige una cuenta e inicia sesión. 6. El sistema redirige al usuario a la pantalla de vinculación con ESP32. 7. El sistema comprueba si el usuario tiene vinculado un ESP32.

	8. El sistema solicita el identificador de su ESP32. 9. El usuario introduce el identificador de su ESP32. 10. El sistema verifica la validez del identificador 11. El sistema redirige al usuario al listado de dispositivos.
Flujos alternativos	1.B.1 Si el usuario ya está autenticado, ir al paso 6 7.B.1 Si el usuario ya tiene un ESP32 vinculado, ir al paso 11 10.B.1 Si el sistema determina que el identificador no es válido, muestra un mensaje de error 10.B.2 Ir al paso 8

Nombre	Visualizar consumo y coste de un dispositivo
Requisitos asociados	RF 9, 11, 12, 13, 14
Actor principal	Usuario
Precondiciones	El usuario tiene asociado un ESP32 a su cuenta El usuario se encuentra en la pantalla del listado de dispositivos
Postcondiciones	El sistema muestra los datos de consumo y coste correspondientes al intervalo y dispositivo seleccionados.
Flujo principal	1. El sistema obtiene los dispositivos conectados al ESP32 del usuario. 2. El sistema muestra la lista de dispositivos. Para cada uno muestra su ID, nombre y última medición. 3. El usuario selecciona uno de los dispositivos. 4. El sistema redirige al usuario a la pantalla de visualización del consumo. 5. El sistema establece por defecto como intervalo las últimas 24 horas. 6. El sistema muestra el nombre del dispositivo y gráficas de potencia instantánea, energía acumulada, precios históricos y coste a lo largo del tiempo en el intervalo. 7. El usuario selecciona la opción de intervalo personalizado. 8. El sistema muestra un calendario. 9. El usuario elige un intervalo y selecciona la opción de confirmar. 10. El sistema cierra el calendario y recalcula las gráficas de acuerdo con el nuevo intervalo. 11. Volver al paso 6.
Flujos alternativos	1.A.1 El sistema no encuentra datos de dispositivos conectados. 1.A.2 El sistema muestra un mensaje indicando al usuario que no hay datos y que verifique la conexión de su ESP32 con los enchufes e internet. 1.A.3 El usuario pulsa el botón "actualizar". 1.A.4 Ir al paso 1. 6.A.1 El usuario selecciona uno de los intervalos predefinidos.

	6.A.2 El sistema recalcula las gráficas de acuerdo con el nuevo intervalo. Volver al paso 5. 6.B.1 El usuario utiliza uno de los botones de navegación "anterior" o "siguiente". 6.B.2 El sistema avanza o retrocede según la duración del intervalo actual. 6.B.3 El sistema recalcula las gráficas de acuerdo con el nuevo intervalo. 6.B.4 Volver al paso 5.

Nombre	Obtener recomendación de consumo
Requisitos asociados	RF 15, 16, 17, 18
Actor principal	Usuario
Precondiciones	El usuario se encuentra en la pantalla de datos de consumo de un dispositivo.
Postcondiciones	El sistema genera una recomendación de consumo.
Flujo principal	1. El usuario selecciona la opción "recomendaciones de consumo" 2. El sistema redirige al usuario a la pantalla de recomendaciones. 3. El sistema muestra una gráfica comparativa de los precios de la energía de las últimas 24 horas y las siguientes 24 horas. 4. El sistema genera una gráfica con el consumo medio por hora del dispositivo de los últimos 7 días. 5. El sistema utiliza el consumo medio y los precios futuros para generar una nueva gráfica con una propuesta de consumo que reduce el coste según el objetivo de ahorro configurado.
Flujos alternativos	5.A.1 El usuario selecciona la opción "objetivo de ahorro" 5.A.2 El sistema solicita el nuevo valor en un cuadro de diálogo. 5.A.3 El usuario introduce el nuevo valor. 5.A.4 El sistema guarda el nuevo valor y cierra el cuadro de diálogo. 5.A.5 Volver al paso 3.

2.4. Diagramas de secuencia

Conexiones del ESP32 con enchufes inteligentes, router y servidor

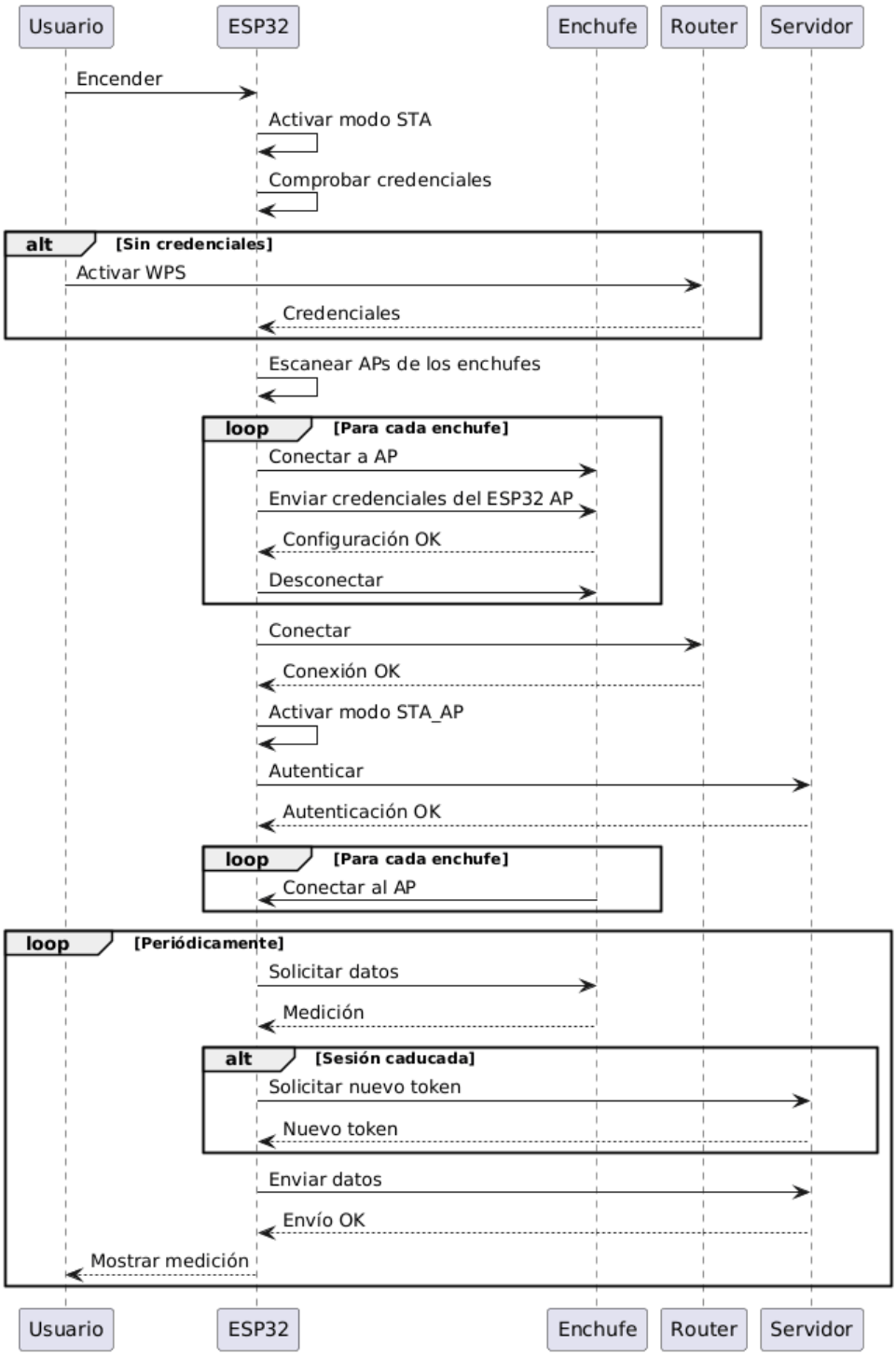

Fig. 14. Diagrama de secuencia de las conexiones del ESP32

Navegación principal de la aplicación, incluyendo inicio de sesión y vinculación

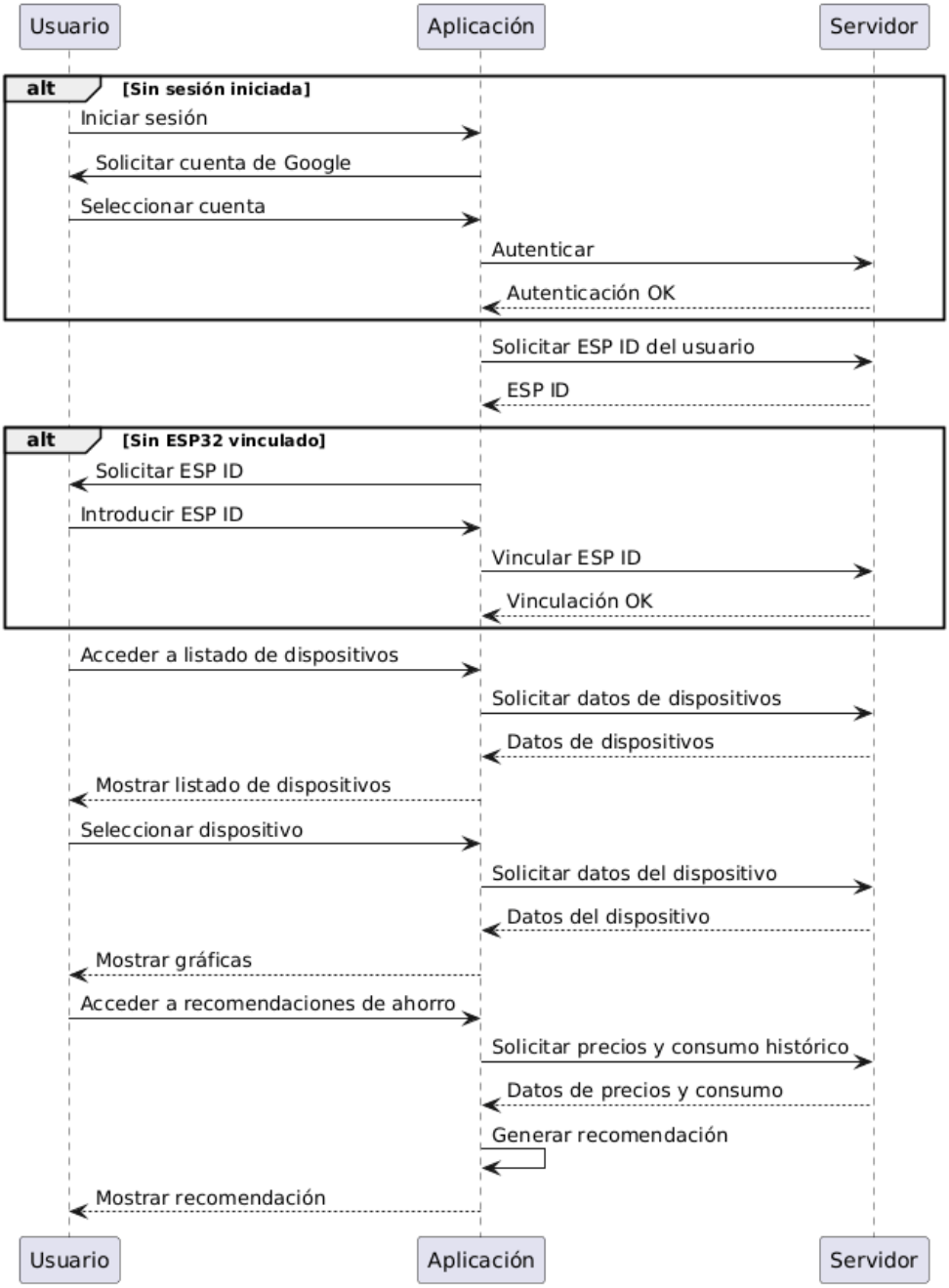

Fig. 15. Diagrama de secuencia de la navegación principal de la aplicación

2.5. Diagramas de componentes

Diagrama de componentes del ESP32 y el servidor

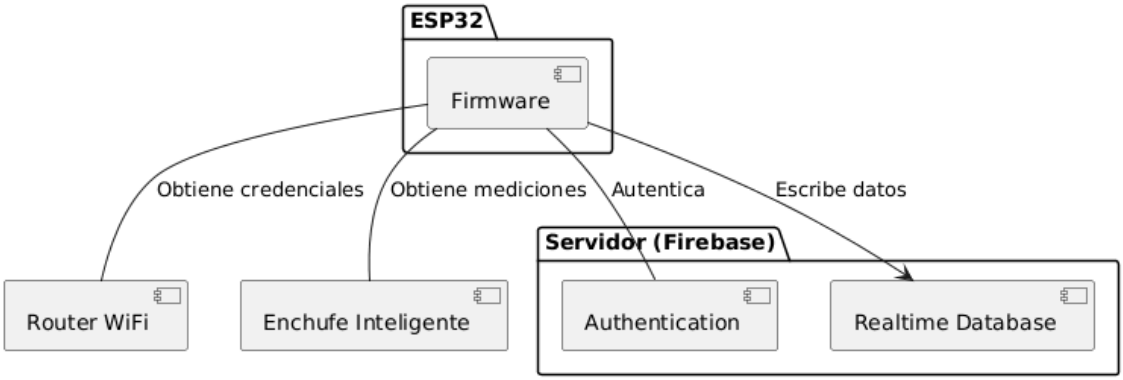

Fig. 16. Diagrama de componentes del ESP32

Diagrama de componentes de la aplicación móvil y el servidor

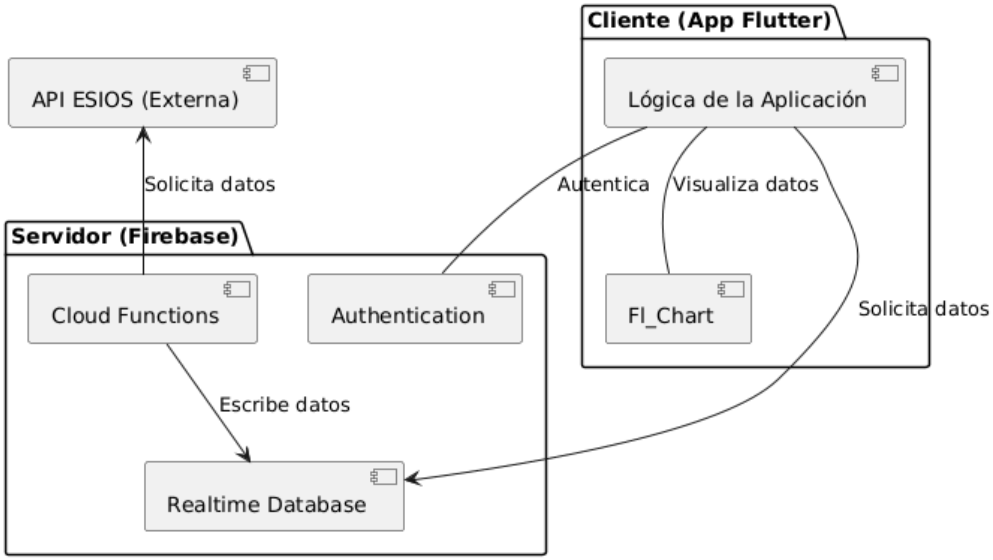

Fig. 17. Diagrama de componentes de la aplicación.

3

Implementación

3.1. Servidor y base de datos

3.1.1. Configuración de Firebase

Para crear un proyecto Firebase, se debe acceder a *firebase.google.com* e iniciar sesión con una cuenta de Google. Seleccionando *ir a la consola* accederemos a la consola de Firebase, donde se podrá crear un nuevo proyecto. En este caso se le dio el nombre *tfg-iot*, para el que Firebase generó el ID de proyecto *tfg-iot-83325*. Se deberán activar tres servicios dentro de la categoría *compilación*: *Authentication* para la autenticación de usuarios, *Realtime Database* para la base de datos, y *Functions* para las solicitudes a la API.

El plan de precios de Firebase inicial es *Spark*, que es gratuito si no se supera el límite de uso mensual. Si se supera, el servicio que haya superado el límite se desactivará hasta que restablezca en el siguiente ciclo de facturación [36]. Pero utilizar el servicio *Functions* requiere cambiar el plan de precios a *Blaze* [37]. que es gratuito hasta el mismo límite que *Spark*. La diferencia es que, si se supera, se cobrará según el uso a una cuenta de *Cloud Billing* que se deberá asociar. Para evitar costes inesperados, se puede configurar un presupuesto que, al superarse, resultará en una notificación por correo electrónico [38]. Sin embargo, el límite gratuito ha sido suficiente durante el desarrollo y las pruebas de este proyecto.

Para crear la base de datos, se seleccionó la región europe-west-1 y se configuraron las reglas de seguridad en modo de prueba. En el apartado 3.1.4 *Reglas de seguridad* se explicarán las reglas definitivas más detalladamente. Tras crearse la base de datos, Firebase generó la URL de conexión: https://tfg-iot-83325-default-rtdb.europe-west1.firebasedatabase.app

Para la autenticación se deben habilitar dos proveedores de usuarios:

- **Correo electrónico/contraseña**, que utilizarán los ESP32 para autenticarse.
- **Google**, para los usuarios de la aplicación móvil. Firebase solicitará un nombre público para la aplicación y un correo electrónico de asistencia.

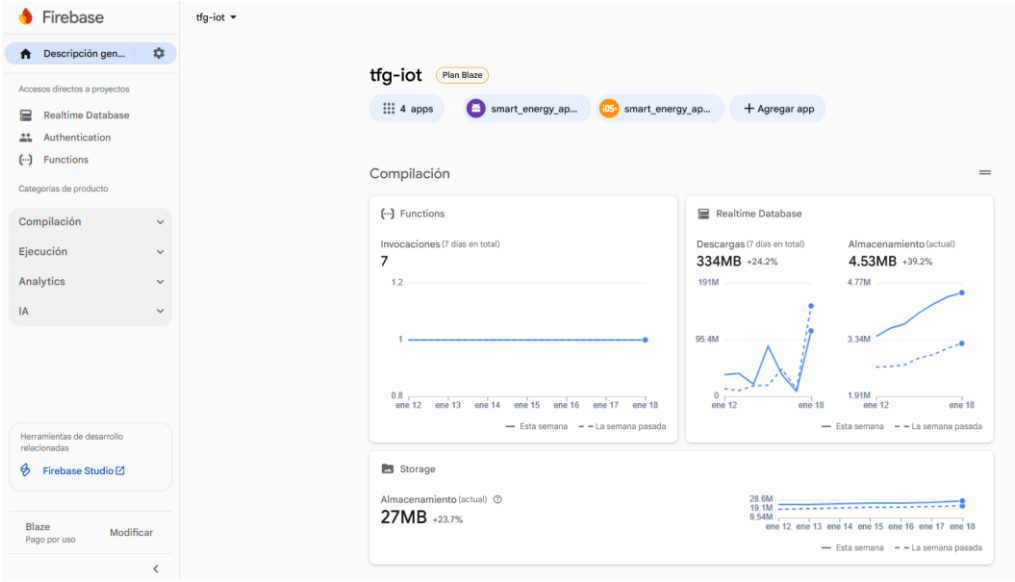

Fig. 18. Apariencia de la consola de Firebase tras configurar los Servicios

Para habilitar la conexión con Firebase desde los ESP32 y Android, se deberán crear dos *apps*: una *app* Web para los ESP32 y otra Android para la aplicación móvil. Se pueden crear accediendo a *Configuración del proyecto -> General -> Tus apps -> Agregar app*. Para ambas apps se proporcionó el nombre *smart_energy_app*. Al generar cada una, se obtendrán parámetros importantes como la *api key*, que se utilizarán más adelante.

Se configuraron además las copias de seguridad con una frecuencia diaria.

3.1.2. Estructura de la base de datos

Realtime Database [39] es una base de datos no relacional, por lo que no utiliza tablas ni relaciones para modelar los datos. En su lugar, los datos se almacenan en formato JSON, formando una estructura jerárquica en forma de árbol donde cada nodo, representado mediante una clave, podrá tener asociado un valor u otros subnodos.

La base de datos contiene cuatro colecciones principales:

- **users**: Asocia a cada usuario registrado con su ESP32.
- **device_info**: Asocia a cada ESP32 con los dispositivos (enchufes inteligentes) conectados a él.
- **devices**: Contiene los datos de todos los ESP32 registrados en la aplicación y a su vez, de todos los dispositivos vinculados a ellos.
- **energy_price**: Contiene los datos históricos del precio de la energía.

Users se utiliza para asociar cada usuario de la aplicación con su ESP32, estableciendo una relación entre los ID de usuario únicos (*uid*) generados por Firebase y los *esp_id*.

```
"WKlt3d7otBhWAViJE2H2NuP3CZG3": {
  "linked_esp": "ESPABCDE"
}
```

Devices contiene los datos de los ESP32. Cada uno se representa con su *esp_id*, y contiene un *owner_uid* que hace referencia al *uid* del usuario al que pertenece. Además, contiene los datos

de cada enchufe o *device*, cada uno representado por un *device_id*. Cada enchufe contiene un nodo *name* con un nombre personalizado dado por el usuario, y un campo *data* donde se almacenan todas las mediciones de consumo. Cada medición se representa por una clave generada por Firebase a la que se asocian tres nodos:

- Power: medición de potencia instantánea en vatios.
- Energy: energía acumulada hasta ese instante en vatios-hora.
- Timestamp: marca de tiempo en formato *UNIX timestamp*, en milisegundos.

Fig. 19. Estructura jerárquica de un ESP32 con dos enchufes asociados.

Device_info: Cuando un ESP32 se conecta a Firebase, *registrará* sus dispositivos escribiendo los *deviceId* de los dispositivos conectados a él en la colección correspondiente a su *espId*.

Fig. 20. Estructura jerárquica de la colección device_info.

Esta colección puede parecer redundante porque sus datos ya están contenidos en *devices*, sin embargo, tiene una utilidad que se explica en el capítulo *3.5.4 Listado de dispositivos*.

Energy_price es una colección de parejas clave-valor, donde cada clave es una marca de tiempo en formato ISO 8601 y el valor es el precio de la energía correspondiente en €/MWh. Los datos se obtienen periódicamente de la API de Red Eléctrica Española.

3.1.3. Usuarios y autenticación

Tanto los ESP32 como la aplicación móvil se conectarán a Firebase para leer y escribir en la base de datos, pero para hacerlo se requiere que estén *autenticados* utilizando el servicio *Firebase Authentication* [40]. Los ESP32 y la aplicación móvil se autentican de forma diferente:

Los **ESP32** se autentican mediante usuario (dirección de email) y contraseña. Para que un ESP32 pueda usarse en el proyecto, deberá existir un usuario correspondiente en Firebase, que deberá ser creado por un administrador. Para crear estos usuarios, se especifican direcciones de email y contraseñas únicas para cada ESP32. Al mismo tiempo, las credenciales del usuario de cada ESP32 deberán incluirse en su firmware, de forma que al iniciarse pueda utilizarlas para autenticarse. Las direcciones de email se utilizan únicamente como nombre de usuario, no es necesario que puedan enviar o recibir correos electrónicos.

Los usuarios de la **aplicación móvil** se autenticarán mediante una cuenta de Google. Si las credenciales de su cuenta son correctas, Firebase creará automáticamente un usuario, asignándole un *uid* (ID de usuario) único.

Para poder acceder a los datos de un ESP32, el usuario deberá conocer su *esp_id* y haberlo vinculado a su cuenta.

3.1.4. Reglas de seguridad

Para limitar el acceso a la base de datos, *Realtime Database* permite especificar reglas de seguridad en formato JSON [41]. El propósito fundamental es impedir que usuarios no autenticados accedan a la base de datos, pero se pueden definir reglas más específicas, de forma que solo se permita el acceso a ciertas colecciones o subcolecciones de datos de acuerdo a condiciones concretas [42]. En el proyecto se define el JSON de reglas como:

```
  "rules": {
    "users": {
      "$uid": {
        ".read": "auth != null && auth.uid === $uid",
        ".write": "auth != null && auth.uid === $uid"
      }
    },
    "devices": {
      "$esp_id": {
        ".read": "auth != null && data.child('owner_uid').val() === auth.uid",
        "$device_id": {
          "data": {
            ".write": "auth != null && newData.exists()",
            ".indexOn": ["timestamp"]
          }
        },
        "owner_uid": {
          ".read": "auth != null",
          ".write": "auth != null && (!data.exists() || (!newData.exists() &&
(data.val() === auth.uid)))",
          ".validate": "newData.exists()? newData.val() === auth.uid : true"
        }
      }
    },
    "energy_price": {
      ".read": "auth != null",
```

```
      ".write": "auth != null",
    }
  }
```

En la colección **users**, se debe impedir que un usuario modifique los datos de otro usuario, lo que implica que, si tiene un cierto *uid,* solo puede acceder a */users/uid* y no a otros. En las reglas de seguridad se expresa de la forma:

```
"users": {
      "$uid": {
        ".read": "auth != null && auth.uid === $uid",
        ".write": "auth != null && auth.uid === $uid"
      }
}
```

Primero se comprueba que *auth* no sea nulo, lo que significa que el usuario está autenticado [42], o bien con Google o bien por usuario y contraseña si es un ESP32. La segunda condición limita que solo pueda acceder a los datos correspondientes a su *uid.*

Las reglas para la colección **devices** sirven para impedir que un usuario acceda a datos de un ESP32 que no le pertenece. Para demostrar que un ESP32 le pertenece, el usuario deberá vincularlo a su cuenta, lo que internamente se traduce a establecer una relación entre su *uid* y su *esp_id* de forma bidireccional:

/users/$uid/linked_esp = esp_id.
/devices/$esp_id/owner_uid = uid

Si un usuario que tiene vinculado un *esp_id* e intenta acceder a *devices/otro_esp_id,* que ya está vinculado con otro usuario, será rechazado por esta regla:

```
"devices": {
      "$esp_id": {
        ".read": "auth != null && data.child('owner_uid').val() === auth.uid",
       // …
       }
}
```

Firebase determinará que el *owner_uid* de ese otro ESP32 es distinto al *uid* del usuario que hace la consulta.
Esta regla establece el requisito de que un usuario deberá vincular un ESP32 antes de acceder a sus datos, pero aún no está garantizada la seguridad porque nada impediría vincular cualquier *esp_id,* incluso si ya está vinculado con otro usuario. Por tanto, se debe añadir más seguridad al proceso de vinculación, concretamente al campo *owner_uid.*

En la vinculación, se escribirá el *uid* del usuario en *owner_uid.* En este caso se debe comprobar que no exista ya otro *owner_uid,* lo que indicaría que ese ESP32 ya está vinculado. Pero hay un caso donde se sí debería autorizar a un usuario a escribir en *owner_uid* aunque ya exista: cuando quiera desvincular. En Realtime Database, asignar *null* a una clave hace que se elimine. [44] Entonces, se debe comprobar que, en caso de intentar escribir *null,* el *owner_uid* ya

existente es igual que el *uid* del usuario que está desvinculando. La regla se implementa de la forma:

```
"owner_uid": {
            ".write": "auth != null && (!data.exists() || !newData.exists() &&
data.val() === auth.uid)",
            ".validate": "newData.exists()? newData.val() === auth.uid : true"
          }
```

Siguiendo la lógica de que para escribir en *owner_uid*, o bien se está vinculando (*owner_uid* no existe) o bien se está desvinculando (se va a escribir *null* para borrar y además los *uid* coinciden)

Aunque *.write* conceda acceso de escritura, se puede evaluar el valor concreto que se está escribiendo con *.validate*. En este caso, se valida que si el *uid* que se va a escribir no es *null*, se corresponde con el *uid* del usuario que escribe.

Aún podría existir otra forma de vincular un ESP32 de forma no autorizada: un usuario malicioso podría, sin utilizar la aplicación, escribir directamente en *users/uid* cualquier *esp_id* porque no hay reglas que se lo impidan. Sin embargo, la colección *users* sirve únicamente para guardar qué *esp_id* está asociado con el usuario y no tenga que introducirlo manualmente cada vez que inicia la aplicación. No determina si tiene acceso o no a los datos de ese ESP32, para eso se utiliza el campo *owner_uid* que sí está protegido.

En cuanto a escrituras, el campo *data* está protegido de forma que solo se puedan insertar nuevos datos y no modificar o borrar datos ya existentes con la regla:

```
".write": "auth != null && newData.exists()”
```

Todos los usuarios podrán leer la colección **energy_price** siempre que estén autenticados, por tanto, no necesita ninguna limitación además de que *auth* no sea *null*. Sin embargo, ningún usuario podrá escribir excepto la *cloud function* encargada de insertar datos desde la API.

```
"energy_price": {
  ".read": "auth != null",
  ".write": false
}
```

La *cloud function* sí podrá escribir porque utiliza una cuenta de administrador, como se explica más detalladamente en el capítulo *3.4.1 Funciones Cloud*.

3.1.5. Índices

Existen otros tipos de reglas con usos distintos a limitar el acceso. Una de ellas es *.indexOn*, que permite optimizar el rendimiento de las consultas definiendo *índices* sobre determinados campos en los datos [45].

En este caso, se define un índice para el campo *timestamp* dentro de las mediciones de consumo en la colección *data* de cada enchufe.

```
"data": {
  ".indexOn": ["timestamp"]
}
```

Definir el índice resulta útil cuando se hacen consultas a las mediciones de consumo, ya que el resultado de la consulta debe estar ordenado y acotado a un cierto rango. Firebase puede utilizar el índice para localizar los datos de forma más eficiente, evitando iterar de forma innecesaria por todos los registros de la colección para determinar si están dentro del rango.

3.2. Dispositivos hardware

3.2.1. Enchufes inteligentes

Para el proyecto se utilizarán enchufes inteligentes Shelly Plug Plus S [46]. Utilizando un ESP32 internamente, incorporan conectividad WiFi y Bluetooth, ofreciendo funcionalidades como medición del consumo de energía, creación de temporizadores [47] y la posibilidad de ser encendidos y apagados de forma remota, pero este proyecto se centrará en la medición del consumo.

Fig. 21. Enchufe inteligente Shelly Plug Plus S

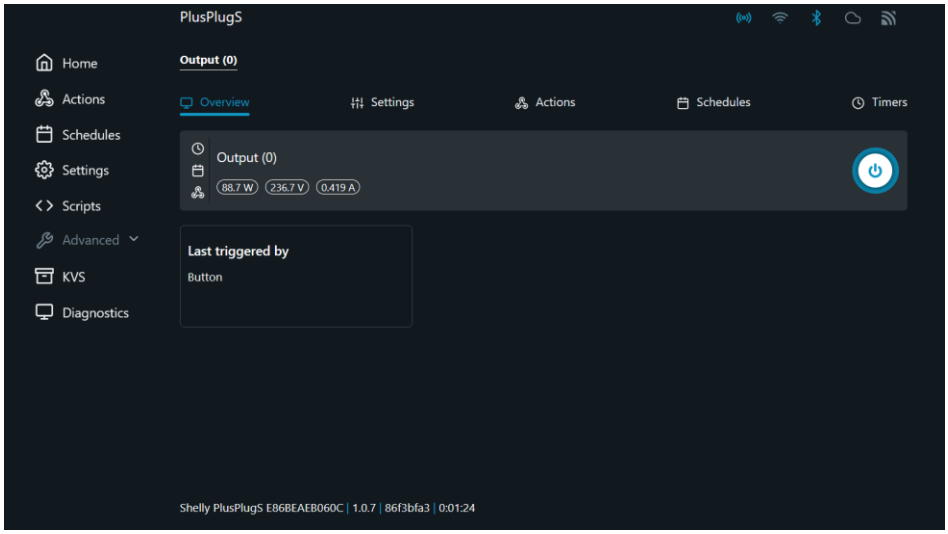

Fig. 22. Página principal de la interfaz web de Shelly

Ofrecen tanto una interfaz web [48] como una aplicación móvil [49] para configurar el enchufe y visualizar los datos. La diferencia entre estos enchufes y otros fabricantes es que Shelly ofrece una API abierta [50], de forma que se podrá simplemente hacer peticiones HTTP a ciertos endpoints y obtener los datos fácilmente, sin necesidad de depender de las aplicaciones de Shelly. Lograr lo mismo con enchufes de otros fabricantes podría requerir flashear su firmware y sustituirlo por una solución de código abierto como *ESPHome* [51] o *Tasmota* [52].

3.2.2. ESP32

Desde la interfaz web de Shelly se puede especificar un endpoint que el enchufe puede usar para enviar sus mediciones mediante el servicio de *Webhooks* [53]. Ese endpoint podría corresponder a una base de datos directamente, pero en su lugar, este proyecto emplea un ESP32 que actuará como nodo intermedio debido a dos principales ventajas:

- **Automatización de la conexión:** Permite automatizar todo el proceso de conexión de forma que el usuario no deberá introducir credenciales. Bastará con simplemente conectar los enchufes y encender el ESP32, y en el caso de la primera conexión, pulsar el botón WPS en el router WiFi.
- **Agrupamiento de dispositivos:** Los ESP32 tendrán un identificador único que servirá a la base de datos para determinar fácilmente que los enchufes conectados a ese ESP32 pertenecen al mismo usuario. Además, podrá autenticar al ESP32 y asociarlo a su usuario, evitando tener que hacerlo para cada enchufe.

3.2.3. Pantalla OLED

Se ha incorporado en el hardware del proyecto una pantalla OLED [54] de 1.3 pulgadas y 128 x 64 píxeles de resolución que muestra al usuario información sobre el proceso de conexión para informar de posibles errores, así como, una vez completada la conexión, muestra el consumo de energía medido.

Fig. 23. Pantalla OLED

La pantalla utiliza cuatro pines para conectarse con el ESP32, dos para alimentación (VDD y GND) y otros dos para transferencia de datos (SCK y SDA) mediante el protocolo serie I2C.

ESP32	Pantalla
G26	SCK
G25	SDA
3V3	VCC
GND	GND

Tabla 1. Correspondencia entre los pines del ESP32 y la pantalla

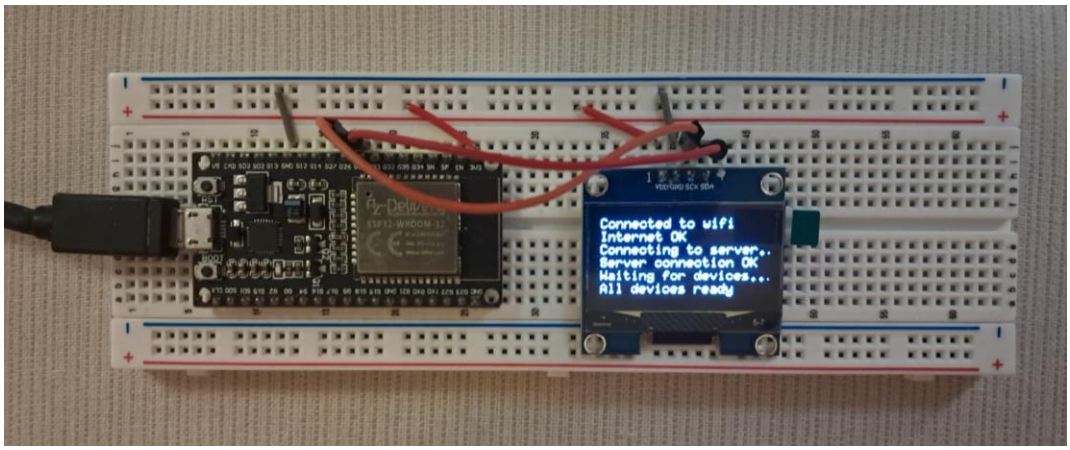

Fig. 24. Pantalla OLED conectada al ESP32

Para trabajar con la pantalla desde el firmware se utilizan las librerías:

- Wire.h: Incluida en el framework de Arduino, sirve para gestionar el protocolo I2C.
- U8g2lib.h [55]: Incluye fuentes preinstaladas y funciones para escribir texto en la pantalla.

La interfaz de la pantalla emula una terminal, es decir, que cuando se imprime una nueva línea, ésta aparece en la parte inferior, desplazándose el resto de líneas una posición hacia arriba. Para implementarlo, se utiliza un búfer de 6 cadenas de texto, que es el número máximo de líneas que caben en la pantalla. Al insertar una nueva línea, se desplazan todos los elementos del búfer una posición a la izquierda, descartando el primero y dejando la última posición disponible para la nueva línea. Después se redibuja toda la pantalla llamando a la función *drawStr* de la librería U8g2, especificando las coordenadas y texto de cada línea. Esta funcionalidad se implementa en la función *displayLine*, que se utiliza en el resto del firmware para mostrar mensajes de estado al usuario.

Tras una conexión exitosa con dos enchufes, la información mostrada será similar a:

```
Scanning devices...
Devices found: 2
Connecting to 97B0...
Connected to 97B0
Configuring device
Device 97B0 config OK
Connecting to 060C...
Connected to 060C
Configuring device
Device 060C config OK
Connecting to MyWifiSSID...
Connected to wifi
Internet OK
Connecting to server...
Server connection OK
Waiting for devices...
All devices ready
97B0: 27.3 W
060C: 140.5 W
97B0: 26.9 W
060C: 123.0 W
...
```

3.3. Firmware del microcontrolador

3.3.1. Proceso de conexión

Cuando un enchufe Shelly de fábrica se conecta a la corriente se activa su modo AP (*Access Point* o punto de acceso). En este modo, crea una red WiFi con un SSID de la forma *ShellyPlusPlugS-<id>*. Ese ID se usará como identificador único para los enchufes (*device_id*). Conectándose a esa red wifi y accediendo con un navegador web a la IP 192.168.33.1, se puede acceder a la interfaz web de Shelly en la que se configuran ciertos parámetros como las credenciales del punto de acceso al que se enviarán las mediciones. Aprovechando la API HTTP que proporciona Shelly, en este proyecto se podrá automatizar el proceso de conexión para que el usuario no tenga que acceder a la interfaz web ni introducir ninguna credencial [56].

Una vez configurado el enchufe, éste pasará a modo STA (*Station*) y se conectará al ESP32, que periódicamente hará peticiones de los datos de consumo.

En cuanto al ESP32, permite dos modos WiFi [57]:

- Modo AP (WIFI_AP). Permite que otros dispositivos WiFi (en modo *Station*) se conecten a él.
- Modo Station (WIFI_STA). Permite al ESP32 conectarse a otros AP.

También soporta un modo mixto WIFI_STA_AP, que combina ambos modos al mismo tiempo.

Teniendo en cuenta lo anterior, el proceso seguido para automatizar la conexión es el siguiente:

Paso 1. Conectar los enchufes Shelly a la corriente y asegurarse de que estén en modo AP. Si no lo estuvieran, se deben restablecer a su estado de fábrica pulsando el botón durante al menos 10 segundos.
Paso 2. Iniciar el ESP32. Éste comprobará si tiene almacenadas las credenciales WiFi del router. En caso contrario, el ESP32 pasará a modo WPS y la conexión se deberá realizar pulsando el botón WPS del router. El ESP32 obtendrá las credenciales y las almacenará.
Paso 3. El ESP32 pasará al modo STA y buscará los AP de los enchufes, cuyos SSID empiezan con *ShellyPlusPlugS*.
Para cada AP encontrado:

- **Paso 4.** El ESP32 se conectará al AP del enchufe.
- **Paso 5.** El enchufe necesita las credenciales de un AP al que enviará los datos, que se pueden enviar al endpoint http://192.168.33.1/rpc/WiFi.SetConfig En formato JSON, se envían el SSID y la contraseña que tendrá el AP del ESP32. Nótese que AP del ESP32 aún no se activa.
- **Paso 6.** No es posible conectar a varios AP al mismo tiempo, así que, una vez enviadas las credenciales, el ESP32 se desconecta del AP del enchufe antes de pasar al siguiente.

Una vez se han configurado todos los enchufes:
Paso 7. Continuando en modo STA, el ESP32 se conecta al AP del router WiFi con las credenciales almacenadas en el paso 2 y se prueba la conexión a internet. Esta conexión debe hacerse después de configurar los enchufes porque en caso contrario, se perdería la conexión al AP del router cuando el ESP32 deba conectarse a los AP de los enchufes para configurarlos.

Paso 8. Finalmente, el ESP32 cambia al modo mixto STA_AP, activándose así su AP y espera a que los enchufes se conecten a él usando las credenciales proporcionadas antes.
La activación del AP del ESP32 debe hacerse al final porque no puede conectarse a otro AP mientras el suyo está activo, pero sí mantener la conexión.
Paso 9. Con los enchufes conectados, ya se pueden recoger datos de consumo. Periódicamente se harán peticiones a cada enchufe usando su IP estática, se procesarán las respuestas y se enviarán al servidor.

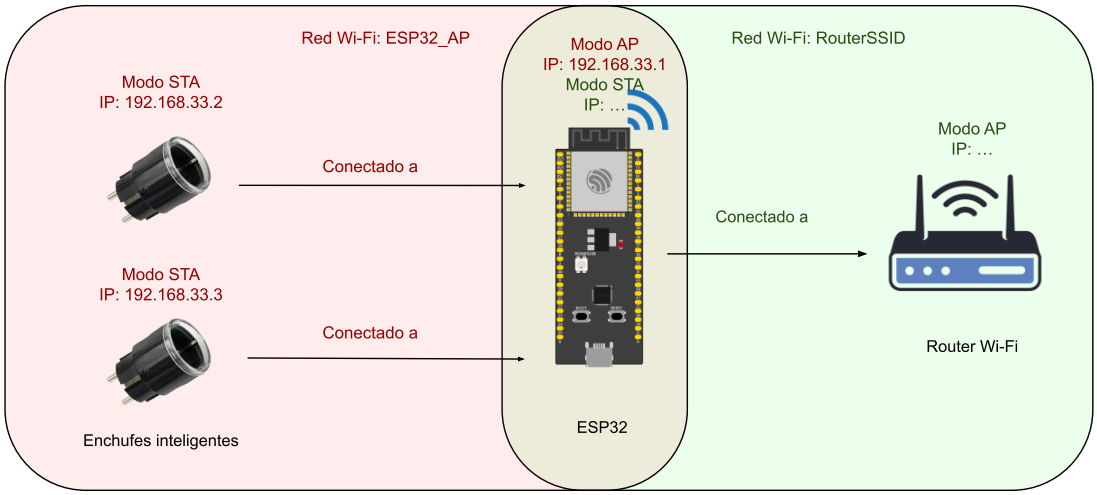

Fig. 25. Conexiones del ESP32 al finalizar el proceso

Los detalles de cada paso del proceso se explican más adelante.

3.3.1.1. WPS y memoria persistente

El ESP32 deberá obtener las credenciales de la red WiFi del usuario. Para obtenerlas fácilmente, se utiliza el protocolo WPS (*WiFi Protected Setup*)[58], que permite conectar dispositivos a una red WiFi sin necesidad de escribir la contraseña. Para hacerlo, se deberá pulsar en el router un botón dedicado que activará el modo WPS temporalmente. Mientras esté activo, los dispositivos que se conecten obtendrán las credenciales automáticamente. Al iniciar el ESP32, comprobará si ya tiene almacenadas las credenciales, y en caso negativo, iniciará el protocolo WPS para obtenerlas. De esta forma, se evita tener que volver a obtener las credenciales tras cada reinicio.

Para conseguir esto, no será suficiente con almacenar las credenciales en la memoria RAM, porque al ser volátil, los datos se eliminan al desconectar la alimentación o reiniciar. Se deberá entonces utilizar la memoria NVS (*Non-Volatile Storage*, Almacenamiento No Volátil), que es persistente entre cada reinicio. Su desventaja es que soporta un número relativamente limitado de escrituras antes de degradarse en comparación con la RAM, pero no resulta un problema en este proyecto ya que solo se utilizará ocasionalmente.

Para poder trabajar con la NVS se utiliza la librería *preferences.h*. Esta librería modela la memoria como un diccionario de parejas clave-valor. Estas parejas se agrupan en espacios de nombres o *namespaces*. Antes de leer o escribir se debe abrir un espacio de nombres, que debe cerrarse al terminar.

```
Preferences preferences;
// false = modo lectura/escritura, true = modo solo lectura
preferences.begin("wifiConfig", false);
preferences.putString("localWifiSSID", (char*)wifi_config.sta.ssid);
preferences.putString("localWifiPass", (char*)wifi_config.sta.password);
preferences.end();
```

Para cambiar el ESP32 a modo WPS, se utilizan las funciones esp_wifi_wps_enable(&config) y esp_wifi_start(0). Se define una variable global *wpsSuccess* que solo será verdadera tras haber realizado la conexión exitosamente. Mientras tanto, el programa se bloquea en un bucle *while (!wpsSuccess)*. Esta variable se marca como *volatile* para evitar que el compilador haga una optimización errónea al suponer que siempre es falsa. Al llamar a *esp_wifi_start*, la función inicia internamente tarea con FreeRTOS donde se realiza la conexión. Esta tarea lanza eventos de tipo *WifiEvent_t* que se deben capturar. Para ello, se define una función *callback* llamada WiFiEvent, a la que la otra tarea llamará cuando se produzca un evento. Antes de iniciar el WPS, se llama a WiFi.onEvent(WiFiEvent) para indicar que este es el *callback* a utilizar. Algunos de los eventos que se pueden producir son: [59][60]

- *ARDUINO_EVENT_WIFI_STA_START*: El ESP32 se ha cambiado a modo Station.
- *ARDUINO_EVENT_WIFI_STA_GOT_IP*: El ESP32 ha obtenido una IP local.
- *ARDUINO_EVENT_WIFI_STA_DISCONNECTED*: Después de conectar se ha producido una desconexión.
- *ARDUINO_EVENT_WPS_ER_FAILED*: Error al conectar. Se reinicia el WPS.
- *ARDUINO_EVENT_WPS_ER_TIMEOUT*: El router lleva mucho tiempo sin responder. Se reinicia el WPS.

El evento más relevante es *ARDUINO_EVENT_WPS_ER_SUCCESS*, que se produce cuando la conexión WPS ha sido exitosa. En este caso, se detiene el WPS con *wpsStop()* y se cambia *wpsSuccess* a verdadero. Se accede a las credenciales obtenidas por WPS utilizando:

```
wifi_config_t wifi_config;
esp_err_t err = esp_wifi_get_config(WIFI_IF_STA, &wifi_config);
```

El SSID se obtiene con *wifi_config.sta.ssid*, y la contraseña con *wifi_config.sta.password*. Estas credenciales se guardan utilizando *preferences*, como se explicó anteriormente.
Con *wpsSuccess* verdadero, el bucle *while* anterior termina y el programa continúa.

3.3.1.2. Conexión con los enchufes inteligentes

El objetivo del proceso de conexión será que se cree una red WiFi local con el ESP32 en modo AP, de forma que los enchufes, estando en modo STA, envíen sus datos de consumo al ESP32. Dentro de esta red, al ESP32 siempre se le asigna la IP 192.168.33.1, y a los enchufes las direcciones 192.168.33.2 y sucesivas. Para lograrlo, se deberán encontrar los enchufes y enviarles unas credenciales para que puedan conectarse al ESP32.

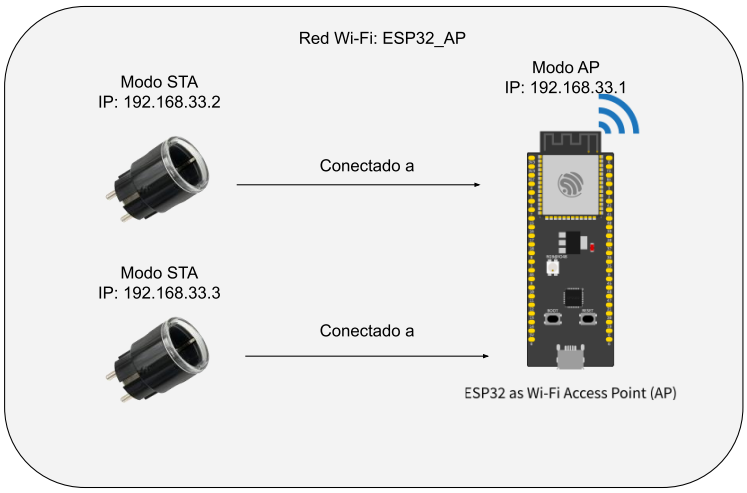

Fig. 26. Resultado después de conectar dos enchufes al ESP32

En el firmware, la conexión con los enchufes inteligentes se gestiona en el fichero *shelly_connection.hpp*. Se utilizan funciones definidas en el archivo de encabezado *WiFi.h*, incluido en el framework de Arduino. El ESP32 se cambia a modo Station con *WiFi.mode(WIFI_MODE_STA)*. Se escanean las redes WiFi utilizando *WiFi.scanNetworks()*, que devuelve el número de redes encontradas o en el caso de haber encontrado un error, un número negativo. A continuación, se itera sobre todas las redes encontradas buscando las que correspondan a puntos de acceso de enchufes Shelly, cuyos SSID siempre empiezan por el prefijo *ShellySmartPlugS-*. Los SSID de esas redes se almacenan en un vector.

Se itera sobre ese vector de SSIDs, y para cada uno:

- El ESP32 intentará conectarse a esa red utilizando *WiFi.begin(ssid)*. No hay que incluir una contraseña porque los puntos de acceso de los enchufes Shelly son abiertos.
- El proceso de conexión puede durar un tiempo. El programa espera en un bucle while hasta que el proceso se complete, comprobando si *WiFi.status() == WL_CONNECTED*. Si el proceso se extiende demasiado, se implementa un tiempo límite que termina el bucle while y continúa el programa para intentarlo con el siguiente enchufe.
- Una vez conectado al enchufe, se puede llamar a sus endpoints para configurarlo en lugar de que el usuario tenga que hacerlo manualmente. El endpoint relevante es */rpc/WiFi.SetConfig*. En este momento, el enchufe tendrá la IP 192.168.33.1 por ser el punto de acceso, así que la petición se deberá hacer a la URL *http://192.168.33.1/rpc/WiFi.SetConfig* [61]. Para hacer peticiones HTTP se emplea el archivo de encabezado *HTTPClient.h*, incluido también en el framework de Arduino, de la forma:

```
HTTPClient http;
String url = "http://192.168.33.1/rpc/WiFi.SetConfig";
http.begin(url);
http.addHeader("Content-Type", "application/json");
String body = …;
int httpCode = http.POST(body);
```

El cuerpo de la petición HTTP, que debe ser POST, se envía en formato JSON y contiene:

```
{
  "config": {
    "sta": {
      "ssid": ESP_SSID,
      "pass": ESP_PASS,
      "enable": true,
    }
  }
}
```

Donde con ESP_SSID y ESP_PASS se especifican las credenciales de la red WiFi local que creará el ESP32 y a la que posteriormente se conectará el enchufe. Después de hacer la petición, el cliente HTTP devolverá el código de respuesta con el que se hará el control de errores.

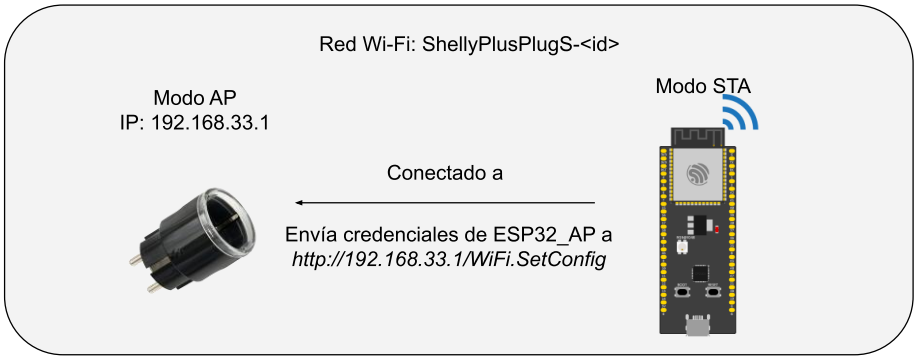

Fig. 27. Envío de credenciales a enchufe inteligente

- Una vez enviada la configuración, el ESP32 se desconecta del enchufe con *WiFi.disconnect()* y se continúa con el siguiente SSID de la lista.

3.3.1.3. Conexión y autenticación con Firebase

Para conectarse con Firebase, el ESP32 deberá conectarse primero al router para conseguir acceso a internet. En este momento ya estarán las credenciales almacenadas, así que es suficiente con leerlas de la memoria NVS con *preferences* y llamar a *WiFi.begin(routerSsid, routerPassword)* En un bucle while, se espera a que la conexión se complete de forma similar a los enchufes.

A continuación, se hace una petición POST a la siguiente URL: https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key=<FIREBASE_API_KEY> utilizando la API key de la app web creada en el apartado *3.1.1 Configuración de Firebase*. El cuerpo de la petición será un JSON con el siguiente formato [62]:

```
{
  “email”: FIREBASE_EMAIL,
  “password”: FIREBASE_PASS,
  “returnSecureToken”: true
}
```

Si la autenticación ha sido correcta, Firebase devolverá una respuesta con código 200 conteniendo un JSON con los campos *idToken, localId, refreshToken* y *expiresIn.* Esos campos se extraerán utilizando la librería ArduinoJSON que se explica en el siguiente apartado. Para leer y escribir datos en Firebase como usuario autenticado, se deberá incluir el *idToken* en cada petición [63].

Periódicamente será necesario renovar la sesión, para ello se utilizan los campos *refreshToken* y *expiresIn*. Se calcula el instante en que la sesión caducará sumando *expiresIn* al tiempo en que se creó la sesión y restando 5 minutos como margen. En cada petición a Firebase, se comprueba si la sesión ha caducado. Si es el caso, se renueva haciendo una petición a *https://securetoken.googleapis.com/v1/token?key=<FIREBASE_API_KEY>* [64] incluyendo el *refreshToken* en el cuerpo de la forma:

```
grant_type=refresh_token&refresh_token=<refreshToken>
```

Google devolverá un nuevo *idToken, refreshToken* y *expiresIn* para renovar la sesión.

3.3.2. Recogida de datos y envío a Firebase

Teniendo la IP estática de un enchufe, podemos obtener sus datos de consumo haciendo una petición al endpoint *http://<SHELLY_IP>/rpc/Switch.GetStatus?id=0* [65]
Se recibe una respuesta en JSON con este formato (algunos campos se han omitido):

```
{
  "output": false,
  "apower": 0,
  "aenergy": {
    "total": 11.679,
    "by_minute": [0, 0, 0],
    "minute_ts": 1654511972
  },
}
```

Esta respuesta se pasa a la función *parseShellyData* como *payload.* Para poder extraer valores de datos JSON, se utiliza la librería *ArduinoJson* [66], que ofrece la función deserializeJson.

```
// documento estático: se especifica un tamaño concreto que no podrá crecer
StaticJsonDocument<1024> inputJson;
DeserializationError error = deserializeJson(inputJson, httpPayload);
if (error) {
  Serial.println("Error parsing shelly JSON");
  return "{}";
}
```

Una vez deserializado, se puede utilizar el documento para acceder a los campos del JSON.

```
float power = inputJson["apower"];
float energy = inputJson["aenergy"]["total"];
```

Con los datos obtenidos, se crea otro documento JSON para enviarlo a firebase. El proceso es el inverso al anterior: en lugar de *deserializar* una cadena para obtener un documento y de ahí extraer los campos, se crea un documento, se insertan campos y finalmente se *serializa* a una cadena:

```
StaticJsonDocument<1024> outputJson;
// asigna el consumo obtenido a la clave power. si no existe, se crea.
outputJson["power"] = power;

JsonObject timestamp = outputJson.createNestedObject("timestamp");
timestamp[".sv"] = "timestamp";

String serializedOutputJson;
serializeJson(outputJson, serializedOutputJson);
return serializedOutputJson;
```

Para obtener la marca de tiempo, se inserta un objeto anidado que contiene una clave *.sv*:

```
"timestamp": {
  ".sv": "timestamp"
}
```

Firebase sustituirá este objeto por una clave *timestamp* a la que asociará la marca de tiempo del instante en que recibió los datos [67].

Este JSON serializado pasa a la función *sendJson*, que se encargará de enviarlo a firebase. Esta función, utilizando *HTTPClient*, hace una petición POST a la URL *<FIREBASE_URL>/devices/<ESP_ID>/<DEVICE_ID>/data.json?auth=<ID_TOKEN>* *donde*

- *FIREBASE_URL* es la URL de conexión a Firebase.
- ESP_ID. Es el identificador único del ESP32 actual.
- DEVICE_ID Es el ID del enchufe al que corresponden los datos, que corresponde al ID único dado por el SSID del punto de acceso del enchufe sin el prefijo.
- ID_TOKEN. El token de acceso obtenido anteriormente tras autenticarse en Firebase.

El cuerpo de la petición será la cadena que devolvió la función *parseShellyData*.

3.4. API de Red Eléctrica

En el proyecto se empleará la API de la Red Eléctrica Española para obtener los datos del precio de la electricidad. Para conseguir un token de acceso hay que solicitarlo escribiendo un correo electrónico a consultasios@ree.es con el asunto *Personal Token Request* [68].

Los endpoints disponibles se pueden consultar en https://api.esios.ree.es. La API ofrece distintos *indicadores*. Se puede ver la lista completa de indicadores haciendo una petición al endpoint

https://api.esios.ree.es/indicators. En todas las peticiones a la API se debe incluir el token de acceso en el header de la forma [69]:

```
Accept: application/json; application/vnd.esios-api-v1+json
Content-Type: application/json
x-api-key: "TOKEN_DE_ACCESO"
```

Al hacer la petición, obtenemos un JSON con todos los indicadores, incluyéndose su nombre (*name*), descripción (*description*), nombre corto (*short_name*) e ID.

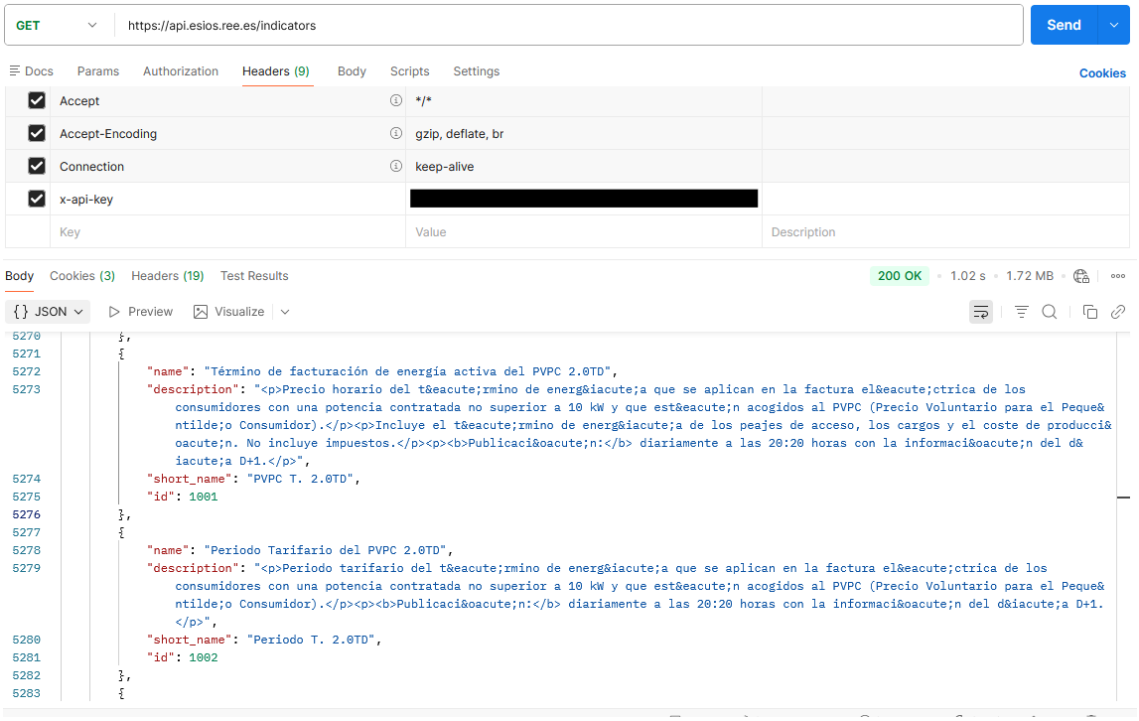

Fig. 28. Resultado de la petición realizada desde Postman

De todos los indicadores, el más relevante para el proyecto es *Término de facturación de energía activa del PVPC 2.0TD*, con ID 1001, que representa el precio de la energía en €/MWh sin impuestos. Los datos se publican diariamente a las 20:20 aproximadamente, proporcionando el precio medio de cada hora hasta las siguientes 24 horas.

Para obtener los datos de este indicador, hay que hacer una petición a https://api.esios.ree.es/indicators/1001 con parámetros que se detallarán en el siguiente apartado.

3.4.1. Funciones Cloud

Llamar a la API directamente desde la aplicación móvil es inseguro porque habría que incluir el token de acceso y existiría riesgo de llegar al límite de peticiones. Por eso, se llamará a la API desde el servidor utilizando una *cloud function* en Firebase. Las llamadas se realizarán desde el servidor periódicamente una vez al día y solo éste conocerá el token de acceso, evitando de esta forma los dos problemas anteriores. La función cloud escribirá los datos del precio obtenidos en la base de datos para que los usuarios puedan acceder a ellos desde la aplicación.

Para poder utilizar funciones cloud en el proyecto, se deben seguir los siguientes pasos [70]:

- Instalar *node*. Se puede verificar si está instalado correctamente con **node -v**
- Instalar la CLI (*Command Line Interface*) de Firebase:

  npm install -g firebase-tools
- Crear un directorio para las cloud functions (en este proyecto: "firebase")
- En ese directorio, ejecutar:

  firebase login
  firebase init functions
- Iniciar sesión en firebase y elegir el proyecto firebase correspondiente (tfg-iot)
- Se creará una carpeta *functions*
- Las peticiones se harán usando *axios*, para instalarlo:

  cd functions
  npm install axios
- La función se implementará en *index.js*

Para evitar incluir el token de acceso de la API en el código, éste se almacenará en Firebase como un *secret*. Se puede crear desde la terminal con **firebase functions:secrets:set ESIOS_TOKEN**, y posteriormente introduciendo el token.

La función utiliza una cuenta de administrador, de esta forma podrá escribir en la colección *energy_prices* aun existiendo la regla `".write": false.`

```
const admin = require("firebase-admin");
admin.initializeApp();
```

En index.js, se define una función cloud llamada *fetchEnergyPrice* que se ejecuta cada 24 horas de la siguiente forma:

```
exports.fetchEnergyPrice = onSchedule(
  {
    schedule: "0 21 * * *",
    secrets: [ESIOS_TOKEN],
    timeZone: "Europe/Madrid",
  },
  async () => {/* Cuerpo de la función... */}
)
```

El parámetro *schedule* se especifica siguiendo el formato de *Unix Crontab*. En este caso, se configura la función para que se ejecute cada día a las 21:00.
También se debe incluir una lista con todos los *secrets* que se vayan a utilizar en el cuerpo de la función. En este caso es *ESIOS_TOKEN*, que se obtiene de Firebase usando:

```
const ESIOS_TOKEN = defineSecret("ESIOS_TOKEN");
```

En el cuerpo de la función, se prepara el header de la petición:

```
const TOKEN = ESIOS_TOKEN.value();
const headers = {
  "Accept": "application/json; application/vnd.esios-api-v2+json",
```

```
  "Authorization": `Token token="${TOKEN}"`,
  "x-api-key": TOKEN,
};
```

Y los parámetros de la petición:

```
const params = {
  start_date: startDateStr,
  end_date: endDateStr,
  geo_ids: [8741],
};
```

Se especifica el intervalo de tiempo deseado, donde el inicio es el instante actual y el final es el instante actual sumadas 24 horas en formato ISO 8601. El número 8741 es un identificador geográfico que representa a la península Ibérica en general [71].

Finalmente se realiza la petición utilizando *axios*:

```
const res = await axios.get("https://api.esios.ree.es/indicators/1001", {
    headers,
    params,
});
```

Los datos obtenidos se insertan en la base de datos

```
const values = res.data.indicator.values;

const db = admin.database();
const updates = {};

values.forEach((v) => {
  updates[v.datetime_utc] = v.value;
});

await db.ref("energy_price").update(updates);
```

Para desplegar la función en la nube se utiliza el comando **firebase deploy --only functions.**

3.5. Aplicación móvil

3.5.1. Configuración de Firebase en Flutter

Para utilizar los servicios de autenticación y base de datos de Firebase en el proyecto Flutter, se deben seguir los siguientes pasos:

Añadir las dependencias correspondientes a *pubspec.yaml*.

```
dependencies:
  flutter:
    sdk: flutter
  firebase_core: ^3.6.0
  firebase_auth: ^5.3.0
  google_sign_in: ^6.2.1
  firebase_database: ^11.3.10
```

En la carpeta del proyecto, ejecutar:

dart pub global activate flutterfire_cli
flutterfire configure

Se deberán elegir las plataformas que se vayan a utilizar, en este caso Android y Web (para pruebas). Haciendo esto, existirá en firebase una *app* para cada plataforma [72].
Dentro de la carpeta *lib* se creará el archivo *firebase_options.dart*.

La autenticación en web se hace mediante OAuth 2.0, para implementarla se debe obtener un ID de cliente (*clientId*). Se obtiene en la consola de Google Cloud accediendo a https://console.cloud.google.com/apis/credentials *-> crear credenciales -> ID de cliente de OAuth.*
El cliente de OAuth se configura con:
Orígenes autorizados:

- http://localhost:5000
- http://localhost:49434

URIs de redireccionamiento autorizados:

- https://tfg-iot-83325.firebaseapp.com/__/auth/handler

Tras configurar el resto de los parámetros (nombre de la aplicación, email de asistencia...), obtendremos el ID del cliente OAuth.

La autenticación por Google deberá estar activada en Firebase, accediendo a:
Firebase Console -> Authentication -> Método de Inicio de Sesión -> Activar Google.

Android tiene un requisito adicional: establecer un *hash SHA-1* de la aplicación. Este hash se puede obtener utilizando la herramienta *keytool*, que se incluye al instalar un JDK (*Java Development Kit*). Una vez obtenido, se añade a la aplicación Android en la consola de Firebase: *Project Settings -> Android App -> Add SHA-1.*

Una vez completado todo el proceso, ya se puede inicializar Firebase en el punto de entrada de la aplicación, la función *main*:

```dart
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp(options: DefaultFirebaseOptions.currentPlatform);
  runApp(const MyApp());
}
```

3.5.2. Autenticación de Usuarios

La autenticación se realiza en el widget *LoginScreen*. Primero se llama a la función *GoogleSignIn*, especificando el *clientId* obtenido anteriormente en caso de que la app se esté ejecutando en web, obteniéndose un objeto *googleUser*. A partir del usuario, se obtienen las credenciales necesarias para la autenticación con Firebase [73].

```
final GoogleSignInAuthentication? googleAuth = await googleUser?.authentication;
// crear credenciales de Firebase
final credential = GoogleAuthProvider.credential(
  accessToken: googleAuth?.accessToken,
  idToken: googleAuth?.idToken,
);
// utilizar credenciales para autenticarse
return await FirebaseAuth.instance.signInWithCredential(credential);
```

Una vez autenticado, se podrá obtener el usuario actual de la forma:

```
final user = FirebaseAuth.instance.currentUser;
```

De forma similar, se puede obtener una referencia a la base de datos para hacer consultas:

```
final db = FirebaseDatabase.instance.ref();
```

3.5.3. Asociación con ESP32.

Tras la autenticación, se comprobará si el usuario tiene asociado un ESP32. Para hacerlo, se hace una consulta a *users/<uid>* y se comprueba si contiene la clave *linked_esp* y su valor no está vacío.

```
final snapshot = await db.child("users/${user!.uid}/linked_esp").get();
if (snapshot.exists && snapshot.value != "") {
      // ...
        _hasDevice = true;
       // ...
}
```

En caso afirmativo, se redirigirá al usuario al listado de dispositivos. En caso negativo, la app solicitará al usuario que introduzca el ID de su ESP32 y lo asociará a su *uid* en la base de datos:

```
  Future<void> _linkDevice() async {
    final espId = _deviceIdController.text.trim();
    if (espId.isEmpty) return;
    final uid = user!.uid;
    // Crear una asociación bidireccional entre el usuario y su ESP32
    await db.child("devices/$espId/owner_uid").set(uid);
    await db.child("users/$uid/linked_esp").set(espId);

    _goToMainPage();
  }
```

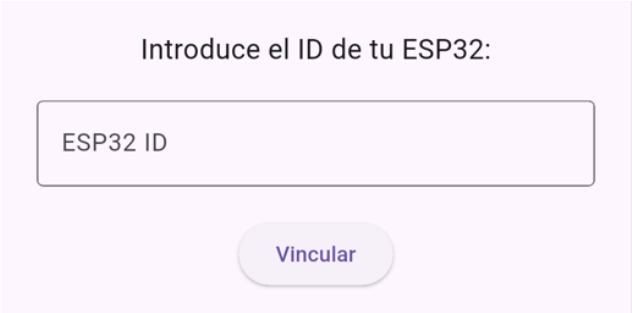

Fig. 29. Pantalla de vinculación

Para desvincular un ESP32, se guardan *owner_uid* y *linked_esp* como *null*, lo que provoca que esas claves y sus valores asociados se borren.

```
await db.child("devices/$espId/owner_uid").set(null);
await db.child("users/${user.uid}/linked_esp").set(null);
```

3.5.4. Listado de dispositivos

En esta pantalla de la aplicación se podrán ver los dispositivos (enchufes) asociados al usuario. Para cada uno, se mostrará su identificador, nombre personalizado y última medición. Se podrá editar el nombre personalizado de cada dispositivo. Se incluyen también botones para actualizar las últimas mediciones, cerrar sesión y desvincular el ESP32 del usuario.

Dado el *uid* del usuario actual obtenido de la sesión, se obtiene el *id* de su ESP32 asociado utilizando la colección *users*.

```
final userEspSnap = await db.child("users/${user!.uid}/linked_esp").get();

if (userEspSnap.exists && userEspSnap.value != "") {
       final espId = userEspSnap.value as String
```

A partir de ese *espId* se deben obtener sus *deviceId* asociados. Una forma posible de conseguirlo es realizar esta consulta:

```
final deviceSnap = await db.child("/devices/$espId").get();
```

Y después, extraer las claves del diccionario *deviceSnap*, que contendrán los *deviceId*. Pero existe un problema: Cuando se llama a *get*, Firebase descargará el nodo */devices/$espId* completo, incluyendo todos los datos de sus subnodos, que contienen las colecciones de mediciones *data*. Estas colecciones podrían contener miles de datos, que son innecesarios. Los SDK de Realtime Database, incluyendo la versión para Flutter, no proporciona una forma directa de descargar solo las claves contenidas por un nodo sin sus correspondientes valores. Solo se puede conseguir realizando una llamada directamente a la API REST [74], pero esa forma implica varias limitaciones.

En este caso, la solución es utilizar la colección *device_info*, que tiene la misma estructura que *devices* pero solo contiene los datos necesarios para obtener los *deviceId* asociados a un *espId*.

```
final deviceSnap = await db.child("/device_info/$espId").get();
```

Una vez obtenidos los *deviceId*, el resto de las consultas se podrá hacer de forma eficiente.

Para cada enchufe se obtiene su última medición, que está formada por una tupla con el consumo en vatios y la marca de tiempo.

```
final snap = await db
      .child("/devices/$espId/$deviceId/data")
      .orderByChild("timestamp")
      .limitToLast(1)
      .get();

    if (!snap.exists) return null;

    // "snap" es un Map con un solo valor, que a su vez es un map con dos claves
    final data = (snap.value as Map).values.first as Map;
    final power = data["power"];
    final timestamp = data["timestamp"];
```

Para mostrarla en pantalla, se utiliza la librería *timeago* [75] para expresar la diferencia entre el instante actual y la marca de tiempo.

```
final formattedDate = timeago.format(DateTime.fromMillisecondsSinceEpoch(timestamp),
locale: "es");
return "${power.toStringAsFixed(1)} W, $formattedDate";
```

De esta forma, se mostrará un mensaje similar a *47.1 W, hace un minuto.*

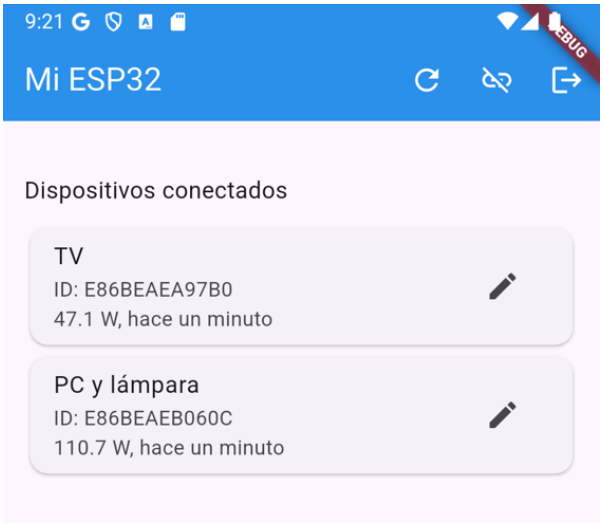

Fig. 30. Listado de dispositivos

Cada dispositivo tiene un botón asociado que permite asignarle un nombre personalizado más fácil de recordar que su *device_id*. Al pulsarlo, la aplicación muestra un cuadro de diálogo para ingresar el nuevo nombre. Al confirmar, se registra en la base de datos.

```
await db.child("devices/$_espId/$plugId/name").set(newName);
```

Esta pantalla también incluye un botón para *actualizar*, que vuelve a obtener las últimas mediciones y renderiza de nuevo el *widget*.

3.5.5. Caché de mediciones

Para poder visualizar las mediciones de consumo, se debe hacer una consulta a la base de datos para obtener los datos dentro del intervalo de tiempo deseado. Pero existe el problema de que cuando el usuario navega por la aplicación y visualiza distintos intervalos, en las consultas se vuelven a descargar los datos de los rangos repetidamente, sin importar si se habían descargado con anterioridad. Para evitar descargar datos innecesariamente y reducir el tamaño de las consultas, Firebase ya ofrece algunas soluciones.

La primera es la función *keepSynced*. Si se llama a esta función sobre una colección, Firebase guardará en memoria una copia local y realizará las consultas sobre la copia en lugar de cargar y descargar datos en la nube. En este caso, se desea mantener sincronizada la colección *data* correspondiente a un dispositivo.

```
await db.child("devices/$espId/$deviceId/data").keepSynced(true);
```

De esta forma, solo se utilizará la nube para datos nuevos que aún no estén en memoria. El problema de este método es que la copia no persiste entre reinicios de la aplicación ya que se almacena en la memoria RAM. La colección *data* podría contener miles de datos con meses de antigüedad y no es deseable descargarlos cada vez si el usuario no va a visualizarlos.

Debido a esto, se requiere utilizar la función de *persistencia* de Firebase, que crea una copia *offline* de las colecciones útil para no perder cambios si se produce una pérdida de conexión, pero en nuestro caso resultará útil para evitar descargar la colección *data* cada vez. Esta copia sí será persistente al estar almacenada en el sistema de archivos del teléfono.
La persistencia se activa mediante esta línea, que debe insertarse en la función *main* antes de iniciar la aplicación [76]:

```
FirebaseDatabase.instance.setPersistenceEnabled(true);
```

Sin embargo, esta solución aumentó el tiempo de las consultas notablemente, llegando en ocasiones hasta los 10 segundos. Este es un problema de Firebase que ha sido reportado por varios usuarios [77][78] y puede deberse a que el volumen de consultas que se desea realizar es demasiado alto para el sistema de archivos, que tiene una velocidad de acceso muy baja comparada con la memoria RAM.

Ante esta situación, se decidió implementar un sistema de caché propio que aunque no es persistente, está diseñado para descargar únicamente los datos relevantes.

La caché se implementa en la clase *MeasurementCache*. Internamente, esta clase almacena una lista de mediciones, con la invariante de que las mediciones siempre se mantendrán ordenadas por su *timestamp*. Las mediciones se modelan con una clase *Measurement* con tres campos:

```
class Measurement {
  final int timestamp;
  final double? power;
  final double? energy;
}
```

La caché implementa dos operaciones básicas:

PutAll: Dada una lista de mediciones ordenada, se insertan en la caché de forma que el resultado se mantenga ordenado. Si la marca de tiempo de la primera medición de la lista dada es posterior a la última medición de la caché, se pueden añadir los nuevos datos al final simplemente utilizando *addAll*. En caso contrario, significa que los datos se deben añadir al inicio de la caché o incluso podría haber solapamiento con los datos ya existentes. En este caso, se debe implementar un algoritmo que, dadas dos listas ordenadas, genere otra lista ordenada que sea la unión de las dos. Este es el mismo algoritmo que se emplea en la ordenación por mezcla o *merge sort*. Se itera sobre las listas *cache* y *data*, seleccionando cada vez la medición con el *timestamp* menor e insertándolo en una lista auxiliar *merged*.

```
putAll(List<Measurement> data) {
    if (data.isEmpty) return;
    if (_cache.isEmpty) {
      _cache = List.from(data);
      return;
    }
    // caso rápido: todos los nuevos datos son posteriores a los datos de la caché
    if (data.first.timestamp > _cache.last.timestamp) {
      _cache.addAll(data);
      return;
    }
    // en caso contrario, cache = mezcla(cache, data)
    final List<Measurement> merged = [];
    // inicializar índices a la primera medición de cada lista
    int i = 0, j = 0;
    while (i < _cache.length && j < data.length) {
      // en cada iteración, comparar las mediciones con índice i y j, añadir a merged
      // la medición con el timestamp menor, e incrementar su índice correspondiente.
      if (_cache[i].timestamp < data[j].timestamp) {
        merged.add(_cache[i]);
        i++;
      } else if (_cache[i].timestamp > data[j].timestamp) {
        merged.add(data[j]);
        j++;
      } else { // en caso de timestamps iguales, incrementar ambos punteros
        merged.add(_cache[i]);
        i++;
        j++;
      }
```

```
    }
    // si se alcanza el final de una lista, añadir a merged el resto de los elementos
    // de la otra lista
    if (i < _cache.length) {
      merged.addAll(_cache.sublist(i));
    }
    if (j < data.length) {
      merged.addAll(data.sublist(j));
    }
    _cache = merged;
  }
```

Si *n* y *m* son el número de elementos de *cache* y *data* respectivamente, la función tiene una complejidad de $O(m)$ en el primer caso y $O(n+m)$ en el caso general.

GetInRange: Dado un rango formado por dos marcas de tiempo *start* y *end,* devuelve los elementos de la caché que se encuentran dentro de ese rango.

```
List<Measurement> getInRange(int start, int end) {
    if (_cache.isEmpty || start > end) return [];

    final int left = _lowerBound(start);
    final int right = _upperBound(end);

    if (left >= _cache.length || left > right) return [];
    final result = _cache.sublist(left, right + 1);
    return result;
  }
```

Utilizando dos funciones *lowerBound* y *upperBound,* se obtienen los índices de las mediciones con el timestamp más cercano al rango deseado, y se devuelve la sub-lista delimitada por esos índices. Tanto *lowerBound* y como *upperBound* se implementan mediante una búsqueda binaria, pero con una diferencia entre ellas:

- *LowerBound* devuelve el primer índice donde se podría insertar una medición con timestamp *start,* manteniendo el resultado ordenado.
- *UpperBound* devuelve el último índice donde se podría insertar una medición con timestamp *end,* manteniendo el resultado ordenado.

Ambas búsquedas se pueden hacer en $O(\log n)$, pero para devolver la sub-lista hay que iterar sobre cada uno de sus elementos, lo que resulta en una complejidad lineal.

La caché debería persistir al navegar entre las distintas pantallas de la aplicación. Para conseguirlo, se podría almacenar en una variable global, pero existe otra solución más correcta.

Utilizando el paquete *provider* [79], se crea la clase *MeasurementStore,* que persiste durante toda la aplicación y almacenará una caché distinta para cada dispositivo del usuario. Esta clase hereda de *ChangeNotifier,* de forma que notificará a los *widgets* que hacen referencia a ella cuando se produzca algún cambio. Se implementan métodos que, dado un *deviceId,* permiten utilizar su caché correspondiente para leer o escribir elementos.

```
class MeasurementStore extends ChangeNotifier {
  // Asocia cada deviceId con su caché de mediciones.
  final Map<String, MeasurementCache> _caches = {};

  /// Inicializa una caché para un deviceId, si ya no lo estaba
  void registerDevice(String deviceId) {
    if (!_caches.containsKey(deviceId)) {
      _caches[deviceId] = MeasurementCache(deviceId);
    }
  }

  void putAllInDevice(String deviceId, List<Measurement> data) {
    registerDevice(deviceId);
    _caches[deviceId]!.putAll(data);
  }

  List<Measurement> getInRangeForDevice(String deviceId, int start, int end){/*...*/}

  (int, int)? getCachedRangeOf(String deviceId) {/*...*/}
}
```

Este *ChangeNotifier* se debe registrar al inicio de la aplicación en la función *main*:

```
runApp(
    ChangeNotifierProvider(
      create: (_) => MeasurementStore(),
      child: const MyApp(),
    )
  );
```

3.5.6. Consultas a la base de datos

Las consultas a la base de datos se implementan en el archivo *db_queries.dart*. En él, se define la función loadPowerAndEnergyData que, dado un intervalo de tiempo deseado *(rangeStart, rangeEnd)*, genera dos listas con las mediciones de potencia y energía dentro de ese intervalo. Inicialmente se implementó el siguiente algoritmo, utilizando la caché.

1. Comparar el intervalo deseado con el intervalo de los datos de la caché.
2. Calcular los intervalos de los datos que no están en la caché
3. Para cada uno de esos intervalos, obtener sus datos de la base de datos.
4. Guardar en la caché los nuevos datos obtenidos.
5. La caché ya contendrá todos los datos necesarios, entonces se utiliza para obtener todos los datos del rango deseado.

Sin embargo, en esta etapa del desarrollo se pudieron solucionar los problemas de rendimiento causados por la persistencia de Firebase modificando las consultas para evitar un problema similar al mencionado en el apartado *3.5.4 Listado de dispositivos* donde se descargaban datos innecesarios, así como realizando las pruebas de aplicación en un dispositivo móvil real en lugar del emulador. Por este motivo, finalmente se descartó la solución de la caché implementada anteriormente.

La consulta para obtener los datos de potencia y energía se realiza de la forma:

```
await db.child("devices/$espId/$deviceId/data").keepSynced(true);

final snap = await db
  .child("devices/$espId/$deviceId/data")
  .orderByChild("timestamp")
  .startAt(rangeStart)
  .endAt(rangeEnd)
  .get();
```

Definir el campo *timestamp* de la colección *data* como índice permite que las consultas ordenadas y acotadas a un rango se puedan realizar de forma eficiente. Para que los datos puedan ser representados en gráficas, se deben proporcionar en listas de *FlSpot*, que es el tipo de datos utilizado por la librería de gráficas (*fl_chart*) para representar puntos con coordenadas *x* e *y*, donde *x* es una marca de tiempo e *y* es su valor correspondiente.

```
List<FlSpot> powerSpots = [];
List<FlSpot> energySpots = [];

for (final child in snap.children) {
  final ts = int.tryParse(child.child("timestamp").value.toString());
  final power = double.tryParse(child.child("power").value.toString());
  final energy = double.tryParse(child.child("energy").value.toString());

  if (ts != null) {
    if (power != null) powerSpots.add(FlSpot(ts.toDouble(), power));
    if (energy != null) energySpots.add(FlSpot(ts.toDouble(), energy));
  }
}
```

Antes de ser dibujados en la gráfica, los datos de energía se *normalizan* para que la energía inicial del rango sea 0 y la energía final corresponda al total de energía consumida en el rango. Además, se asegura que la función de energía nunca sea decreciente ya que el cálculo del coste sería incorrecto si hay intervalos con consumo negativo. Esta situación puede darse cuando el usuario desconecta un enchufe de la corriente, lo que puede provocar que se reinicie su contador interno de energía acumulada.

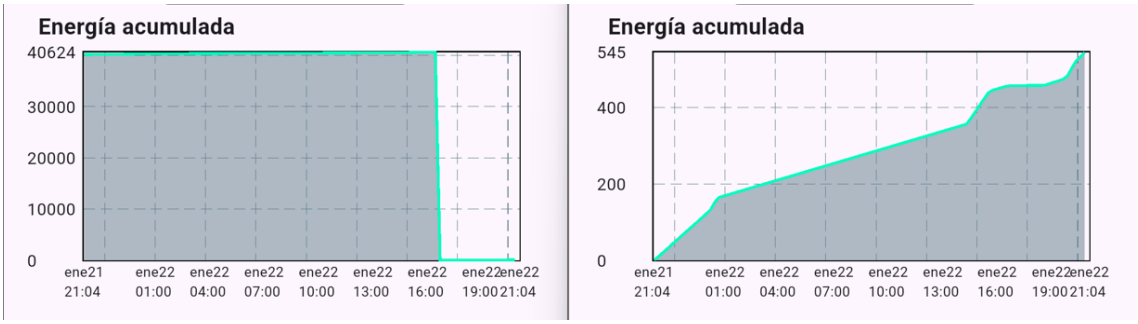

Fig. 31. Comparación entre gráficas de energía sin normalizar (izquierda) y normalizada (derecha).

Los datos del precio de la energía se obtienen de forma similar:

```
await db.child("energy_price").keepSynced(true);
final priceSnap = await db
    .child("energy_price")
    .orderByKey()
    .startAt(asFirebaseKey(range.start))
    .endAt(asFirebaseKey(range.end))
    .get();
```

Donde *asFirebaseKey* es una función auxiliar que convierte las marcas de tiempo en milisegundos al formato *ISO 8601* usado por la API de ESIOS. Los datos se transforman en una lista de *FlSpot* de forma similar a la potencia y energía.

3.5.7. Gráficas

En el widget *PlugChartPage* se visualizan varias gráficas de un dispositivo: el consumo (potencia y energía), el precio de la energía y combinando las anteriores, el coste a lo largo del tiempo. Las gráficas se mostrarán en un rango determinado que se almacena en el campo de clase *_range*, que se inicializa a un rango entre las 24 horas anteriores y el instante actual.

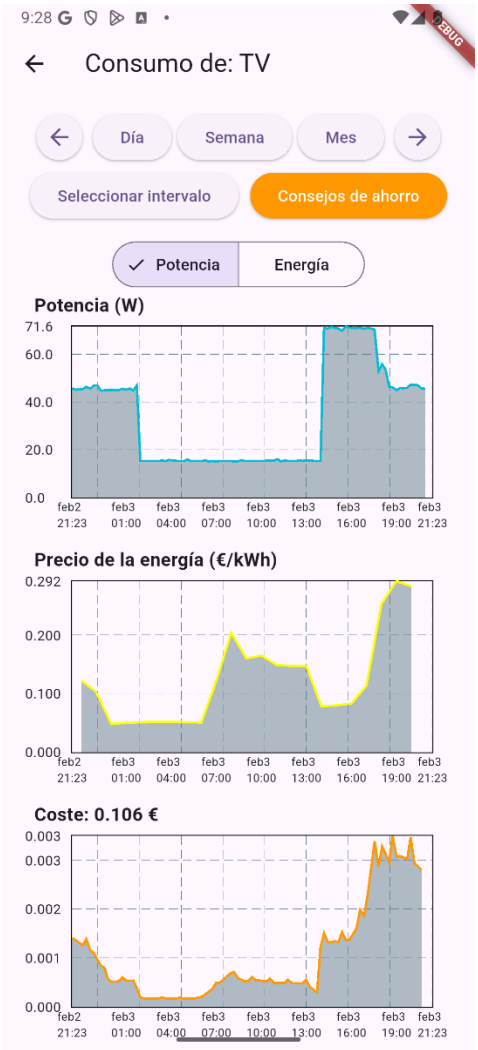

Fig. 32. Pantalla con las gráficas de consumo

Cuando el widget se inicializa en *initState*, se llama a *reapplyAllFilters*, donde se obtienen los datos relevantes para el rango seleccionado y se dibujan las gráficas utilizando la librería *fl_chart* [80].

Si el rango de tiempo seleccionado es muy amplio, es posible que se trabajen con muchos datos, lo que afectaría al rendimiento si se dibujan todos en la gráfica. Para solucionarlo, se escoge un subconjunto de los datos aplicando *downsampling*.

```
List<FlSpot> downsample(List<FlSpot> points, int maxPoints) {
  if (points.length <= maxPoints) return points;

  double step = points.length / maxPoints;
  List<FlSpot> result = [];

  for (double i = 0; i < points.length; i += step) {
    result.add(points[i.toInt()]);
  }

  return result;
}
```

Para crear las gráficas, se utiliza el widget personalizado *Chart*, que actúa como envoltorio del widget base de la librería, *FlChart*. Es posible que los rangos resultantes de computar los consumos y los precios no sean iguales, lo que provocaría que los instantes de tiempo no estén alineados entre las gráficas. Para corregirlo, en el widget *Chart* se insertan puntos adicionales al inicio y al final con marcas de tiempo que corresponden al rango seleccionado. Para separar estos nuevos puntos de los ya existentes, se insertan puntos *nulos* (*nullSpot*), que sirven para indicar a *fl_chart* que es el inicio de un *segmento* diferente, de esta forma no dibujará líneas conectando puntos de segmentos distintos.

```
List<FlSpot> _expandToRange(List<FlSpot> spots, DateTimeRange range) {
    final startMs = range.start.millisecondsSinceEpoch;
    final endMs = range.end.millisecondsSinceEpoch;
    return [
      FlSpot((startMs / 1000).toDouble(), 0), // punto adicional al inicio del rango
      FlSpot.nullSpot, // punto nulo
      ...spots, // puntos ya existentes
      FlSpot.nullSpot, // punto nulo
      FlSpot((endMs / 1000).toDouble(), 0), // punto adicional al final del rango
    ];
```

Esta función tiene la precondición de que todos los puntos de *spots* deben estar dentro de *range*, que se cumple porque es el mismo rango que se utiliza para las consultas, y éstas nunca devuelven puntos fuera de ese rango. Con este procedimiento y utilizando el mismo rango para todas las gráficas, se soluciona el problema anterior.

En las mediciones de potencia y energía, es posible que haya intervalos sin datos debido a desconexiones momentáneas de los enchufes. Para asegurar que estos intervalos se muestran

como vacíos, se implementa sobre la gráfica un procedimiento de *segmentación*. Si este procedimiento encuentra un intervalo entre mediciones con duración superior a un umbral, insertará puntos nulos para que *FlSpot* los muestre como intervalos separados, sin una línea que los conecte.

```
List<FlSpot> _splitIntoSegments(List<FlSpot> spots, DateTimeRange range) {
    List<FlSpot> result = [];
    final gapThreshold = range.duration.inMilliseconds / 20;
    for (int i = 0; i < spots.length - 1; i++) {
      final currT = spots[i].x;
      final nextT = spots[i + 1].x;
      if (nextT - currT > gapThreshold) {
        result.addAll([spots[i], FlSpot.nullSpot]);
      } else {
        result.add(spots[i]);
      }
    }
    return result;
  }
```

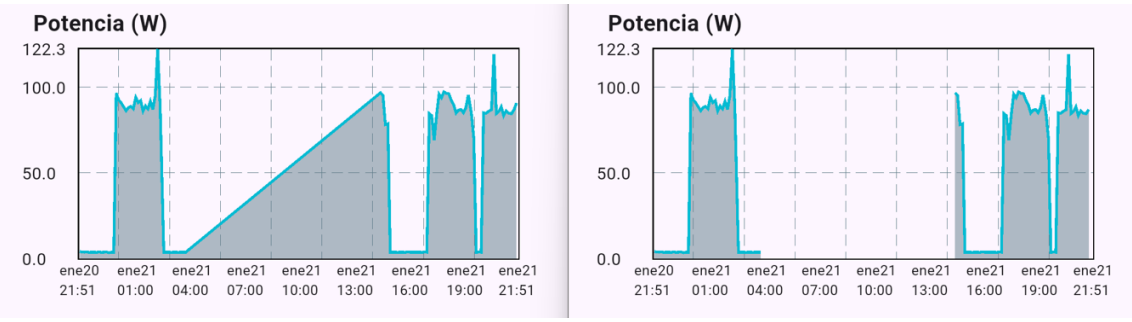

Fig. 33. Comparación entre una gráfica sin segmentación (izquierda) y con segmentación (derecha)

El *widget* está diseñado para permitir dibujar varias colecciones de datos superpuestas en la misma gráfica, donde cada una podrá tener su propio color. Esta funcionalidad se utiliza en la pantalla de recomendaciones energéticas para poder comparar los precios de dos días consecutivos.

Debido a que el *widget* personalizado *Chart* se utiliza para varios tipos de gráficas, para que se pueda usar de forma modular evitando la repetición de código, se han incluido en su constructor parámetros de configuración, incluyendo un título (*label*) y número de decimales y unidad con la que se mostrarán los valores. Por defecto se muestran la fecha y hora de las marcas de tiempo, pero hay situaciones donde solo es relevante mostrar las horas y minutos, por lo que este parámetro también es configurable.

3.5.8. Cálculo del coste

La aplicación implementa una función que, dada una colección $e_0, e_1, \dots, e_{N-1}$ de mediciones de consumo de un dispositivo y otra colección $p_0, p_1, \dots, p_{M-1}$ con los precios de la energía en €/kWh, calcula el coste total del dispositivo en ese intervalo. Las colecciones estarán compuestas de puntos donde la coordenada x es una marca de tiempo en milisegundos y la

coordenada y es el valor correspondiente (energía o precio). A alto nivel, el algoritmo es el siguiente:

Para todos los intervalos formados por parejas de mediciones consecutivas:
- Calcular la energía consumida en ese intervalo.
- Multiplicar la energía por el precio correspondiente a ese intervalo, obteniendo el coste.

El resultado será la suma de los costes de todos los intervalos.

Con mediciones de potencia instantánea en tiempos t_0 y $t_1$se puede estimar la energía total de la forma:

$$E(t_0, t_1) = \int_{t_0}^{t_1} P(t)\, dt$$

Donde $P(t)$ es una función que describe la potencia a lo largo del tiempo. Sin embargo, solo se dispone de medidas de potencia en los extremos, así que se debe aproximar esta función como una constante, cuyo valor será la media de esas medidas. Esta solución presenta un problema: Es posible que, debido a desconexiones de los enchufes, algunos intervalos de mediciones tengan una duración muy larga. Como no existirá información del consumo excepto en los extremos, la aproximación podría ser muy poco precisa.

Es este el motivo por el que, además de la potencia instantánea, se recoge de los enchufes la energía acumulada. De esta forma, se puede calcular la energía consumida en cualquier intervalo simplemente restando la energía inicial a la energía final, sin importar su duración.

El algoritmo se realiza iterando sobre las mediciones de energía para cada $i = 0,1,2, \dots, N-2$, trabajando con intervalos definidos por mediciones consecutivas e_i y e_{i+1}.
Se definen los tiempos t_0 y t_1 como:

$$t_0 = e_i.x$$
$$t_1 = e_{i+1}.x$$

Buscamos un índice j para un precio p_j de forma que su marca de tiempo $p_j.x$ sea lo más cercana a t_0. En el código se implementa con un bucle while que incrementa una variable *priceIndex* hasta encontrar el valor deseado. Su valor se inicializa a 0 fuera del bucle principal y no se reinicia entre cada iteración.

Una vez obtenido el índice j, obtenemos una pareja de precios p_j y p_{j+1}. Necesitamos obtener el precio en los tiempos t_0 y t_1, pero estos tiempos no se corresponderán con los de los precios p_j y p_{j+1} debido a que las colecciones de consumo y precio tienen resolución temporal distinta. Entonces, obtenemos unos precios aproximados utilizando **interpolación lineal.**

Dados dos puntos (x_0, y_0) y (x_1, y_1) que describen una recta, podemos encontrar qué coordenada y corresponde a una coordenada x dada, de forma que el punto (x, y) esté contenido en la recta anterior.

$$y = lerp(x_0, y_0, x_1, y_1, x) = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0}$$

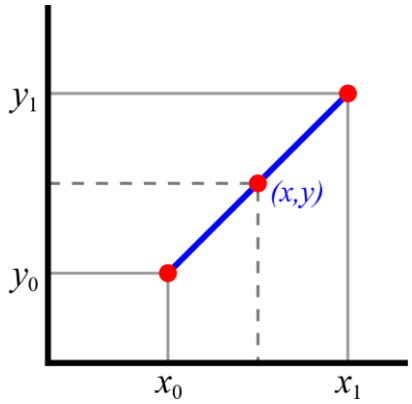

Fig. 34. Representación gráfica de la interpolación lineal [43]

Es decir, si conocemos el precio en dos instantes de tiempo dados, podemos deducir el precio en otro instante intermedio, suponiendo que el precio cambia de forma lineal entre los instantes dados.

De esta forma, podemos calcular un precio aproximado en un tiempo t, denotado $p(t)$

$$p(t) = lerp(p_j.x, p_j.y, p_{j+1}.x, p_{j+1}.y, t)$$

Obtenemos $p(t_0)$ y $p(t_1)$.
La energía consumida entre los instantes t_0 y t_1 es:

$$E(t_0, t_1) = e_{i+1}.x - e_i.x.$$

Calculamos también el precio medio de los precios aproximados:

$$\overline{p} = \frac{p(t_0) + p(t_1)}{2}$$

Finalmente, obtenemos el coste en el intervalo actual multiplicando el consumo de energía por el precio:

$$C(t_0, t_1) = E(t_0, t_1) \cdot \overline{p}$$

La energía tiene $W \cdot s$ como unidad, y el precio viene dado en $€/(Kw \cdot h)$. Para que el resultado quede en euros, debemos convertir $W \cdot s$ a $KW \cdot h$:

$$1\ W \cdot s \cdot \frac{1\ h}{3600\ s} \cdot \frac{1\ KW}{1000\ W} = \frac{1}{3600000} KW \cdot h$$

Aplicando ese factor de conversión a la energía, al calcular el coste obtendremos el resultado en euros. El coste total se calcula sumando los costes correspondientes a todos los intervalos formados por parejas de puntos de consumo consecutivos.

Para cada intervalo, la función guarda su coste en una lista que se utilizará para dibujar la gráfica del coste a lo largo del tiempo.

3.5.9. Sugerencias de ahorro

Las sugerencias utilizan las predicciones de precios del día siguiente. Estas predicciones se publican en la API a las 20:20 aproximadamente. Para que la aplicación disponga de estos datos con certeza, ésta generará una nueva recomendación cada día a las 21:00.

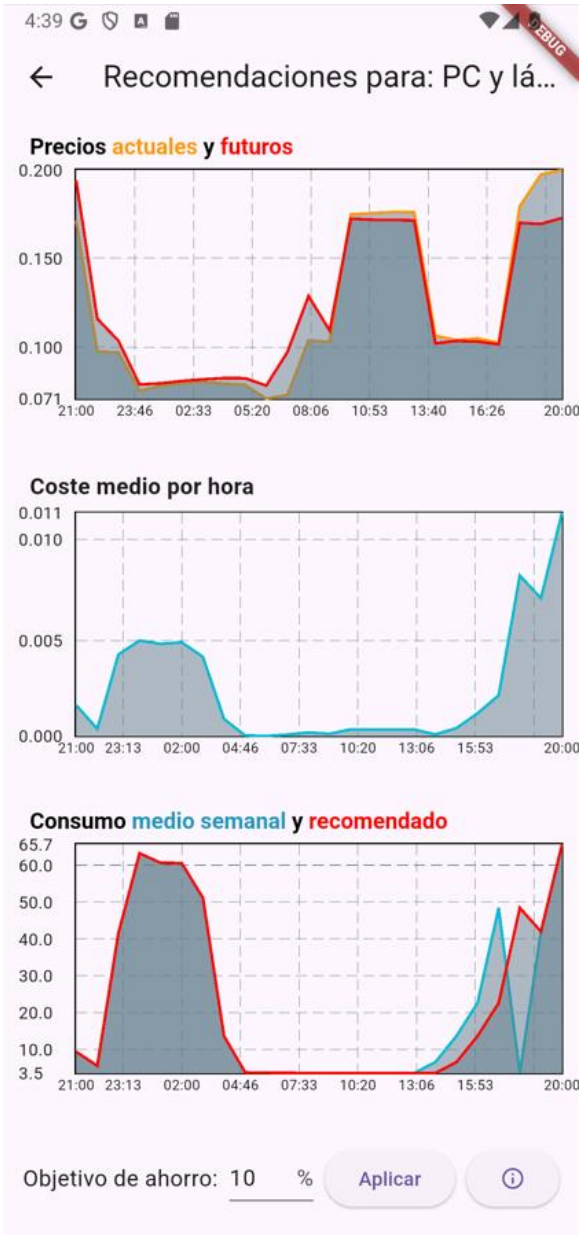

Fig. 35. Pantalla de recomendaciones

La primera gráfica compara los precios actuales con los precios siguientes. De esta forma, el usuario puede tener una idea general de cómo evolucionará el precio y la duración de las horas más costosas.

Para generar la recomendación, el algoritmo determina los hábitos de consumo del usuario calculando el consumo medio por hora de los últimos 7 días. Para ello, se itera sobre las mediciones de potencia de los últimos 7 días y se clasifican en una tabla hash de acuerdo con la hora del día de su marca de tiempo, creándose 24 claves, cada una asociada a la lista de

mediciones correspondiente a esa hora del día. Finalmente, devuelve una lista conteniendo las medias de todas las listas anteriores.

Combinando los hábitos con los precios del día siguiente, se genera otra gráfica con el coste del dispositivo por hora, que ayuda al usuario a determinar cómo afectan los precios al coste no solo en general, sino ajustado al consumo de su dispositivo.

Durante el cálculo de las sugerencias, resulta útil trabajar con listas circulares, que modelan intuitivamente el comportamiento cíclico de los datos. En lugar de definir funciones auxiliares, se pueden utilizar los *métodos de extensión* de Dart para añadir nuevos métodos a clases externas, en este caso a la clase *List* que contenga un tipo genérico *T*.

```
extension Cyclic<T> on List<T> {
  T cyclicGet(int index) {
    int cyclicIndex = ((index % length) + length) % length;
    return this[cyclicIndex];
  }

  void cyclicSet(int index, T elem) {
    int cyclicIndex = ((index % length) + length) % length;
    this[cyclicIndex] = elem;
  }
}
```

Se definen dos funciones *get* y *set* cíclicas, de forma que al utilizar índices fuera de rango, la función asegura que el índice está dentro de la lista mediante aritmética modular. Si se utiliza un índice negativo, la función empieza a contar hacia atrás desde el último elemento.

A partir de los hábitos de consumo, el algoritmo busca *bloques* de consumo, es decir, intervalos de tiempo donde sea más probable que el usuario vaya a utilizar su dispositivo. Por ejemplo, dada esta gráfica de consumo:

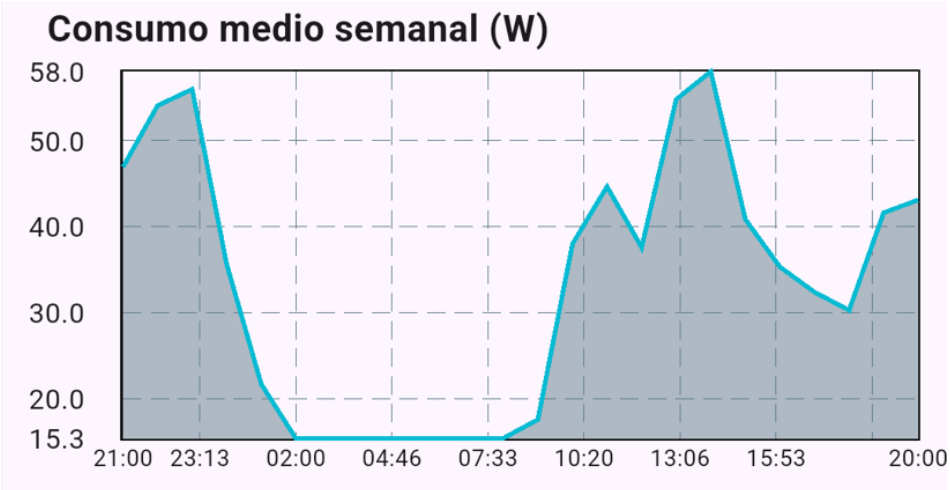

Fig. 36. Gráfica del consumo medio

El algoritmo determina estos bloques:

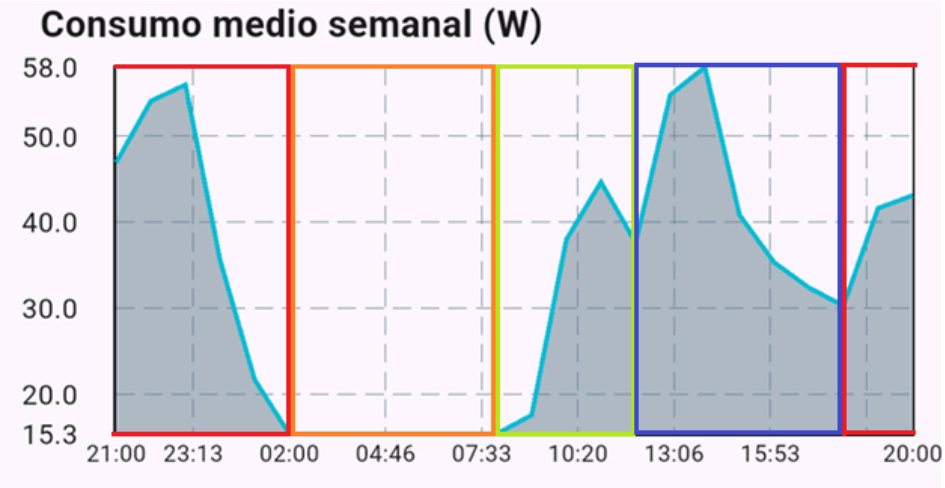

Fig. 37. Gráfica del consumo medio con los bloques resaltados

El algoritmo considera que la gráfica está formada por 24 *muestras*, donde a las 21:00 le corresponde el índice 0, a las 22:00 le corresponde el índice 1, y así sucesivamente.
En el código, los bloques se definen como instancias de la clase *UsageBlock*, que contienen los dos índices que lo delimitan, además de algunos métodos auxiliares.

```
class UsageBlock
  int start;
  int end;
  // métodos auxiliares...
}
```

Debido al comportamiento cíclico, es posible que el índice inicial sea mayor al índice final, como es el caso del primer ciclo.
Se define una función *findBlocks* que recibe una lista de muestras y devuelve una lista de bloques. Para delimitar los bloques, la función busca muestras donde exista un *mínimo local*.

```
List<UsageBlock> findBlocks(List<double> power) {
  // buscar mínimos locales
  final List<int> localMinima = [];
  for (int i = 0; i < power.length; i++) {
    double prev = power.cyclicGet(i - 1);
    double curr = power.cyclicGet(i);
    double next = power.cyclicGet(i + 1);
    if (prev > curr && curr < next) {
      localMinima.add(i);
    }
  }
  // construir los bloques
  final List<UsageBlock> blocks = [];
  for (int i = 0; i < localMinima.length; i++) {
    final block = UsageBlock(start: localMinima.cyclicGet(i), end:
localMinima.cyclicGet(i + 1) - 1);
```

```
    blocks.add(block);
  }
  return blocks;
}
```

Una vez obtenidos los bloques $b_0, b_1, \dots, b_N$, el algoritmo encuentra una lista de *desplazamientos* correspondiente $d_0, d_1, \dots, d_N$ que resulte en el coste final más cercano al objetivo del usuario. Además, intentará reducir el número de desplazamientos total. La operación desplazar o *shift* se define en la clase *UsageBlock*, donde dado un número d de muestras, se desplazará el consumo d muestras hacia delante si $d > 0$ o d muestras hacia atrás si $d < 0$.
Es posible que el dispositivo tenga un consumo base o *baseline* simplemente por estar conectado a la corriente y no se pueda disminuir. La función tiene esto en cuenta y no sobreescribe el consumo con ceros. Solo desplaza el consumo que se encuentre por encima del consumo base que, para asegurar transiciones suaves entre bloques, se define como el consumo inicial del bloque.

Con los bloques y la operación desplazar definida, el algoritmo de optimización en pseudocódigo es el siguiente:

```
optimizar(consumos, precios, objetivo) {
  costeInicial = cost(consumos, precios);
  costeObjetivo = costeIncial * (1 - objetivo);
  bloques = findBlocks(consumos);
  ordenarPorCoste(bloques)

  maxDesplazamientos = 12;
  mejorSolucion = consumos;
  for (nDesp = 0; i < nDesp; i++) {
    solucion = buscarConMaxDesplazamientos(nDesp, bloques, precios, objetivo);
    if (cost(solucion, precios) < cost(mejorSolucion, precios)) {
      mejorSolucion = solucion;
    }
    if (cost(solucion, precios) < costeObjetivo) {
      return solucion;
    }
  }
  return mejorSolucion;
}
```

El algoritmo da prioridad a los bloques con mayor *coste*, es decir, los bloques cuyo consumo contribuye más al gasto energético del usuario. Se define un número máximo de desplazamientos total, que el algoritmo intentará distribuir entre todos los bloques. Empezará con un máximo de 1 desplazamientos. Si no se alcanza el objetivo, utilizará un máximo de 2 desplazamientos y así sucesivamente hasta que se alcance el objetivo o llegue al límite máximo. En cada iteración se guarda la mejor solución hasta el momento. Combinado con lo anterior, esto garantiza que:

- Si se encuentra una solución que alcanza el objetivo, tendrá el menor número de desplazamientos posible.
- Si se llega al límite de desplazamientos sin alcanzar el objetivo, la solución generada será la más cercana al objetivo.

A continuación, se define la función *buscarConMaxDesplazamientos*, que intenta encontrar una solución distribuyendo un número determinado de desplazamientos entre los bloques. La solución se define como una vector $d_0, d_1, \dots, d_N$ de desplazamientos, donde la suma en valor absoluto de cada uno no supere el límite. Además, se define 3 como el desplazamiento máximo por bloque en ambas direcciones, de forma que $-3 \leq d_i \leq 3$.

Para encontrar la solución, se utiliza un algoritmo de búsqueda en profundidad con vuelta atrás. En cada estado se calcula el coste actual. Si es inferior al mejor coste hasta el momento, se actualiza la mejor solución. Si además el coste es inferior al objetivo, se considera solución óptima y la búsqueda termina. En caso de no haber encontrado solución óptima se selecciona un bloque y se prueba a aplicar todos los desplazamientos posibles sin superar el límite por bloque, repitiendo la búsqueda recursivamente para cada uno. En pseudocódigo, la búsqueda se realiza de la siguiente manera:

```
buscarConMaxDesplazamientos(maxDesps, bloques, consumos, precios, costeObjetivo) {
  // el vector de desplazamientos, inicializados a 0.
  solucion = List.filled(bloques.length, 0);
  // dejará de ser nulo si se encuentra una solución con coste menor al objetivo
  solucionOptima = null;
  mejorSolucion = solucion;
  mejorCoste = cost(consumos, precios);

  // función anidada que tiene acceso a los parámetros y variables locales de la
función que la contiene
  busquedaEnProfundidad(indiceBloque, consumosActuales, despsRestantes) {
    // cancelar la búsqueda si ya existe solución óptima
    if (solucionOptima != null) return;
    costeActual = cost(consumosActuales, precios);
    // comprobar si se ha encontrado una mejor solución
    if (costeActual < mejorCoste) {
      mejorCoste = costeActual;
      mejorSolucion = solucion;
    }
    // comprobar si se ha encontrado la solución óptima
    if (costeActual < costeObjetivo) {
      solucionOptima = mejorSolucion;
      return;
    }
    if (indiceBloque >= bloques.length) return;
    // seleccionar el bloque
    bloque = bloques[indiceBloque];
    // probar todos los desplazamientos posibles para ese bloque
    maxDespPorBloque = 3;
```

```
    for (desp = -maxDespPorBloque; desp <= maxDespPorBloque; desp++) {
      // si este desplazamiento supera el límite, saltar
      if (abs(desp) > despsRestantes) continue;
      // probar a utilizar este desplazamiento en la solucion
      solucion[indiceBloque] = desp;
      if (desp == 0) {
        // si se prueba desplazamiento 0, simplemente repetir búsqueda con el
siguiente bloque
        busquedaEnProfundidad(indiceBloque + 1, consumosActuales, despsRestantes);
      } else {
        // en otro caso, aplicar el desplazamiento al consumo
        consumosTrasDesp = shiftBlock(consumosActuales, bloque, desp);
        // tras aplicarlo, repetir búsqueda con el siguiente bloque
        busquedaEnProfundidad(indiceBloque + 1, consumosTrasDesp, despsRestantes -
abs(desp));
      }
      // comprobar si tras la búsqueda recursiva, este desplazamiento ha generado
solución óptima
      if (solucionOptima != null) return;
    }
    // ningún desplazamiento para este bloque mejoró la solución (vuelta atrás)
    solucion[indiceBloque] = 0;
  }

  // iniciar búsqueda con el primer bloque (el más costoso)
  busquedaEnProfundidad(0, consumos, maxDesps);
  if (solucionOptima != null) {
    // se ha encontrado solución que alcanza el objetivo
    return solucionOptima;
  } else {
    // no se ha encontrado solución que alcance el objetivo
    return mejorSolucion;
  }
}
```

En una llamada recursiva es posible que se haya alcanzado el objetivo, en ese caso se devuelve la solución. Si no existe solución que alcance el objetivo dentro del límite de desplazamientos, ya se habrán probado todas las posibilidades y se devolverá la mejor solución encontrada.

Durante el desarrollo se exploró la posibilidad de aumentar la resolución de las muestras para obtener recomendaciones más precisas, es decir, trabajar con 48 muestras en vez de 24, donde cada hora estaría representada por dos muestras. Sin embargo, esta opción presentó varios problemas:

En primer lugar, aunque el algoritmo intenta encontrar la solución óptima rápidamente priorizando los bloques que más contribuyen al coste, la búsqueda en profundidad tiene una complejidad exponencial $O(d^N)$ donde N es número de bloques encontrados y d es el número de posibles desplazamientos para cada bloque. Duplicar la resolución hace que se generen más

bloques y cada bloque tenga el doble de desplazamientos posibles lo que, combinado con la complejidad exponencial, redujo drásticamente el rendimiento del algoritmo. Para solucionarlo, se probaron las siguientes optimizaciones:

- Reducir el número de bloques combinando bloques pequeños entre sí.
- Precalcular el *potencial de ahorro máximo* de cada bloque. Este potencial se obtiene calculando el ahorro de todos los posibles desplazamientos del bloque sobre el consumo y eligiendo el mejor. En la búsqueda, si se selecciona un bloque que no alcanza el objetivo aun restando su potencial al coste actual, inmediatamente se descarta esa rama completa. Esta optimización es similar a la utilizada en algoritmos de tipo *ramificación y poda*, y consiguió reducir el número de búsquedas en un 99%.

Con las optimizaciones anteriores, el rendimiento se acercaba más al algoritmo original, pero presentaba un segundo problema: En general, los resultados eran menos favorables. En las pruebas realizadas, el algoritmo original obtenía un ahorro medio del 7-8%, mientras que al incrementar la resolución el ahorro disminuía al 5%. Por estos motivos se decidió volver a la solución original y utilizar una muestra por cada hora.

Una vez obtenido el vector de desplazamientos, la aplicación muestra al usuario una nueva gráfica donde se han aplicado los desplazamientos al consumo medio inicial. También se muestra en un cuadro de diálogo la lista de los desplazamientos en un formato intuitivo para el usuario, especificando el tramo horario y el número de horas hacia delante o hacia atrás que el usuario debería desplazar su consumo.

Fig. 38. Ejemplo de recomendación generada

En el caso de que el algoritmo genere desplazamientos que el usuario considere demasiado agresivos o difíciles de implementar, éste podrá reducir el objetivo de ahorro de forma que la aplicación pueda generar una nueva solución con menos desplazamientos.

3.6. Pruebas

En este capítulo se resumen las pruebas realizadas sobre el sistema durante el desarrollo y los resultados obtenidos, centrándose en los casos excepcionales que podrían dar lugar a errores:

Prueba realizada	Resultado obtenido
Un enchufe inteligente pierde la conexión con el ESP32.	El ESP32 intenta conectar con el enchufe y muestra el mensaje *can't connect to device <device_id>*. El resto de dispositivos seguirán funcionando. El enchufe volverá a conectarse automáticamente cuando la señal tenga fuerza suficiente.
El ESP32 pierde la conexión con el router WiFi.	El ESP32 mostrará el mensaje *can't connect to wifi*. Intentará volver a conectar hasta que la señal tenga fuerza suficiente.
El usuario desconecta un enchufe inteligente de la corriente.	Es posible que el enchufe reinicie su contador interno de energía acumulada, pero la aplicación normaliza los datos de energía antes de utilizarlos.
El usuario intenta vincular un ESP32 que ya está vinculado con otro usuario.	La aplicación impide la vinculación y muestra un mensaje de error.
El usuario introduce un ID de ESP32 incorrecto.	La aplicación mostrará que no hay dispositivos vinculados. El usuario podrá desvincular y volver a intentarlo con el ID correcto.
El usuario selecciona un intervalo sin mediciones o en el futuro.	La aplicación muestra las gráficas como vacías.
El usuario selecciona un rango de fechas donde existen intervalos sin mediciones debido a desconexiones momentáneas de los enchufes inteligentes.	La aplicación mostrará en las gráficas esos intervalos como vacíos para indicar que no hubo mediciones.
El usuario selecciona un intervalo de mediciones muy amplio.	La aplicación podría tardar más en cargar los datos, pero una vez cargados la fluidez de la aplicación no se ve afectada al limitarse el número de datos que se muestran en las gráficas.
El usuario navega rápidamente por muchos intervalos de mediciones distintos.	La aplicación hace las consultas de la base de datos sobre la caché local siempre que sea posible, por lo que el tráfico de la red y el servidor no se ve afectado notablemente.
El usuario elimina los datos de la aplicación en el teléfono.	La aplicación continúa funcionando correctamente, volviendo a crear la caché local.
El usuario cambia la zona horaria de su teléfono.	La aplicación muestra los datos ajustando las fechas a la zona horaria local.
El usuario selecciona un objetivo de ahorro demasiado alto.	La aplicación mostrará la recomendación con el porcentaje de ahorro más cercano al objetivo.

Tabla 2. Resumen de pruebas realizadas.

4

Conclusiones y líneas futuras

En este proyecto se ha explorado cómo desarrollar un sistema de Internet de las Cosas completo, implementando todo el flujo de datos desde que se recogen de los sensores hasta que el usuario pueda visualizarlos en una aplicación. El sistema se ha diseñado para admitir múltiples usuarios. Se han involucrado múltiples campos del desarrollo de software, incluyendo programación de sistemas embebidos, computación en la nube y desarrollo de aplicaciones móviles.

En primer lugar, se ha programado un firmware para el microcontrolador ESP32 que permite la conexión inalámbrica con múltiples enchufes inteligentes *Shelly* mediante WiFi, asegurando que el proceso de conexión sea automatizado para simplificar su uso. Se ha incluido soporte para el protocolo WPS con el fin de conseguir acceso a internet sin necesidad de introducir las credenciales de un router WiFi, permitiendo la conexión, autenticación y envío de datos al servidor.

En cuanto a computación en la nube, se utilizado la plataforma *Firebase* diseñar para un esquema de base de datos adaptado a las necesidades del sistema, manejar la autenticación de usuarios, implementar reglas de seguridad y gestionar las llamadas a la API externa *esios* para obtener los precios de la energía.

Por último, se ha diseñado una aplicación móvil Android utilizando *Flutter*, que incluye inicio de sesión, y un sistema de vinculación que permite al usuario acceder a los datos recopilados por su ESP32. Se han empleado paquetes creados por la comunidad como *fl_chart* para la visualización de la información recogida de la base de datos a lo largo del tiempo, incluyendo el consumo en potencia instantánea, energía acumulada, y precio de la energía. Con esta información, la aplicación calcula y visualiza el coste del dispositivo, e incorpora un algoritmo que ayuda al usuario a reducir su coste en energía generando recomendaciones a partir de los precios futuros obtenidos de la API.

El proyecto ha generado como resultado un sistema funcional y versátil, pero aún existe un margen de mejora. Para mejorar aspectos del proyecto y añadir nuevas funcionalidades, se plantean estas líneas futuras:

- **Mejora de la escalabilidad:** En el proyecto se ha utilizado *Realtime Database* como base de datos, que está diseñada para que cada usuario mantenga una copia local de la parte relevante de la base de datos y obtenga actualizaciones de forma automática. Este enfoque ha sido apropiado para el volumen de datos manejado durante el desarrollo, pero *Realtime Database* no es la mejor solución si se quiere trabajar con volúmenes de datos muy grandes. En ese caso, se podría migrar a la otra alternativa que *Firebase* ofrece como base de datos: *Firestore*, que incorpora funciones más avanzadas [81].

- **Aumentar el número máximo de dispositivos conectados:** Con el firmware desarrollado, un ESP32 soporta un máximo de 4 dispositivos conectados a su punto de acceso, que es el máximo por defecto que establece la librería de WiFi utilizada. Existen maneras de aumentar el límite, pero es posible que se exceda la capacidad de memoria o CPU del ESP32 y provocar inestabilidad en la conexión. Como solución, se puede utilizar otra arquitectura de forma que el ESP32 proporcionen a los enchufes las credenciales del router WiFi para que se conecten a él. De esta forma, el ESP32 no tendría enchufes conectados directamente, sino que utilizaría el router como nodo intermedio para hacer las peticiones.

- **Mejoras de seguridad:** Desde la aplicación móvil, se asegura que un usuario no pueda acceder a los datos de otros, aun conociendo sus *esp_id*. Sin embargo, se recomienda que un usuario no comparta su *esp_id* con nadie, porque es posible que un usuario malicioso utilice una herramienta como *postman* para hacer peticiones a los endpoints del servidor directamente. Aun así, no podría leer, modificar ni borrar los datos de otro usuario sabiendo su *esp_id*, pero sí podría escribir datos nuevos arbitrariamente. Esto se debe a cómo se ha diseñado el proceso de vinculación y las reglas de seguridad. Para solucionarlo, se podría rediseñar la vinculación para que utilice un código temporal. Por otro lado, se podría implementar un límite de solicitudes para asegurar que un usuario no pueda sobrecargar el servidor.

- **Añadir funcionalidades a la aplicación:** Se podrían añadir más funcionalidades a la aplicación móvil, por ejemplo:
    - En lugar de obtener el precio de la energía de la península en general, utilizar la API para obtenerlo de una región concreta configurable.
    - Añadir la opción de eliminar cuentas de usuario y datos de consumo, incluyendo la eliminación de datos antiguos después de un periodo configurable.
    - Obtener los datos en tiempo real para que el usuario no tenga que actualizar manualmente.
    - Añadir localización para soportar varios idiomas.

    - Aprovechar la capacidad de los enchufes inteligentes de ser encendidos y apagados en remoto para que la aplicación lo haga automáticamente según las recomendaciones de consumo generadas.

- **Mejora de la optimización del coste**: El algoritmo implementado cumple con la función de reducir el coste y generar recomendaciones intuitivas para el usuario. Sin embargo, existe un gran margen de mejora si se exploran técnicas más avanzadas como modelos estadísticos o *machine learning*. Además, la aplicación no considera que el patrón de consumo puede variar según el tipo de dispositivo. Por ejemplo, una lámpara tiene un patrón *binario*: su consumo solo depende de si está encendida o no. Sin embargo, un aire acondicionado puede variar su consumo mientras esté encendido, dependiendo de la temperatura u otros factores. Un algoritmo más avanzado podría tener esta información en cuenta para generar recomendaciones más relevantes.

Referencias

[1] Espressif, «ESP32 Datasheet,» [En línea]. Available: https://www.esp32.dk/esp32_datasheet_en.pdf
[2] Espressif Systems, «ESP-IDF API Reference,» [En línea]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/index.html
[3] Espressif Systems, «FreeRTOS Overview,» [En línea]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/freertos.html
[4] Espressif, «Arduino core for the ESP32 family of SoCs,» [En línea]. Available: https://github.com/espressif/arduino-esp32
[5] Espressif Systems, «Establish Serial Connection with ESP32,» [En línea]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/get-started/establish-serial-connection.html#establish-serial-connection-with-esp32
[6] Espressif, «Repositorio Github de esptool,» [En línea]. Available: https://github.com/espressif/esptool.
[7] Google, «Firebase Products,» [En línea]. Available: https://firebase.google.com/products-build?hl=es-419.
[8] Google, «Firebase Authentication,» [En línea]. Available: https://firebase.google.com/docs/auth?hl=es-419
[9] Google, «Firebase Realtime Database,» [En línea]. Available: https://firebase.google.com/docs/database?hl=es-419.
[10] Google, «Cloud Functions para Firebase,» [En línea]. Available: https://firebase.google.com/docs/functions?hl=es-419.
[11] Google, «Plataformas, frameworks, bibliotecas y herramientas compatibles,» [En línea]. Available: https://firebase.google.com/docs/libraries?hl=es-419
[12] Google, «Flutter: Build for any screen,» [En línea]. Available: https://flutter.dev/
[13] Google, «Flutter: Beautiful apps for every screen,» [En línea]. Available: https://flutter.dev/multi-platform
[14] Google, «Welcome to Skia: The 2D graphics library,» [En línea]. Available: https://skia.org/
[15] Google, «Dart: An approachable, portable and productive language for high-quality apps on any platform,» [En línea]. Available: https://dart.dev/
[16] Google, «Flutter Docs: Hot reload,» [En línea]. Available: https://docs.flutter.dev/tools/hot-reload
[17] Google, «Android Studio,» [En línea]. Available: https://developer.android.com/studio?hl=es-419
[18] Google, «Pub.dev, the official package repository for Dart and Flutter apps.,» [En línea]. Available: https://pub.dev/
[19] Google, «Add interactivity to your Flutter app: Stateful and stateless widgets,» [En línea]. Available: https://docs.flutter.dev/ui/interactivity#stateful-and-stateless-widgets
[20] Google, «Flutter cookbook: Navigate to a new screen and back,» [En línea]. Available: https://docs.flutter.dev/cookbook/navigation/navigation-basics
[21] Microsoft, «Visual Studio Marketplace: PlatformIO IDE,» [En línea]. Available: https://marketplace.visualstudio.com/items?itemName=platformio.platformio-ide
[22] Dart Code, «Visual Studio Marketplace: Dart,» [En línea]. Available: https://marketplace.visualstudio.com/items?itemName=Dart-Code.dart-code
[23] D. Code, «Visual Studio Marketplace: Flutter,» [En línea]. Available: https://marketplace.visualstudio.com/items?itemName=Dart-Code.flutter
[24] G. community, «Git,» [En línea]. Available: https://git-scm.com/.
[25] Github, «Github,» [En línea]. Available: https://github.com/.
[26] Postman, Inc, «Postman API Platform,» [En línea]. Available: https://www.postman.com/product/
[27] PlantUML, «PlantUML Github,» [En línea]. Available: https://github.com/plantuml/plantuml
[28] PlantUML, «Versión Web de PlantUML,» [En línea]. Available: https://www.plantuml.com/plantuml/uml/
[29] Shelly Spain, «Shelly Spain,» [En línea]. Available: https://shellyspain.com/

[30] Shelly, «Shelly Smart Control Guide,» [En línea]. Available: https://www.shelly.com/blogs/documentation/shelly-smart-control-guide
[31] Red Eléctrica de España, «ESIOS: Red eléctrica,» [En línea]. Available: https://www.esios.ree.es/es#
[32] Hello Watt, «Hello Watt,» [En línea]. Available: https://www.hellowatt.es/
[33] Hello Watt, «Reduce tus facturas de energía con la App de Hello Watt,» [En línea]. Available: https://www.hellowatt.es/seguimiento-consumo-energetico/
[34] Aya Sayed, Yassine Himeur, Abdullah Alsalemi, Faycal Bensaali, Abbes Amira, «Intelligent Edge-Based Recommender System for Internet of Energy Applications,» *IEEE Systems Journal,* vol. 16, nº 9, p. 5001–5010, 2022.
[35] O. H. Foundation, «Home Assistant,» [En línea]. Available: https://www.home-assistant.io/
[36] Google, «Planes de precios de Firebase,» [En línea]. Available: https://firebase.google.com/docs/projects/billing/firebase-pricing-plans?hl=es-419
[37] Google, «Planes de precios de Firebase,» [En línea]. Available: https://firebase.google.com/pricing?hl=es-41
[38] Google, «Configura correos electrónicos con alertas de presupuesto,» [En línea]. Available: https://firebase.google.com/docs/projects/billing/avoid-surprise-bills?hl=es-419#set-up-budget-alert-emails
[39] Google, «Introducción a Firebase Realtime Database,» [En línea]. Available: https://firebase.google.com/docs/database?hl=es-419
[40] Google, «Firebase Authentication,» [En línea]. Available: https://firebase.google.com/docs/auth?hl=es-419
[41] Google, «Introducción a las reglas de seguridad en Realtime Database,» [En línea]. Available: https://firebase.google.com/docs/rules?hl=es-419
[42] Google, «Usa condiciones en las reglas de seguridad de Realtime Database,» [En línea]. Available: https://firebase.google.com/docs/database/security/rules-conditions?hl=es-419
[43] Berland, «Wikimedia commons – Linear interpolation,» [En línea]. Available: https://commons.wikimedia.org/wiki/File:LinearInterpolation.svg
[44] Google, «Firebase: Lee y escribe datos en la web,» [En línea]. Available: https://firebase.google.com/docs/database/web/read-and-write?hl=es-419#delete_data
[45] Google, «Firebase: Indexa tus datos,» [En línea]. Available: https://firebase.google.com/docs/database/security/indexing-data?hl=es-419
[46] Shelly Spain, «Página de producto del enchufe inteligente Shelly Plug Plus S,» [En línea]. Available: https://shellyspain.com/shelly-plus-plug-s.html
[47] Shelly, «Shelly Plus Plug S Knowledge Guide,» [En línea]. Available: https://kb.shelly.cloud/knowledge-base/shelly-plus-plug-s-device-smart-control#ShellySmartControlguidecommonparts-Schedule
[48] Shelly, «Shelly Plus Plug S web interface guide,» [En línea]. Available: https://kb.shelly.cloud/knowledge-base/shelly-plus-plug-s-web-interface-guide
[49] Shelly, «Shelly Smart Control en Google Play Store,» [En línea]. Available: https://play.google.com/store/apps/details?id=cloud.shelly.smartcontrol&pli=1
[50] Shelly, «Documentación de la API del Shelly Plus Plug S,» [En línea]. Available: https://shelly-api-docs.shelly.cloud/gen2/Devices/Gen2/ShellyPlusPlug
[51] O. H. Foundation, «Web del firmware ESPHome,» [En línea]. Available: https://esphome.io/
[52] T. Arends, «Documentación del firmware Tasmota,» [En línea]. Available: https://tasmota.github.io/docs/
[53] Shelly, «Shelly Components and Services: Webhook,» [En línea]. Available: https://shelly-api-docs.shelly.cloud/gen2/ComponentsAndServices/Webhook/
[54] AZ-Delivery, «Datasheet de la pantalla OLED,» [En línea]. Available: https://cdn.shopify.com/s/files/1/1509/1638/files/1_3_Zoll_Display_Datenblatt_AZ-Delivery_Vertriebs_GmbH_rev.pdf?v=160616452
[55] Olikraus, «Repositorio en Github de la librería U8g2,» [En línea]. Available: https://github.com/olikraus/U8g
[56] Shelly, «Shelly Family API Overview,» [En línea]. Available: https://shelly-api-docs.shelly.cloud/gen1/#shelly-family-overview
[57] Espressif, «ESP32 Wi-Fi API,» [En línea]. Available: https://docs.espressif.com/projects/arduino-esp32/en/latest/api/wifi.html
[58] W.-F. Alliance, *Wi-Fi Protected Setup Specification Version 1.0h,* 2006
[59] Espressif, «ESP32 Wi-Fi API: Wi-Fi events example,» [En línea]. Available: https://docs.espressif.com/projects/arduino-esp32/en/latest/api/wifi.html#wi-fi-events-example

[60] P. Cherukupalli, «ESP32 Wi-FI WPS example, espressif/arduino-esp32 Github Repository,» [En línea]. Available: https://github.com/espressif/arduino-esp32/blob/master/libraries/WiFi/examples/WPS/WPS.ino
[61] Shelly, «Shelly API documentation: WiFi.SetConfig example,» [En línea]. Available: https://shelly-api-docs.shelly.cloud/gen2/ComponentsAndServices/WiFi#wifisetconfig-example
[62] Google, «Google Identity Platform: accounts.SignInWithPassword,» [En línea]. Available: https://docs.cloud.google.com/identity-platform/docs/reference/rest/v1/accounts/signInWithPassword
[63] Google, «Autentica solicitudes de REST: Tokens de ID de Firebase,» [En línea]. Available: https://firebase.google.com/docs/database/rest/auth?hl=es-419#firebase_id_tokens
[64] Google, «Google Identity Platform: Exchange a refresh token for an ID token,» [En línea]. Available: https://docs.cloud.google.com/identity-platform/docs/use-rest-api#section-refresh-token
[65] Shelly, «Shelly Switch API, Switch.GetStatus example,» [En línea]. Available: https://shelly-api-docs.shelly.cloud/gen2/ComponentsAndServices/Switch#switchgetstatus-example
[66] B. Lanchon, «Repositiorio en Github de la librería ArduinoJson,» [En línea]. Available: https://github.com/bblanchon/ArduinoJson
[67] Google, «Firebase: Guarda datos – Escribe valores de servidor,» [En línea]. Available: https://firebase.google.com/docs/database/rest/save-data?hl=es-419#section-rest-server-values
[68] Red Eléctrica de España, «API para descarga de información,» [En línea]. Available: https://www.esios.ree.es/es/pagina/api#
[69] Red Eléctrica de España, «Indicator API: Getting a list of indicators,» [En línea]. Available: https://api.esios.ree.es/doc/indicator/getting_a_list_of_indicators.html
[70] Google, «Cloud Functions: Escribe, prueba e implementa tus primeras funciones,» [En línea]. Available: https://firebase.google.com/docs/functions/get-started?hl=es-419
[71] Red Eléctrica, «REData API,» [En línea]. Available: https://www.ree.es/en/datos/apidata
[72] Google, «Agregar Firebase a tu app de Flutter,» [En línea]. Available: https://firebase.google.com/docs/flutter/setup?hl=es-419&platform=android
[73] Google, «Administrar usuarios en Firebase,» [En línea]. Available: https://firebase.google.com/docs/auth/flutter/manage-users?hl=es-419
[74] Google, «Cómo recuperar datos: Shallow,» [En línea]. Available: https://firebase.google.com/docs/database/rest/retrieve-data&hl=es-419#shallow
[75] A. Araujo, «Timeago – pub.dev,» [En línea]. Available: https://pub.dev/packages/timeago
[76] Google, «Habilita funciones sin conexión,» [En línea]. Available: https://firebase.google.com/docs/database/flutter/offline-capabilities?hl=es-419
[77] S. Fortmann-Roe, «Firestore Persistence Loading Speed,» [En línea]. Available: https://github.com/firebase/firebase-js-sdk/issues/2457
[78] «Firebase Google Group: setPersistenceEnabled() on Android slow,» [En línea]. Available: https://groups.google.com/g/firebase-talk/c/G50gHcoSZ30
[79] R. Rousselet, «Provider – pub.dev,» [En línea]. Available: https://pub.dev/packages/provider
[80] flchart.dev, «fl_chart – pub.dev,» [En línea]. Available: https://pub.dev/packages/fl_chart
[81] Google, «Cómo elegir tu base de datos: Cloud Firestore o Realtime Database,» [En línea]. Available: https://firebase.google.com/docs/database/rtdb-vs-firestore?hl=es-419

Apéndice A. Manual de Usuario

A.1. Configuración inicial

Dispositivos necesarios:

- Uno o varios enchufes inteligentes Shelly Smart Plug Plus S, hasta un máximo de cuatro.
- Un ESP32 con el firmware instalado.
- Un dispositivo móvil con Android 14 o superior.

Pasos a seguir:

1. Instale la aplicación en su dispositivo móvil.
2. Inicie sesión con su cuenta de Google.
3. La aplicación le pedirá vincular su dispositivo ESP32. Para hacerlo, introduzca el ESP_ID que viene incluido en su ESP32.
4. Conecte sus enchufes inteligentes y pulse el botón para encenderlos. Verifique si están en el modo correcto comprobando con cualquier dispositivo WiFi que existe una red con nombre que empiece por *ShellyPlusPlugS* por cada uno de los enchufes. En caso contrario, pulse el botón del enchufe inteligente durante al menos 10 segundos para restablecerlo a su estado de fábrica. Si el dispositivo conectado no recibe corriente, pulse el botón del enchufe.
5. Conecte su dispositivo ESP32. Si es la primera vez que se conecta, aparecerá el mensaje "WPS required". En ese caso, pulse el botón WPS de su router WiFi y espere unos segundos hasta que se complete la conexión.
6. El ESP32 mostrará el mensaje *devices found* y el número de dispositivos detectados. Si no detecta todos sus dispositivos, pruebe a reiniciar el ESP32 pulsado el botón *Reset* o cámbielo de posición a una zona con mayor cobertura de señal.
7. El ESP32 mostrará el mensaje *All devices ready* cuando se haya completado la conexión correctamente. Transcurridos unos minutos, mostrará las mediciones obtenidas periódicamente.

A.2. Uso de la aplicación

A.2.1 Pantalla *Mis Dispositivos*

En la pantalla *Mis Dispositivos* podrá visualizar una lista con los dispositivos conectados y sus últimas mediciones, así como botones para actualizar, desvincular y cerrar sesión en la barra superior.

Si la lista aparece vacía:

- Espere unos minutos hasta que el ESP32 empiece a enviar mediciones al servidor y pulse el botón *actualizar*.
- Si aún no aparecen dispositivos, utilice el botón desvincular para volver a introducir el ID de su ESP32, asegurándose de que sea correcto.

Pulsando sobre el icono de lápiz de un dispositivo, puede asignarle un nombre más fácil de recordar. Al pulsar sobre un dispositivo, podrá acceder a su pantalla de *Consumo*. Es posible que la primera vez que se usa la aplicación deba esperar unos segundos mientras se crea la copia local de la base de datos.

A.2.2 Pantalla *Consumo*

En esta pantalla, se visualizan tres gráficas correspondientes al consumo de ese dispositivo en un intervalo de tiempo determinado, establecido por defecto a las últimas 24 horas. Puede cambiar la duración del intervalo utilizando los botones *Día, Semana* y *Mes*. Utilizando los botones de flecha, puede navegar adelante y atrás en el tiempo según la duración del intervalo seleccionado.

Al pulsar sobre el botón *Seleccionar Intervalo*, se mostrará una ventana con un calendario donde podrá introducir un rango de días personalizado que se aplicará a las gráficas tras pulsar el botón *confirmar* situado en la esquina superior derecha.

Las tres gráficas mostradas son:

- Consumo del dispositivo a lo largo del tiempo: Utilizando el selector *Potencia/Energía*, podrá visualizar el consumo instantáneo en Vatios o el consumo acumulado en Vatios-Hora.
- Evolución del precio de la energía a lo largo del tiempo en €/kWh. En esta gráfica podrá visualizar cuáles son las horas con mayor demanda y por tanto, mayor coste.
- Coste del dispositivo a lo largo del tiempo combinando el consumo con los precios. En esta gráfica podrá visualizar a qué horas del día su dispositivo está contribuyendo más coste a su factura de electricidad.

Pulsando sobre cualquier punto de las gráficas obtendrá el valor exacto correspondiente al instante de tiempo seleccionado.

A.2.3 Pantalla *Consejos de Ahorro*

Pulsando sobre el botón *Consejos de Ahorro*, la aplicación generará una recomendación que le ayude a reducir su coste energético. Las recomendaciones se generan una vez al día a las 21:00 y se aplican a las siguientes 24 horas. La pantalla mostrará otras tres gráficas:

- Comparación de los precios actuales con los precios futuros, donde podrá observar cómo evolucionarán los picos de demanda.
- Coste medio por hora del dispositivo durante los últimos 7 días.
- Consumo medio por hora del dispositivo durante los últimos 7 días, superpuesto con la recomendación generada a partir de los precios futuros.

Pulsando sobre el botón de información podrá visualizar los cambios sugeridos por la aplicación, indicando tramos horarios de consumo para desplazar y reducir el coste. Si la aplicación genera demasiados cambios o no alcanza el objetivo, pruebe a reducir el objetivo de ahorro y generar una nueva recomendación pulsando el botón *Aplicar*.

Apéndice B. Manual de instalación

B.1. Firmware del ESP32

El repositorio de código (tfg-iot) está estructurado en tres carpetas principales:

- esp32: Firmware del ESP32
- firebase: Código de la *función cloud*
- app: Código de la aplicación móvil

Para compilar y subir el firmware del ESP32, es necesario instalar Visual Studio Code.
En Visual Studio code, abra el *workspace /tfg-iot/esp32*.

Nota: Será necesario añadir unas credenciales proporcionadas por un administrador de la base de datos al archivo *secrets.template.h* y renombrarlo a *secrets.h*.

Se requiere la extensión *PlatformIO*, deberá instalarla cuando Visual Studio Code lo solicite. Una vez instalada, PlatformIO detectará el archivo de configuración *platformio.ini*, donde se indican las dependencias a instalar. Pulse sobre el icono *Build* en la barra inferior para instalar las dependencias y compilar.

Para subir el firmware, conecte su ESP32 mediante un cable USB y pulse el botón *Upload*. Si no se detecta el ESP32, pruebe a:

- Pulsar el botón *BOOT* del ESP32 durante unos segundos cuando aparezca el mensaje *connecting...* en la consola.
- Instale los drivers UART del ESP32 (CP210x o CH340 según el modelo)

Una vez se ha subido el Firmware, podrá utilizar el ESP32 conectado a su ordenador y monitorizar los eventos generados en el puerto serie pulsando sobre el icono *Serial Monitor*, o bien hacer que funcione de forma independiente alimentando el ESP32 con un cargador USB y opcionalmente añadiendo una pantalla como se explica en *3.2.3 Pantalla Oled*.

B.2. Aplicación móvil

B.2.1 Ejecutar la aplicación en móvil a partir de APK

Transfiera el APK a su dispositivo, por ejemplo, utilizando un cable USB. Utilizando el navegador de archivos, abra el APK y pulse *instalar*. Para que la instalación no sea bloqueada, asegúrese de que tiene activado el ajuste *Activar Aplicaciones desconocidas*. Si ya dispone de un emulador Android configurado, puede simplemente arrastrar el archivo APK a la ventana del emulador y se instalará automáticamente.

B.2.2 Compilar la aplicación a partir del código fuente

Es necesario instalar

- Visual Studio Code
- Flutter SDK. Puede utilizar el comando **flutter doctor** para detectar problemas relacionados con la instalación.

Si además se desea probar la aplicación en emulador:

- Android Studio

En Visual Studio Code, abra el *workspace* /tfg-iot/app. Se requieren las extensiones *Flutter* y *Dart Code*, deberá instalarlas cuando Visual Studio Code lo solicite.

Utilice la terminal para navegar al directorio *smart_energy_app*

cd smart_energy_app

Instale las dependencias con

flutter pub get

Los pasos siguientes dependerán de si desea ejecutar la aplicación en un emulador o su propio dispositivo móvil.

B.2.2.1 En emulador

Deberá crear un dispositivo virtual en Android Studio. Los pasos dependerán de la versión de Android Studio, en el proyecto se utilizó *Android Studio Narwhal 4* versión 2025.1.4.

- Abra Android Studio
- Seleccione *More Actions -> Virtual Device Manager*
- Pulse el botón *Create Virtual Device* situado en la esquina superior izquierda.
- Elija el dispositivo *Medium Phone*.
- Elija la versión de Android a utilizar. En el proyecto se utilizó *API 35 Vanila Ice Cream* (Android 15)
- Descargue la imagen del sistema operativo si se solicita y pulse *Finish*.
- Vuelva a *Virtual Device Manager* y ejecute el emulador pulsando el botón *Start*.

En la consola de Visual Studio, en el directorio *smart_energy_app*, compruebe si Flutter detecta correctamente el emulador:

flutter doctor

Si es así, debería aparecer Android en la lista de dispositivos disponibles.

Ejecute la aplicación con

flutter run –release

Seleccione el emulador cuando Flutter solicite elegir un dispositivo.

B.2.2.2 En dispositivo móvil

Deberá activar el modo desarrollador. Para ello:

- En ajustes, navegue hacia *Acerca del teléfono*
- Pulse 7 veces sobre *Número de compilación*
- Introduzca el PIN de desbloqueo y se mostrará el mensaje *Ahora eres desarrollador*
- Active *Ajustes -> Opciones de Desarrollador -> Depuración USB*

Conecte su dispositivo móvil a un ordenador mediante USB. Pulse *permitir* cuando aparezca un mensaje solicitando autorización para depuración USB.

En la terminal, verifique que su dispositivo móvil aparece correctamente ejecutando

flutter devices

Ejecute la aplicación con el comando

flutter run --release

Seleccione su dispositivo cuando Flutter lo solicite.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga