UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA
INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Control de consumo energético usando Internet of Things

Realizado por

Daryl Serrano Hipolito

Tutorizado por

Francisco Javier Hormigo Aguilar
Andrés Rodriguez Moreno

Departamento
Arquitectura de Computadores

MÁLAGA, (Junio 2020)

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA INFORMÁTICA

Control de consumo energético usando Internet of Things

Control of energy consumption using Internet of Things

Realizado por
Daryl Serrano Hipolito

Tutorizado por
Francisco Javier Hormigo Aguilar
Andrés Rodríguez Moreno

Departamento
Arquitectura de Computadores

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2020

1. Resumen y palabras clave

1.1. Resumen

Diseño e implementación de un sistema Internet de las Cosas para el hogar, que permite registrar, controlar y calcular el consumo de energía eléctrica de los dispositivos de un hogar. Para realizar esta tarea, se han usado enchufes inteligentes con firmware personalizado, utilizando a bajo nivel los sensores y actuadores de dichos enchufes inteligentes donde se conectan los dispositivos del hogar. Los datos de los sensores y los metadatos de los enchufes inteligentes se han almacenado en bases de datos Time Series Database y No Relacionales. Se ha empleado la API REST del precio de energía eléctrica para calcular el precio gastado por hora de los dispositivos. Se ha definido diferentes maneras de automatizar el encendido o apagado de los enchufes inteligentes. El cliente puede acceder la interfaz web del sistema que hemos diseñado, donde se muestra el estado de los enchufes y gráficas del valor de los sensores. Estas gráficas son dinámicas y reciben los datos a intervalos de tiempo programado. La interfaz web se comunica con un servidor web que provee una API REST. Los enchufes se interconectan por el protocolo MQTT y envían sus datos de los sensores a través de este protocolo. El sistema utiliza la tecnología de contenedores para eliminar dependencias, instalar, desplegar y manejar los componentes del sistema. Además se ha diseñado e implementado un flujo de trabajo que testea la aplicación y la despliega a un repositorio de imágenes Docker.

Palabras clave Internet de las Cosas, Hogar Inteligente, Consumo electrico, Automatizacion, Tecnologías Web.

1.2. Abstract

Design and implementation of a smart house IoT system that controls, registers and calculates the price of the electricity consumed by the devices of a home. In order to do this, we have used smart plugs with a custom firmware, using the low level sensors and actuators of these smart plugs connected with the devices of the home. The data of the sensor and metadata of the smart plugs are stored on Time Series Databases and Non Relational Databases. The system uses and makes requests to the API REST of the price of the electricity and uses the data to calculate the price spended by hour for every smartplug.

A front-end where the client can connect to and interact with, has been designed and it shows the state of the smartplugs and displays charts using the data of the sensors. These charts are dynamic and receive data of the sensors on specified periods of time. The front-end talks with a backend web server that serves an API REST. The smartplugs are interconnected by MQTT and send sensor data using this protocol. The system uses container technology for installing, deploying, controlling the components of the system and alleviates dependencies of software. A pipeline of processes that tests and deploy the application to an image repository of Docker has been designed and implemented.

Keywords Internet of Things, Smart Home, Electrical consumption, Automatization, Web Techologies

Índice

1. Resumen y palabras clave **3**

1.1. Resumen . . . 3

1.2. Abstract . . . 3

2. Introducción **7**

2.1. Motivación y objetivos . . . 7

2.1.1. Motivacion . . . 7

2.1.2. Objetivos . . . 7

2.2. Estudio del estado del arte . . . 8

2.3. Metodología a utilizar . . . 13

3. Elección y modificacion de los dispositivos inteligentes **15**

3.1. Elección de enchufe inteligente . . . 15

3.2. Modificacion y configuración de enchufes inteligentes . . . 18

4. Diseño y desarrollo del sistema **21**

4.1. Diseño físico del sistema . . . 21

4.2. Almacenamiento de datos . . . 23

4.3. Aplicación principal . . . 25

4.4. Descubrimiento y registro de enchufes inteligentes . . . 30

4.5. Automatizaciones del sistema . . . 31

4.5.1. Automatizaciones por tiempo . . . 33

4.5.2. Automatización según los valores de los sensores del enchufe inteligente . . . 34

4.5.3. Automatización por precio de la electricidad . . . 36

4.5.4. Automatización por precio gastado . . . 38

4.6. Diseño de los modelos de datos . . . 40

4.6.1. Devices y DevicesGroup . . . 40

4.6.2. Automatizations y triggers . . . 43

4.6.3. Dashboards . . . 45

4.6.4. Site-Settings . . . 45

4.7. Diseño de una REST API para la estación base . . . 45

4.8. Diseño de la interfaz de usuario y comprobación del estado del sistema . . 47
4.8.1. Dashboard 48
4.8.2. Comprobación del estado del sistema 52
4.9. Despliegue y testeo de la aplicación 54

5. Conclusiones y Líneas Futuras 61
5.1. Conclusiones 61
5.2. Lineas futuras 63

6. Apéndice 69
6.1. Manual de instalación 69
6.1.1. Instalacion y configuracion inicial de la estación base 69
6.1.2. Cambio de firmware y configuración inicial de los enchufes inteligentes 70
6.2. Detalles de implementación 79
6.2.1. Repositorios del código e imágenes Docker 79
6.2.2. Ip fija entre contenedores del sistema 80
6.2.3. Task template 80

2. Introducción

2.1. Motivación y objetivos

2.1.1. Motivacion

El motivo principal de este trabajo es desarrollar un sistema de control y monitorización de la energía de consumo doméstico en un hogar. Usando el Internet de las Cosas para poder monitorizar el consumo y controlar los distintos dispositivos y dotarlo de una inteligencia al sistema para controlar el consumo de energía. Apoyándose de la API que provee la Red Eléctrica Española para obtener el precio del Precio Voluntario al Pequeño Consumidor (PVPC) para poder calcular el consumo obtenido a partir de los dispositivos conectados al sistema. Existen varias alternativas para montar un sistema para controlar los dispositivos de un hogar tanto comerciales como de código abierto que usan estándares abiertos, se desea construir uno de estos tipos de sistemas desde cero. Siguiendo cada proceso de diseño del sistema desde la capa física hasta la implementación de la aplicación principal que utilizará el cliente. Es decir diseñar e implementar un sistema Internet de las Cosas, un sistema *smart home* que es capaz de interconectar dispositivos, controlar, medir y automatizar el uso de los dispositivos de un hogar doméstico apoyándose en el uso de los datos de los precios del PVPC a través de la API de la Red Eléctrica Española.

2.1.2. Objetivos

Se desea diseñar e implementar un sistema Internet de las Cosas en el que los objetivos principales y secundarios son:

- Interconectar los dispositivos a un sistema de control de consumo energético
- Obtener información del consumo de energía de los dispositivos.
  - Proveer una interfaz de usuario para mostrar la información de los dispositivos a distancia.
- Controlar el consumo de dichos dispositivos.
  - Proveer al usuario diferentes maneras de controlar el dispositivo tanto manualmente como automáticamente.

- Calcular el gasto del consumo generado por los dispositivos.

- Instalación y configuración relativamente sencilla.

2.2. Estudio del estado del arte

«Una solución de Internet de las Cosas consiste en dar inteligencia a objetos y dispositivos para que puedan comportarse de forma autónoma» Radovici, Culic y Rusu [17].

A día de hoy el internet de las cosas consiste en dispositivos *things* que se interconectan entre ellos a través de una red informática, dichos dispositivos publican información y con esa información dotar de inteligencia a los dispositivos para que interactúen de forma esperada o adecuada con el mundo físico.

Citado por Barton, Salgueiro y Hanes [1], segun Cisco, a partir de 2020 se estima que sobre 50 billones de *things* estarán conectados a Internet y que además se estima que esto llevara a unos beneficios y ahorros de 19 trillones de dólares. Esto implica que el Internet de las Cosas tiene un gran impacto sobre el mundo, el Internet de las Cosas se utiliza en muchas áreas diferentes para facilitar tanto procesos comerciales como la vida humana. Se usa tanto en industria como en las ciudades con edificios inteligentes, *smart buildings*, donde se utilizan sensores de presencia para controlar las luces de un edificio, calcular y saber la cantidad de tiempo en el que una habitacion en un edificio se utiliza, en carreteras conectadas, dodne se colocan sensores en intersecciones y a partir de los datos de los sensores inferir el tráfico e informar a vehículos inteligentes, de modo que entre estos vehículos compartir información entre sí y decidir que camino sería optimo para evitar tráfico, el Internet de las Cosas se utiliza incluso en criaturas animales, *smart creatures*, a los animales se les dota de chips que envían información de sus sensores y a partir de estos datos, por ejemplo obbtener estadísticas sobre una granja de animales, o encontrar supervivientes tras un desastre natural, a través de los sensores adheridos a un insecto para encontrar supervivientes a través de los sensores de sonido que porta. [1]

Todas las soluciones e implementaciones de un sistema de Internet de las Cosas se basan en una arquitectura definida, lo habitual en un sistema de Internet de las Cosa es que esté compuesta principalmente de dispositivos inteligentes con sensores y/o actuadores, un banco donde almacenar los datos de la información recogida, interconexiones entre los dispositivos y un punto de procesamiento de los datos, aunque existen muchas diferencias entre cada solución de Internet de las Cosas [17]. El sistema que vamos a desarrollar

principalmente contiene esos componentes. En la figura 1 podemos apreciar una gráfica con los principales de un sistema de Internet de las Cosas.

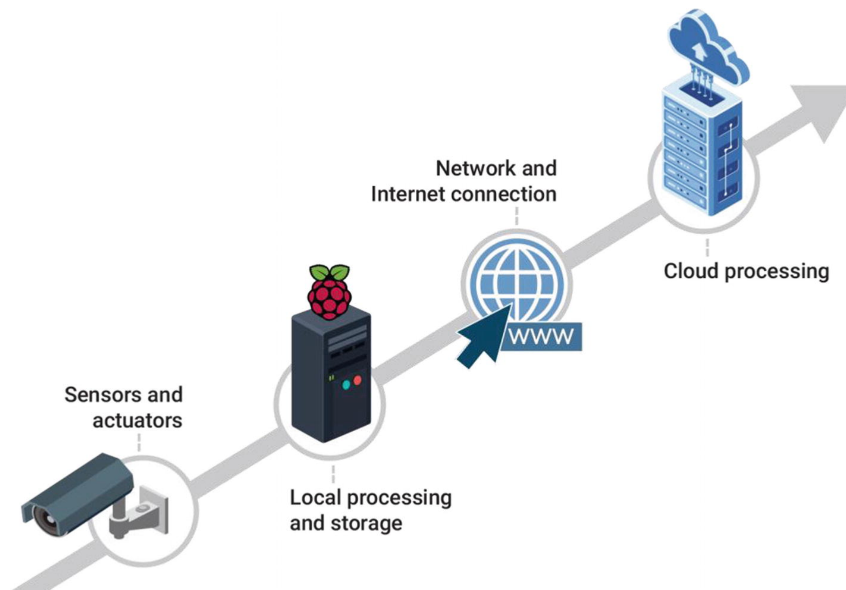

Figura 1: Radovici, Culic y Rusu [17] Diagrama de componentes de un sistema de Internet de las Cosas

En relacion con dispositivos inteligentes de un sistema de Internet de las Cosas, estos pueden contener uno o varios sensores y actuadores. Los sensores son los encargados de recoger datos sobre el medio, permiten sentirlo y recopilar datos sobre él. Los sensores pueden ser desde sensores de temperatura hasta cámaras que graban video. Los actuadores son los que permiten interactuar con el medio físico, se encargan de realizar acciones físicas en un dispositivo. Los dispositivos inteligentes además de tener sensores y actuadores tienen un elemento para que puedan conectarse a una red informática normalmente a una red inalámbrica y deben de tener una pequeña unidad de procesamiento [1].

Los dispositivos inteligentes tienen que estar conectados a una red informática de comunicación para transmitir los datos recogidos y recibir información sobre cómo actuar, para ello los dispositivos inteligentes transmiten los datos y los reciben a través de protocolos para transmitir datos, los protocolos más populares se pueden dividir según la capa del modelo OSI en las que se implementan estos protocolos. Para la capa de física o de bajo nivel los más populares son ethernet, WiFi, Z-Wave, ZigBee, Bluetooth. Mientras que para la capa de datos y aplicación están MQTT, CoAP, STOMP y AMQP [17].

El protocolo que se utiliza en este trabajo, MQTT es un protocolo muy popular en el Internet de las Cosas, es un protocolo de publish/subscribe y diseñado para transportar

mensajes en conexiones remotas y de poco bando de ancha. El protocolo está diseñado para funcionar sobre redes TCP/IP. Los clientes se conectan a un servidor/broker. Estos clientes se suscriben y publican los datos en diferentes topics. El protocolo permite tener autenticación y encriptación por SSL. Además, tiene tres modos de calidad de servicio para transmitir los datos, en el nivel 0, no importa que el receptor no haya recibido el mensaje, en el 1, se garantiza que el mensaje lo haya recibido el cliente al menos una vez y en el nivel 2, que haya recibido el mensaje y que no sea duplicado.

Dentro de los sistemas de Internet de las Cosas aparece también la figura de *Gateway*, es el encargado de ser el componente que interconecta los dispositivos inteligentes y el que proporciona una comunicación bidireccional entre los dispositivos inteligentes y otros componentes del sistema Internet de las Cosas. Cabe descatacar que los dispositivos inteligentes utilizan diferentes protocolos de comunicación, con lo cuál el *Gateway* se encarga de interconectar por un sistema heterogéneo de protocolos de comunicación. Los *Gateway* a veces también son los que se encargan de realizar el procesamiento de datos recogidos por los sensores de los dispositivos inteligentes. Estos datos procesados se pasan a otros *gateway* o directamente con el resultado de los datos procesados, aplica acciones a los dispositivos inteligentes [18]. Ya que los *gateway* pueden tener la capacidad de procesar datos, normalmente son dispositivos con capacidad computación microcontroladores tales como Arduino, UDOO NEO, productos de Internet de las Cosas de Intel, Raspberry Pi y BeagleBone [10].

Otro componente que también se tiene en cuenta en un Sistema Internet de las Cosas son los servicios de la nube, dependiendo de cómo queramos escalar el sistema Internet de las Cosas utilizar servicios de la nube puede ser crucial, tanto para mantener los datos como para realizar procesamiento que no se podría hacer en un *gateway*. Un ejemplo, utilizar servicios de la nube para machine learning o big data. Los *gateway* normalmente son los encargados de traspasar los datos de los sensores a la nube, pueden pasar todos los datos o solo el resultado de los datos procesados desde el propio *gateway*. Cabe destacar también que es preferible utilizar la nube para aplicaciones en las que los dispositivos inteligentes publican sus datos de sus sensores en diferentes partes, de este modo se consigue un punto central donde procesar los datos [10]. En este trabajo no se utilizan servicios de la nube aunque parte del sistema desarrollado puede llevarse a la nube.

Por último, otro componente dentro de un sistema de Internet de las cosas son las in-

terfaces de usuario, son un componente importante para los clientes finales. Proporcionan una manera para que los clientes/usuarios puedan interactuar con el sistema de Internet de las Cosas, estas interfaces pueden ser desde paneles táctiles usados, usualmente usado para las casas inteligentes, hasta aplicaciones web que se comunican con el sistema a través de una API REST.

En cuanto el desarrollo software para Internet de las Cosas, depende de las características del hardware del sistema Internet de las Cosas [17]. Cuando se desarrolla software para dispositivos inteligentes lo normal es que las herramientas y lenguajes de programación tiendan a ser de bajo nivel puesto que estos dispositivos inteligentes trabajan con sensores y actuadores directamente. El lenguaje por defecto y más adecuado es uno de bajo nivel como C y C++ aunque existen proyectos que proveen la posibilidad de programar a bajo nivel utilizando lenguajes interpretados como como JerryScript y MicroPython.

Si se trata de la programación para interfaces de usuario es decir el componente más cercano a el de los clientes/usuarios de un sistema de Internet de las Cosas, normalmente se desarrollan aplicaciones web con lenguajes como Javascript, Python o PHP o servicios web programados.

Como el procesamiento de los datos es importante en una arquitectura de un sistema Internet de las Cosas, y a veces hay que almacenar los datos para posterioremente procesarlos o guardar información sobre los dispositivos inteligentes conectados al sistema, se utilizan bases de datos para guardar los datos de forma persistente. Las bases de datos se dividen en dos tipos las bases de datos Relacionales y No Relacionales aunque hay otro tipo de base de datos relevante para Internet de las Cosas, llamado Time Series Databases. Las bases de datos Relacionales proveen una manera estructurada para recoger y guardar datos, emplean el lenguaje *Structured Query Language(SQL)*, los datos están estructurados en forma de tablas con filas y columnas [2]. Varios ejemplos de sistema de bases de datos Relacionales pueden ser, MySQL, PostgreSQL y SQLite. Los datos de una base de datos Relacional se caracterizan por estar definidos por un esquema de datos. Las bases de datos No Relacionales como su nombre indica no guardan datos con una forma estructurada definida, en las bases de datos No Relacionales no existe un esquema de datos definido, el esquema puede cambiar en cualquier momento, normalmente son *document oriented databases*. Las colleciones de documentos reemplazan las tablas, un documento es una fila dentro de la tabla y dentro de cada documento, contienen un con-

junto de pares clave-valor, estos pares pueden cambiar en cualquier momento [3]. Unos ejemplos de bases de datos No relacionales son MongoDB, CouchDB, DynamoDB. Las Time Series Databases son unas bases de datos especializadas para datos *time-stamped* es decir, datos asociados a un punto temporal, los datos son medidas o eventos, por ejemplo métricas de server, datos de sensores y diferentes tipos de datos para análisis. Las time series database tienen diferentes propósitos de uso pero principalmente se utiliza para análisis y métricas de los datos al igual que para detectar patrones y alertar de cambios significantes en los datos tomados. Unos ejemplos de Time Series Databases son InfluxDB, Amazon Timestream, Prometheus, Graphite y RRDtool [12].

De entre todos los ámbitos en los que se puede aplicar Internet de las Cosas, el ámbito de interés de este trabajo es el doméstico, más en concreto los sistemas *home automation/smart home* o hogar inteligente. Como todo sistema Internet de las Cosas, está compuesto por sensores, actuadores, *gateway* y un banco donde procesar los datos, aparte de esto cabe destacar que existen otros elementos más como plataformas de integración, estos son frameworks o plataformas especializados para interconectar dispositivos inteligentes diferentes entre sí. Hay diferentes plataformas de integración unas pueden ser comerciales como Amazon Alexa, Samsung Smartthings y Apple HomeKit. Y otras plataformas de integración pueden ser de código abierto y que utilizan protocolos abiertos, unos ejemplos son openHAB, Domoticz y Home Assistant [10].

Los *gateway* específicos para un hogar inteligente proporcionan una API/SDK para añadir o crear nuevas funcionalidades y/o aprovecharse de la interfaz que sirven para comunicarse con los demás dispositivos inteligentes dentro de la red. [10]

En cuanto a dispositivos inteligentes para obtener información del consumo energético de otros dispositivos no inteligentes domésticos en un hogar. Unos ejemplos de estos tipos de dispositivos son los enchufes inteligentes (*smart plugs*) y tomas de corriente (*sockets*). Los más fáciles de utilizar son los enchufes inteligentes. Los enchufes inteligentes son dispositivos que contienen unos sensores que permiten medir tanto el voltaje de la corriente, el amperaje, la potencia y la energía eléctrica. Son muy convenientes puesto que también contienen un interruptor(actuador) que permite cortar el paso de la energía eléctrica hacia el dispositivo que esté conectado al enchufe o toma de corriente. Aparte de esto, proveen la posibilidad de ser manejados a distancia, a través de una red de comunicación, en estos casos en un red Wifi. Algunos de estos dispositivos incluso proveen

automatizaciones sencillas como cortar y dejar pasar la corriente eléctrica en determinado periodos de tiempo.

2.3. Metodología a utilizar

Para desarrollar el sistema se ha empleado el método agile que consiste en un conjunto de metodologías, métodos y buenas prácticas para trabajar en equipo más eficiente y hacer mejores decisiones [19]. Aunque esta metodología está centrada para equipos software, parte de ella puede usarse para proyectos software donde solo hay un desarrollador. En concreto se ha adaptado el uso del sistema Kanban y el tablero Kanban. El sistema Kanban es un sistema para comprobar el progreso del proyecto mientras que el tablero Kanban es la visualización representativa del progreso del proyecto y su proceso de desarrollo [9].

El sistema Kanban consiste en tres principios: visualizar el trabajo con la posibilidad de comprobar el estado de cada item del flujo de trabajo, limitar el número de trabajo en proceso estableciendo un limite de items que se realiza a la vez y manejar el flujo de trabajo rápidamente sin ninguna interrupción en el flujo de trabajo, esto se puede hacer gracias a que hay una visualización general de todo el flujo del trabajo y con ello, saber en qué punto mejorar dicho flujo o aumentar o bajar el límite de tareas simultáneas [9].

Se ha elegido este sistema por ser el más rápido de utilizar y de implementar. facil de comprobar el estado del desarrollo y tener un punto en el que guardar las ideas y tareas a realizar para el trabajo.

Asi, en el desarrollo del trabajo se ha establecido un flujo de trabajo en el que cada columna del tablero Kanban consiste en una fase del desarrollo del proyecto software. Se ha empezado el proyecto con tres columnas básicas para el flujo de desarrollo cada item dentro de el tablero Kanban consiste en historias, requerimientos o tareas. Durante el desarrollo del trabajo se ha tomado la desicion de poder partir estos items para que sean más manejables, a diferencia de que en un sistema Kanban, los ítems tienen que ser unidades de trabajo autocontenidas para que puedan tener seguimiento durante todo el flujo de trabajo [19]. En el caso de este trabajo las item no son autocontenidas y unas dependen de otras, pero se ha realizado así para poder delimitar bien la cantidad de trabajo a realizar a la vez. En cuanto a las columnas, la primera columna, la columna Tareas(TODO), que consiste en todos los requisitos y tareas del proyecto, la siguiente columna a su derecha llamada columna En proceso(WIP), que consiste en las tareas o

ítems y por último la columna de Hecho(Done) consiste en las items acabadas.

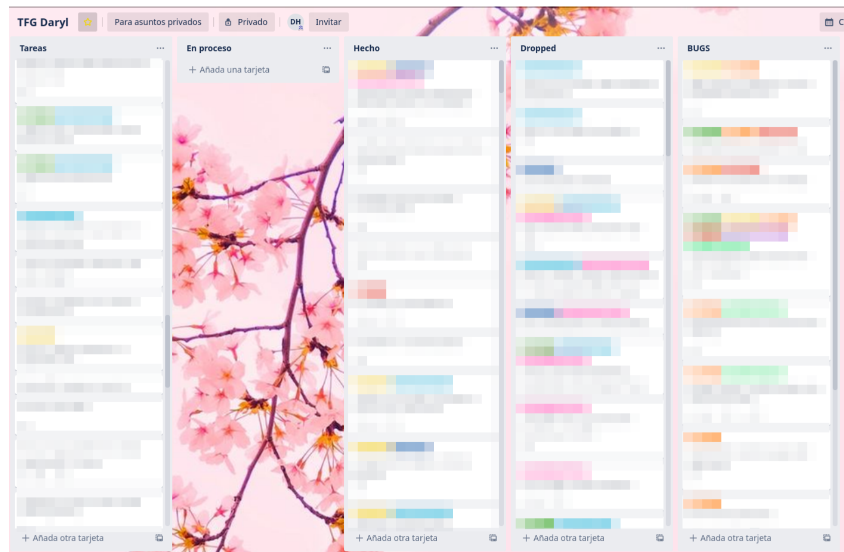

Figura 2: Tablero Kanban utilizado durante el desarrollo

Como el sistema Kanban es un sistema en el que continuamente el flujo de trabajo cambia y mejora, a mitad del desarrollo del proyecto se han añadido otras columnas más al tablero Kanban cambiando así el flujo de trabajo, las columnas añadidas son las de Bugs y de Dropped. La columna de Bugs sirve para tener un seguimiento de que bugs y errores que se han encontrado en el sistema y la columna de Dropped sirve para tener seguimiento de ítems que se han propuesto y que no pueden realizarse en el tiempo en el que se ha estimado la entrega del producto software, estos ítems se quedan como ideas de ampliación del trabajo realizado. La figura 2 refleja el tablero Kanban utilizado durante el desarrollo del trabajo.

Se ha utilizado la herramienta web Trello para este caso, ya que proporciona un tablero Kanban online.

3. Elección y modificacion de los dispositivos inteligentes

3.1. Elección de enchufe inteligente

Como se explicó en el apartado 2.2, un sistema Internet de las Cosas contiene dispositivos inteligentes, en este caso los dispositivos inteligentes que se utilizan en este trabajo deben de poseer sensores para obtener la información de la energía electrica consumida, además, deben de tener actuadores para controlar el paso de dicha corriente electrica.

Para esta fase se ha realizado una búsqueda de diferentes enchufes inteligentes comerciales y se ha seleccionado uno de entre esos enchufes inteligentes. El motivo por el que se ha decidido escoger dispositivos comerciales es debido a que ya incorporan el hardware necesario, tales como los sensores para la monitorización de la corriente y los actuadores para controlar el uso de la corriente eléctrica además tienen la posibilidad de ser conectados por medio de una red informática. Se han seleccionado cuatro enchufes inteligentes comerciales: TP-Link HS110, BlitzWolf BW-SHP2, D-Link DSP-W215.

El enchufe inteligente TP-Link HS110 es un dispositivo provisto con monitorización de energía con un botón manual de encendido y apagado. No solo se ha seleccionado este enchufe inteligente por ser un producto fabricado por TP-Link, una marca comercial famosa, sino que principalmente por la característica anteriormente mencionada, la monitorización de energía. Esto implica que este dispositivo ya viene incluido con los sensores necesarios. Este dispositivo usa un software propietario para poder ser manejado, en concreto usa una aplicación móvil compatible con Android e iOS, esta aplicación se llama TP-Link Kasa. Además, tiene la posibilidad de ser integrado con Amazon Alexa, Google Assistant e IFTTT.

En cuanto al enchufe inteligente BlitzWolf BW-SHP2 al igual que el enchufe inteligente TP-Link, está provisto con la característica de monitorización de la energía, botón manual de encendido y apagado e integraciones con Amazon Alexa, Google Assistant e IFTTT. Otra característica interesante de este dispositivo es la protección que ofrece, si el aparato conectado a él tiene una potencia mayor de la soportada, este se apaga. Se ha seleccionado este enchufe inteligente debido a que es un producto con un precio asequible a pesar de ser un producto fabricado por una marca moderadamente popular. La aplicación que es utilizada para controlar el dispositivo es Smart Life, una aplicación

propietaria desarrollada por Tuya, una compañía desarrolladora de plataformas Internet de las Cosas.

Por último el enchufe inteligente D-Link DSP-W215, que igualmente tiene casi las mismas características que los otros dos enchufes inteligentes nombrados anteriormente, monitorización de energía, botón manual de encendido y apagado e integración con Amazon Alexa e IFTTT. Igualmente como el enchufe BlitzWolf, este dispositivo viene provisto de una protección del dispositivo, solo que es diferente de el otro enchufe anteriormente mencionado puesto que es una protección por recalentamiento del enchufe. El enchufe lleva integrado un sensor de temperatura que apaga automáticamente el enchufe si se sobrecalienta. Al igual que los otros enchufes inteligentes, se maneja por medio de un aplicación de tercero denominada mydlink desarrollada por D-Link. El principal motivo por el que se ha seleccionado este es debido a la capacidad monitorización de la energía.

Finalmente se ha optado por elegir el enchufe inteligente BlitzWolf BW-SHP2 gracias a que este dispositivo está provisto por un microcontrolador ESP8266, este microcontrolador tiene la capacidad de realizar conexiones Wi-Fi y se usa mayormente para dispositivos Internet de las Cosas. Al ser un microcontrolador muy famoso en Internet de las Cosas existen firmwares alternativos y proyectos de código abierto que pueden ser usados para modificar el firmware existente del microcontrolador ESP8266. Actualmente existen dos proyectos de código abierto que permiten utilzar los sensores y controlar el enchufe inteligente BlitzWolf BW-SHP2, ESPurna y Tasmota.

Se ha decidido cambiar el firmware del enchufe inteligente que hemos elegido puesto que esto nos provee de la total libertad de control sobre el dispositivo ya que no dependemos de una aplicación de terceros y que este se conecte a un servicio de la nube prefijado por el propio fabricante de el enchufe. Además ya no se dependerá del fabricante y a la espera de que estos lancen una actualización del firmware puesto que nosotros mismos podríamos encargarnos de realizar cambios en el firmware alternativo que hemos instalado. Otro motivo por el que se ha elegido este enchufe inteligente es su bajo precio del mercado, haciéndolo el más barato y asequible de los tres. Sobre todo otro motivo más importante por el que se ha elegido este enchufe es la seguridad, tanto los enchufes TP-Link HS110 y D-Link DSP-W215 han sido examinados sobre la vulnerabilidades que presentan. Con respecto al enchufe inteligente TP-Link HS110, como «cualquier persona que esté dentro de la misma red local que el dispositivo tiene control absoluto y total del dispositivo debido

a que no existe autenticación para los usuarios locales, además de poseer un cifrado que es posible romper y de este modo observar las acciones que se realizan con este dispositivo.» [20] En el caso del enchufe inteligente D-Link DSP-W215 existe una vulnerabilidad que permite obtener el control completo y total del dispositivo [5, 6]

En conclusión, el dispositivo a utilizar para la implementación del sistema de control de energía es BlitzWolf BW-SHP2, cumple con las necesidades primaria de control del uso de energía y la capacidad de sentir y recoger la información de su uso. Además realizaremos una modificación del firmware de fabrica y lo reemplazamos por un firmware alternativo. La comparación de las diferencias técnicas entre los tres enchufes inteligentes se pueden observar de manera intuitiva desde el cuadro 1

	TP-Link HS110	D-Link DSP-W215	BlitzWolf BW-SHP2
Protocolo red	IEEE 802.11b/g/n 2.4GHz	IEEE 802.11b/g/n 2.4GHz	IEEE 802.11b/g/n 2.4GHz
Botones	Botón de encendigo/apagado y Botón reset	Botón de encendigo/apagado	Botón de encendigo/apagado
Voltaje de entrada	100-240VAC	100-240V	110-240V
Voltaje de salida	100-240VAC	-	-
Monitorización de energía	Si	Si	Si
Alimentación Máxima	3.84W	3.84W	3.84W
Temperatura de funcionamiento	0Cº˜4Cº	0Cº˜4Cº	-10Cº˜60Cº
Modo de control	App TP-Link Kasa	App mydlink	App Smart life
Integraciones	Amazon Alexa, Google Asistant, IFTTT	Amazon Alexa, IFTTT	Amazon Alexa, Google Asistant, IFTTT
Vulnerabilidades	Control absoluto	Control absoluto	-
Firmware alternativo	Librerias de terceros	Librerias de terceros	Firmware alternativo

Cuadro 1: Diferencias técnicas entre los enchufes inteligentes seleccionados

El hecho de que el enchufe inteligente BW-SHP2 no tenga vulnerabilidades no significa que esté protegido del todo, aún así es mejor elegir un enchufe inteligente en el que no se haya revelado por ahora un problema de seguridad.

3.2. Modificacion y configuración de enchufes inteligentes

Una vez elegido el dispositivo inteligente en concreto a usar para nuestro sistema y además tras haber argumentado las razones por las que modificar el firmware existente del fabricante se presentan dos opciones: utilizar un firmware alternativo ya desarrollado o desarrollar firmware alternativo. Se ha decidido optar por el uso de un firmware alternativo existente ya desarrollado debido a que el desarrollo de un firmware propio para el enchufe inteligente es complejo, está fuera de el alcance del proyecto y su desarrollo no es el objetivo de este proyecto.

Como se mencionó anteriormente, existen dos grandes proyectos de código abierto que proveen un firmware alternativo para el enchufe inteligente que hemos seleccionado: ESPurna y Tasmota, ambos son compatibles con una gran variedad de dispositivos inteligentes, esto puede ser una gran ventaja puesto que de este modo se puede utilizar no solo para nuestro dispositivo elegido si no que se puede ampliar nuestro sistema e incorporar nuevos dispositivos que urilizen estos firmwares. Ambos tienen unas características comunes como:

- Conexión Wifi en modo AP (Punto de acceso) o STA (Station)
- Soporte de MQTT para controlar los actuadores y sensores.
- Integración con Google Assistant y Amazon Alexa.
- Soporte para Integración con software de domótica como Domoticz, openHAB, Home Assitant o Thingspeak...
- Soporte para actualizaciones OTA (Over-The-Air)
- Binarios del firmware pre-compilados
- Soporte para acciones simples de control como encendido y apagado por tiempo.
- Sistema de logs.

Finalmente se ha decidido elegir ESPurna debido a que en comparación a Tasmota, ofrece distintos binarios pre-compilados y específicos para cada dispositivo, esto en contraste con los binarios pre-compilados que ofrece Tasmota que incluye todo el soporte para todos los dispositivos y sensores que indica en el proyecto como consecuencia existe

código que nunca se ejecutará puesto que dicho código está diseñado para ejecutarse en otro dispositivo diferente el enchufe inteligente que hemos elegido.

La modificación e instalación del firmware y la configuración inicial se recogen en el apendice 6.1.2.

4. Diseño y desarrollo del sistema

4.1. Diseño físico del sistema

Una vez decidido el tipo de enchufe inteligente que se utilizará para controlar los dispositivos eléctricos de un hogar doméstico. Hay que diseñar la infraestructura física y el software del sistema *smart home*. Empezando primero por la infraestructura física, como se ha mencionado antes, tenemos los enchufes inteligentes. Necesitamos un medio con el que comunicarnos con los enchufes inteligentes, es decir una red informática donde los enchufes inteligentes pueden conectarse, se decide por una red WiFi puesto que es la única que soporta los enchufes inteligentes y es el método más utilizado para dispositivos inteligentes del Internet de las Cosas.

Para poder comunicarse con todos los enchufes inteligentes, debe de haber un punto central en el que recoger la información publicada por los sensores de los enchufes inteligentes y controlarlos. Este punto central será el *gateway* o estación base, debe de interconectar los enchufes inteligentes y proveer un medio en el que el cliente pueda ver el estado de los enchufes inteligentes, la información de sus sensores y controlarlos. Debe también ser un dispositivo capaz de conectarse a una red WiFi y tener una capacidad de cómputo adecuada para cumplir estos objetivos. Necesitamos una *single board computer*. Existen varios *single board computer* como Raspberry Pi, Arduino, BeagleBoard, Banana Pi. Finalmente se ha decidido por elegir una Raspberry Pi, en concreto la Raspberry Pi 4 B puesto que es la más nueva en el mercado y contiene suficiente capacidad de cómputo, además de poseer un procesador (ARM v8) de 64-bit. Además de posee tanto la posibilidad de conexión inalámbrica WiFi como conexión por ethernet. Otra ventaja que posee es un amplio soporte técnico y una comunidad activa tanto de desarrolladores como de entusiastas [16]. Al tener la suficiente capacidad de cómputo y tener un procesador de 64-bits podemos desplegar contenedores de aplicaciones deseadas para el sistema que se ha desarrollado, en concreto MongoDB, una base de datos utilizada para almacenar información del sistema y que se detalla su uso en el apartado 4.2.

Una vez definida la infraestructura física del sistema debemos elegir un protocolo en el que se realizará la interconexión y comunicación de los enchufes inteligentes con la estación base. Puesto que no solo basta con tener una conexión WiFi donde conectar todos los dispositivos, se necesita una manera de pasar mensajes o comandos entre los

enchufes inteligentes y la estación base. Dicho protocolo debe de funcionar sobre una red de Internet, IP que es el protocolo usado por los enchufes inteligentes conectados. Existen varias protocolos para realizar el traspaso de mensajes y comandos, como por ejemplo MQTT, CoAP, AMQP, Websockets, XMPP. Al final se ha decidido por elegir el protocolo MQTT puesto que los enchufes inteligentes soportan solo ese protocolo de entre todos los protocolos anteriormente mencionados. Como se explicó anteriormente en el apartado 2.2, el protocolo MQTT necesita un Broker/Servidor que reciba todos los mensajes de los clientes conectados a el que a su vez enrutan a los diferentes clientes que estén esperando a mensajes en el *topic* correspondiente. Podemos identificar como clientes los enchufes inteligentes y la propia estación base.

Los enchufes inteligentes son los encargados de enviar información de sus sensores a través de los MQTT en los *topics* correspondientes. El Broker, estación base, recibe estos mensajes y la propia estación base estará pendiente de los *topic* de cada enchufe inteligente. En concreto la aplicación que se desarrollara y que se aloja en la estación, será la encargada de escuchar y recoger estos mensajes. Podemos observar en la figura 3 un diagrama de la intercomunicación entre los clientes/enchufes inteligentes y el Broker/estación base.

Existen varios brokers que elegir y de entre ellos se ha elegido Eclipse Mosquitto principalmente porque existe una imagen Docker de dicha, aplicación, en concreto una imagen para ARM32 y ARM64, justo para usarse en una Raspberry Pi.

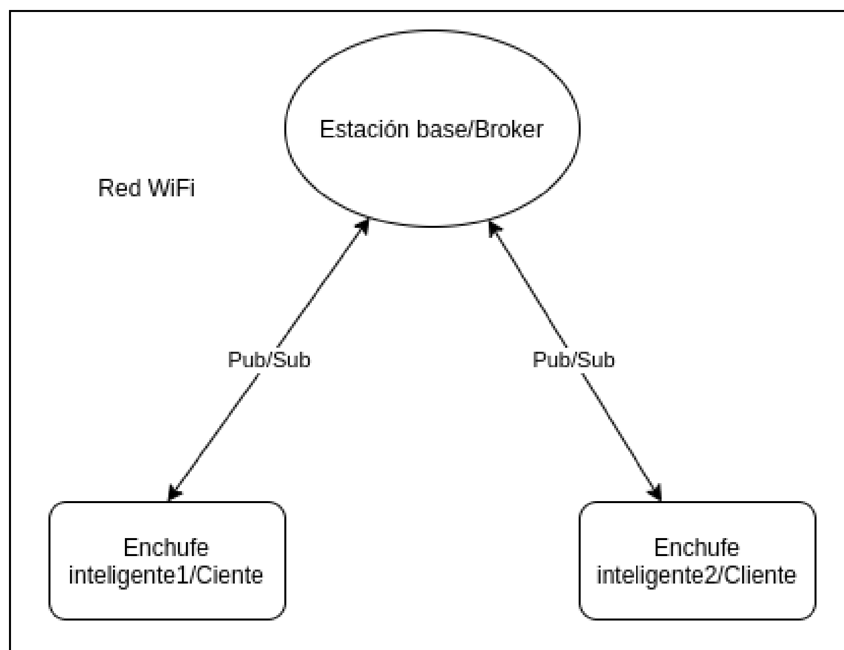

Figura 3: diagrama fisico del sistema diseñado

4.2. Almacenamiento de datos

El sistema puede comunicarse con los enchufes, interactuar con ellos y recoger los datos de estos enchufes inteligentes pero una vez que el sistema se reinicie, este perderá los datos sobre que dispositivo inteligente hay conectado al sistema. El sistema debe de almacenar datos permanentes sobre estos dispositivos inteligentes, debe de saber cómo identificarlos para después registrarlos en el sistema y guardar esa información de identificación. Dicha información de identificación debe de ser persistente y única. Para solucionar este problema podemos emplear ficheros con formato sencillo para guardar esta información sobre los dispositivos inteligentes que hay en el sistema. Pero esto llega a ser una solución complicada puesto que para acceder a los datos, debemos de programar una manera para recuperar los datos, guardarlos en un formato específico y parsearlos de nuevo para usarlos en otro programa [2]. Teniendo en cuenta esto y que no estamos muy restringidos en cuanto a hardware disponible ya que usamos una Raspberry Pi 4, vamos a utilizar una base de datos para guardar los datos sobres los enchufes inteligentes.

Como se explicó en el apartado 2.2, existen diferentes tipos de bases de datos, las Relacionales y las No Relacionales, se ha optado por elegir una base de datos No Relacional. Esto es debido a que las bases de datos Relacionales tienen diversos problemas en el caso de sistemas Internet de las Cosas, tienen problemas de escalabilidad esto puede suponer un problema teniendo en cuenta que empleamos una base de datos en un sistema de Internet de las cosas, donde continuamente se guarda datos y con ello lleva a problemas de rendimiento, mientras que las bases de datos No Relacionales están diseñadas para escalar y escalar de forma horizontal. Otro problema que se presenta es en el esquema de datos, en un sistema Internet de las Cosas el modelo de datos seguramente cambiará con el tiempo con lo cual es deseable tener un esquema dinámico el problema que tienen las bases de datos Relacionales es que sus esquemas de datos han de diseñarse correctamente desde el inicio al contrario que con las bases de datos No Relacionales, estas pueden cambiar fácilmente su esquema de datos [1].

La base de datos No Relacional que se ha elegido es MongoDB, puesto que en ella los datos están en forma de documentos con pares de clave, valor y además los documentos son similares a objetos JSON, de este modo es fácil transformar los datos en objetos de Javascript [21]. También encaja adecuadamente con el apartado 4.7, donde se explica el diseño e implementación de una API REST donde la representación de los recursos está

en JSON, la base de datos MongoDB contiene documentos de los recursos del sistema Internet de las Cosas. De este modo tenemos una manera de guardar de forma persistente los datos e información sobre los enchufes inteligentes conectados al sistema.

Aparte de guardar la información sobre los enchufes inteligentes, debemos de guardar los datos de los sensores de los enchufes inteligentes. Debido a que los enchufes inteligentes publican los datos de sus sensores de forma continua en los *topics* correspondientes a sus sensores, estos datos también deben de ser almacenados para posteriormente acceder a ellos y procesarlos. Como se trata de un gran volumen de cantidad de datos en el que el tiempo de la toma de los datos es relevante junto con el valor, se utilizará una base de datos *Timed series database*. Una time series data base está diseñada para guardar grandes cantidades de datos de medida que nunca van a ser editados de nuevo. [8]. Aunque también existen maneras específicas para guardar time series en unas bases de datos Relacionales y No Relacionales, se ha decidido por utilizar una base de datos especializada en time series. En concreto se ha decidido utilizar InfluxDB puesto que es una de las bases de datos time series más maduras. Otro motivo por el que se ha elegido InfluxDB es por el hecho de que los desarrolladores de InfluxDB, a parte de ofrecer la base de datos InfluxDB, ofrecen aplicaciones que la complementan, estas aplicaciones son Telegraf y Kapacitor.

En el sistema de Internet de las Cosas se necesita antes realizar el paso de obtención de los datos a partir de los *topics* de el Broker de MQTT de la estación base para posteriormente guardarlos en InfluxDB. La aplicación principal debe escuchar los datos publicados en MQTT y guardarlos en InfluxDB. Telegraf es una aplicación que puede realizar esta acción, por tanto, se ha decidido utilizarla. Telegraf es un agente de recogida de datos, permite recoger datos de cualquier servicio/fuente y enviarlos a InfluxDB, esto supone que no necesitamos desarrollar la lógica de recogida y guardado de datos en InfluxDB. Telegraf soporta la recogida de datos de MQTT. Además de poder recoger los datos de MQTT que puede recoger datos de otras fuentes como por ejemplo HTTP, Docker, el propio Sistema operativo sobre el que está corriendo, información del servidor de InfluxDB, Kubernetes, AWS. Aparte de poder enviar los datos a InfluxDB, Telegraf también puede enviar los datos recogidos a otras aplicaciones y servicios que él mismo soporta a través de plugins, además de que se pueden desarrollar plugins de recogida y envío de datos, esto permite que Telegraf pueda ser utilizado de forma independiente. La lista de plugins de recogida de datos y salida de datos está en su Documentación. Al emplear la aplicación Telegraf

en el sistema esto permite eliminar carga a la aplicación principal al no implementar la lógica para recoger los datos desde MQTT en la aplicación principal. Otro beneficio es que existe una imagen Docker para la aplicación Telegraf y que podemos utilizar en el sistema.

Kapacitor, es otra de las aplicaciones que proporciona los desarrolladores de InfluxDB y se utiliza en el sistema. Es una herramienta para procesamiento de datos y alerta de eventos. Permite recoger y procesar un flujo de datos de InfluxDB en tiempo real o en grupos/lotes de datos e implementar una lógica para procesar y alertar basándose en los datos obtenidos. Esto permite que podamos implementar automatizaciones en base a los datos almacenados en InfluxDB y procesados por Kapacitor. Un ejemplo es el de apagar el enchufe inteligente si el sensor llega a un valor de potencia mayor al que debería. Kapacitor permite alertar por medio de diferentes servicios como por ejemplo, MQTT, HTTP, Telegram, Email. Además, Kapacitor también permite mandar los datos procesados de nuevo a InfluxDB y puede funcionar como alternativa a Telegraf aunque los desarrolladores recomiendan usar el propio Telegraf. En la figura 4 podemos comprobar cómo las aplicaciones de Telegraf y Kapacitor interactúan con InfluxDB y que junto a Chronograf, forman el TICK Stack.

4.3. Aplicación principal

Una vez diseñado el sistema físico, cómo interconectar los enchufes inteligentes y almacenar los datos de estos. Hay que diseñar la aplicación principal, es la que se encargará de gestionar toda la lógica necesaria para cumplir los objetivos propuestos en este trabajo. La aplicación principal se encargará de orquestar toda la lógica de comunicación con los dispositivos inteligentes.

La aplicación se ha desarrollado usando el lenguaje de programación Javascript, en concreto Typescript. Desarrollando una aplicación de Node.js. Aunque en el anteproyecto se indicó que se iba a desarrollar con Node-Red, se ha decidido desarrollar una aplicación de Node.js en Typescript puesto que permite más flexibilidad de desarrollo que Node-Red además, Node-Red en sí es una aplicación de Node.js que consiste en un servidor y herramientas para facilitar el desarrollo con dispositivos hardware Internet de las Cosas. Aunque debido a esto, el desarrollador tiene que estar sujeto a dichas herramientas para poder desarrollar. Con el fin de desarrollar una aplicación desde cero, ampliar los cono-

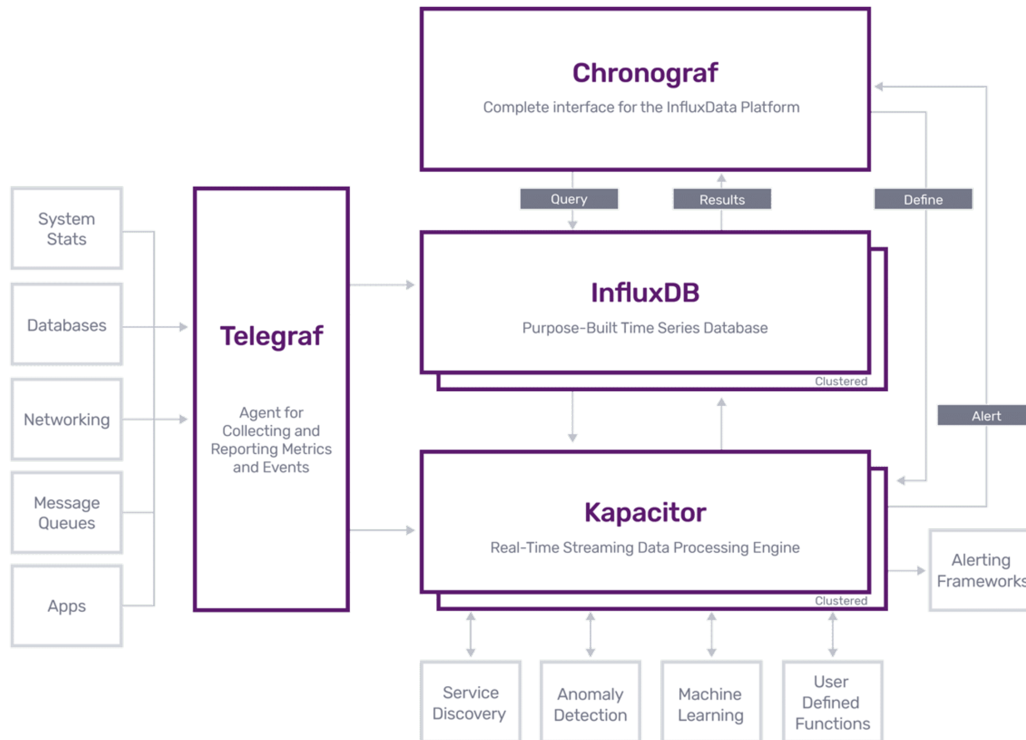

Figura 4: InfluxData [11] Diagrama de las aplicaciones del TICK Stack, describiendo cómo las aplicaciones interactúan entre sí

cimientos y de tener una flexibilidad de desarrollo, se ha decidido desarrollar en Node.js con Typescript/Javascript.

Javascript nació como un lenguaje scripting para navegadores web y debido a los avances que ha recibido, ahora se considera un lenguaje de programación en sí. Utilizado mayormente en servidores web y aplicaciones web. Además algunos proyectos de dispositivos embebidos permiten correr un subconjunto de Javascript en microcontroladores, un ejemplo de esto son proyectos como Espruino y Kinoma.JS. Otra de las ventajas que proporciona utilizar Javascript en Internet de las Cosas es que Javascript utiliza funciones asíncronas, las funciones asíncronas son ideales para programación dirigida por eventos en hardware, además Node.js, provee una manera de llamar a librerías desarrolladas en C o C++ e importarlas en Node.js y utilizarlas como un módulo de Node.js. De esta manera podemos usar apis nativas o integrar librerías de tercero implementadas en C/C++ en Node.js. Esto se puede conseguir con N-API [15]. Otro motivo por el que se utiliza Javascript es por la facilidad de programar en el, además, como utilizamos la Raspberry Pi como *gateway* existe suficiente capacidad de computación.

Node.js es un entorno de de ejecución de Javascript, este entorno corre sobre el motor

V8 de Google. Node.JS permite ejecutar Javascript fuera de un navegador web, además, el propósito de Node.js es el de dar un acceso a bajo nivel del event-loop de Javascript y a los recursos del sistema operativo. Node.js realiza operaciones de entrada/salida no bloqueantes empleando el event loop y con ello se consigue un paralelismo. Cada operación de entrada, salida tiene asociada un callback que se ejecuta una vez que la operación haya acabado [23]. Node.js a menudo es usado para programas en el lado del servidor aunque como se explicó anteriormente, Node.js también tiene su aplicación en el Internet de las Cosas. El hecho de utilizar Node.js es ideal porque es un entorno que puede usarse en dispositivos diferentes, en nuestro caso al usar una Raspberry Pi, Node.js puede instalarse fácilmente. Otra ventaja es que al utilizar Node.js y al ser un entorno que ejecuta Javascript, todas las aplicaciones del sistema se han desarrollado con Javascript, desde el servidor web del backend hasta el servidor web del fontend y el frontend mismo. Además Node.js contiene una cantidad enorme de librerías de terceros para todo tipo de acciones que se desean hacer, todas estas librerias estan alojadas en NPM.

Typescript es un superconjunto de Javascript, añade chequeo de tipado a Javascript para evitar posibles problemas en tiempo de ejecución, contiene varias adiciones como Interfaces y la posibilidad de utilizar nuevas mejoras de la especificación ECMAScript para el lenguaje Javascript sin preocuparse por la compatibilidad entre diferentes clientes web o entornos de Node.js. Es un lenguaje transpilado, es decir Typescript es compilado a Javascript, un lenguaje del mismo nivel de abstracción que Typescript. Todos los programas en Javascript son programas Typescript [22]. El uso de Typescript es ideal para proyectos de grandes envergaduras. La única desventaja es el hecho de que tiene que ser compilado primero a Javascript y que, para proyectos pequeños o script cortos, añadir todo el equipaje que se obtiene con Typescript no tiene sentido.

De los principales objetivos enunciados en el apartado 2.1, para poder cumplir esos objetivos debemos de plantear sus soluciones y cómo implementarlas en la aplicación a desarrollar.

Tanto el objetivo de **interconectar los dispositivos a un sistema de control de consumo energético** y el objetivo de **obtener información del consumo de energía de los dispositivos** están cubiertos tanto por la red WiFi y MQTT, lo único que faltaría implementar es el descubrimiento de los enchufes inteligentes presentes en el sistema. Aunque los enchufes inteligentes ya están conectados por MQTT, primero la aplicación

tiene que descubrir quiénes están conectados al Broker MQTT y después registrarlos en el sistema/aplicación. Debido a esto, la aplicación que desarrollemos tiene que ser capaz de escuchar los *topics* de MQTT de los enchufes inteligentes.

Para el objetivo de **proveer una interfaz de usuario para mostrar la información de los dispositivos a distancia**, debemos de desarrollar una interfaz gráfica para que un usuario/cliente pueda ver que enchufes inteligentes hay registrados, ver el estado de estos enchufes y comprobar los datos de los sensores. Se debe de implementar un cuadro de mandos que el cliente/usuario pueda acceder a distancia. Se ha decidido implementar una aplicación web en el que un servidor web de Node.js servirá a los clientes. De este modo los clientes pueden acceder al cuadro de mandos a distancia dentro de la red privada WiFi del cliente. Se ha implementado un *frontend* desarrollado en Javascript usando la librería React debido a que ya se tiene familiaridad con el desarrollo de aplicaciones web con React. El *frontend* se comunica con el *backend* a través de una API REST.

Para proveer una manera de **controlar el consumo de dichos dispositivos**, y **controlar los dispositivos de forma manual y automática**, la lógica de la aplicación debe de ser capaz de controlar los enchufes inteligentes, esto se realiza a través de MQTT, puesto que los enchufes propios proveen unos *topics* en el que estos escuchan y en el que un cliente MQTT puede enviar comandos para controlarlos. De esta manera podemos controlar uno o varios de los dispositivos de manera manual o automática. Para controlarlos de forma automática se han investigado varias manera para cumplir este objetivo, estas se detallan en el apartado 4.5 de automatizaciones del sistema.

Para el objetivo de **calcular el gasto del consumo generado por los dispositivos** se puede cumplir gracias a que los sensores de los enchufes inteligentes proveen el consumo energético de estos. Solo haría falta saber el precio de la energía eléctrica en determinado tiempo, esto se puede conseguir gracias a que la Red Eléctrica Española provee una API REST con la que consultar el Precio Voluntario al Pequeño Consumidor (PVPC), con ella podemos obtener el precio de la luz por horas. Esto naturalmente sólo funciona si el cliente tiene contratada una tarifa por de luz PVPC.

Por último, para el objetivo de **instalación y configuración relativamente sencilla**, esto significa que la aplicación y todo el sistema pueda instalarse de una forma sencilla. Para ello se ha decidido utilizar Docker y Docker-compose. Las aplicaciones estarán desplegadas como contenedores que utilizan imágenes de Docker de estas aplicaciones, de

manera que cualquier cliente pueda desplegar las aplicaciones desarrolladas con un solo comando o menos, sin necesidad de instalarlos directamente en su Raspberry Pi. Esto también supone otra ventaja, la de poder desplegar las otras aplicaciones necesarias en el sistema como la base de datos y el broker MQTT juntas a las aplicaciones desarrolladas. Esto se logra con otra aplicación de Docker, Docker-compose, con simplemente ejecutar un comando se puede desplegar toda la infraestructura software del sistema. Para tener una fácil configuración, el cliente solo deberá de editar algunas pocas cosas del fichero de configuraciones. En cuanto a configurar la propia aplicación del sistema, el *frontend* debe de proveer unos ajustes del sistema como variables de entorno o ajustes desde la aplicación web *frontend*.

Una vez definidas que soluciones tomar para cumplir los objetivos, dividimos la aplicación desarrollada en dos partes. El *backend*, encargado de interconectar los enchufes inteligentes, descubrir los enchufes y registrarlos, comunicarse con el *frontend*, controlar los dispositivos tanto manual y de forma automática utilizando la información obtenida a partir de la API REST de la Red Eléctrica Española. El *frontend*, una aplicación web encargada de proveer una interfaz gráfica al cliente, además que se comunicara con el *backend*, permitirá la posibilidad de que el cliente pueda ver el estado del sistema, interactuar con los enchufes inteligentes y obtener una visualización de los datos obtenidos por los sensores de los enchufes inteligentes. Tanto *backend* y *frontend* están desplegados en la Raspberry Pi junto con las otras aplicaciones necesarias del sistema.

Como la aplicación está separada entre el servidor web *backend* y *frontend*, se ha decidido implementar una API REST servida por el *backend*, de modo que el *frontend* pueda consumir esta API REST. Una ventaja de esto es el hecho de que otro desarrollador pueda crear su propia aplicación web que consuma esta API REST o directamente no utilizar una aplicación gráfica e interactuar con el *backend* solo por medio de peticiones HTTP.

Existen tres componentes del sistema que se han desarrollado, el *backend* el *frontend* y el *dashboard*. *Backend* es el corazón principal de la aplicación, interactúa con todos los componentes del sistema. El *frontend* él es encargado de proporcionar la interfaz gráfica a un cliente y el *dashboard* es un *backend* solo dedicado a el cuadro de mandos que se muestra en el *frontend*, esto se detalla más en el apartado 4.8.

4.4. Descubrimiento y registro de enchufes inteligentes

Como se ha mencionado anteriormente en el apartado 4.3, el *backend* debe de ser descubrir que enchufes inteligentes están conectados al broker MQTT del sistema, a continuación se explica cómo se realiza esto en el backend.

Se parte desde un escenario en el que se supone que el enchufe inteligente a descubrir y registrar ya está conectado a el broker MQTT. La configuracion de conexion de los enchufes inteligentes a el broker MQTT, se abarca en el apéndice 6.1.2. Según la documentación de ESPurna, los enchufes inteligentes publican información sobre sí mismos y sobre su estado en los *topic* de *Heartbeat*, donde publican información como por ejemplo: la Ip, el hostname, la mac, la versión del firmware de ESPURNA. Lo que nos interesa es el *topic* ***/mac***. Este *topic* es el que se encarga de publicar la información de la mac de el enchufe inteligente. Esta información es única y sirve como identificador de cada enchufe inteligente guardado en la base de datos. El *backend* proporciona un método de descubrimiento de enchufes inteligentes. La lógica del descubrimiento es recoger los datos de los enchufes que hay conectados en el broker y devolverlos. Otro método que provee el backend es para registrar un enchufe inteligente. El *backend* debe de escuchar los *topics* *Heartbeat* sea cual sea de cualquier enchufe inteligente conectado al broker, como cada enchufe inteligente publica su mac en un *topic* del estilo ***{root topic}/mac***, para ello el *backend* escuchará el *topic* ***+/mac***.

Para el método de *discovery* escuchara mensajes durante un determinado tiempo, los recogerá y devolverá al cliente.

Una vez que el cliente tenga los datos de los enchufes descubiertos, deberá utilizar el método de registro de un enchufe, el cliente debe de indicar los metadatos del enchufe a registrar, pueden ser los que previamente ha devuelto tras haber hecho un descubrimiento de los enchufes conectados al Broker o si el cliente sabe los metadatos, los puede proporcionar de forma manual. El *backend* procederá a comprobar si dicho enchufe inteligente está ya registrado en la base de datos. Si no lo está, procederá a comprobar si existe un enchufe inteligente conectado al broker que utilice la misma mac. Si el enchufe inteligente existe y está conectado al broker MQTT, lo registrará en la base de datos, modificará la configuración de Telegraf para que escuche los *topics* con el *root topic* correspondiente al *mac* de el enchufe inteligente para que de este modo Telegraf recoge los mensajes que el enchufe inteligente publique en los *topics* de sus sensores, tras esto, reiniciará el contene-

dor de Telegraf. Podemos observar el diagrama de secuencia de tanto de los métodos de scan como el register. Ambos son llamadas a la API REST que el *backend* proporciona.

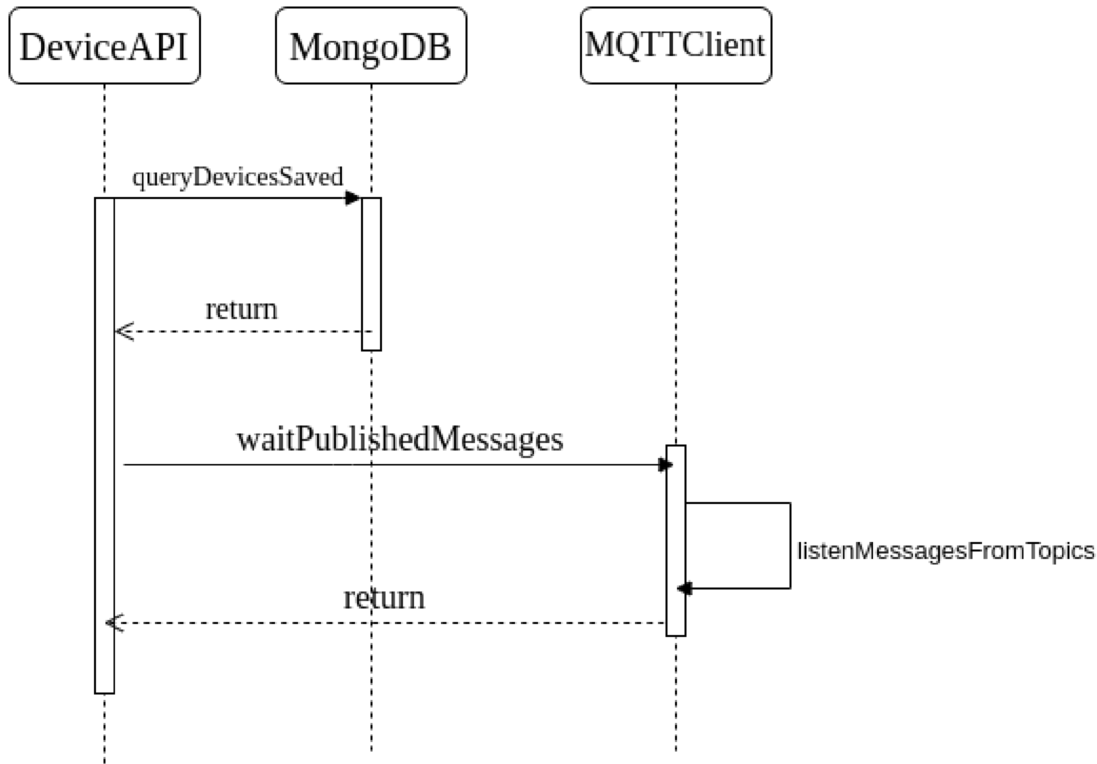

Figura 5: Diagrama de secuencia de el escaneo de enchufes inteligentes

4.5. Automatizaciones del sistema

Uno de los objetivos principales de la aplicación es la de proporcionar una manera de automatizar el uso de los enchufes inteligentes asimismo también poder controlar de forma manual los enchufes inteligentes. Los enchufes inteligentes poseen un relé/*switch* del que pueden cortar o permitir el paso de la corriente eléctrica hacia los dispositivos conectados al enchufe inteligente. ESPurna provee una manera de controlar estos relés, además también se pueden controlar por medio de *topics* de MQTT. Proporciona dos *topics*, uno para ver el estado del relé y otro para cambiar su estado. Ambos *topics* son **{root topic}/relay/0** y **{root topic}/relay/0/set** respectivamente. Para controlar de forma manual basta con enviar un mensaje con el contenido 0 o 1 al *topic* **{root topic}/relay/0/set** para encender o apagar el enchufe inteligente. De esta manera el backend solo tiene que enviar por MQTT esos contenidos a los *topics* correspondiente de un enchufe inteligente. Gracias a eso podemos manejar los enchufes inteligentes de forma manual. El cliente solamente tendrá que interactuar con el frontend y el frontend enviará

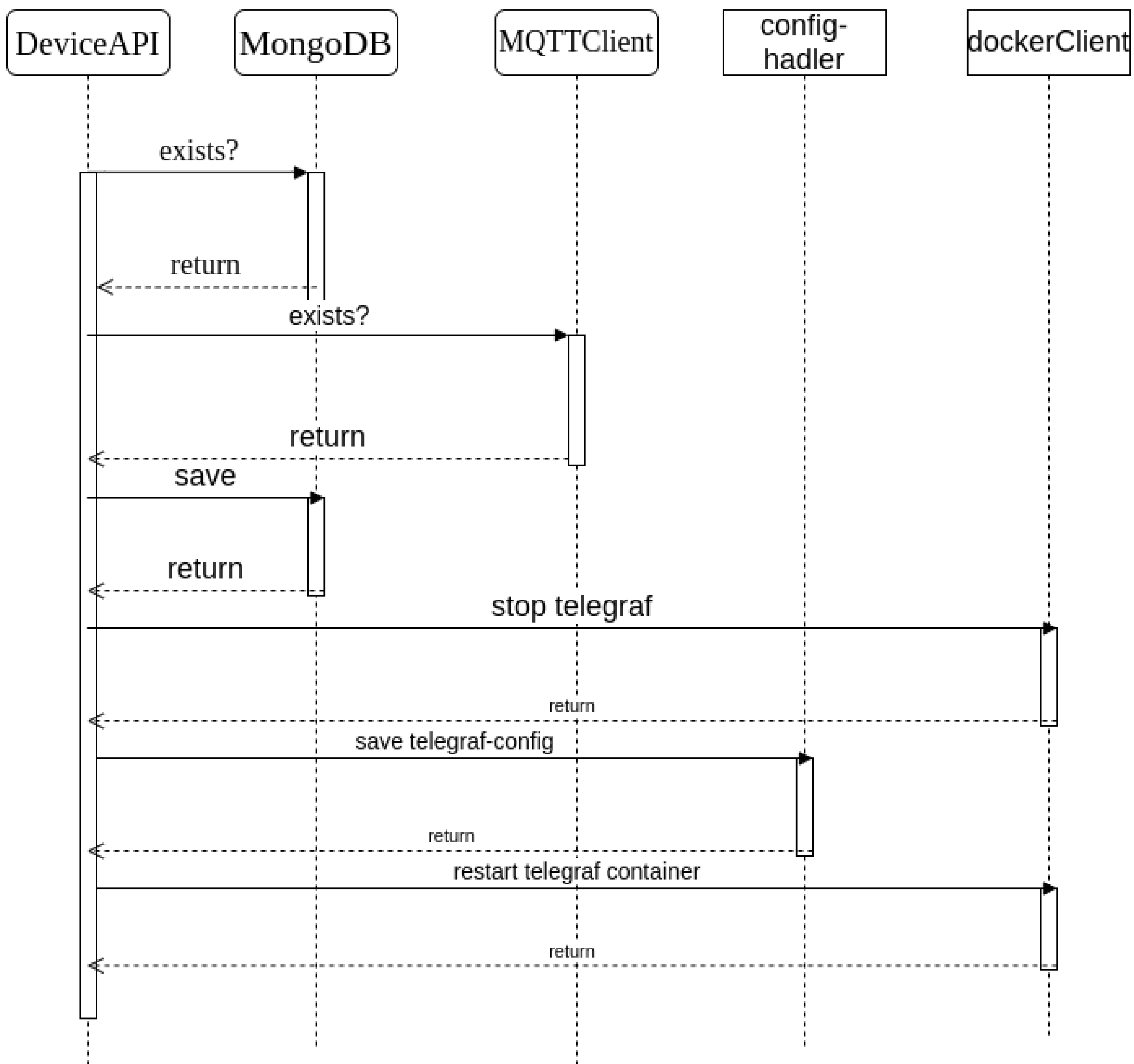

Figura 6: Diagrama de secuencia de el registro de enchufes inteligentes

una petición a el backend utilizando la API REST, después a su vez el backend enviaría un 0 o 1 a el *topic* correspondiente de el enchufe inteligente.

Para crear automatizaciones se ha propuesto una serie de posibles casos de automatización. Los posibles casos son:

- Automatización por tiempo.
- Automatización según los valores de los sensores de el enchufe inteligente.
- Automatización por precio de la electricidad.
- Automatización por precio gastado.

Una automatización está compuesta por unos **triggers** y **actions**. Las *actions* son una lista de operaciones que realizarán los enchufes inteligentes conectados al sistema cuando una automatización es disparada. Las acciones son apagar o encender el enchufe inteligente, es decir cambiar el estado del relé del enchufe inteligente. Los triggers son disparadores que indican cuando una automatización debe de ejecutarse. Cada automatizacion puede entrar en estado de *scheduled*, es decir se programa la automatización para que en determinadas circunstancias, la automatización se dispare por los triggers y por tanto este ejecute las acciones que tiene asociadas.

La única gran diferencia que existe entre cada automatizacion son los diferentes tipos de *triggers* que pueden haber definidos en una automatización. Siguiendo los posibles casos de automatización listados anteriormente se ha implementado una lógica para cada una de estas automatizaciones.

4.5.1. Automatizaciones por tiempo

Las automatizaciones por tiempo son las más sencillas puesto como su nombre indica, en determinado tiempo disparan una automatización. La manera de disparar la automatización depende de la hora en la que se quiere disparar y los días en el que se quieren disparar. Se ha desarrollado un componente denominado Scheduler, un componente encargado de encolar Schedules. Los Schedules son objetos que utilizan la función *setTimeout()* de Javascript y saltan tras un determinado de tiempo pasado, cada Schedule tiene una función de callback asociada a ese *setTimeout()*. Cuando una automatización por tiempo es es programada, se crea un Schedule y se encola al Scheduler, el Scheduler calcula cuánto tiempo en milisegundos tiene que pasar desde ahora hasta el tiempo determinado para que se dispare la automatización y se establece el temporizador de *setTimeout()*. Cuando *setTimeout()* salte, llamará a su función callback asociada, esta función callback se encarga de ejecutar las acciones de la automatización por tiempo. El diagrama de clases de estas dos clases se puede comprobar en la figura 7

Los triggers que disparan la automatización por tiempo los hemos denominado como *timerTriggers*.

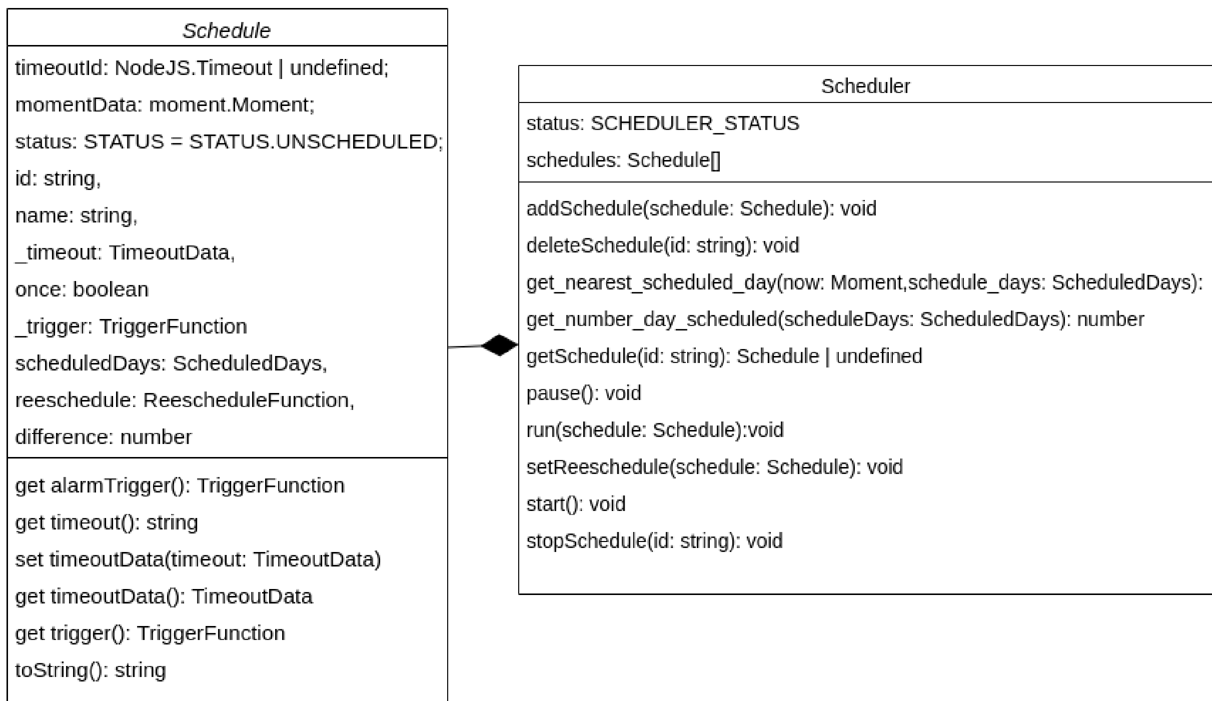

Figura 7: Diagrama de clases de Schedule y Scheduler

4.5.2. Automatización según los valores de los sensores del enchufe inteligente

Estas automatizaciones definen un trigger que se dispara la automatización si cierto valor de los sensores de los enchufes inteligentes es mayor o menor a un valor definido por el cliente. Los parámetros de los sensores de los enchufes inteligentes son:

- current
- voltage
- power
- apparent
- reactive
- factor
- energy

Este tipo de triggers los llamamos *influxTrigger*. El cliente podrá indicar cuando una automatización se dispara dependiendo del valor de estos parámetros. Esto implica que el *backend* debe de estar atento a los cambios de estos valores. Aunque esto se soluciona

con el uso de Kapacitor, una aplicación que hemos introducido en el apartado 4.2. Esta aplicación permite crear **tasks** que disparan alertas según qué valores hay dentro de una *measure* de la base de datos de InfluxDB. Estos tasks se definen usando un script de lenguaje de dominio específico de Kapacitor, denominados TICKscript. Se ha definido un *template* para crear *tasks* que comprueban los valores de una measure de InfluxDB y disparan una petición HTTP POST al *backend* que a su vez el backend, cuando maneje dicho POST, se encargará de disparar la automatización correspondiente. El backend crea los tasks a partir de la plantilla tick script desarrollada, contacta con Kapacitor a partir de peticiones HTTP usando la API REST de Kapacitor. En la figura 8 podemos ver las clases de *influxTrigger* relacionado con la clase de *Kapacitor*. La clase *Kapacitor* es la que se encarga de realizar las peticiones HTTP a Kapacitor mientras que la clase *influxTrigger*, llamará a estos métodos proveídos por la clase *Kapacitor*.

En la figura 9 podemos ver cual es el flujo de acciones que se realiza en la aplicación una vez que desde Kapacitor se haya disparado una alerta de un task, se puede observar que la task de Kapacitor hace un POST a la API REST del backend a través de la URI *:idfire*. El backend al manejar este POST, utiliza el método de fireActions de la clase *automatizationDAL*, que es la encargada de ejecutar las acciones de una automatización y enviar los comandos a los enchufes inteligentes correspondientes.

Kapacitor permite definir *tasks*, que son un conjunto de acciones que debe de hacer Kapacitor para extraer información de InfluxDB, procesarla o comprobar cambios en la información obtenida desde InfluxDB la detección del cambio o como procesar la información lo define un tickscript, un script de lenguaje específico de dominio. El tickscript selecciona qué y cómo recibir la información de Influxdb y a partir de ahí alertará de los cambios o procesa los datos y generar otros. El principal objetivo por el que se usa los task de Kapacitor son las alertas. Una alerta puede ser manejada de diferentes maneras por ejemplo enviando un mensaje por email, por slack, o por mensaje de telegram. En este caso el manejo de la alerta hace que envíe un POST a la API REST del backend como habíamos mencionado anteriormente. Las task templates son plantillas para crear tasks de Kapacitor. A la hora de crear un influxTrigger el backend crear un nuevo task utilizando el task template que se ha desarrollado. En el apéndice 6.2.3 se puede comprobar como se ha desarrollado dicho task template y cómo funciona. En la figura 10 podemos observar el grafo de cómo fluyen los datos en un task asociado a un influxTrigger, este grafo se

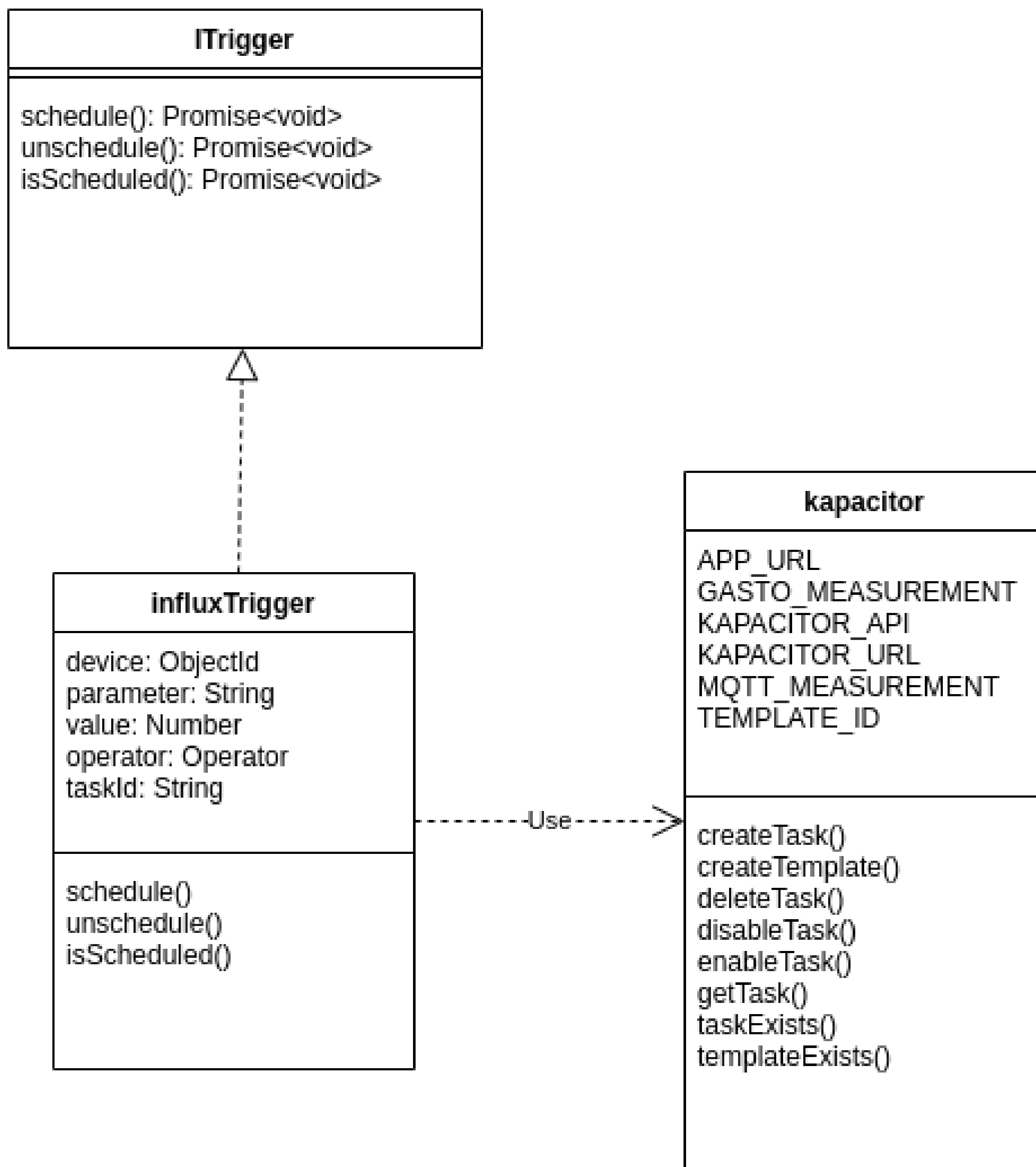

Figura 8: Diagrama de clases de la interfaz ITrigger y las clases influxTrigger y Kapacitor

explica con más claridad en el apartado 6.2.3.

4.5.3. Automatización por precio de la electricidad

Gracias a que la Red Eléctrica Española nos proporciona el valor del PVPC, podemos saber el precio de la energía eléctrica cada franja horaria del día. Existen tres tarifas.

- Sin discriminacion horaria (2.0A) **(GEN)**

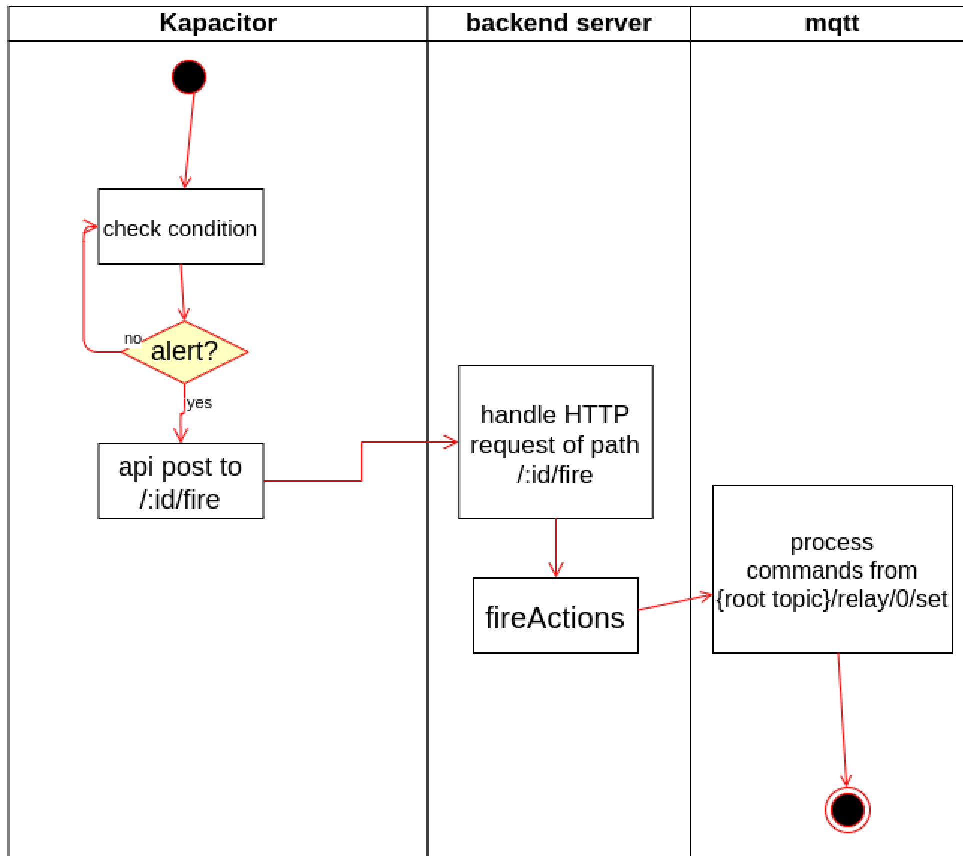

Figura 9: Diagrama flujo de acciones e interacciones entre Kapacitor, el *backend* y los enchufes inteligentes una vez que se ha disparado una alerta desde Kapacitor

- Con discriminación horaria de dos periodos (2.0DH) **(NOC)**

- Con discriminación horaria supervalle (2.0DHS) **(VHC)**

En función de estas tarifas, se puede saber a qué hora costará menos o costará más la energía eléctrica. Y hemos llamado timerPriceTrigger a los triggers que dependen de cuando el coste de la energía eléctrica sea baja o alta. Se ha reutilizado la lógica ya implementada en las clases de *Scheduler* y *Schedule*. Cuanto una automatización con un timerPriceTrigger es programada, se crea un *Schedule* para el día deseado y la hora deseada, así tras dispararse el Schedule, llamará al método de *fireActions* de la automatization y realizará las acciones correspondientes para encender o apagar los enchufes inteligentes.

En el diagrama de secuencias 11 podemos observar la secuencia de acciones que se desencadenan una vez que se programa una automatización con un trigger de tipo timerPriceTrigger. Y el diagrama 12 muestra la cadena de acciones que se dan una vez un Schedule dentro del Scheduler es disparado.

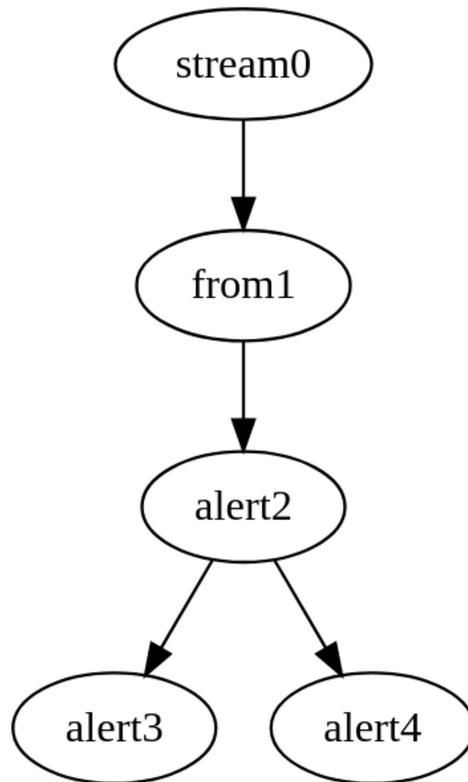

Figura 10: Grafo de flujo de los datos del task template desarrollado

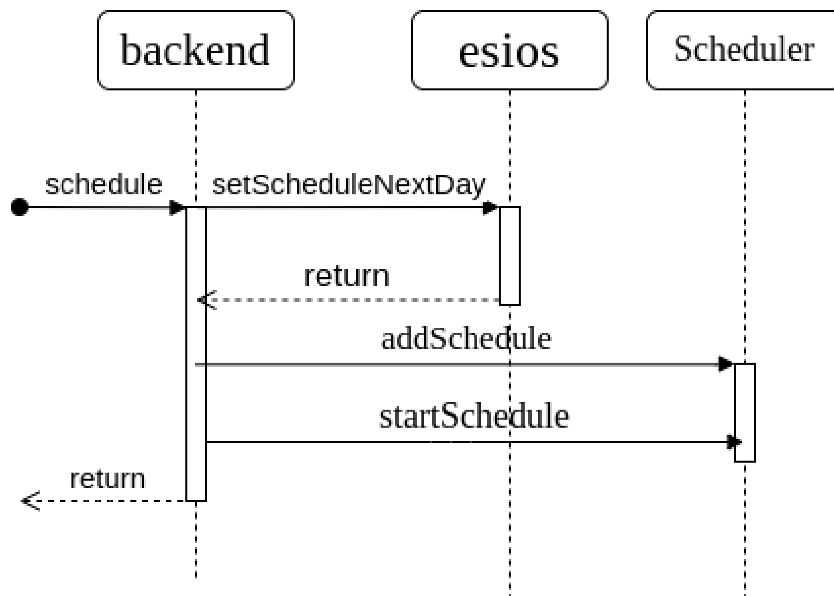

Figura 11: Diagrama de secuencia sobre cómo el backend utiliza la información de la API de la Red Eléctrica Española y crea un Schedule que despues inicia en el Scheduler

4.5.4. Automatización por precio gastado

Otro tipo de automatización que se ha desarrollado depende de el precio gastado por parte de un enchufe inteligente, según si el valor del gasto es mayor o menor disparamos

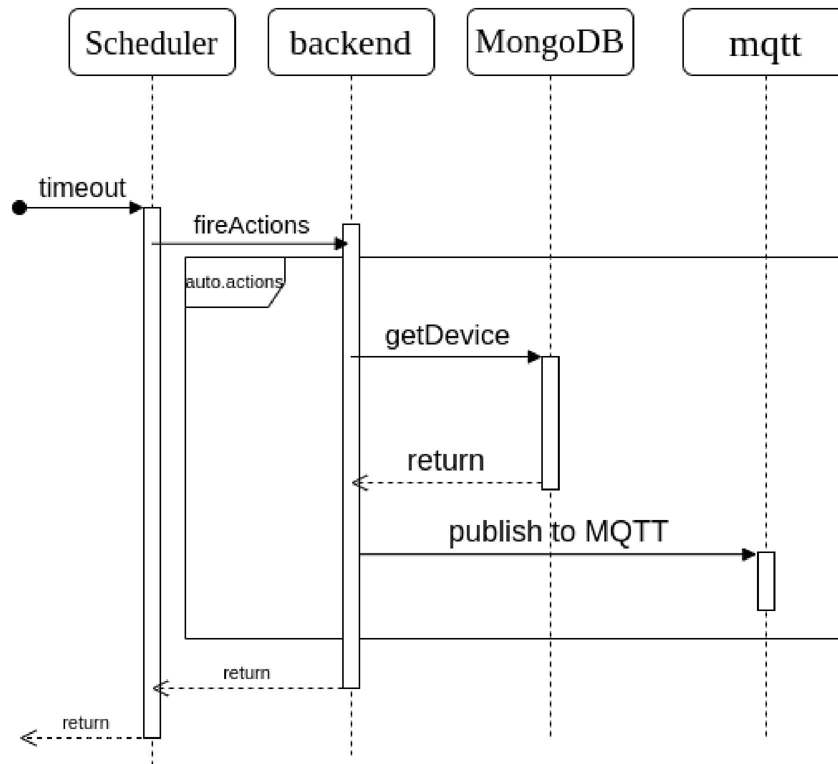

Figura 12: Diagrama de secuencia de las acciones que se realizan una vez que un Schedule se dispara y ejecuta las actions asociadas a una automatización

la automatización o no. Estos triggers son denominados como priceTrigger. Reutilizan la lógica de los influxTriggers además también utilizan la API del PVPC de la red eléctrica Española. Cómo a partir de la API del PVPC sabemos el coste de la energía eléctrica en cada franja y como los sensores de los enchufes inteligentes devuelven los datos de la energía consumida, podemos calcular el precio de la energía consumida por enchufe inteligente en cada franja horaria y guardarla en InfluxDB en un *measurement* denominado *device_gasto*. De este modo podemos crear un task de Kapacitor que esté atento a los valores de *measure* que tengan un *tag* de valor correspondiente al *deviceId*, al id del enchufe electrónico guardado en la base de datos de MongoDB. A partir de sigue la misma lógica que con los influxTriggers, alertará cuando la condiciones sean verdaderas en el task de Kapacitor.

Hay que tener en cuenta que debe de haber un componente que cumpla con la función de estar continuamente calculando el precio gastado y guardarlo a InfluxDB. Para ello se ha desarrollado un componente denominado *PriceCalculatorPooler*. Este componente se encarga de calcular el precio gastado de cada enchufe inteligente registrado en la aplicación. Cada hora realizara una petición a la red Eléctrica Española para obtener el precio

y por cada enchufe inteligente consultará los datos de sus sensores guardados en InfluxDB de la hora anterior, en concreto la consulta consiste en que devuelva los datos del topic *energy*. Con el precio y el valor del topic energy, calculará el precio gastado y luego lo guardará en InfluxDB de nuevo.

Como el *topic* ***{root topic}/energy*** contiene la energía agregada en kilovatios hora (kWh) y el precio de la energía eléctrica obtenida desde la API de la red eléctrica es de euros entre Megavatios por hora basta con obtener lo consumido entre la franja horaria y multiplicarlo con el precio de dicha franja horaria, es decir el cálculo del gasto de la energía consumida sería el de la ecuación 1.

$$\text{Gasto}\,€ = (\text{energy}_{\text{max}}\,\text{kW h} - \text{energy}_{\text{min}}\,\text{kW h}) \times \text{coste}€/\,\text{kW h} \tag{1}$$

Una vez que ya tenemos el gasto por hora de cada enchufe inteligente guardado en InfluxDB, lo único que tendrá que hacer el cliente es crear una automatización que tenga un influxTrigger que esté atento al valor del measurement *price* de determinado enchufe inteligente y este trigger se disparará siguiendo la misma lógica que un influxTrigger. El *PriceCalculatorPooler* es instanciado al iniciar el servidor *backend*, en la figura 13 podemos ver el diagrama de secuencia sobre las acciones que realiza el *PriceCalculatorPooler* a determinado intervalo de tiempo ejecutando el método *intervalHandler* de su clase.

4.6. Diseño de los modelos de datos

La aplicación necesita guardar datos de forma persistente sobre los enchufes inteligentes registrados, las automatizaciones, la información sobre cómo presentar las gráficas sobre los datos de los sensores y los ajustes propios de la aplicación y el frontend. Se ha diseñado el modelo de datos para estos recursos y utilizado la librería Mongoose, una librería ODM para Nodejs. Mongoose proporciona la ventaja de poder validar los documentos de MongoDB proporciona una asociación de documentos a objetos de javascript.

4.6.1. Devices y DevicesGroup

El modelo de datos para los enchufes inteligentes registrados en la aplicación se llama *Device*, se ha decidido este nombre puesto que en un futuro la aplicación podría mejorarse a poder soportar otros dispositivos que no sean enchufes inteligentes pero que usen el

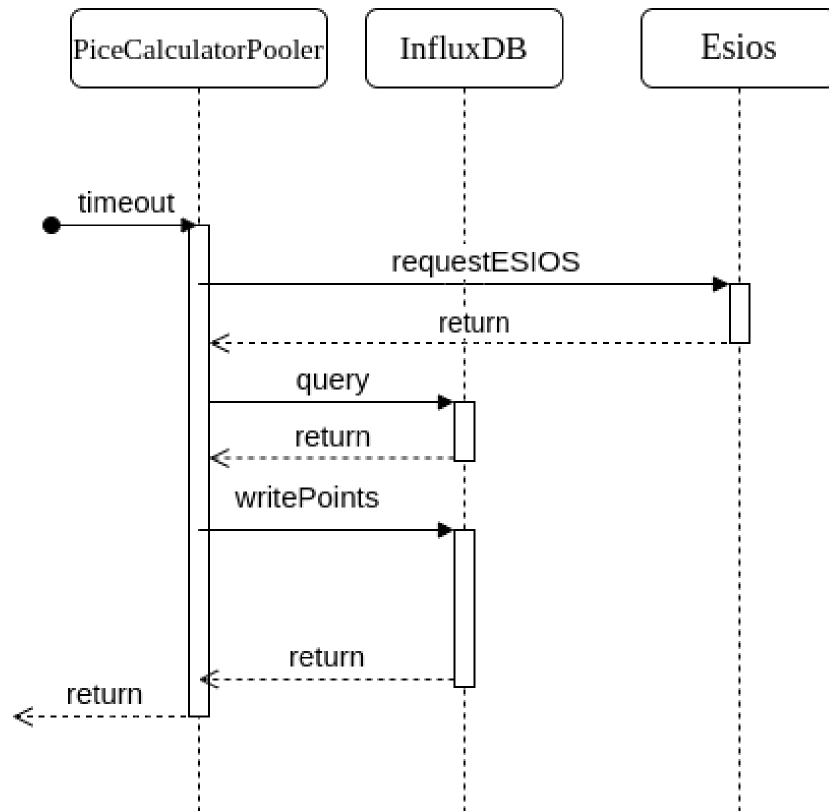

Figura 13: Diagrama de secuencia de acciones e interacciones entre el *PriceCalculatorPooler*, la clase de *esios* y InfluxDB cada vez que el pooler ejecuta el método intervalHandler

mismo firmware de ESPurna. El modelo de datos contiene una *field* llamada type, sirve para distinguir el tipo de dispositivo que es. En este caso como solo usamos enchufes inteligentes solo puede haber un tipo: **SMARTPLUG**.

Un Device tambien tiene como *field* la información de los *topics* de *Heartbeat* de MQTT como mac, host, app, version. También guarda como *field* el estado del dispositivo. El estado del dispositivo puede ser **ON** o **OFF**. Otra field importantes que contiene un Device son las *actions*, las actions son una la lista de acciones que puede realizar el dispositivo, en este caso solo podemos encender (**ON**), apagar (**OFF**) y alternar el estado (**TOGGLE**) de un dispositivo.

Aparte de estos fields, Mongoose también proporciona una manera de tener métodos para una instancia de un documento de MongoDB, esto permite añadir lógica y métodos propios a un documento de Mongoose. Un método de instancia a destacar es el de *fireAction*, que recibe una acción y envía el dato correspondiente a el *topic* correspondiente de MQTT en el que el enchufe inteligente o dispositivo escucha para realizar dicha acción.

Se ha diseñado otro modelo de datos para crear agrupaciones de Devices, este modelo

de datos se llama *DevicesGroup*, que solamente contiene como *fields* un nombre y una lista de referencias a Devices. Este modelo de datos permite dar una facilidad para manipular varios dispositivos en una automatización o facilidad para crear gráficas sobre los datos de sensores de varios dispositivos. Podemos comprobar en la figura 14 el modelo de datos de Device y sus métodos junto con la relación de devicesGroups.

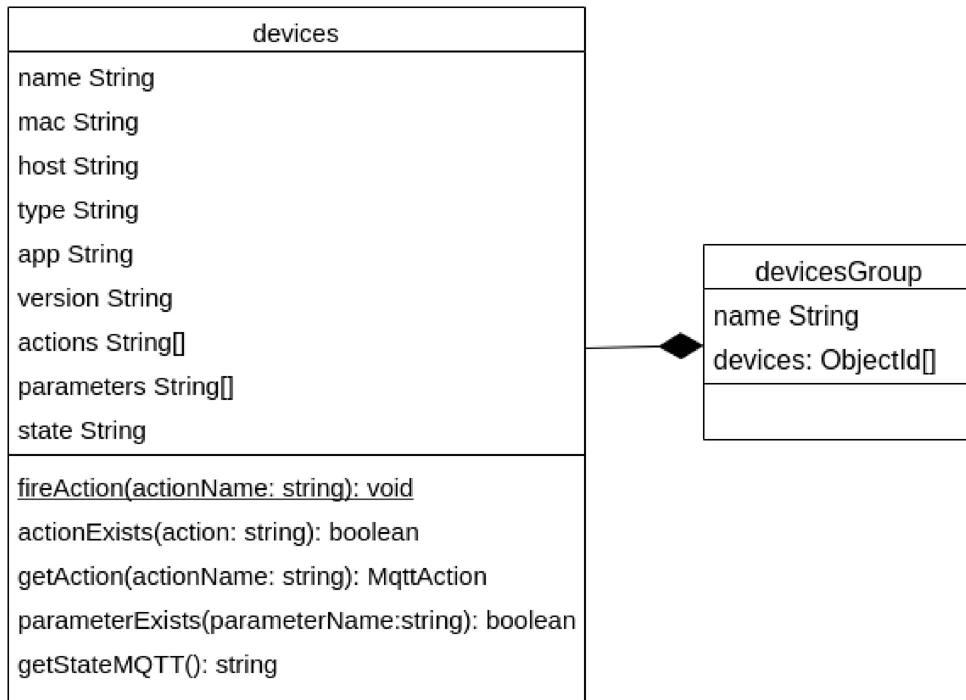

Figura 14: Diagrama de clases de devices y devicesGroup

Los devices también tienen un *field* llamado *parameters*, este *field* consiste en una lista de strings. Esta lista contiene el nombre de los *topics* de los sensores de un enchufe inteligente que Telegraf debe de escuchar para obtener los datos recibidos en ese *topic* y guardarlos en InfluxDB. La tabla 2 es una tabla de correspondencia entre sensor y topic muestra que lo único que hay de diferencia entre ellos es el {root topic}. La tabla es algo similar a la que se muestra en la documentación de MQTT de ESPurna, se ha añadido otro parámetro que se tiene en cuenta en la aplicación y es el de *price*. Este es un parámetro especial puesto que no corresponde con un sensor y topic del enchufe inteligente sino que corresponde con el precio guardado en InfluxDB, se ha añadido este parámetro para hacer más conveniente la manera de seleccionar influxtriggers a partir del precio gastado por un enchufe inteligente.

Topic de ESPurna	Parametro en el *field* de devices	Valor de ejemplo	Explicación
{root topic}/current	current	0.35	Corriente, tension en Amperios
{root topic}/voltage	voltage	227	Voltaje, en V
{root topic}/power	power	430	Energía activa en W
{root topic}/apparent	apparent	320	Energía aparente en W
{root topic}/reactive	reactive	100	Energía reactiva en W
{root topic}/factor	factor	95	Factor de energía en %
{root topic}/energy	energy	253654	Suma total de la energía consumida en kWh
*	price	253654	Precio de la energía gastada en una franja horaria, en euros

Cuadro 2: Comparacion de *topics* de ESPurna con *parameters* de devices en la aplicación

4.6.2. Automatizations y triggers

La aplicación tiene que guardar también datos sobre las automatizaciones creadas, para ello se diseña el modelo de datos de una automatización. Este modelo de datos es el más complejo de entre todos. Ya sabemos que una automatización debe de contener unas acciones y unos triggers. El *field* de actions es una lista de subdocumentos que contienen el dispositivo y la acción a realizar sobre ese dispositivo. El *field* de scheduled sirve para comprobar si se ha programado o no la automatización. Los fields de timerTriggers, influxTriggers y timerPriceTriggers son subdocumentos con los datos necesarios para los tipos de triggers definidos anteriormente en el apartado 4.5. Los subdocumentos de trigger implementan la interfaz de *ITrigger*, que obligan a implementar los métodos de instancia

- schedule
- unschedule
- isScheduled

Estos tres métodos también los implementa el propio Modelo de datos de automatization, en un documento de automatization cuando se ejecuta uno de estos métodos de instancia,

llama a los métodos de instancia correspondientes de cada trigger. Un método de instancia a destacar es el de fireActions, se llama igual que el método de instancia de un device solo que este método por cada actions guardado en una automatización, llama al método fireAction correspondiente a el device asociado a la acción guardada en la automatización Una automatización tiene también un *field* llamado *repetitive*, permite indicar si la automatización se vuelve a repetir una vez que ya se haya disparado y ejecutado las acciones que tiene asociada. En la siguiente figura 15 podemos apreciar el diagrama de clases de automatización y los triggers.

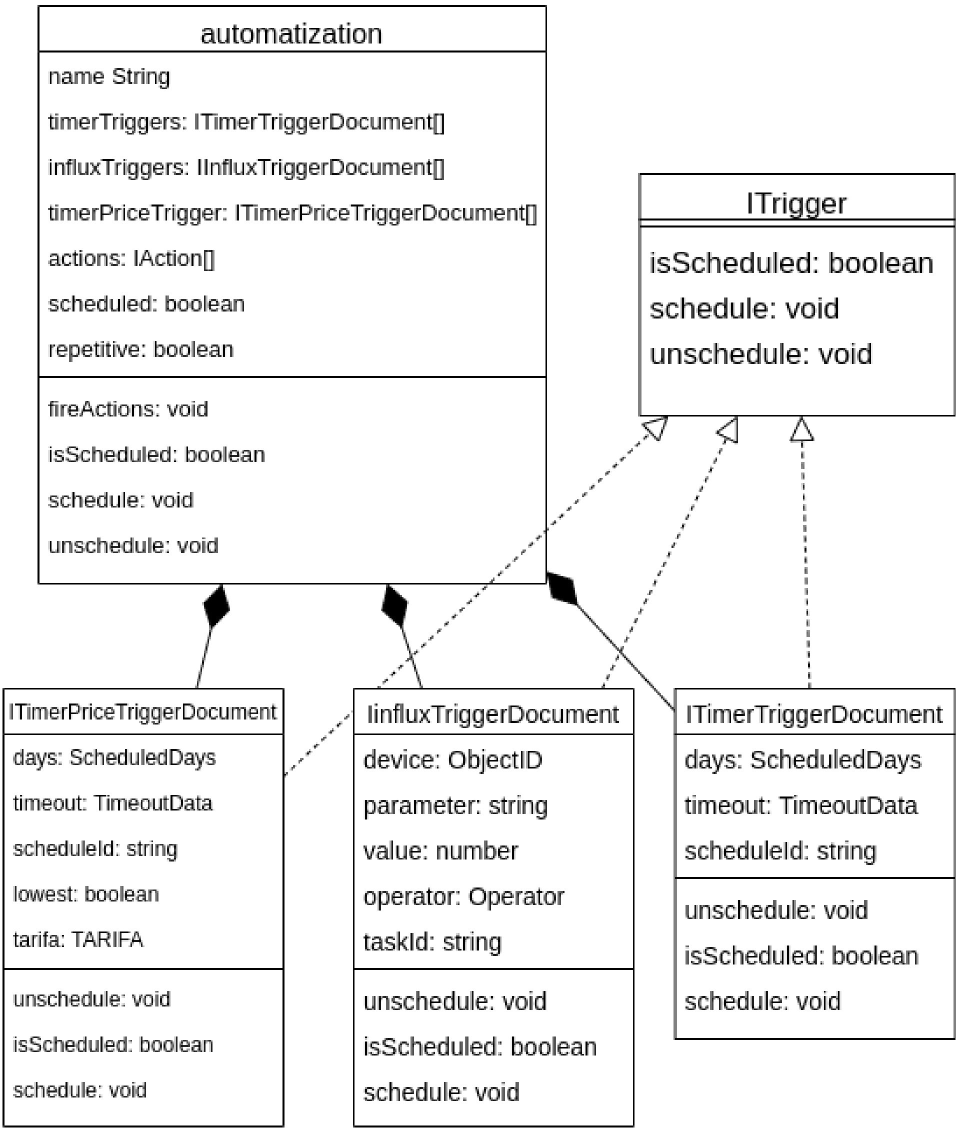

Figura 15: Diagrama de clases de automatization y triggers

4.6.3. Dashboards

Uno de los objetivos de la aplicación es la de proporcionar la vista a los valores de los sensores, dichos valores se mostrarán en forma de gráficas. Esto conlleva a también definir el modelo de datos para una *Dashboard*, gráficas donde mostrar esos datos. Más adelante se explica con más detalle la lógica de un Dashboard en el apartado 4.8.1. Se ha definido dos tipos de *dashboards*, usan gráficas de tipo *AREA*, *LINE* y *BAR*. Cada *dashboard* tiene una lista de subdocumentos que contienen de *field* un *device* y el *parameter* que se quiera mostrar en la gráfica. En la figura 16 podemos ver la definición de su clase.

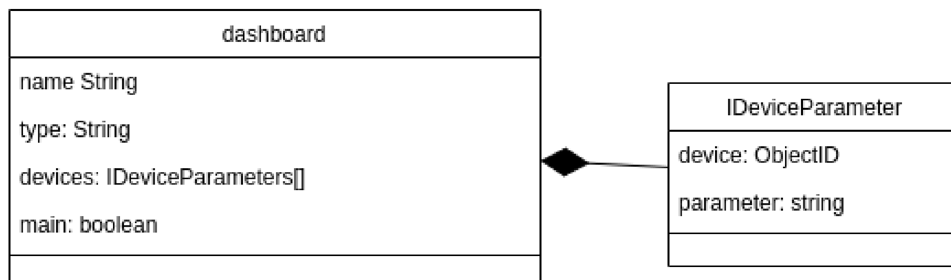

Figura 16: Diagrama de clases de dashboard

4.6.4. Site-Settings

La aplicación necesita guardar de forma persistente el token necesario para realizar las peticiones a la API de la Red Eléctrica Española para obtener los precios de PVPC y también guarda el tiempo de *pooling* de los estados de los enchufes inteligentes, las automatizaciones y la tarifa usada para calcular el precio del gasto de la energía eléctrica de cada uno de los dispositivos registrados en el sistema. Todos estos datos se guardan en un único documento llamado Site-Settings, se ha implementado la lógica para que solo haya un documento de Site-Settings.

4.7. Diseño de una REST API para la estación base

Como se mencionó antes en el apartado 4.3, se ha decidido implementar el *backend* como un servidor web que provee una API REST, esto supone que el servidor del *backend* puede estar desacoplado de el *fronted* e incluso podemos interactuar con el sistema sin llegar a dar uso del *frontend*. Incluso el propio cliente podría desarrollar su propia interfaz gráfica.

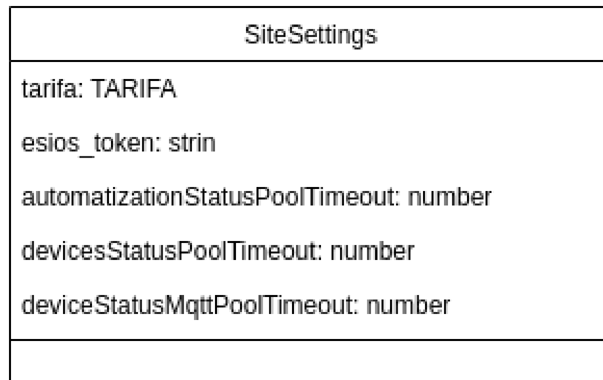

Figura 17: Diagrama de clases de de los ajustes de la aplicación

Una API no es nada más que una definición de cómo una pieza de software se puede comunicarse con otra pieza de software, define qué formatos de datos y protocolos se utilizan para que dos piezas de software se comunican entre ellos [4]. «REST es un estilo de arquitectura que sirve de ayuda en el proceso de crear y organizar sistemas distribuidos»[7] REST implica que la manera de comunicarse entre dos piezas de software ha de ser a través de peticiones de HTTP, la arquitectura define que debe de haber recursos, una manera de actuar con esos recursos, una manera de acceder a ellos y de obtener una representación de esos recursos. Los recursos corresponden con parte del modelo de datos que se ha explicado anteriormente en el apartado 4.6:

- *devices y devicesGroup*
- *automatizations*
- *dashboards*
- *site-settings*

La representación de estos recursos es a través de documentos JSON, es posible representarlos en diferentes formatos como en el formato XML pero se ha elegido JSON puesto que encaja bien con el hecho de que el almacenamiento de los datos de los recursos se realicen como documentos JSON en la base de datos MongoDB.

Para acceder a estos recursos, se obtienen a traves de peticiones HTTP hacia una URI (unique resource identifier), las URIS no son nada más que un identificador para acceder a un recurso en concreto. En el caso de la API REST diseñada, los nombres de las colecciones en la base de datos de MongoDB son las URI para acceder a recursos de

dicho tipo, luego se combinan con el ID de cada documento para solo acceder a un recurso en concreto.

Para interactuar con los recursos, modificar el estado de estos, añadir otros recursos y eliminar los recursos se realiza a partir de los métodos de HTTP. Los métodos *GET*, *PATCH*, *DELETE* son los verbos HTTP usados por la API REST. Estos métodos se utilizan tanto para obtener un recurso, modificarlo y eliminarlo, esto encaja con el término CRUD (create, retrieve, update, and delete) y con la manera de manejar documentos dentro de una base de datos. Los documentos se guardan en una colección y posteriormente se pueden obtener, modificar o eliminar de la colección.

Para servir la API REST desde el *backend* se ha utilizado el framework de aplicaciones web, express.js. Express es un framework minimalista y muy popular. Se ha definido las diferentes rutas de la API REST y utilizando la lógica de middlewares para manejar peticiones HTTP.

Cada componente del sistema tiene un fichero .js en el que se implementan las rutas de GET, POST Y PATCH para dichos recursos y donde dentro de cada una de ellas, se interactúa con la base de datos de MongoDB para modificar sus datos, es decir el estado de el componente. Cabe destacar que se ha tenido que romper algunas recomendaciones de diseño de API REST debido a algunos inconvenientes presentados a la hora de desarrollar el sistema.

4.8. Diseño de la interfaz de usuario y comprobación del estado del sistema

El *Frontend* es una aplicación web que permite al usuario interactuar con el sistema desde su navegador web. El *frontend* está desarrollado con ReactJS, un librería de javascript que permite desarrollar interfaces de usuario, se ha utilizado más en concreto Next.js, un framework para desarrollar aplicaciones web con ReactJS. Next.js permite desarrollar aplicaciones ReactJS renderizadas en el lado del servidor y desarrollar páginas web estáticas. Se ha elegido este framework puesto que ya directamente provee el servidor web para servir la aplicación web desarrollada en ReactJS, la misma aplicación podría haberse servido desde el propio servidor web del *backend* del sistema pero esto no sería ideal puesto que no tendríamos un sistemas desacoplado entre *backend* y *frontend*. El *frontend* se comunica con el *backend* a través de la API REST que sirve. El diseño se ha

realizado con ayuda de la librería de UI Ant Design.

4.8.1. Dashboard

Ya se comentó anteriormente en el apartado 4.6.3 que el frontend debe de mostrar los datos de los sensores de los enchufes inteligentes, esto se consigue a partir de representaciones gráficas de los datos de los sensores guardados en InfluxDB. Para mostrar estos gráficos se ha utilizado la librería reaviz, una librería de componentes de ReactJS para mostrar gráficos.

Otro requisito que se ha impuesto es que estos gráficos deben de actualizarse dinámicamente tanto por una determinada longitud de tiempo como por hacer una consulta de los datos de los sensores en InfluxDB a través del frontend. Esto se podría lograr por medio de timers en el frontend realizando peticiones HTTP a la API REST del backend, pero se ha optado por utilizar la libreria Socket.IO que usa websockets para comunicarse con el backend a tiempo real y pedir los datos de los sensores de los enchufes inteligentes a tiempo real. Esto permite que el backend envíe datos de los sensores después de un determinado tiempo al frontend y así actualizar las gráficas. En el frontend se utiliza el cliente de Socket.IO con el que se comunica con el servidor web de Socket.IO. Tenemos entonces otro servidor web que se encarga de comunicarse con el cliente Socket.IO del frontend, recibe los mensajes/eventos del cliente y este a su vez envía mensajes/eventos con los datos de los sensores del dispositivo asociado al dashboard.

En el apartado de modelo de datos de un dashboard 4.6.3 se ha mencionado que un dashboard contiene una lista de subdocumentos con *fields* de device y parameter. Device se refiere a que enchufe inteligente y parameter se refiere a qué parámetro del enchufe inteligente queremos añadir a la gráfica del dashboard, así podemos tener gráficas en las que solo se ve los datos de un solo parámetro de un dispositivo o varios parámetros de diferentes dispositivos, de esta manera se puede comparar los datos de los sensores de diferentes dispositivos. El tipo de gráficas que se permiten realizar en la interfaz web son por líneas, área o barras. La figuras 18, 19 y 20 muestra un ejemplo de cada tipo e gráfica que se permiten crear en la interfaz web.

Se ha desarrollado otro servidor web exclusivo para el backend de Socket.IO, es un servidor web Node.js que utiliza express.js y que crea una instancia de servidor de Socket.IO que escucha conexiones de clientes Socket.IO. En la figura 21 podemos ver como el

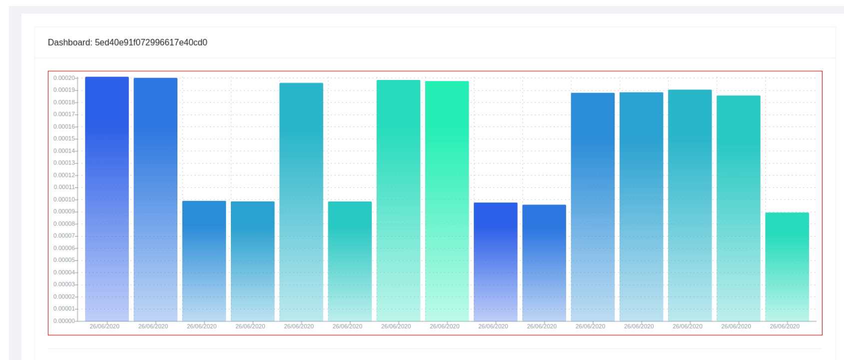

Figura 18: Captura de pantalla de una gráfica de barras

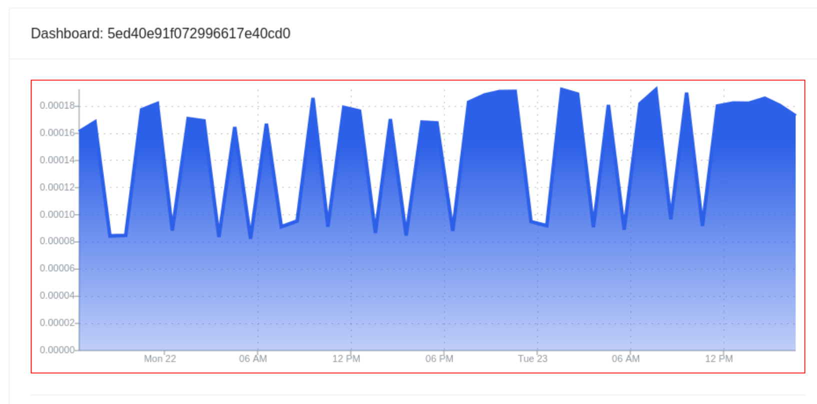

Figura 19: Captura de pantalla de una gráfica de area

Figura 20: Captura de pantalla de una gráfica de lineas

cliente se comunica con este servidor y con el propio servidor backend que provee la API REST además de las relaciones de estos con los servidores de bases de datos, MongoDB y InfluxDB.

Cuando un cliente entra en la url */dashboard/:id* de un dashboard en el frontend, el cliente de Socket.IO se conecta con el servidor de Socket.IO. El servidor de Socket.IO en

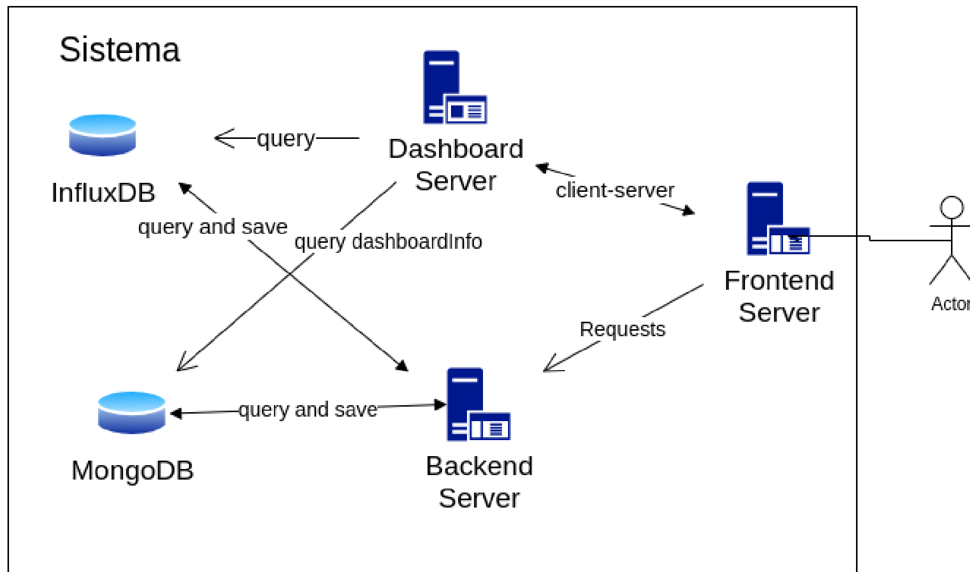

Figura 21: Diagrama de los diferentes componentes del sistema y cómo interactúan con ellos

cuanto un cliente se conecte a él, añadirá dicho cliente a el *room* correspondiente a la dashboard elegida a través del uso de él id del dashboard proporcionado en la url. Los *room* de Socket.IO son canales en los que los sockets de los clientes pueden unirse y el servidor puede enviar mensajes broadcast a estos clientes unidos a la *room*. La razón por la que se ha decidido crear *rooms* para cada dashboard es para optimizar la manera de enviar a tiempo real los datos. En vez de tener un timer para cada cliente conectado al servidor Socket.IO, se ha decidido tener un solo timer por dashboard y enviar los datos de los sensores a los clientes de Socket.IO que escuchen los eventos de dicho dashboard. Para ello en el servidor web de Socket.IO se ha desarrollado un administrador de las conexiones de los clientes de Socket.IO a el servidor (*dashboardManager*) y un gestor de cada *room* del servidor de Socket.IO (*room*). Además en el lado del cliente también se ha desarrollado una clase para manejar los eventos y enviar eventos de Socket.IO al servidor Socket.IO (**DashboardApp**). La lista de eventos posibles son:

- data
- Dashboard-error
- getData

El evento de **data** lo utiliza el servidor Socket.IO para enviar datos al cliente, el cliente escucha estos eventos. El evento de **getData** sirve para que el cliente pida al servidor los

datos a partir de un rango determinado de tiempo.

En la figura 22 podemos ver que las acciones que realiza el servidor Socket.IO en cuanto un cliente se conecta. En la figura 23 podemos ver como el room envía los datos de los sensores de los enchufes inteligentes cada cierto tiempo a los clientes conectados Y en la figura 24 podemos comprobar las clases desarrolladas tanto para el servidor Socket.IO como el cliente Socket.IO de el frontend.

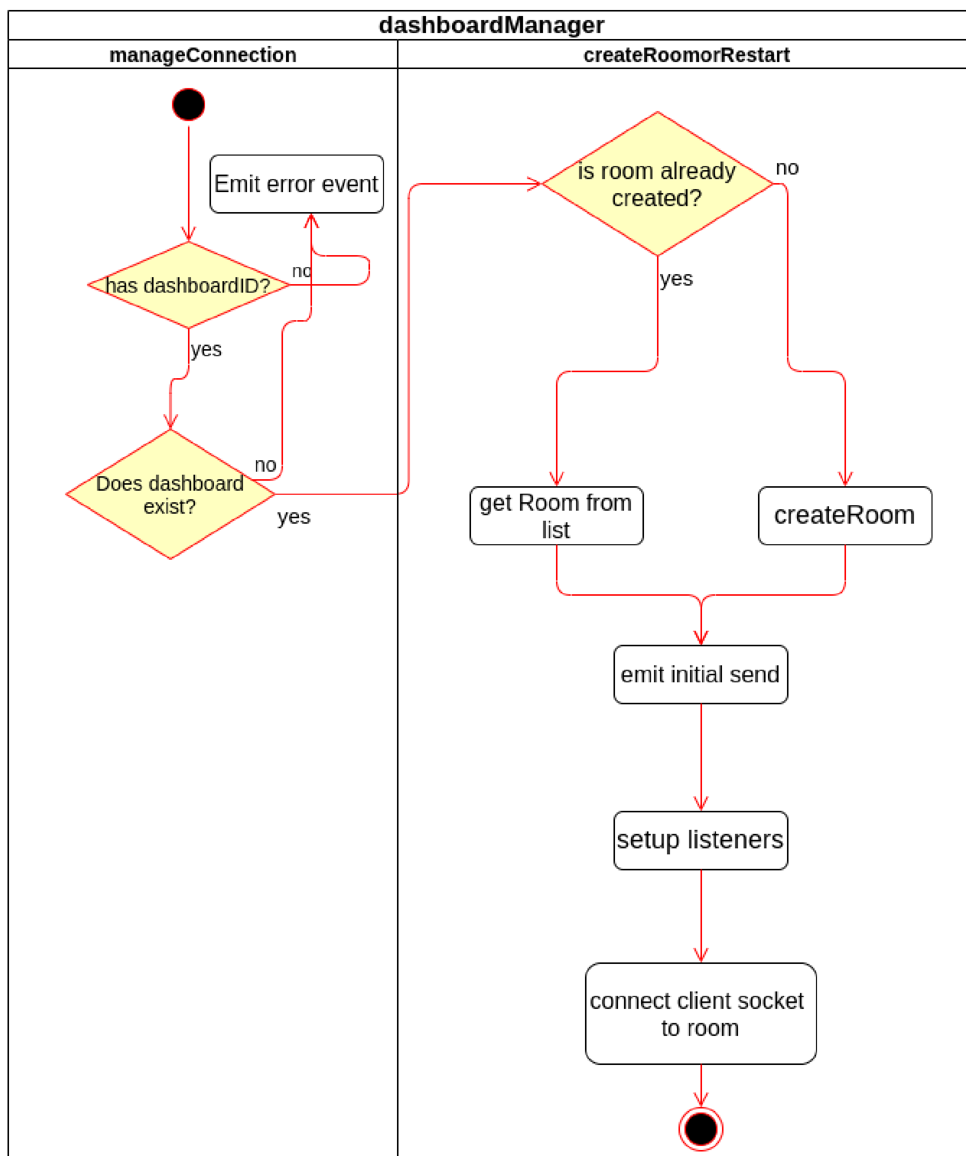

Figura 22: Diagrama de flujo de las acciones que realiza el servidor de Socket.IO una vez que un cliente se conecte a él

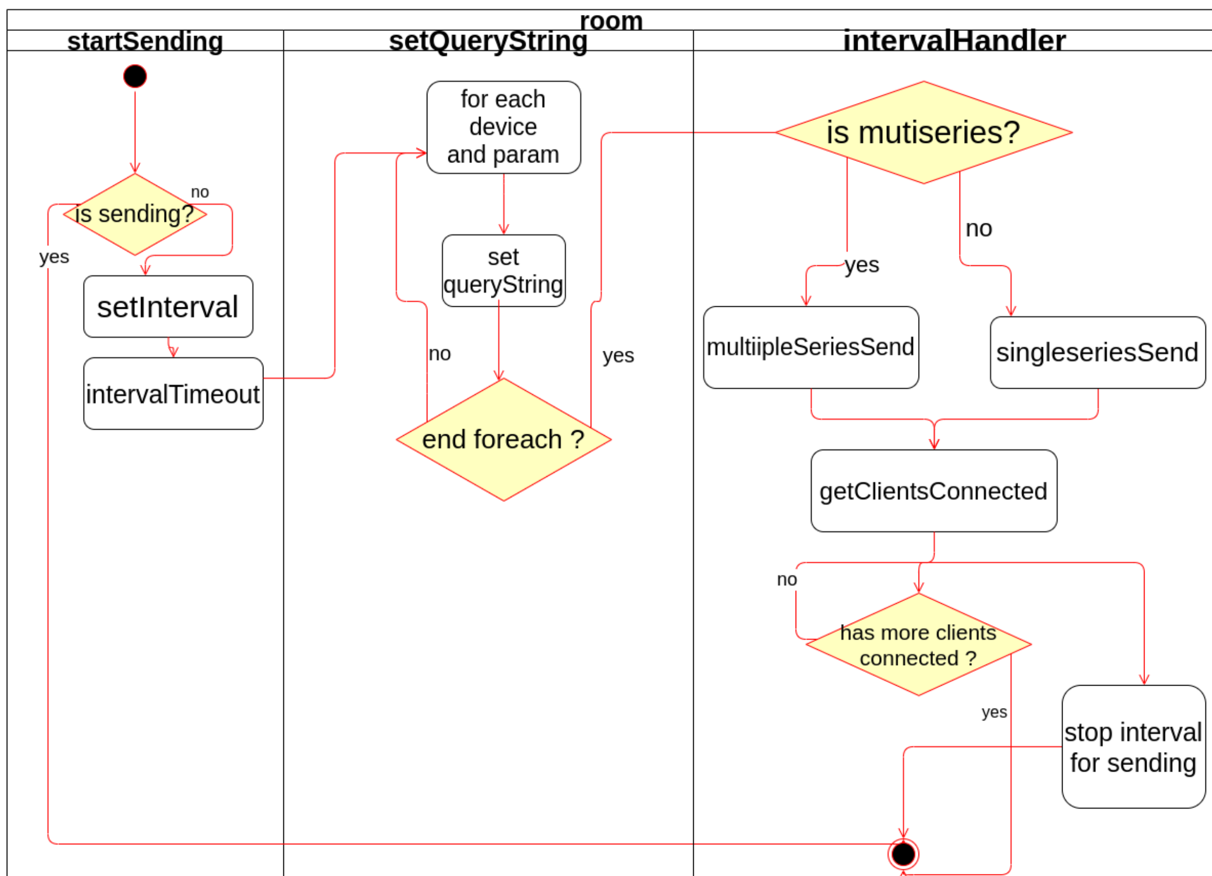

Figura 23: Diagrama de flujo desde que una room empieza a enviar a intervalos, los datos de las query de los devices y parámetros en una dashboard

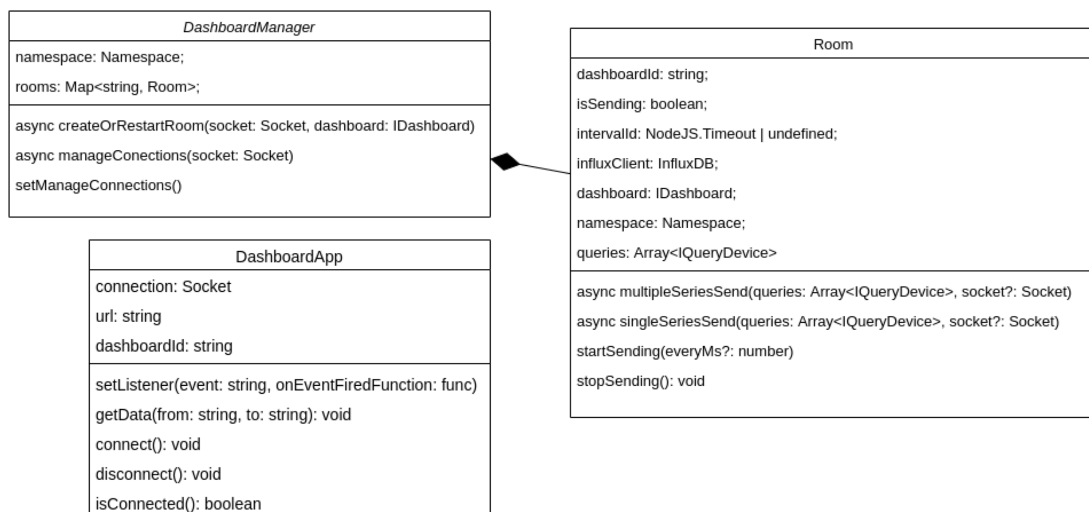

Figura 24: Diagrama de clases de DashboardManager y Room del servidor Socket.IO y DashboardApp del cliente Socket.IO en el frontend

4.8.2. Comprobación del estado del sistema

Uno de los requisitos de la aplicación es comprobar y mantener el estado de los enchufes inteligentes, es decir comprobar si si están encendidos o no. También se mantiene y

comprueba el estado de las automatizaciones definidas en el sistema, si están programadas o no.

Para realizar esto se han desarrollado dos componentes que hacen *pooling* de los estados de los enchufes inteligente y las automatizaciones y actualizan su documento en la base de datos (**AutomatizationStatusPooler** y **DeviceStatusPooler**). Ambos tienen una lógica similar, tras un determinado de tiempo comprueban el estado de los enchufes y las automatizaciones. Para el DeviceStatusPooler, este escucha cualquier el último mensaje del *topic* ***{root topic}/relay/0*** y establece el estado actual en el que está el enchufe inteligente. Podemos comprobar la secuencia que realiza cada vez que pasa un determinado de tiempo y comprueba el estado de los enchufes inteligentes, en la figura 25.

Para el AutomatizationStatusPooler cada cierto tiempo ejecuta el método *isScheduled* de la automatización. Una automatización estará en estado *scheduled* si hay un Schedule en el componente de Scheduler o si hay un task en activo de Kapacitor asociado a un influxTrigger. Como dijimos, la lógica entre AutomatizationStatusPooler y DeviceStatus-Pooler son las mismas, debido a esto el diagrama de clases es más o menos similar. En la figura 26 tenemos el diagrama de clases de el DeviceStatusPooler.

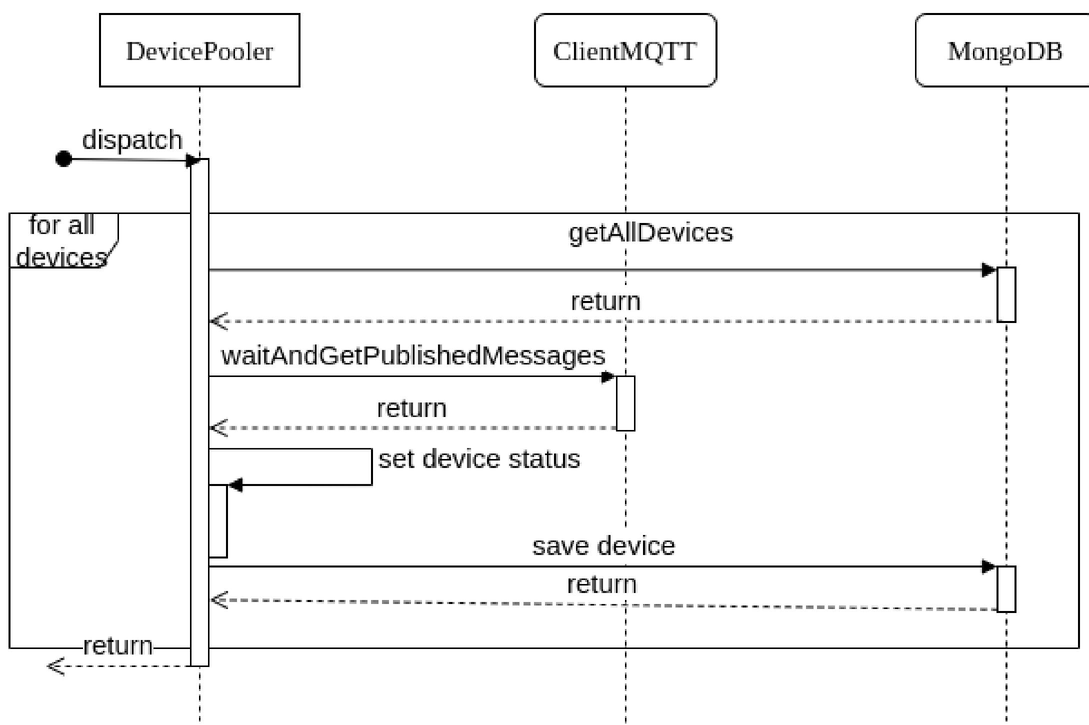

Figura 25: Diagrama de secuencia de DeviceStatusPooler con la ejecución de acciones que realiza cada vez que tiene que comprobar el estado de los enchufes inteligentes

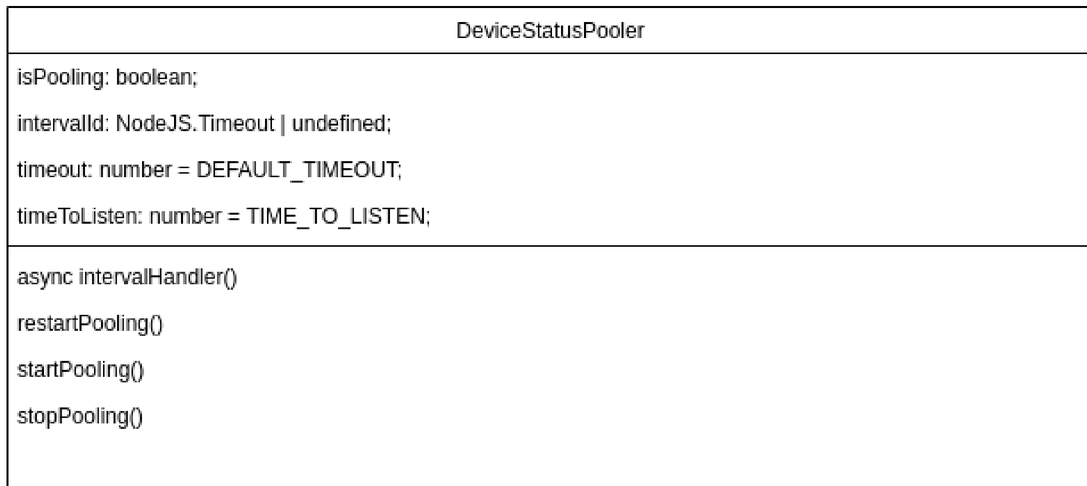

Figura 26: Diagrama de clases de DeviceStatusPooler

4.9. Despliegue y testeo de la aplicación

El requisito de que la aplicación sea de Instalación y configuración relativamente sencilla. Se puede alcanzar gracias a Docker.

Docker es una herramienta que ayuda a resolver problemas comunes como insertar, eliminar, actualizar, distribuir y correr software. Permite crear entornos aislados donde los programas que corren dentro de un Docker contenedores hablan directamente con el el kernel de linux a diferencia de maquinas virtuales, Docker no utiliza virtualización de hardware. La tecnología contenedores es una característica del sistema operativo linux. Docker solo hace más fácil el uso de la tecnología contenedores puesto que esta ya existe en Linux, los namespaces y cgroups. En sí los Docker contenedores son cajas donde correr las aplicaciones y sus dependencias de forma aislada de otras aplicaciones. [13]

En el caso de un sistema Internet de las Cosas, la tecnología de contenedores tiene sentido para las aplicaciones del *gateway* o aplicaciones que dependan de un sistema operativo puesto que Docker utiliza la tecnología de contenedores de Linux, de esta manera los clientes no tienen que preocuparse por dependencias y problemas de software la aplicación será igual sea donde sea el entorno en el que se despliegue.

Docker también tiene un gran beneficio para el desarrollo de aplicaciones y el testeo, debido a que es fácil de iniciar y parar contenedores, un desarrollador puede tener el entorno de desarrollo desplegado con solo unos pocos comandos, puede emular también un entorno de producción donde testear la aplicación que está desarrollando, en el caso de este trabajo Docker ha sido una tecnología muy importante para el desarrollo y testeo

de la aplicación.

En este trabajo, todo componente del sistema es un contenedor de Docker. Los componentes que se han desarrollado: servidor web del **backend**, servidor web del **frontend**, servidor web de **Socket.IO**, son contenedores que utilizan una imagen que parten de Node.js provista por los propios desarrolladores de Node.js. Así se obtiene un contenedor con los paquetes y dependencias e incluye una versión de Node.js instalada. De esta manera el cliente no necesita descargarse el código de la aplicación para compilarlo en su propia Raspberry pi. La única dependencia que el cliente debe de instalar es el programa de Docker y Docker-compose.

A menudo un sistema está compuesto de varias aplicaciones ejecutándose. A la hora de manejar un sistema de ese tipo y para poner en marcha las aplicaciones, configurarlos y pararlos de una forma automática, Docker-Compose es una herramienta ideal. En el contexto de la herramienta Docker-compose, los contenedores de las aplicaciones pasan a ser llamados servicios, estos servicios pueden pararse, iniciarse, modificarse, escalar, actualizar, comprobar que dichos servicios están vivos y realizar balanceo de carga entre los servicios y sus réplicas.

Docker-Compose es una herramienta, que permite definir un conjunto de de contenedores/servicios a desplegar e iniciar, todo esto se define a partir de un fichero en formato YAML, normalmente llamado *Docker-compose.yml* el fichero Docker-compose utilizado para el sistema permite definir el conjunto de servicios de Docker que han de levantarse, asimismo también permite definir y anclar la versión de la imagen de cada contenedores definido en este fichero, definir sus variables de entorno, la red de Docker a la que se conecte y sus puertos expuestos. Además podemos indicar también que dichos servicios se reinician cada vez que caigan, esto permite aumentar la disponibilidad del servicio. La principal ventaja es que el cliente con solo ejecutar un comando puede crear y desplegar todos los servicios necesarios del sistema:

- El servidor web del **backend**
- El servidor web del **frontend**
- El servidor web del dashboard **dashboard Socket.IO**
- La base de datos **MongoDB**
- Time series database **InfluxDB**

- Broker **Eclipse Mosquitto**
- Aplicacion **Telegraf**
- Aplicacion **Kapacitor**

En el apéndice 6.1.1 se explica como instalar Docker, Docker-compose e iniciar el sistema completo. El repositorio tfg_system contiene un Docker-compose.yml con la definición de los contenedores necesarios y las versiones fijas usadas para desarrollar el sistema y la aplicación principal. Las imágenes Docker de los componentes desarrollados del sistema están alojadas en un repositorio de imágenes Docker, Docker Hub. Para que así sea fácil de encontrar y de descargar en cualquier máquina de un cliente que tenga conexión a internet. La para la creación de estas imágenes Docker se han definido Dockerfiles. Los Dockerfiles son ficheros de texto que contienen los comandos necesarios para crear la imagen Docker. Como se mencionó al principio, cada componente del sistema parte de una imagen de Node.js, los Dockerfile del sistema contienen los comandos para instalar las dependencias de los proyectos de cada componente y la compilación de estos. Con ejecutar el comando docker build, se generan dichas imágenes Docker, para publicarlas a el repositorio de Docker Hub se utiliza el comando docker push.

Este proceso de generar la imagen y publicarla a el repositorio de Docker Hub consiste en el proceso de entrega de la aplicación. Para automatizar este proceso se podría utilizar scripts de bash pero se ha decidido utilizar un servicio de integración continua y despliegue continuo *CI/CD* para entregar de forma automática estas imágenes Docker. Se ha elegido Travis CI puesto que posee una buena integración con Github, el servicio para hostear repositorios Git, y donde está alojado el código de la aplicación.

Travis CI dispara una *build* cada vez que un desarrollador haga un *push* al repositorio de Github. Una vez la build es iniciada, Travis CI ejecuta una serie de comandos que se indican en un fichero yaml, nombrado *.travis.yml*. El fichero .travis.yml alojado en cada repositorio de cada componente del sistema contiene una serie de acciones que se llevan a cabo en una build, una de ellas es la de crear la imagen Docker del componente y subirla al repositorio de Docker Hub. De esta manera se garantiza una entrega automática de las imágenes Docker de los componentes del sistema.

El proceso de compilación y despliegue es un posible proceso que se puede automatizar con un servicio CI/CD. CI/CD son procesos para el desarrollo de un producto software

en los que se puede automatizar y aliviar la carga humana de dichos procesos. Integración continua *CI* es una fase de un proceso en el que el cambio de los códigos de los desarrolladores en cada rama del proyecto git es unida en la rama principal y testeada. La integración continua utiliza test desarrollados para comprobar que la pieza de código desarrollado funciona, si esta no funciona, se notifica para que haya intervención humana y arregle los cambios. Mientras que despliegue continuo *CD* es un proceso en el que una vez los cambios se han unido en la rama principal y validadas en la fase de integración continua, se despliegan los artefactos del producto software que se está desarrollando para que los clientes puedan descargar la última versión del producto [14]. De esta manera al implementar un pipeline de CI/CD automatizamos el proceso de testeo y despliegue de la aplicación.

En el desarrollo de cualquier aplicación software se ha de garantizar que dicho software funciona y actúa concorde a lo los objetivos y requisitos para ello conlleva desarrollar casos de prueba. Existen varias manera de realizar casos de prueba para el código desarrollado.Se puede comprobar que cada pieza de código funciona bien individualmente a través de test unitarios. A lo largo del desarrollo de cada componente del sistema se ha desarrollado test unitarios usando el framework de testing Jest junto con la librería de mocks, stubs y spy Sinon.js.

Puesto que los componentes del sistema tienen una dependencia con los otros componentes del sistema, por ejemplo, el servidor web del backend depende de: eclipse mosquitto, Kapacitor, Telegraf, InfluxDB y MongoDB. Los test que se han desarrollado crean datos en MongoDB, guardan datos y recogen datos de InfluxDB, publican mensajes MQTT y definen tasks en Kapacitor, todo esto con un sistema limpio sin configurar. Para solucionar el problema de las dependencias, poder desarrollar tests y ejecutar estos tests sobre un sistema limpio, se ha aprovechado de las ventajas de Docker y Docker-compose. Para ello se ha definido un fichero Docker-compose.yml con los contenedores e imágenes necesarias para levantar un sistema y testear sobre él. De este modo nada más que para ejecutar tests basta con levantar los servicios usando docker−compose up −d y para iniciar los tests utilizar el comando docker−compose run app run test. Esto es similar a lo que el fichero .travis.yaml realiza, primero levanta los servicios y después inicia el test. La build de Travis CI primero ejecuta los tests, y si dichos tests pasan, entonces se realiza el despliegue/entrega de la aplicación, es decir crea las imágenes Docker y después las sube

al repositorio de Docker Hub. De este modo tenemos definido un pipeline, el conjunto de procesos para testear la aplicación y entregarla/desplegarla en un repositorio de Docker. Al ser automatico el desarrollador solo necesita centrarse en codificar el programa y desarrollar tests y que dicho código pasen los tests obteniendo así una eficiencia en el desarrollo de un programa software.

Para los casos de prueba del código del servidor web de backend prueban la manipulación del modelo de datos, la API REST, la lógica de las automatizaciones, la lógica de pooling del estado del sistema y la comunicación por MQTT.

Para el frontend debido a restricciones de tiempo solo se ha realizado test unitarios sobre la lógica de el cliente Socket.IO, utilizando un servidor mock de Socket.IO. Para testear que el Frontend muestra correctamente los datos de el backend, se ha optado por mockear el servidor del backend, más en concreto la API REST que provee, aunque no en su totalidad, a partir del uso de un servidor de mockeo llamado json-server para comprobar que las páginas de Next.JS funcionan correctamente y que el diseño de cada componente quede correcto. Para desarrollar, testear visualmente cada componente individual de ReactJS se vean visualmente correctos y funcionen, se ha utilizado la herramienta llamada Storybook, que permite desarrollar de forma aislada componentes del frontend. Gracias a esta herramienta, se ha decidido desarrollar el frontend usando el método de desarrollo *bottom up*, empezando por componentes individuales y luego juntarlos hasta tener una página completa del frontend. Finalmente para el servidor de dashboard los test unitarios prueban la lógica de las clases *DashboardManager* y *Room*.

Todos los archivos de test y Docker-compose.yml utilizados para realizar los tests se encuentran en las carpetas de test de los repositorios correspondiente a cada componente desarrollado.

Aparte de los test unitarios, existen otros tipos de test como los de integración, de sistema y de aceptación. Debido a restricciones de tiempo no se ha podido desarrollar tests automáticos de integración sistemas y aceptacion. Los test de integración envolverán la interacción entre los diferentes componentes desarrollados es decir que la interacción entre los servidores backend, frontend y dashboard. Y los tests de sistema en sí por completo como un producto final. Tanto los de integración como los test de sistema se han hecho a mano desde el navegador con el sistema total en sí desplegado. No se ha llegado a automatizar o a desarrollar test automatizados debido a restricciones de tiempo y que

está fuera de los objetivos del trabajo.

Otro punto más a añadir es el nivel de cobertura de código, permite saber cuánto porcentaje del código ha sido cubierto por el test, de esta manera podemos saber si entre las distintas bifurcaciones del código de un programa los tests pasan sobre esas bifurcaciones o no. Asegura un porcentaje de calidad de test sobre el código. El framework de test Jest provee también la opción de recogida de cobertura de código y mostrar un reporte sobre la cobertura de código. Esta cobertura puede ser recogida y guardada para tener en constancia como evoluciona dicho porcentaje de cobertura de código. Se ha optado por utilizar el servicio de Coveralls porque también tiene una buena integración con Travis CI. En una build de Travis CI, a parte de testear y desplegar las imágenes Docker de los componentes desarrollados, también recoge la cobertura de código y las publica a Coveralls.

5. Conclusiones y Líneas Futuras

5.1. Conclusiones

Se ha desarrollado un sistema para el control de energía eléctrica en el que se ha logrado:

- Obtener información sobre el consumo eléctrico se ha empleado enchufes inteligentes, hemos conseguido encontrar una manera de medir cuanto un dispositivo o dispositivos domésticos consumen energía eléctrica, por medio de estos enchufes inteligentes, además estos enchufes inteligentes han sido modificados con un Firmware de código abierto diferente de él de fábrica para evitar usar una aplicación de terceros sin la inseguridad de saber si dichos datos lo recogen o no los desarrolladores de la app de terceros evitando que estos datos se guarden en la nube.

- Realizar la la interconexión de los enchufes inteligentes del sistema gracias a que estas se conectan a un broker MQTT donde publicar información de sus sensores en unos topics de MQTT y que la aplicación principal maneje la lógica entre ellas.

- Almacenar tanto los datos de consumo, los sensores y metadatos de los dispositivos registrados en el sistema respectivamente en MongoDB e Influxdb haciendo uso de bases de datos No relacionales como time series databases aprendiendo a manejar estos tipos de bases de datos, unas bases de datos muy diferentes a las Relacionales, que son las que más hincapié se ha dado en la formación de la carrera.

- Permitir que el sistema controle de forma automática los dispositivos apoyándose en las distintas automatizaciones desarrolladas:

  - Automatizaciones por tiempo utilizando la programación asíncrona y el paradigma de programación por eventos.

  - Automatizaciones según el precio de la energía eléctrica dado en un tiempo en concreto usando los datos de los precios del PVPC que provee la API de la red eléctrica española, además, esta automatización se ha implementado haciendo uso de conocimientos de programación web y API REST.

  - Automatización por aumento y descenso de valores de los sensores de los enchufes inteligentes o por el aumento y descenso del gasto económico de la energía

eléctrica consumida por los enchufes inteligentes.

- Que el cliente pueda controlar de manera manual, el estado de los dispositivos a través de una interfaz gráfica. Esta interfaz gráfica se ha desarrollado utilizando React JS un framework de Javascript moderno y muy utilizado hoy en día.

- Permitir que el cliente pueda comprobar el estado de los dispositivos y el valor que recoge los sensores de los enchufes inteligentes. Esto es través de gráficas además de permitir crear diferentes gráficas respecto a los datos de los sensores obtenidos. Las gráficas devuelven nuevos datos actualizados tomados por los sensores a ciertos intervalos de tiempo e incluso el cliente puede pedir datos desde un punto en el tiempo a otro. Para ello se ha tenido que aprender a utilizar una librería de comunicación a tiempo real, SocketIO.

Aparte de realizar los objetivos y desarrollar un sistema Internet de las Cosas, se ha adquirido diferentes conocimientos no dados en la carrera o enriquecido algunos conocimientos que ya se habían aprendido en la carrera:

- Se ha aprendido a desarrollar test unitarios y de integración para cada componente del sistema, garantizando así el funcionamiento del sistema completo.

- Los servidores web del sistema se han desarrollado utilizando Node.js junto con Javascript y Typescript.

- El sistema desarrollado hace uso de un pipeline configurado para testear, subir cobertura de código y desplegar el software, aplicando así un pipeline de CI/CD, un concepto que está teniendo un aumento de popularidad enorme en estos tiempos.

- Además se ha aprendido la herramienta Docker y conceptos de tecnologías de contenedores, una tecnología muy útil para desarrollo, tests, aplicaciones en la nube y mucho más. Y que, igualmente está en auge además de que es otra tecnología mencionada en la carrera pero no vista desde un aspecto útil para desarrollo software.

- Se han aplicado conocimientos anteriores como la programación orientada por objetos y diseño software, se han ampliado más conocimientos sobre el uso de control de versiones gracias a este proyecto.

Respecto a el sistema desarrollado existen muchas maneras de mejorarlo pero comparado con los demás proyectos de código abierto que proporcionen una manera de controlar dispositivos Internet de las Cosas para consumir menos, existen mejores alternativas. La solución planteada aquí es mucho más específica puesto que usa determinados enchufes inteligentes y no tiene soporte para demás enchufes inteligentes, además de que se ha reinventado la rueda debido a que ya existen sistemas de código abierto más robustos con mucha más flexibilidad de configuración. En contraste a esto, el sistema desarrollado es más sencillo de utilizar porque es más específico para una solución en concreta. En general el proyecto ha servido mucho más para ampliar conocimientos aunque lo que se haya desarrollado no sea un aporte muy revolucionario entre las diferentes soluciones para controlar el consumo.

5.2. Lineas futuras

La aplicación desarrollada cumple con su principal propósito de controlar la energía consumida aunque la aplicación esté en un estado completo, existen muchos aspectos en los que se puede mejorar y que, por restricción de tiempo no se han podido desarrollar más. Estas mejoras de desarrollo pueden servir para aplicarse en otras áreas de conocimiento que no se exponen y explican en este trabajo. A continuación se proponen unas posibles mejoras para el trabajo desarrollado.

Uno de los aspectos que muchas veces se deja un poco de lado o para al final cuando se desarrolla una solución software es la seguridad de una aplicación. El sistema desarrollado tiene algunos puntos de flaqueza en relación a la seguridad.

La aplicación no tiene un modo de autenticación y autorización. Cualquier persona conectada a la misma red donde esté la estación base y y los enchufes inteligentes, puede acceder y controlar con total libertad los enchufes inteligentes. Para evitar esto, se debe de aplicar autenticación y autorización a diferentes componentes del sistema, a la API REST del servidor del backend, proveer un login en el frontend y proveer autenticación para SocketIO. Aparte de de proporcionar autenticación y autorización, otro foco de problema que existe en el sistema desarrollado es la privacidad, los servidores web no establecen una conexión cifrada con el cliente o los clientes que acceden a él, solo establecen una conexión HTTP, Al forzar a los clientes a usar HTTPS permite que ningún atacante conectado a la red pueda observar los datos en plano. Aparte de utilizar HTTPS en los servidores web del

sistema, el Broker MQTT también debe de utilizar SSL proporcionando así autenticación en el lado de los enchufes inteligentes y en MQTT. Para ello ESPurna también proporciona soporte para SSL aunque habría que compilar un Firmware con los flags necesarios. Otra manera de aumentar la seguridad de los datos es imponer autenticación y roles de usuario dentro de las bases de datos tanto de MongoDB como Influxdb además Kapacitor también provee la misma posibilidad de autenticar y autorizar usuarios que se conecten a él.

Ya que se ha mencionado que el Broker MQTT utilice SSL y autenticación, otro cambio de configuración que se podría hacer a el Broker MQTT sería el de aumentar la calidad de servicio, actualmente está configurada a 0, si se quiere garantizar que los dispositivos conectados a el Broker MQTT se aseguren de que los mensajes y datos de sus sensores de verdad hayan sido recibidos por el servidor Broker, podemos aumentar esta opción.

En el aspecto de accesibilidad de el frontend existen varios puntos de mejora como la internacionalización de la aplicación web esto se ha intentado realizar mientras se estaba desarrollando el frontend y existe un prototipo en una rama de desarrollo del repositorio Git del frontend para ello se decidió utilizar i18n, una herramienta que proporciona una solución para tener palabras clave traducidas en una aplicación web.

Otra posible mejora del frontend es la de optimizar la visualización del frontend en diferentes dispositivos de diferentes resoluciones, es decir una mejora en el diseño, para que el frontend sea responsive. Además se puede proporcionar por ejemplo temas oscuros y claros.

Otra mejora que podría realizarse para tanto el frontend y el servidor de Socket IO es la de convertir la conexión de Socket IO desde el cliente, global, es decir que esté conectado al servidor de Socket IO a nivel global de la aplicación web, puesto que cada vez que entramos en la ruta **/dashboards/:id**, el cliente Socket IO del frontend hace la conexión con el servidor de SocketIO. Además la lógica implementada está limitada a obtener solamente los datos de la dashboard correspondiente obtenida al entrar en la ruta **/dashboards/:id** del frontend. Podemos ampliar su funcionalidad dotando de la posibilidad de mostrar notificaciones sobre algunas acciones realizadas en el sistema como por ejemplo, que una automatización haya disparado, que el estado de un enchufe inteligente ha cambiado o notificar el hecho de de que ha caido la conexion con algún componente del sistema.

En el dashboard también se podría implementar más variedades de gráficas, puesto

que reaviz permite una variedad de diferentes gráficas pero solo se han utilizado 3 tipos de entre todas las gráficas disponibles.

Otro punto de mejora para las gráficas y el dashboard es permitir que la vista de los datos sean agregadas, uno de los problemas de pedir los datos de un rango determinado de tiempo grande es que se devuelvan tantos datos que haga que el rendimiento debido a una petición de grandes tamaños de datos. Por eso se desea elegir una manera de condensar esos datos ya sea usando una media o una suma agregada o otros métodos para reducir la cantidad de datos que ha de transferirse.

Durante el desarrollo de la aplicación se ha decidido recoger reportes de cobertura de código, otro punto que se podría mejorar sobre el código desarrollado es la calidad del código, existen herramientas para medir la calidad del código de un proyecto software, no es una obligación estricta tener un control de calidad de código pero tenerlo implica obtener sugerencias sobre cómo mejorar el código y como hacer el código más mantenible, esto provee beneficio a largo plazo además de que también puede prevenir posibles errores o bugs futuros en cuanto se quiera ampliar la funcionalidad del sistema. Una posible herramienta a utilizar es SonarQube, que además se puede combinar con el pipeline CI/CD desarrollado para tener notificaciones automáticas de posibles mejoras del código con cada commit y push.

Uno de los objetivos del trabajo es que la configuración del sistema sea sencilla y fácil, existe otra posibilidad más para mejorar este aspecto y tener una configuración mucho más automatica existen unos procesos de la primera instalación que no se han automatizado, por ejemplo los enchufes inteligentes deben de ser primero configurados para que se conecten a la red wifi, conectarse a la red wifi y conectar en el broker MQTT. Como se explica en el apéndice 6.1.2 el enchufe inteligente, una vez instalado el Firmware de ESPurna inicia en modo AP y hay que configurarlo conectándose a la red wifi que cree y configurarlo desde la página «http://192.168.4.1» o «http://192.168.4.1». Espurna provee una manera de configurar el enchufe inteligente por HTTP, se puede automatizar la configuración haciendo otro componente del sistema que escanee redes Wifi, busque las que tengan un SSID similar a «ESPURNA-XXXXXX», se conecte a dicha Wifi de un enchufe nuevo y la configure a través de su API HTTP además se puede rizar más el hilo porque ESPurna también tiene soporte para telnet, otra manera de configurarlo en conectándose por telnet y configurandolo. De esta manera el cliente solo necesita tener un

enchufe inteligente con el Firmware de ESPurna, conectarlo a la red eléctrica de la casa y dentro del frontend indicar que busque un dispositivo con firmware ESPurna, se conecte a él y que lo configure de forma automatica.

Existen también posibles puntos de mejora relacionados en la implementación y diseño del sistema.

Debido a restricciones de tiempo se ha implementado algunas soluciones no ideales, por ejemplo, una de las variables de entrada de el task template es la variable «url», esta variable aloja la url donde está la aplicación del backend. Esta url varía según en qué red privada estemos conectados, además al no tener DNS siempre se utiliza directamente la dirección IP de la Raspberry PI y el usuario ha de configurarla antes en el Docker-compose.yml. Una manera de evitar esto y tener resolución de nombre de servidor sería utilizar un reverse proxy. Un solo servidor se encarga de redireccionar las peticiones HTTP hacia la Raspberry PI y a los servidores de los contenedores ejecutándose dentro de esta. Otro problema que existe en el frontend y relacionado con esto esque las direcciones IP también están incluidas en el código de compilación y creación de la imagen Docker de modo que requeriría crear una nueva imagen Docker si estás IP cambian, que lo más probable es que así sea.

El proyecto de el componente de Dashboard depende de el proyecto del servidor del Backend, dentro del repositorio hace referencia a el repositorio del Backend puesto que depende del modelo de datos de Dashboard alojado en el repositorio del Backend, una manera de eliminar esta dependencia es convertir en un módulo o paquete privado NPM de todo el modelo de datos alojado en el proyecto del Backend. De esta manera el Dashboard solo necesitaría incluir como dependencia dentro de su código el módulo npm.

Se puede añadir más tipos de automatizaciones gracias a que Kapacitor permite definir task templates para crear tasks, solo habría que buscar que otros métodos se pueden utilizar para automatizar incluso dentro de estos task templates incluir la posibilidad de alerta a un cliente desde telegram o email. Otra posibilidad de crear diferentes tipos de automatizaciones sería aprovechándose de los datos guardados sobre los sensores de los enchufes inteligentes y el consumo gastado por estos enchufes, haciendo uso de machine learning o métodos de inteligencia artificial para dotar el sistema un aprendizaje del comportamiento de consumo de los enchufes inteligentes y buscar patrones donde haya un mayor consumo de energía y gasto para después responder a estos patrones.

El proyecto podría ampliarse a utilizar Docker Swarm y replicar contenedores de las aplicaciones y formar un cluster con ellos, ampliando el nivel de disponibilidad de la aplicación. Otra ampliación sería alojar parte de los contenedores del sistema en la la nube de este modo la la estación base solo tendría que alojar los contenedores de el broker MQTT y Telegraf. Los datos estarían en la nube y tanto el frontend y backend también en la nube, aliviando nivel de cómputo para la estación base.

Otro de los puntos flojos del sistema desarrollado es el nivel de cobertura de código en los demás componentes desarrollados, una ampliación sería implementar más casos de prueba sobretodo para los proyectos de Frontend y Dashboard, así ampliar más los conocimientos en implementación de pruebas sobre el Framework ReactJS. También se podrían implementar pruebas de integración para comprobar que los componentes funcionan bien juntos.

6. Apéndice

6.1. Manual de instalación

6.1.1. Instalacion y configuracion inicial de la estación base

Aunque la propia compañía de raspberry proporciona una imagen suya propia de debian para raspberry, vamos a utilizar Ubuntu Server para raspberry pi puesto que hay una versión de 64 bits en contraposición a Raspbian Buster que solo es de 32 bits. Dicho esto, se ha optado por descargar Ubuntu Server 20.04 LTS para Raspberry Pi 4 64 bits.

La propia página web tiene un tutorial de como instalar e iniciar el sistema operativo en la Raspberry pi 4 solo hace falta seguir el tutorial. Al entrar usando ssh nos pedirá que cambiemos la contraseña, debemos cambiarla por una que deseemos. Una vez instalada y una vez hemos accedido a la raspberry por ssh, la actualizamos 1.

```
1 sudo apt update
2 sudo apt upgrade
```

Listing 1: Comando para actualizar Ubuntu

Cambiamos configuración regional de tiempo para europa/madrid, usando los comandos 2.

```
1 ubuntu@ubuntu:~$ timedatectl list-timezones |  grep Madrid
2 Europe/Madrid
3 ubuntu@ubuntu:~$ timedatectl set-timezone "Europe/Madrid"
```

Listing 2: Comando para listar regiones y cambiar la configuración regional

Cambiamos Ip dinamica a fija siguiendo la documentacion oficial de Ubuntu server Documentation oficial. Una vez realizado esto, debemos de instalar tanto Docker como Docker-compose. Podemos seguir la documentation oficial de Docker para este trabajo de fin de grado se ha optado por la manera de instalar por el script de instalación rapida **convenience script** reiniciamos y comprobamos que funciona con el comando 3.

```
1 Docker contenedores run hello-world
```

Listing 3: Commando hello world de Docker

Ahora procedemos a instalar Docker-compose, como se indica anteriormente basta con seguir la documentation oficial de Docker-compose en este caso lo hemos instalado por medio de **pip3**, primero debemos de instalar **pip3** y después instalar Docker-compose a traves del uso de estos comandos 4.

```
1 sudo apt-get install python3-pip
2 sudo pip3 install Docker-compose
3 Docker-compose --version
```

Listing 4: Commandos para instalar Docker-compose en Ubuntu

Al acabar con la instalación de Docker y Docker-compose realizamos un clone del repositorio **tfg_system**. Para ello usamos git clone. E instalamos git si no lo tenemos instalado Una vez clonado el repositorio configuramos el Docker-compose.yaml con sus variables de entornos Una vez configurado el Docker-compose.yaml debemos de iniciarlo. Para ello usamos el comando 5

```
1 Docker-compose up -d
```

Listing 5: Comando para desplegar todo el sistema y contenedores

Con esto iniciamos todo el sistema de la estación base. Para acceder al panel de control de la aplicación debemos ir a la IP de la raspberry ejemplo **http://192.168.1.54:3002**. Una vez dentro se procederá a realizar los registros de los enchufes inteligentes y creación de paneles de control o automatizaciones

6.1.2. Cambio de firmware y configuración inicial de los enchufes inteligentes

Este apéndice tiene como propósito enseñar a cualquier usuario medio los pasos necesarios para realizar la una modificación del firmware incorporado en el enchufe inteligente BlitzWolf BW-SHP2 y reemplazarlo por ESPurna. Se partirá desde el soldado de los cables a él microcontrolador ESP8266 hasta el reemplazo(flashing) del firmware y la configuración inicial. Este proceso es general para el enchufe inteligente que hemos elegido y podemos instalar otro firmware diferente de el de ESPurna.

Soldar los cables a el microcontrolador Antes de conectar el enchufe inteligente a el ordenador primero hay que establecer los medios para que este se pueda conectar a un

ordenador. Para ello hay que abrir el enchufe inteligente y soldar los cables a los pines del ESP8266

En el momento actual en el que se está realizando este trabajo de fin de grado, el enchufe inteligente BlitzWolf BW-SHP2 ha recibido una revisión de hardware denominada V2.3 en el que los pines de conexión están localizados en otra posición diferente, este apéndice sólo contempla la primera versión de hardware del enchufe inteligente BlitzWolf aunque los pines están localizados en diferentes posiciones, son los mismos pines que hay que conectar. Si se quiere obtener mas información sobre donde están localizados los pines para la revisión de hardware V2.3 puede visitar el repositorio de ESPurna y buscar el dispositivo o acceder a la pagina web `https://github.com/arendst/Sonoff-Tasmota/wiki/Gosund-SP1#new-hardware-revision`.

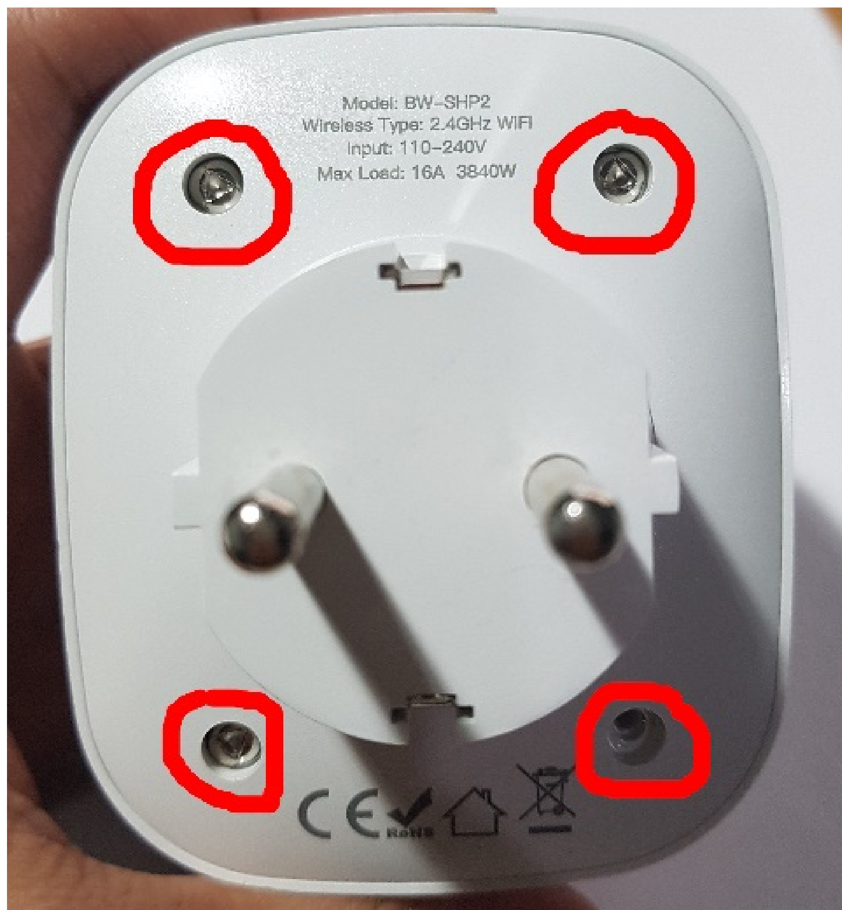

Figura 27: Tornillos del enchufe inteligente BLITZWOLF

Para abrir el enchufe hay que utilizar unas herramientas especiales puesto que los cuatro tornillos que que tiene el enchufe inteligente, véase la ilustración 27 son de rosca en forma de estrella.

Una vez abierto podemos observar el microcontrolador ESP8266, tal y como se muestra en la ilustración 28, detrás de la placa, están los pines del ESP8266 que hay que soldar para unir los cables que se insertarán en los pines del programador. Como se puede observar en la ilustración 29, existe un pin especial GPIO0 que se usará para iniciar el enchufe inteligente en modo flashmode.

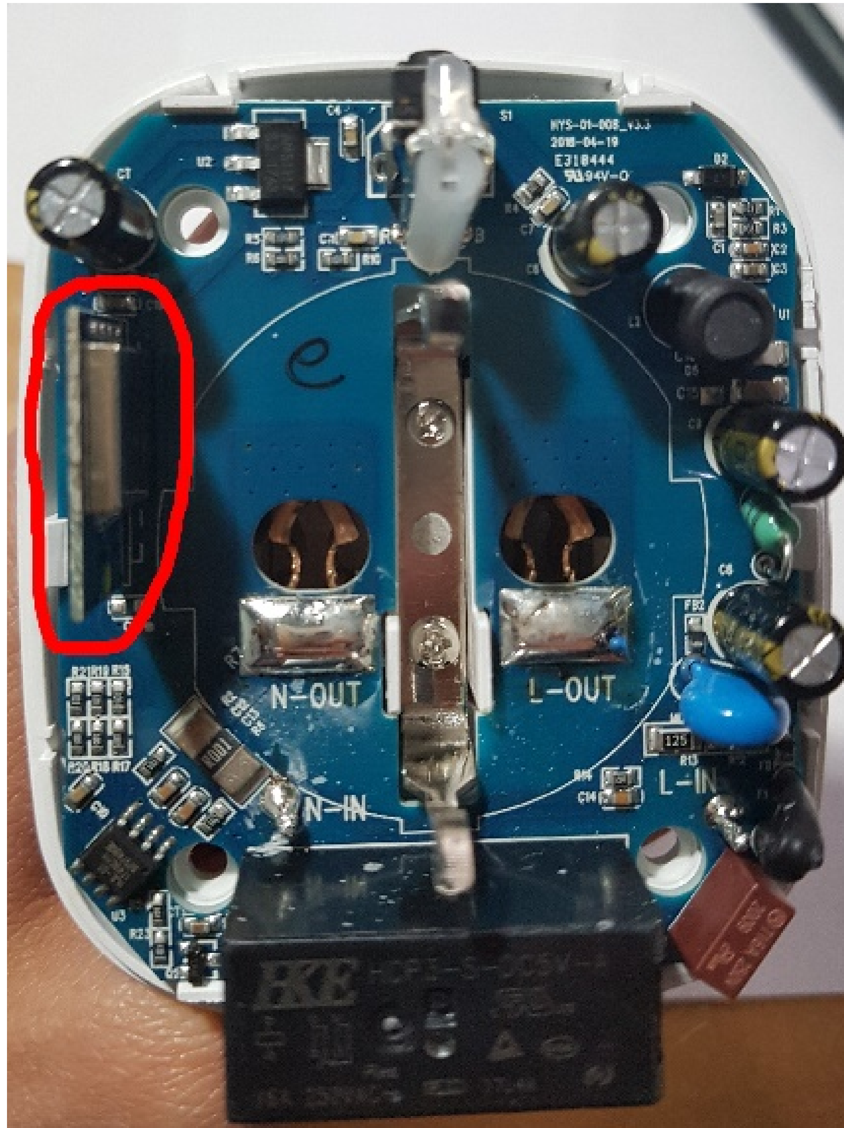

Figura 28: Localización del microcontrolador ESP8266

El pin GPIO0 debe de conectarse y realizar un corto circuito a el pin GND al conectar el enchufe al programador para poder entrar al modo flashmode y en ese modo poder realizar el reemplazo de el firmware del fabricante almacenado en el enchufe inteligente.

Conexión del enchufe inteligente con un ordenador y modo flash Para realizar un flash de la memoria de del enchufe inteligente, hay que instalar el programa esptool.py,

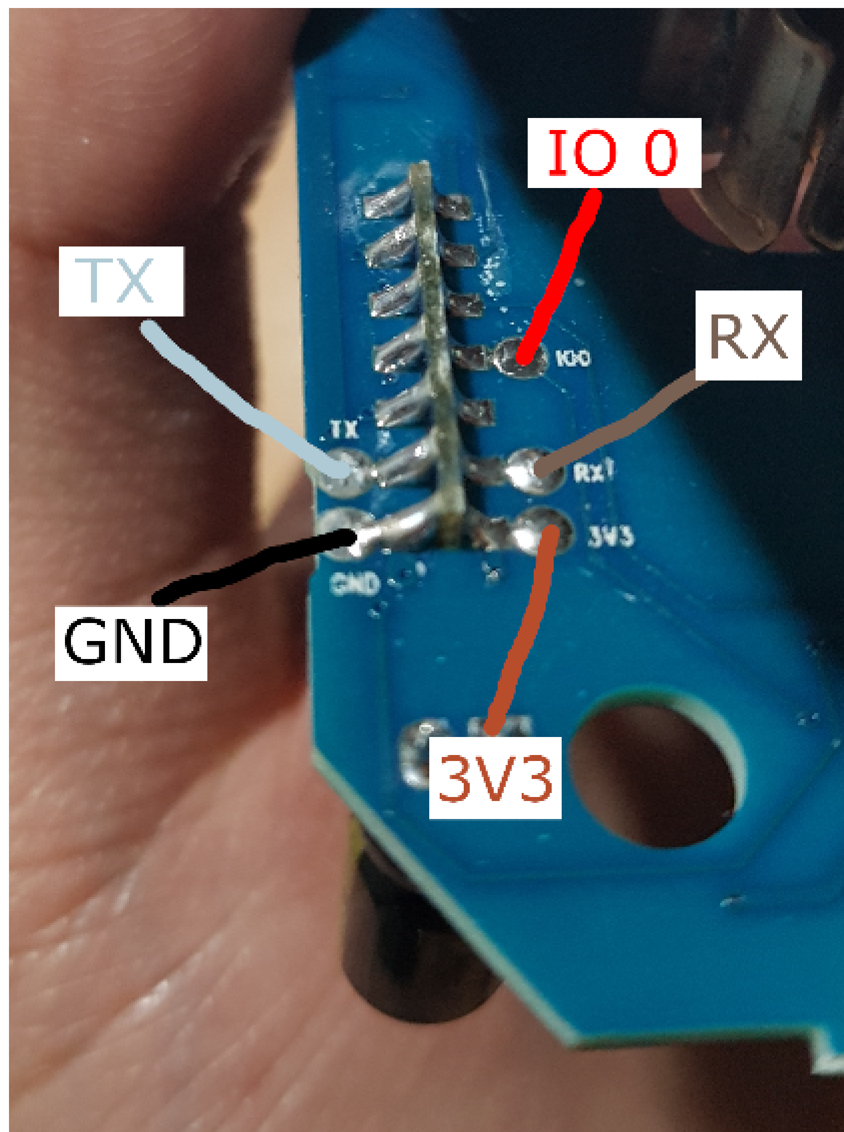

Figura 29: Pines de el microcontrolador ESP8266 que deben ser conectados

este programa requiere tener python no se abarcara la instalación de este programa puesto que se puede referir a la documentación oficial del repositorio de este programa.

Para conectar el enchufe inteligente a el ordanor hace falta un dispositivo convertidor USB a UART que suministre 3.3V como por ejemplo un Basic FTDI USB-to-Serial Converter/Programmer, Adaptador FTDI USB a serial TTL con capacidad de que suministre 3.3V como por ejemplo este amazon

Las correspondencias los pines del microcontrolador ESP8266 con los pines de los del adaptador son:

- Pin 3v3 del ESP8266 conectado con el pin VCC/3v3 del adaptador.
- Pin TX del ESP8266 conectado con el pin RX del adaptador.

- Pin RX del ESP8266 conectado con el pin TX del adaptador.
- Pin GND del ESP8266 conectado con el pin GND del adaptador.

Como se puede observar los pines RX y TX se conectan de forma opuesta entre el microcontrolador ESP8266 y el adaptador.

El ESP8266 tiene varios modos de arranque, el normal por defecto ejecuta el programa almacenado en la memoria flash y modo flashmode(modo UART) en el que se nos permite borrar el contenido de la memoria flash o escribir un programa en la memoria flash.

Para entrar en el modo flashmode hay que conectar el pin GPIO0 a GND cuando el ESP8266 se conecte a una fuente de 3.3V es decir cuando el adaptador esté conectado al ordenador, una vez hecho esto, observamos que el led del enchufe inteligente se encenderá con un color rojo fuerte, una vez quitemos la conexión del pin GPIO0 a GND el led seguirá con un color rojo tenue. A partir de ahí podemos utilizar el programa esptool.py para borrar la memoria flash, leer la memoria flash o escribir la memoria flash.

Copia de seguridad del firmware fabricante Antes de realizar el reemplazo de el firmware del fabricante por el firmware alternativo es deseable realizar una copia de seguridad del firmware del fabricante guardado en el enchufe inteligente para en casos de querer volver a usar el firmware del fabricante y dejar de usar el firmware alternativo.

Primero procederemos a iniciar el enchufe inteligente en flashmode realizando los pasos que se indicaron en el apartado de A.3, una vez en que el enchufe inteligente esté en flashmode para determinar cuánto ocupa la imagen de la que vamos a realizar la copia de seguridad ejecutamos el comando 6.

```
1 esptool.py flash_id
```

Listing 6: Commando para examinar informacion del firmware

Con el comando 6 automáticamente se conectará al dispositivo que hay conectado al ordenador y mostrará en la ventana de comandos la información sobre el microcontrolador ESP8266 como el fabricante, el puerto serie en el que está conectado y el tamaño de la memoria flash.

Una vez que hemos identificado el tamaño de la memoria flash y el puerto en el que está conectado el enchufe inteligente, reiniciamos el enchufe inteligente y de nuevo lo iniciamos en modo flashmode. Debemos ejecutar otro comando 7 para poder realizar la copia de

seguridad especificando en las opciones del comando el puerto en el que está conectado el enchufe inteligente al ordenador y el tamaño de la memoria flash. Para Linux los puertos normalmente están en la dirección /dev/ttyUSB* y para Windows en COM*.

```
1 esptool.py --port /dev/ttyUSB0 read_flash 0x00000 0x100000 esp-1
    MB-backup.bin
```

Listing 7: Commando para restaurar la copia de seguridad del firmware de fabricante

El comando varía según el puerto serie y el tamaño de memoria flash a la que se hará una copia de seguridad y el nombre que le queramos dar al fichero donde se guardará la copia de seguridad. El comando usado aquí es para Linux pero en Windows solo hay que reemplazar /dev/ttyUSB* por COM*. Una vez ejecutado el comando esptool.py nos devuelve información del estado de la ejecución del comando y al finalizar ya tendremos el fichero con la copia de seguridad.

Si queremos restaurar la copia de seguridad basta con volver a conectar el enchufe inteligente al ordenador e iniciarlo en flashmode y ejecutar el comando 8 introduciendo el puerto serie y el fichero de la copia de seguridad como opciones en el comando.

```
1 esptool.py --port /dev/ttyUSB0 write_flash 0x00000 esp-1MB-
    backup.bin
```

Listing 8: Commando para restaurar la copia de seguridad del firmware de fabricante

Realización del reemplazo por el firmware alternativo Para realizar el reemplazo por el firmware alternativo primero debemos de descargar el archivo binario correspondiente a el enchufe inteligente, en este caso vamos a instalar el firmware alternativo ESPurna, para ello hay que dirigirse a la página de las publicaciones de versiones de ESPurna a la siguiente pagina web, https://github.com/xoseperez/espurna/releases. La versión que se está usando para este trabajo de fin de grado es la versión [1.13.5] 2019-02-27.

Esta página web nos muestra todos los binarios precompilados para todos los dispositivos que ESPurna soporta. En este caso como se mencionó en el apartado 6.1.2 del apéndice, existen dos revisiones hardware de el mismo enchufe inteligente, dependiendo de cual tengamos, descargamos el binario correspondiente, por tanto debemos elegir entre el binario espurna-*-blitzwolf-bwshpx.bin o espurna-*-blitzwolf-bwshpx-v23.bin según la versión de revisión del enchufe inteligente que tengamos.

```
1 esptool.py --port /dev/ttyUSB0 erase_flash
```

Listing 9: Comando para formatear la memoria flash

Una vez descargado el archivo binario hay que conectar el enchufe inteligente al ordenador e iniciarlo en modo flashmode siguiendo los pasos del apartado 6.1.2, una vez iniciado en dicho modo, hay que borrar primero el contenido de la memoria flash, esto se realiza con el fin de tener una memoria flash vacía sin restos de los datos de el firmware del fabricante y que pueden dañar la instalación del nuevo firmware alternativo. Para ello hay que ejecutar el comando 9

En la opción port especificamos el puerto serie por el que el enchufe inteligente está conectado, para Linux normalmente es /dev/ttyUSB* y para Windows es COM*. Una vez que hemos borrado la memoria flash, hay que instalar el firmware alternativo en el enchufe inteligente, para ello de nuevo debemos reiniciar el dispositivo en modo flashmode y ejecutar el comando 10

```
1 esptool.py --port /dev/ttyUSB0 write_flash --flash_mode dout 0
    x00000 espurna-1.13.5-blitzwolf-bwshpx.bin
```

Listing 10: Comando para instalar el firmware de Espurna

Igual que se indicó anteriormente en la opción port debe coincidir con el puerto serie en el cual el enchufe inteligente está conectado y el nombre del archivo del firmware debe ser el correspondiente al que se ha descargado una vez ejecutado el comando y tras acabar esptool.py nos indica que ha acabado tal y como se puede apreciar en la figura 30, finalmente podemos realizar la primera configuración inicial del enchufe inteligente.

Configuración inicial de ESPurna Una vez que ya hemos reemplazado el firmware del fabricante por el alternativo, ESPurna, hay que ensamblar de nuevo el enchufe inteligente y conectarlo a la red eléctrica, una vez esté conectada podemos realizar la primera configuración inicial tanto desde un móvil o desde un ordenador con una tarjeta red que soporte WIFI 2.4GHz.

El enchufe inteligente una vez conectado a la red eléctrica iniciara en modo AP(Punto de acceso Inalámbrico), para poder configurarlo hay que acceder a dicho punto de acceso, este punto de acceso tendrá un nombre con formato similar a “ESPURNA-XXXXXX”.

```
➜ Downloads sudo esptool.py --port /dev/ttyUSB0 write_flash --flash_mode dout 0
x00000 espurna-1.13.5-blitzwolf-bwshpx.bin
esptool.py v2.6
Serial port /dev/ttyUSB0
Connecting....
Detecting chip type... ESP8266
Chip is ESP8266EX
Features: WiFi
MAC: dc:4f:22:36:16:ba
Uploading stub...
Running stub...
Stub running...
Configuring flash size...
Auto-detected Flash size: 1MB
Compressed 492880 bytes to 352587...
Wrote 492880 bytes (352587 compressed) at 0x00000000 in 31.3 seconds (effective
125.9 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
➜ Downloads
```

Figura 30: Captura de pantalla de la ejecución del comando para instalar el firmware alternativo en el enchufe inteligente

La clave para acceder al punto de acceso es "fibonacci", una vez se ha accedido al punto de acceso, se debe de utilizar un navegador para acceder a "http://192.168.4.1" o a "http://192.168.244.1", tras esto, para acceder se debe introducir el usuario "admin" y la clave "fibonacci". Después ESPurna nos avisará que hay que cambiar la contraseña por defecto por otra una propia nuestra, finalmente una vez que la cambiamos debemos de nuevo volver a autenticarnos como "admin" y usando la nueva clave introducida. Una vez dentro se nos presenta una página web con la que configurar el enchufe y ver los datos del enchufe.

Para realizar la conexión del enchufe inteligente a una red wifi debemos primero acceder al punto de acceso del enchufe inteligente y entrar en la página web de configuración del enchufe inteligente a través de "http://192.168.4.1" o a través de "http://192.168.244.1", una vez dentro acceder al menú de wifi pinchando desde la barra lateral izquierda se nos presentará un menú de configuración. En esta página podemos realizar un escaneo de los puntos de acceso wifi disponibles alrededor de el enchufe inteligente.

Para añadir una red wifi que deseemos debemos pulsar en el botón "Add Network", tras pulsarlo, se añadirán nuevos campos donde introducir la información como se puede observar en la figura 31 sobre nuestra red wifi una configuración recomendada es usar una IP estática puesto que siempre debemos acceder a el enchufe inteligente y gracias a el uso de una IP estática, esta no cambiará y será para siempre asignada a el enchufe inteligente.

Una vez configurado la red wifi a usar, hay que pulsar el botón "Save" y tras ello, finalmente el enchufe inteligente estará configurado para que se conecte a esa red wifi siempre.

Network SSID

Password

Static IP 192.168.1.108

Leave empty for DHCP negotiation

Gateway IP 192.168.1.1

Set when using a static IP

Network Mask 255.255.255.0

Usually 255.255.255.0 for /24 networks

DNS IP 8.8.8.8

Set the Domain Name Server IP to use when using a static IP

Delete network

Figura 31: Configuración de una red wifi en ESPurna

Configuración de los sensores Debemos realizar un cambio en las unidades de medida de el enchufe inteligente para ello hay que entrar en la pestaña de configuración de los sensores "SENSORS" y cambiar la unidad de medida de Energía a kWh (por defecto en Julios). Además podemos configurar también el intervalo en el que el enchufe inteligente realiza una medición con los sensores y el intervalo de tiempo en el que debe de transcurrir entre reportes de estas medidas captadas a través de los sensores. Debe de quedar como la figura 32

General

Read interval 6 seconds

Select the interval between readings. These will be filtered and averaged for the report. Please mind some sensors do not have fast refresh intervals. Check the sensor datasheet to know the minimum read interval. The default and recommended value is 6 seconds.

Report every 10

Select the number of readings to average and report

Save every 0

Save aggregated data to EEPROM after these many reports. At the moment this only applies to total energy readings. Please mind: saving data to EEPROM too often will wear out the flash memory quickly. Set it to 0 to disable this feature (default value).

Power units Kilowatts (kW)

Energy units Kilowatts·hour (kWh)

Figura 32: Configuración de unidad de medida de los sensores

Conexión a broker MQTT Para que el enchufe inteligente se conecte con el Broker del sistema debemos ir a la configuración de MQTT y activarla, después hay que cambiar el *root topic* por {mac}. Tal y como se muestra en la figura 33.

MQTT QoS 0: At most once

MQTT Retain NO YES

MQTT Keep Alive 300

MQTT Root Topic {mac}

This is the root topic for this device. The {hostname} and {mac} placeholders will be replaced by the device hostname and MAC address.

- **<root>/relay/#/set** Send a 0 or a 1 as a payload to this topic to switch it on or off. You can also send a 2 to toggle its current state. Replace # with the switch ID (starting from 0). If the board has only one switch it will be 0.
- **<root>/status** The device will report a 1 to this topic every few minutes. Upon MQTT disconnecting this will be set to 0.
- Other values reported (depending on the build) are: **firmware** and **version**, **hostname**, **IP**, **MAC**, signal strenth (**RSSI**), **uptime** (in seconds), **free heap** and **power supply**.

Figura 33: Configuracion de conexion al broker MQTT

6.2. Detalles de implementación

Esta apartado tiene como propósito discutir algunos detalles sobre la implementación del sistema desarrollado.

6.2.1. Repositorios del código e imágenes Docker

Hay varias maneras de organizar el código de un proyecto software, en forma de un repositorio monolítico o en varios repositorios, esto permite separar componentes del sistema, además la configuración de CI/CD se hace menos complicada al no tener que distinguir entre diferentes componentes dentro de un repositorio monolítico.

Los repositorios que alojan el código son:

- tfg_backend: Es el repositorio que contiene el código de el servidor de *backend*, el servidor que sirve la API REST.
- tfg_frontend: Es el repositorio con el código del servidor que sirve *frontend*.
- tfg_dashboard: Es el repositorio con el código del servidor que sirve el *dashboard*, el servidor de SocketIO.
- tfg_system: Es el repositorio que contiene el fichero Docker-compose.yaml y las configuraciones de Kapacitor, Eclipse mosquitto y Telegraf usado para desplegar el sistema completo desde cero y mantenerlo.

Las imágenes Docker correspondientes a cada componente del sistema desarrollado son:

- tfgbackend

- tfgdashboard
- tfgfrontend

Las imágenes de Docker de cada componente tienen una tag rasp, esta tag indica que es una tag para usarse en la Raspberry Pi, las demás tags son para Linux de 64 bits. Travis CI permite realizar builds en ARM-64 pero debido a que es una característica aún en alfa, se ha tenido que realizar los despliegues desde la Raspberry Pi utilizada para desplegar los contenedores del sistema. Una posible línea futura es utilizar otro servicio de CI/CD que permita hacer builds de arm64 o esperar a que esta característica salga de la versión alfa. Otra punto a tener en cuenta tiene que ver con el apartado 6.2.2, la imagen del componente frontend con la tag rasp tiene las direcciones ip fijas.

6.2.2. Ip fija entre contenedores del sistema

Uno de los primeros problemas que se ha presentado en el proyecto es el hecho de que las Ip de cada red privada de un hogar puede cambia, lo ideal habría sido utilizar un DNS para resolver las Ip de los contenedores del sistema aunque por limite de tiempo no se ha llegado a implementar un tercer contenedores que sirva para proxy entre el sistema de contenedores dentro de la Raspberry Pi y el navegador web del cliente, con lo cual las Ip están fijas en todo el desarrollo del sistema, incluso fijada en código, en el código del servidor web que sirve el *frontend*, tanto la dirección ip del servidor backend y del servidor de dashboard están fijadas en el código.

6.2.3. Task template

Como ya se ha comentado en el apartado 4.5.2, se ha tenido que definir un task template para crear de forma automática task asociado a los influxTriggers de una automatización. El task template desarrollado contiene varias variables de entrada El script empieza recogiendo datos desde InfluxDB, más en concreto desde un *measurement* pasado como variable de entrada, en este caso en la aplicación, el *measurement* donde recoge información puede ser «MQTT_consumer» o «device_gasto». Se define el nodo data que es un stream/flujo de datos donde recoger información del *measurement* de InfluxDB, la variable «where_filter» se usa aquí para seleccionar qué topic de MQTT filtrar, un ejemplo es el de filtrar solo el MQTT topic del parámetro energy. {*root topic*}*/energy*. Esos datos

pasan en forma de flujo de datos/stream hacia el nodo de alerta, en el template tenemos tres nodos de alerta, esto nodos de alerta utiliza la variable de entrada «crit», que consiste en un lambda expresion de los tick script para indicar cuando una alerta debe de saltar. Un ejemplo de expresión lambda se puede comprobar en la linea 5 del código 11 en el comentario sobre la variable crit. De los tres nodos de alerta el más importante es el de **post HTTP**, este es el que realiza la petición HTTP POST hacia el URI path */:id/fire* para que la automatización ejecute las acciones que tiene asociadas a ella. Los otros dos nodos de alerta sirven para para alertar por telegram y para guardar en un fichero de logs las alertas que se hayan disparado durante todo el tiempo que el task haya estado habilitada y corriendo.

```
1  // Which measurement to consume
2  var measurement string
3  // Where filter for selecting the topic (must be defined upon
     creating a task for automation) ex: lambda: "topic" == 'SHIHO
     -SWITCH2'
4  var where_filter lambda
5  // Critical criteria for alerting. Ex: lambda: "value" > 60
6  var crit lambda
7  // Message used for telegram on alert
8  var message_telegram string
9  // Message used for HTTP POST, should be a json stringified
10 var message_httpost string
11 // URL to post to
12 var url = 'http://app:8080/'
13 // Name of the log file
14 var log_filename = 'value.log'
```

Listing 11: Comando para instalar el firmware de Espurna

Este template está incrustado en el código del backend por conveniencia aunque podría haberse guardado como un fichero y que el propio servidor backend lea dicho template. En el proceso de inicio del backend, el backend crea el template en Kapacitor por medio de una petición POST usando su API REST. El proyecto de el backend contiene un script en python para pasar el tick script a un string de una línea para que pueda ser enviado

por medio de un POST hacia Kapacitor.

En la figura 10 es el grafo de cómo fluye los datos en el task template desarrollado, el nodo stream0 representa el origen de los datos que se procesan en el task template como se explicó antes, consiste en un nodo de datos, el nodo from permite filtrar los datos que pasarán al siguiente nodo, en este caso se filtraran por measurement y por la expresión utilizada en el where. Por último los tres últimos nodos, son los nodos de alerta. Ambos disparará una alerta si pasan ciertas condiciones, en este caso estas condiciones son las lambda expresión que pasamos como variable de entrada.

Índice de cuadros

1. Diferencias técnicas entre los enchufes inteligentes seleccionados 17

2. Comparacion de *topics* de ESPurna con *parameters* de devices en la aplicación 43

Índice de figuras

1. Radovici, Culic y Rusu [17] Diagrama de componentes de un sistema de Internet de las Cosas . . . 9
2. Tablero Kanban utilizado durante el desarrollo . . . 14
3. diagrama fisico del sistema diseñado . . . 22
4. InfluxData [11] Diagrama de las aplicaciones del TICK Stack, describiendo cómo las aplicaciones interactúan entre sí . . . 26
5. Diagrama de secuencia de el escaneo de enchufes inteligentes . . . 31
6. Diagrama de secuencia de el registro de enchufes inteligentes . . . 32
7. Diagrama de clases de Schedule y Scheduler . . . 34
8. Diagrama de clases de la interfaz ITrigger y las clases influxTrigger y Kapacitor . . . 36
9. Diagrama flujo de acciones e interacciones entre Kapacitor, el *backend* y los enchufes inteligentes una vez que se ha disparado una alerta desde Kapacitor 37
10. Grafo de flujo de los datos del task template desarrollado . . . 38
11. Diagrama de secuencia sobre cómo el backend utiliza la información de la API de la Red Eléctrica Española y crea un Schedule que despues inicia en el Scheduler . . . 38
12. Diagrama de secuencia de las acciones que se realizan una vez que un Schedule se dispara y ejecuta las actions asociadas a una automatización . 39
13. Diagrama de secuencia de acciones e interacciones entre el *PriceCalculatorPooler*, la clase de *esios* y InfluxDB cada vez que el pooler ejecuta el método intervalHandler . . . 41
14. Diagrama de clases de devices y devicesGroup . . . 42
15. Diagrama de clases de automatization y triggers . . . 44
16. Diagrama de clases de dashboard . . . 45
17. Diagrama de clases de de los ajustes de la aplicación . . . 46
18. Captura de pantalla de una gráfica de barras . . . 49
19. Captura de pantalla de una gráfica de area . . . 49
20. Captura de pantalla de una gráfica de lineas . . . 49
21. Diagrama de los diferentes componentes del sistema y cómo interactúan con ellos . . . 50

22. Diagrama de flujo de las acciones que realiza el servidor de Socket.IO una vez que un cliente se conecte a él 51
23. Diagrama de flujo desde que una room empieza a enviar a intervalos, los datos de las query de los devices y parámetros en una dashboard 52
24. Diagrama de clases de DashboardManager y Room del servidor Socket.IO y DashboardApp del cliente Socket.IO en el frontend 52
25. Diagrama de secuencia de DeviceStatusPooler con la ejecución de acciones que realiza cada vez que tiene que comprobar el estado de los enchufes inteligentes 53
26. Diagrama de clases de DeviceStatusPooler 54
27. Tornillos del enchufe inteligente BLITZWOLF 71
28. Localización del microcontrolador ESP8266 72
29. Pines de el microcontrolador ESP8266 que deben ser conectados 73
30. Captura de pantalla de la ejecución del comando para instalar el firmware alternativo en el enchufe inteligente 77
31. Configuración de una red wifi en ESPurna 78
32. Configuración de unidad de medida de los sensores 78
33. Configuracion de conexion al broker MQTT 79

Listings

1. Comando para actualizar Ubuntu 69
2. Comando para listar regiones y cambiar la configuración regional 69
3. Commando hello world de Docker 69
4. Commandos para instalar Docker-compose en Ubuntu 70
5. Comando para desplegar todo el sistema y contenedores 70
6. Commando para examinar informacion del firmware 74
7. Commando para restaurar la copia de seguridad del firmware de fabricante 75
8. Commando para restaurar la copia de seguridad del firmware de fabricante 75
9. Comando para formatear la memoria flash 76
10. Comando para instalar el firmware de Espurna 76
11. Comando para instalar el firmware de Espurna 81

Referencias

[1] Rob Barton, Gonzalo Salgueiro y David Hanes. *IoT Fundamentals: Networking Technologies, Protocols, and Use Cases for the Internet of Things*. Cisco Press, 2017.

[2] Charles Bell. *MySQL for the Internet of Things*. Apress, 2016.

[3] Kristina Chodorow, Eoin Brazil y Shannon Bradshaw. *MongoDB: The Definitive Guide, 3rd Edition*. O'Reilly Media, Inc., 2019.

[4] Brajesh De. *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*. Apress, 2017.

[5] devttys0. *Hacking the D-Link DSP-W215 Smart Plug*. URL: `http://www.devttys0.com/2014/05/hacking-the-d-link-dsp-w215-smart-plug/`.

[6] devttys0. *Hacking the D-Link DSP-W215 Smart Plug*. URL: `http://www.devttys0.com/2014/05/hacking-the-d-link-dsp-w215-smart-plug/`.

[7] Fernando Doglio. *REST API Development with Node.js : Manage and Understand the Full Capabilities of Successful REST Development*. Apress, 2018.

[8] Ted Dunning y Ellen Friedman. *Time Series Databases: New Ways to Store and Access Data*. O'Reilly Media, Inc., 2014.

[9] Marcus Hammarberg y Joakim Sundén. *Kanban in Action*. Manning Publications, 2014.

[10] Qusay F. Hassan. *Internet of Things A to Z*. Wiley-IEEE Press, 2018.

[11] InfluxData. *InfluxDB 1.x*. URL: `https://www.influxdata.com/time-series-platform/`.

[12] InfluxData. *Time series database (TSDB) explained*. URL: `https://www.influxdata.com/time-series-database/`.

[13] Stephen Kuenzli y Jeffrey Nickoloff. *Docker in Action, Second Edition*. Manning Publications, 2018.

[14] Brent Laster. *Continuous Integration vs. Continuous Delivery vs. Continuous Deployment*. O'Reilly Media, Inc., 2017.

[15] Patrick Mulder y Kelsey Breseman. *Node.js for Embedded Systems*. O'Reilly Media, Inc., 2016.

[16] Tim Pulver. *Hands-On Internet of Things with MQTT*. Packt Publishing, 2019.

[17] Alexandru Radovici, Ioana Culic y Cristian Rusu. *Commercial and Industrial Internet of Things Applications with the Raspberry Pi: Prototyping IoT Solutions*. Apress, 2020.

[18] Mayur Ramgir. *Internet of Things*. Pearson Education India, 2019.

[19] Andrew Stellman y Jennifer Greene. *Learning Agile*. O'Reilly Media, Inc., 2014.

[20] Lubomir Stroetmann y Tobias Esser. *Reverse Engineering the TP-Link HS110*. URL: https://www.softscheck.com/en/reverse-engineering-tp-link-hs110/.

[21] Vasan Subramanian. *Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node*. Apress, 2019.

[22] Dan Vanderkam. *Effective TypeScript*. O'Reilly Media, Inc., 2019.

[23] Jim Wilson. *Node.js 8 the Right Way*. Pragmatic Bookshelf, 2018.

UNIVERSIDAD DE MÁLAGA | uma.es

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

E.T.S. DE INGENIERÍA INFORMÁTICA