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

Integración y gestión de un nuevo tipo de dispositivo Raspberry en el IoT Server de WSO2

Autor: Luis Valencia Pichardo
Tutor: Isabel Román Martínez

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

Sevilla, 2019

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

Integración y gestión de un nuevo tipo de dispositivo Raspberry en el IoT Server de WSO2

Autor:
Luis Valencia Pichardo

Tutor:
Isabel Román Martínez
Profesora colaboradora

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2019

Trabajo Fin de Grado: Integración y gestión de un nuevo tipo de dispositivo Raspberry en el IoT Server de WSO2

Autor: Luis Valencia Pichardo

Tutor: Isabel Román Martínez

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2019

El Secretario del Tribunal

A mi familia

A mis maestros

A mis compañeros y amigos

Agradecimientos

En primer lugar, a mi familia, compañeros y amigos que durante este camino me ha apoyado y han creído en mi, pues sin su apoyo ni su ayuda no me hubiera sido posible llegar a este lugar.

En segundo lugar, a mis profesores, sin los cuales no habría llegado a apreciar el camino recorrido, ni a mejorar como persona, pues como dijo Einstein *"El arte supremo del maestro consiste en despertar el goce de la expresión creativa y del conocimiento."*

Luis Valencia Pichardo

Sevilla, 2019

Resumen

Hoy en día, existe una gran variedad de dispositivos y tecnologías basadas en el “internet de las cosas”. Gracias a ello, se tiene de forma económica, simple y flexible una gran cantidad de opciones para afrontar retos y problemas del día a día. Por ejemplo, podemos convertir un dispositivo Raspberry Pi en un sensor con capacidad para conectarse e interactuar con otros sistemas.

Uno de los conceptos que debemos desarrollar, es el concepto de agente. Un agente es aquel programa o software, que actúa en representación de otra entidad o programa actuando conforme a lo dictaminado por la otra entidad, como por ejemplo la recolección y transmisión de datos.

El objetivo de este trabajo de fin de grado consiste en desarrollar un agente que se implementará en una Raspberry Pi que contiene tanto un sensor de temperatura como una matriz de leds. El papel de la entidad gestionadora la tomará el servidor que nos ofrece WSO2 para el “Internet de las cosas”, a pesar de ser este un caso de uso específico, cabe recalcar que una vez realizado el trabajo y obtenido un resultado, es altamente extrapolable a distintos sensores, ya sea para fines domóticos, medicinales, recreativos, etc.

Por último, destacar el uso de tecnologías, librerías y estándares definidos para este tipo de sistemas, como por ejemplo el protocolo MQTT.

Abstract

Nowadays there are a lot of devices and technology based on the Internet of Things. For this reason, we have at hand many cheap, simple and flexible approach for most of the problems that we can face in our daily life. Based on this, we can use for example, a Raspberry Pi device, and obtain sensor capabilities and connect it with other systems.

One of the concepts that we will need to develop is the word agent. An agent is a software code which will act in accordance with other entity, for example retrieving and transmiting data.

The objective of this “Trabajo de fin de grado” is to develop a Raspberry Pi agent with a temperature sensor and a Leds matrix to connect it with the server developed by WSO2 for “Internet of Things” and use many of the technology, protocols and specifications available for this subject. Even the specific scope of this project, it is usefull to say that it is meant to be exported to other cases as domotic, health care, etc.

As a last comment, it is important to highlight the usage of multiple libraries, standards and technologies as it is the MQTT protocol.

Índice

ÍNDICE DE FIGURAS

ÍNDICE DE CÓDIGO

Notación

`código`	Código de un fichero
Inglés	Palabra en inglés
`comando`	Comando de bash
Fichero	Nombre de fichero

1 INTRODUCCIÓN

Nuestra recompensa se encuentra en el esfuerzo y no en el resultado. Un esfuerzo total es una victoria completa.

- Mahatma Gandhi-

Desde que en 1999 en el Instituto Tecnológico de Massachusetts (MIT) [1] se definiera el concepto de internet de las cosas, se han ido desarrollando distintas tecnologías destinadas a facilitar el desarrollo de software y estandarizar grandes aspectos del internet de las cosas, así como dispositivos cada vez más económicos y con mejores especificaciones. Como ejemplo de ello, podemos poner uno de los más famosos dispositivos, la Raspberry Pi, que desde que fuera lanzada en 2012 por la Fundación Raspberry Pi [2], con el objetivo de destinarlo a la enseñanza, ha pasado de tener un procesador *single-core* a 700 MHz (en la primera versión) a un procesador de nueva generación con 1,4GHz (en su versión 3).

Que estos dispositivos tengan una gran compatibilidad con distintos sensores y puedan correr sistemas operativos basados en Linux, ha hecho que surja una gran comunidad en internet dedicada a seguir y publicar distintos proyectos que usen esta plataforma.

Por otra parte, WSO2 [3], es una empresa especializada en software de código abierto, que está enfocada en facilitar la gestión de sistemas basados en *SOA* [4]. Entre sus productos de código abierto, se encuentra un servidor enfocado en la temática de *IoT* que nos otorga la flexibilidad necesaria para usarlo en la gestión y monitorización de dispositivos, entre ellos Raspberry Pi.

Este sector se encuentra en auge, tiene una gran flexibilidad de opciones y un enorme espacio de aplicación para ayudar en el día a día de las personas. Estas y otras son las razones de ser de este trabajo de fin de grado, que se centra en el uso de dispositivos Raspberry Pi como plataforma a gestionar y monitorizar mediante un servidor proporcionado por WSO2.

1.1 Motivación

Este trabajo de fin de grado, surge de la curiosidad propia por gestionar uno o múltiples dispositivos Raspberry Pi con algún sistema *Front End,* para lo que se ha elegido un sistema ya maduro e innovador como es WSO2.

El caso de uso elegido es el de control remoto y en tiempo real de la temperatura de un entorno. Este sería aplicable tanto si se necesita poca precisión, como por ejemplo un hogar, como cuando se necesite un alto nivel de precisión en la medida, como sería por ejemplo la monitorización de la temperatura de un ser vivo.

A pesar de que el alcance del proyecto quede limitado a este ámbito, cabe destacar que el resultado es totalmente exportable a otros tipos de sensores con un mínimo esfuerzo de desarrollo.

1.2 Objetivos

El objetivo a perseguir por este trabajo de fin de grado es la integración y gestión de un nuevo tipo de dispositivo Raspberry pi para el IoT Server de WSO2, así como el desarrollo de un agente Raspberry pi donde se implementará toda la lógica de sensores.

El escenario de pruebas se situará en un entorno real en el hogar, donde el sensor pueda controlar durante un periodo de tiempo extenso la temperatura y mandar la información al servidor. También podremos controlar el dispositivo desde el servidor; reiniciándolo, configurando el tiempo de muestreo o apagando/encendiendo la matriz de leds.

1.3 Metodología

Para el desarrollo del proyecto, vamos a necesitar un conjunto de elementos que vamos a enumerar y detallaremos en secciones posteriores. Para el código vamos a necesitar un IDE potente y multipropósito, ya que usaremos una gran variedad de lenguajes como Java, Javascript, Json, Handlebars y Python.

Usaremos Java, Javascript y Json para la parte del servidor, ya que WSO2 está implementado en Java. Usaremos Javascript para tareas en la parte del cliente, mientras que Handlebars [4] es el esquema usado junto con Jaggery [5] por WSO2 para la interfaz de usuario del servidor.

Python será usado en el módulo del agente en la Raspberry Pi, puesto que es un lenguaje a la vez potente y sencillo de aprender, así como multiplataforma y que cuenta con una extensa documentación y librerías, incluyendo las necesarias para la gestión de sensores.

El proyecto constará de dos partes bien distinguidas, por un lado, el complemento a desarrollar en el servidor de WSO2 y en el otro lado, el agente Raspberry Pi, para el que se ha elegido una estructura multihilos. Uno de los hilos, el principal, se encargará del arranque, recolección de datos de sensores y su envío al servidor, mientras que un segundo hilo estará suscrito al topic MQTT, quedando a su cargo el tratar las operaciones entrantes y actuar en consecuencia.

Para el proyecto hemos utilizado los siguientes elementos:

- Raspberry Pi 2
  - Sistema Operativo: Raspbian 8 (Jessie) con Kernel: 4.1.7+.
  - Entorno de ejecución: Terminal Unix de comandos Bash.
  - Intérprete: Python 2.7.13
- Portátil Lenovo B590 – Servidor y entorno de desarrollo
  - Sistema Operativo: Debian 9 (Stretch) con Kernel 4.9.110.
  - Terminal avanzado: Guake con Tmux y zsh + powerline (con implementación de git) con atajos.
  - Editor de texto: Vim, configuración avanzada con plugins (nerdtree, airline, emberhbs)
  - IDE: Atom versión 1.29.0

1.4 Estructura de la Memoria

Esta memoria se dividirá en 6 bloques, partiendo de esta misma introducción, pasando a descripción del sistema y herramientas, trabajo previo, trabajo realizado, pruebas y verificación, para terminar con las conclusiones finales.

En el apartado 1, Introducción, trataremos información sobre el objetivo, motivo y alcance del proyecto.

En el apartado 2, Descripción del sistema y herramientas, hablaremos brevemente del servidor, de las

tecnologías utilizadas en este proyecto, así como de las herramientas utilizadas en el desarrollo.

En el apartado 3, Trabajo previo, se explicará aspectos de la configuración de la Raspberry pi, sus sensores y del servidor WSO2 IoT.

En el apartado 4, Desarrollo de la solución, se analizará en profundidad el código desarrollado y usado en el proyecto.

En el apartado 5, Pruebas y validación, pasaremos a analizar las distintas pruebas de validación realizadas.

En el apartado 6, Conclusiones finales, haremos un análisis final del proyecto y comentaremos posibles mejoras y desarrollos futuros.

2 ESTADO DE LA TÉCNICA Y HERRAMIENTAS

Una experiencia nunca es un fracaso, pues viene a demostrar algo.

- Thomas Alva Edison -

El diseño de un sistema agente-servidor, con una plataforma como la desarrollada por WSO2 para el internet de las cosas, nos lleva a tener que usar una variedad bastante compleja de tecnologías y lenguajes de programación. En esta sección, recorreremos los distintos elementos que componen el servidor, también describiremos los métodos de interacción entre agente y servidor, así como varias tecnologías que nos serán de utilidad para apartados posteriores.

Como ya se ha comentado, una parte principal en este proyecto, es la comunicación entre el agente y el servidor, pues de ahí sale la información útil para el servidor y las órdenes a ejecutar por el agente, esto se logra gracias a un conjunto de protocolos, librerías y estilos que son usados por WSO2 para ese fin.

Queremos que el agente sea lo más simple posible, para poder ejecutarlo de forma segura en dispositivos con baja capacidad de procesamiento, mientras que el servidor tiene unas prestaciones mucho mayores, lo que lleva a que necesitemos un hardware más exigente.

2.1. Arquitectura del Servidor WSO2 IoT

El servidor de WSO2 tiene unos requisitos mínimos bastante exigentes, a continuación, se muestran los requisitos necesarios para poder desplegarlo:

- Memoria RAM: 8GB con 1024MB de tamaño de pila para Java
- Procesador: *Dual Core* o superior.
- Disco Duro: 5GB mínimo.

Como vemos, las características nos llevan a buscar un hardware potente para ello. En el caso de este trabajo fin de grado, para simular un escenario real, se ha usado como plataforma para el servidor un portátil, que cuenta con suficiente capacidad; dado que no es el hardware más indicado para ejercer de servidor, se propone usar para un despliegue real un *barebone* [6] con las características siguientes:

- 8Gb de RAM
- 128Gb de SSD
- Procesador I5 3317U
- Rango de precios: 350 – 400€
- Wifi: Antena dual 802.11 b/g/n

Figura 2.1 - Barebone

La arquitectura del servidor WSO2 se muestra en la figura 2.2, donde podemos distinguir 4 bloques principales que detallaremos:

- *Connected Device Management Core* – Núcleo de gestión de dispositivo conectado
- *API Layer* – Capa de API
- *Security and Integration Layer* – Capa de integración y seguridad
- *Interaction Layer* – Capa de interacción

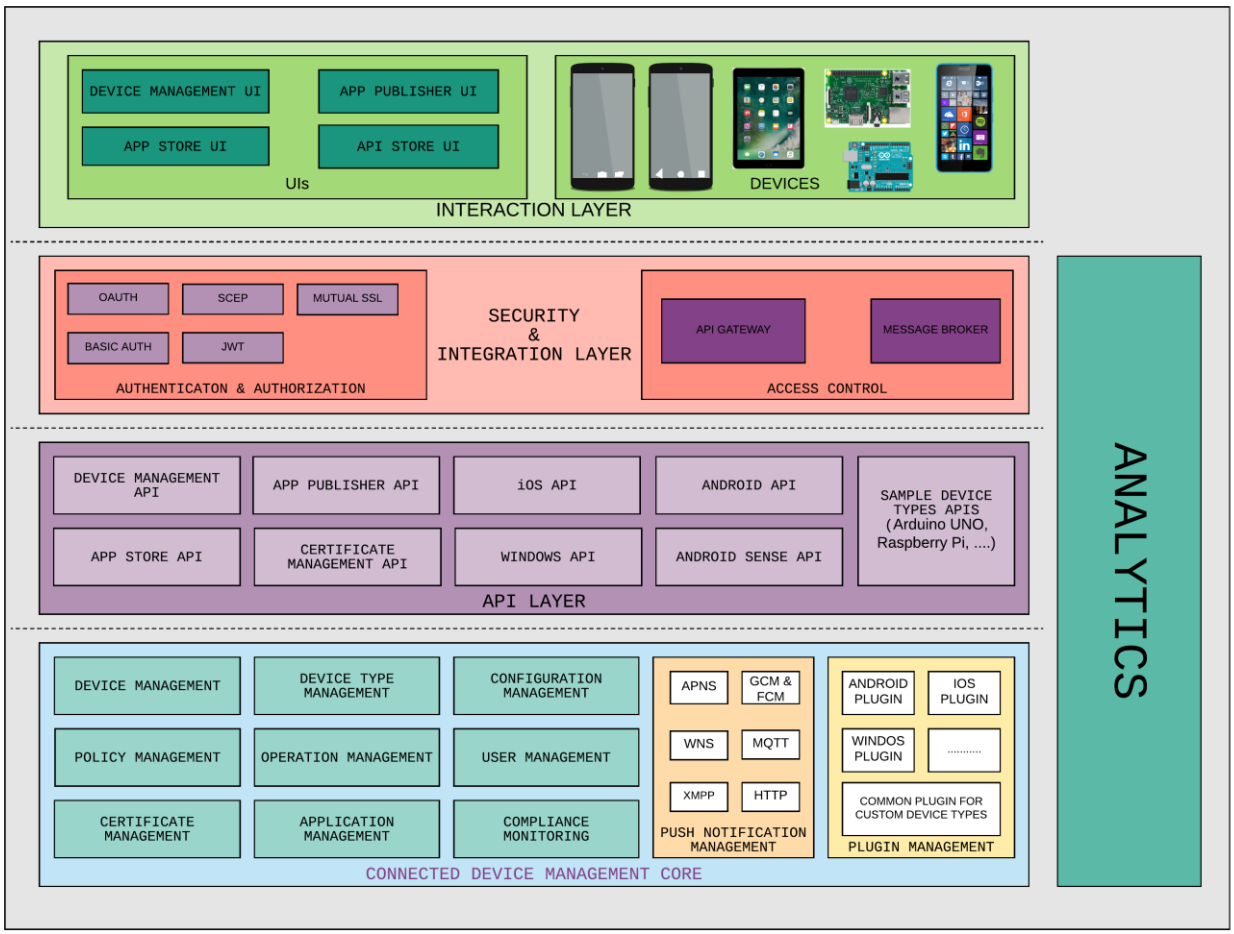


Figura 2.2 – Arquitectura del servidor IoT de WSO2

2.1.1 Connected Device Management Core – Núcleo de Gestión de Dispositivo Conectado

Es el núcleo del servidor, se encarga de controlar todas las funcionalidades que éste tiene. Es el cerebro del servidor y a su vez se divide en distintos módulos, que son totalmente extensibles comentaremos los que más relación tienen con este proyecto.

2.1.1.1 Device Management – Gestión de Dispositivo

Es el módulo encargado de registrar/eliminar los dispositivos en el servidor. Gestiona la información del dispositivo, como el estado.

2.1.1.2 Device Type Management – Gestión de Tipo de Dispositivo

Tiene la capacidad de añadir/retirar nuevos tipos de dispositivos en el servidor, permite describir el tipo de dispositivo, capacidades y operaciones que van a tener y/o realizar facilitando así la gestión de dispositivos por categorías.

Para hacer uso de este módulo, tenemos el framework de gestión de dispositivos conectados (CDMF), escrito en java sobre el que se conectan los *plugin* de dispositivos, estos *plugin* son contenedores OSGi.

2.1.1.3 Operation Management – Gestión de Operación

Las operaciones son órdenes enviadas al dispositivo indicando una acción o un comportamiento determinado. Estas operaciones, son definidas y gestionadas por este módulo antes del envío al dispositivo o dispositivos en cuestión.

2.1.1.4 Push Notification Management – Gestión del Envío de Notificaciones

Permite el uso de diferentes tecnologías para la comunicación con el dispositivo (MQTT, XMPP, HTTP, WNS, GCM/FCM, APNS), en este proyecto, usaremos MQTT [7] el cual proporciona una forma simple, fiable y segura para comunicación indirecta entre dispositivos *IoT*.

2.1.2 API Layer – Capa de API

Es la parte del servidor donde se ubican las distintas API de los dispositivos, estas deben contar con la notación *swagger* [8]. Cabe destacar que todas las API están tras una capa de protección, que exige dos niveles de autorización para ser accedidas.

2.1.3 Security and Integration Layer – Capa de Seguridad e Integración

Para WSO2, la seguridad es un concepto muy importante, por lo que este módulo provee de varios métodos para autenticar y autorizar dispositivos y usuarios. Hay dos niveles de autorización cuando se quiere invocar una operación, el primero es ver si el usuario tiene permitido acceder a la API en cuestión y el segundo es comprobar si el usuario está autorizado en un dispositivo concreto.

En cuanto a la integración, se encarga de gestionar por un lado las peticiones y respuestas de HTTP de entidades con la API, por otro lado, el agente de comunicación, *message broker,* se encarga de gestionar el intercambio de mensajes MQTT entre dispositivos, asegurándose de que el mensaje llegue al conjunto de dispositivos que debería recibirlo.

2.1.4 Interaction Layer – Capa de Interacción

Esta capa se subdivide en dos, por un lado, describe como interaccionan los dispositivos con el servidor, mientras que, por otro lado, define la interacción con sistemas terceros en caso de que los haya, siendo necesaria una autenticación para ello.

2.1.5 Analytics Layer – Capa de Análisis de Datos

El análisis de datos, es una parte importante de la gestión de dispositivo, este módulo nos da la opción de gestionar los datos ya sea en tiempo real o no, para poder actuar en consecuencia. Podemos hacer esto de 3 formas distinta:

- Análisis en tiempo real, el cual nos permite tomar decisiones de forma rápida en base a datos recibidos.
- Análisis del conjunto de datos, nos permite almacenar y representar de forma resumida los datos obtenidos.
- Análisis en el dispositivo, nos da la capacidad de que el dispositivo trate los datos y comunicar los resultados al servidor.

2.2 Herramientas utilizadas

Una parte muy importante a la hora de desarrollar código, independientemente del lenguaje, es el conjunto de herramientas que se usa a lo largo de todas las etapas del proyecto. Hay una gran variedad de entornos de desarrollo, tantos como gustos. Cada entorno proporciona un conjunto de funcionalidades que lo diferencian del

resto. Para realizar este proyecto se ha usado *Atom* [9] y *Vim* [10], el primero para desarrollar a en Python, mientras que el segundo se ha usado para desarrollar distintos *Shellscripts*.

Por otro lado, hemos usado *Eclipse* [11], como plataforma de desarrollo en *Java*, *Javascript* y *Handlebars*, así como plataforma para realizar las distintas pruebas diseñadas con el fin de probar la funcionalidad del código.

Como terminal, se ha usado *Guake* [12], el cual implementa Tmux y Zsh como intérprete de comandos con Oh-My-Zsh! [13] incorporado para editar el aspecto.

Se ha usado también un gestor de versiones tan potente y usado como lo es *git* [14], subido al servidor de G*itHub*[15] (véase la URL del proyecto: https://github.com/lvalencia1/Tfg-LVP-IOT-RPI).

Por último, ha sido necesario un conjunto de *Scripts* para agilizar ciertos procedimientos que deben realizarse de forma muy continua y repetitiva, a lo largo de la fase de desarrollo.

2.2.1 Gestor de terminal Tmux y Guake

El entorno de desarrollo del proyecto ha sido una distribución Debian, con objetivo de hacer más amigable el terminal de Linux, se ha elegido usar un programa de terminal distinto al que nos ofrece por defecto, este es *Guake*.

Guake es un terminal *drop-down* que mediante la tecla F12, aparece en la mitad superior de la pantalla, siendo posible ponerlo a pantalla completa con F11, o modificar su aspecto como si fuera un terminal tradicional, quedando todo su uso, dentro del teclado y evitando al máximo el uso del ratón.

Para instalar *Guake* desde los repositorios oficiales solo hay que ejecutar el siguiente comando:

```
sudo apt-get install guake
```

Por otro lado, le incorporamos un multiplexor de terminales llamado Tmux. La razón de este multiplexor, es por la versatilidad y gran capacidad de personalización que nos otorga. En mi caso, le he configurado una serie de atajos de teclado. La forma de instalar *Tmux* también usa los repositorios oficiales.

```
sudo apt-get install tmux
```

En principio, usamos *Ctrl*+q como combinación para escapar los atajos, tal y como se muestra en la figura siguiente, usaremos '-' para dividir horizontalmente el terminal y '|' para dividir verticalmente. También se han configurado otros atajos que nos pueden ser útiles, como 'm' para activar el uso del ratón y Ctrl+l para sincronizar todas las subdivisiones del terminal.

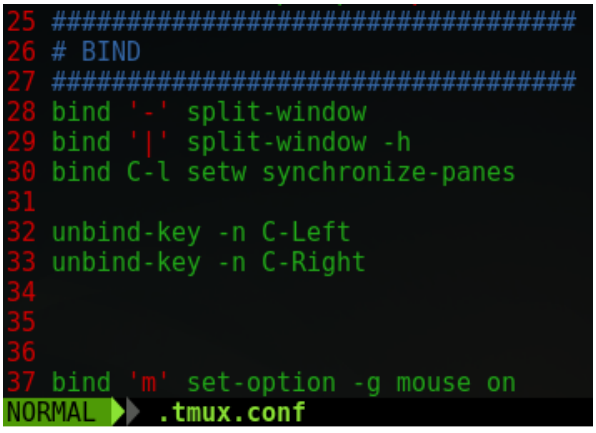


Figura 2.4 – Ejemplo de configuración de Tmux

También se ha configurado una serie de información en la parte baja de *Guake*, considerada de especial interés, como usuario e *IP*, fecha, hora y uso del procesador. Podemos ver esta información en la siguiente figura.

Figura 2.5 -Configuración inferior de Tmux

2.2.2 Intérprete de comandos Zsh

Como intérprete de comandos, se ha elegido *Zsh*, el cual nos da una serie de ventajas con respecto al intérprete tradicional que es *Bash*, estas mejoras las enumeramos a continuación:

1. Más eficiente.
2. Completado de tabulador mejorado.
3. Expansión de nombres de fichero mejorado.
4. Mejor manejo de *arrays.*
5. Totalmente personalizable.

Vamos a instalar Zsh de forma muy parecida a la que ya hemos usado con *Guake* y *Tmux*, haciendo uso de los repositorios oficiales:

```
sudo apt-get install zsh
```

También debemos configurarlo como intérprete por defecto con el siguiente comando:

```
chsh -s $(which zsh)
```

Ya tendremos configurado como intérprete por defecto *Zsh.* Se ha personalizado, para usar el framework de configuración *Oh-My-Zsh!*. La razón de usar este *framework* es que tiene una gran comunidad respaldándolo, lo que lo hace un producto bastante maduro, así mismo, tiene un estilo que es compatible con g*it*.

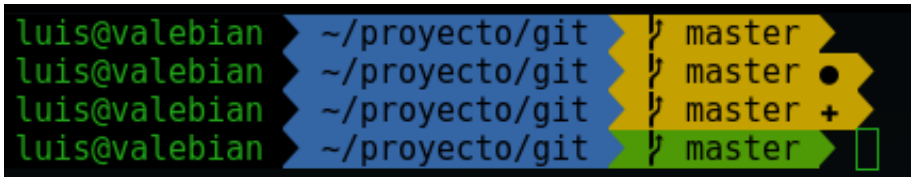


Figura 2.6 – Tema Agnoster de Oh-My-Zsh!

Como vemos en la figura anterior, nos indica ciertos estados de *git* que nos va a ser muy útil a la hora de usarlo. Analicemos estos estados en el orden mostrado:

- Rama "master", con ficheros sin supervisar, hace falta hacer un *git add*.
- Rama "master", con cambios por supervisar, hace falta hacer un *git add*.
- Rama "master", cambios añadidos y pendientes para asentir, haría falta un *git commit*.
- Rama "master" sin nada modificado.

Para instalar el *framework* ya comentado, es necesario el comando *wget;* veamos la ejecución a continuación:

```
sh -c "$(wget https://raw.github.com/robbyrussell/oh-my-
zsh/master/tools/install.sh -O -)"
```

Es necesario tener en cuenta, que ciertos estilos usan caracteres especiales, para que estos se muestren, tendremos que instalar también *Powerline* [16], podemos encontrar su instalación en el anexo de referencias.

2.2.3 Editor de texto Atom

Atom es un editor de código abierto, creado por *Github* esto hace que tenga una muy buena integración con git y con *Github*. Es de código abierto y a su vez se puede configurar como IDE con gran soporte para múltiples lenguajes y una comunidad activa desarrollando multitud de *plugins*, su uso se centrará en el desarrollo de la parte del agente en Python.

2.2.4 Editor de texto Vim

Vim es un editor de texto de código libre, que surge como una mejora del editor tradicional Vi, ambos cuentan con varios modos de funcionamiento, que se alternan para hacer distintas operaciones. Es un editor que se controla por línea de comandos, siendo solo necesario el teclado para manejarlo. Entre sus funcionalidades se incluye la navegación por pestañas, completado de órdenes, puede editar código fuente y llamar a compiladores externos, pudiendo mostrar errores de compilación.

Este editor permite la ejecución de Scripts de Bash desde el mismo, lo que hace que sea una gran opción para desarrollarlos. Sin embargo, tiene curva de aprendizaje muy pronunciada, pues es totalmente distinto a los editores comunes como nano, gedit, etc.

2.3 Tecnologías y protocolos

2.3.1 REST - Transferencia de estado representacional

Surgió en el año 2000 de la mano de Roy Fielding, no es más que un conjunto de restricciones que estandarizan el desarrollo de forma eficiente de servicios web. Hoy en día está plenamente extendido la provisión de APIs REST por una gran cantidad de empresas, siendo accesible mediante estas los distintos servicios web que proporcionen.

El intercambio de información es conforme a los métodos HTTP, donde el cliente manda una petición HTTP al servidor (GET, PUT, POST, etc) y el servidor le da una respuesta siguiendo el protocolo HTTP.

Para entender bien el concepto de Rest, hay que explicar que es una API o *Application Programing Interface*, no es más que una capa de abstracción entre dos aplicaciones que permite el intercambio de información y la interacción entre ellas.

A los sistemas que siguen una arquitectura Rest, se les denomina sistemas *Restful*, en nuestro caso, será necesario el desarrollo de una API Rest, con la que otras entidades pueden invocar acciones en el servidor. Será desarrollada mediante el uso de *swagger*, podemos observar un ejemplo en la siguiente figura.

```
@SwaggerDefinition(
    info = @Info(
        version = "1.0.0",
        title = "",
        extensions = {
            @Extension(properties = {
                @ExtensionProperty(name = "name", value = "connectedcup"),
                @ExtensionProperty(name = "context", value = "/connectedcup"),
            })
        }
    ),
    tags = {
        @Tag(name = "connectedcup", description = "")
    }
)
```

Figura 2.7 – Ejemplo de API en Java con Swagger

2.3.2 MQTT – MQ Telemetry Transport

Es un protocolo de comunicaciones *M2M (Machine to machine)* desarrollado por Andy Stanford-Clark (*IBM*) en 1999. Es un protocolo simple y ligero al extremo, que se basa en subscripciones y publicaciones. Este protocolo es perfecto para la arquitectura de *IoT*, donde los dispositivos suelen tener bajas capacidades, las redes latencia alta y variable; siendo capaz de asegurar una transferencia fiable a pesar de estas características.

2.3.2.1 Funcionamiento

El funcionamiento es de punto-multipunto, donde se usa un sistema de suscrición a un tema y el bróker, o sistema de traducción/distribución de la información se encarga tanto de obtener por parte del sensor los datos, como de distribuirlo a los sistemas finales que se hallen suscritos al tema en cuestión. En la siguiente figura podemos ver un esquema de funcionamiento:

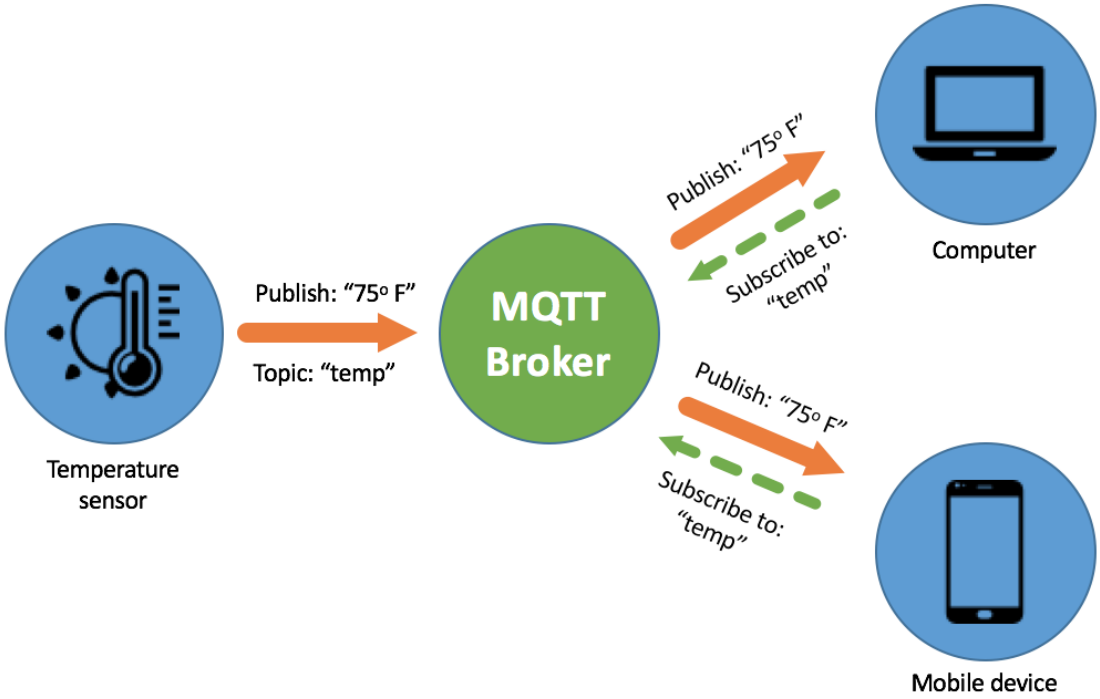


Figura 2.8 – Diagrama de MQTT

2.3.2.2 Calidad de servicio

También cabe distinguir que asegura 3 calidades de distribución de mensajes:

- Nivel 0: "A lo sumo una vez" donde usa el sistema de reenvío de TCP, pero admitiendo pérdidas.
- Nivel 1: "Al menos una vez" asegura los mensajes, pero permite duplicados.
- Nivel 2: "Una vez exactamente" Asegura la llegada de mensajes sin pérdidas ni duplicados, independientemente del medio de comunicación y de la red.

2.3.2.3 Mensajes:

Veamos los tipos de mensajes que define el protocolo:

- Connect: Contiene credenciales de usuario y es enviado por el cliente para conectarse a un servidor. El cliente solo puede suscribirse una vez a un tópic específico, cualquier otro intento al mismo, lo

desconectará por violar el protocolo.

- ConnACK: Como su nombre indica, es un asentimiento al mensaje de conexión, es el primer mensaje que envía el servidor, si el cliente no lo recibe en un periodo de tiempo razonable, debe cortar la conexión.
- Publish: Puede ser enviado tanto desde el cliente al servidor, como desde el servidor al cliente; contiene un mensaje de aplicación que puede incluir por ejemplo datos de sensores en un formato legible por la otra entidad.
- PublishACK: Respuesta al mensaje de *publish* cuando se configura el nivel de calidad 1.
- PubRec: Mensaje de respuesta en el nivel 2 de calidad, es el segundo mensaje en este nivel, durante el proceso de publicación.
- PubRel: Respuesta a PubRec en el nivel 2, es el tercer mensaje en este nivel durante la publicación.
- PubComp: Cuarto y último mensaje tras PubRel en el nivel 2, finaliza la publicación.
- Subscribe: Suscribe un cliente a un tema, el cliente recibirá del servidor o le enviará, datos de aplicación de ese tema.
- SubACK: Respuesta de afirmación al mensaje de suscripción.
- UnSubscribe: Borra la suscripción de un cliente a un tema, iniciado por el cliente.
- UnSubACK: Respuesta de asentimiento al mensaje de borrado de suscripción.
- PingReq: Se envía desde el cliente por varios motivos, mantener viva la conexión ante falta de mensajes, pedir que el servidor confirme que está vivo o comprobar que la red lo está, es el inicio de un proceso de *Keep Alive*.
- PingResp: Es la respuesta del proceso de *Keep Alive*.
- Disconnect: Desconexión por parte del cliente, de una forma limpia.

2.3.3 Swagger

Surgió en 2011 por parte de Tony Tam y es un conjunto de herramientas, notaciones, restricciones y reglas que nos ayuda a la hora de diseñar y desarrollar una API *RESTful*, con el objetivo de hacerla fácilmente accesible para otros usuarios o desarrolladores.

Incluye *Swagger UI*, documentación automática, generación de código y soporte para casos de pruebas, es importante recalcar que es un proyecto que da una gran cantidad apoyo al código libre.

Tiene una curva de aprendizaje bastante poco pronunciada y un nivel de complejidad bastante bajo, lo que hace que sea perfecto para el desarrollo de API, de hecho, está plenamente integrado en WSO2.

Swagger nos proporciona un sistema de notación en Java, que nos permite de forma sencilla definir una API, como hemos visto en la figura 2.7.

2.3.4 Maven

Maven surgió en 2001 como un proyecto *Open source* por parte de *Apache*, su intención fue la de estandarizar, unificar y facilitar la construcción y despliegue del código en proyectos basados en Java. Entre sus cualidades está el ayudar en la gestión de dependencias, de documentación y lanzamientos.

El proyecto Maven está definido en los ficheros *pom.xml* que es un acrónimo de *Project Object Model* o Modelo objecto de proyecto. Estos ficheros *xml* son ficheros que contienen información del proyecto Maven y su configuración para la construcción del mismo. Cuando Maven ejecuta una acción, busca el fichero *pom.xml* en el directorio actual, lo lee y actúa en consecuencia, según la configuración contenida en el fichero.

Los ficheros *pom.xml* permiten la herencia de otros proyectos, pudiendo así implementar dependencias, lista de plugins y la configuración de los proyectos heredados, definidos como proyectos “padres”.

Maven define los *Archetype* o Arquetipos como la plantilla de un proyecto, esta plantilla, mediante la interacción con el usuario, genera el esqueleto de un proyecto funcional con las necesidades básicas del usuario. De esta forma, podemos partir de plantillas definidas con la estructura necesaria para así poder facilitar el desarrollo conforme a las necesidades del proyecto. Se pueden generar distintos arquetipos para Maven, tanto para proyecto completo, como para partes modulares del proyecto, un ejemplo de esto lo encontraremos en este Trabajo de Fin de Grado, donde se partirá de un arquetipo definido por WSO2.

2.3.5 OSGi – Open Service Gateway initiative

Creado en marzo de 1999 con el objetivo de definir especificaciones de software para plataformas compatibles, inicialmente pensado para redes domésticas. OSGi define su propia arquitectura, que podemos observar en la figura 2.9.

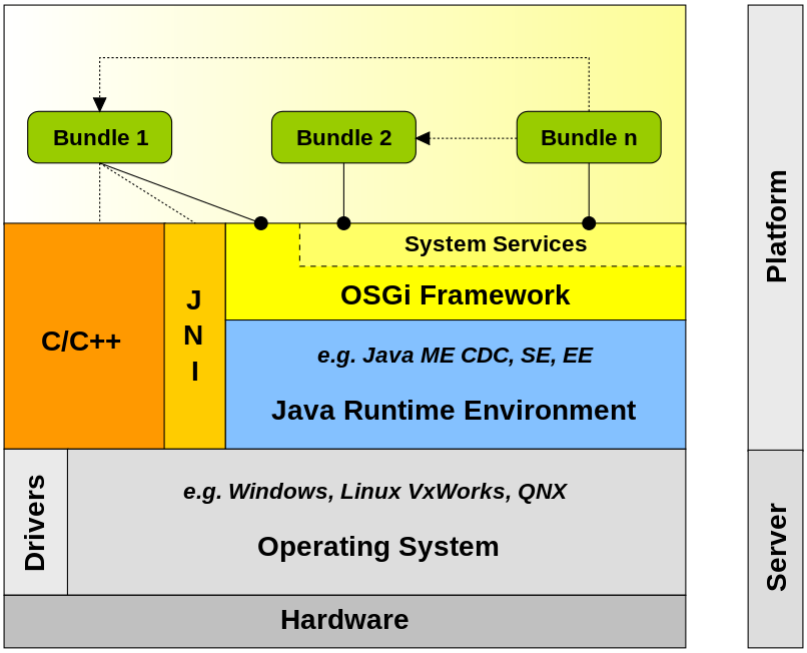


Figura 2.9 – Arquitectura OSGi

OSGi es una capa que permite la creación *bundles* o componentes y que estos puedan comunicarse entre sí en tiempo de ejecución. En OSGi, cada componente (parecido a un jar tradicional + metadatos) se diferencia del resto por su propio *classpath*. La arquitectura OSGi está orientada a servicios donde estos pueden consumirse o registrarse dentro de la plataforma base

OSGi ofrece una plataforma para la gestión de módulos en tiempo de ejecución, encargándose de la instalación, arranque, parada, actualización y la desinstalación sin necesitar parar la plataforma base.

Finalmente, OSGi nos permite reducir la complejidad del desarrollo, pues se basa en los componentes que ocultan su contenido entre sí, comunicándose mediante servicios e interfaces bien definidas. También nos otorga la capacidad de reutilizar los componentes desarrollados, gracias a la independencia entre componentes.

3 TRABAJO PREVIO

Cabe hacer especial hincapié en esta parte, tal vez sea con diferencia la parte más pesada y dura del proyecto, pues requiere de hacer una gran labor de investigación tanto de la documentación oficial del proyecto del servidor de IoT como del código contenido en los repositorios oficiales del proyecto en GitHub.

3.1 Preparación de Raspberry Pi

Lo primero a realizar desde el punto de vista de la Raspberry Pi, es investigar toda la documentación existente para ser capaz de interconectar el sensor y la matriz de leds, utilizando los pines de entrada/salida existentes en todo modelo de Raspberry Pi, para ello nos vamos a valer de los *datasheet* de cada dispositivo y de la documentación oficial de Raspberry Pi.

La Raspberry Pi a usar es la versión 2, cabe destacar que todo lo realizado con esta versión, es compatible con otros modelos de Raspberry Pi, puesto que los pines que usamos son iguales en todos los modelos y la arquitectura del dispositivo se mantiene.

Los periféricos a usar en el proyecto consistirán en un sensor DS18B20 (modelo acuático) y una matriz de leds 8x8 con un driver MAX7219. Estos periféricos son económicos y fácilmente accesibles en el mercado, también cuenta con una comunidad a sus espaldas que ha desarrollado librerías para Python. En internet podemos encontrar el datasheet de estos elementos y esquemas para el cableado necesario. En la siguiente figura (figura 3.1) podemos ver el aspecto de los periféricos a usar.

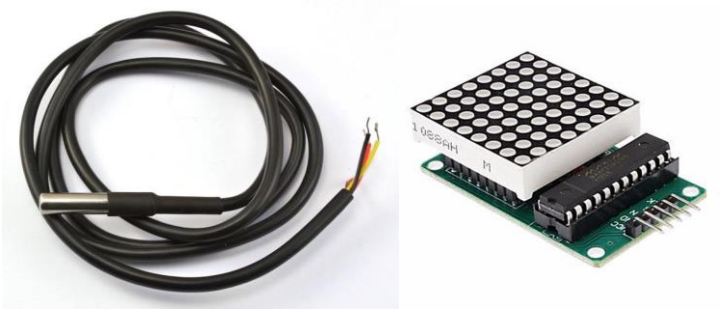


Figura 3.1 – Sensor de temperatura DS18B20 y matriz de leds con *driver* MAX7219

3.1.1 Conexión a la Raspberry Pi

La Raspberry Pi cuenta con unos pines de entrada y salida conocidos como GPIO, estos pines sirven para interconectar, alimentar y comunicarse con una gran variedad de periféricos electrónicos. Algunos de estos pines son para alimentación 3,3v y otros a 5v. También tiene pines para la comunicación SPI o I2C.

A continuación (figura 3.2) mostramos el conjunto de pines que incluye la Raspberry Pi 2, que aumenta considerablemente con respecto a su modelo anterior.

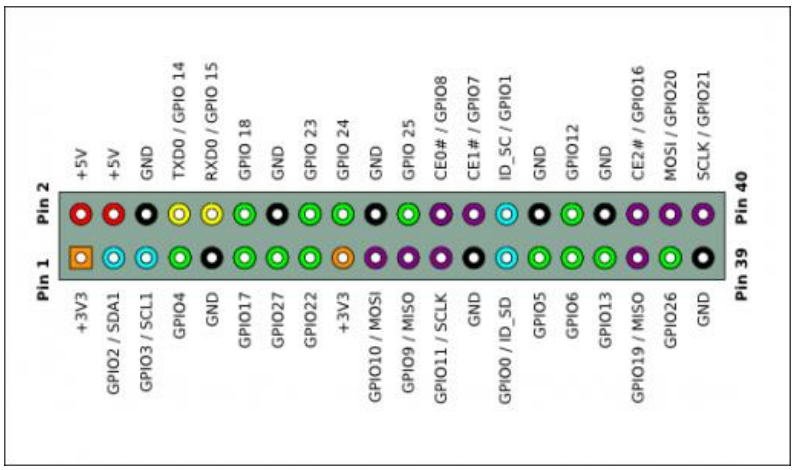


Figura 3.2 – GPIO de Raspberry Pi

A continuación (figura 3.3), veremos el esquema de cableado para conectar los periféricos de interés a la Raspberry Pi, sabiendo que la matriz de leds, usa los pines de SPI y el sensor de temperatura necesita una resistencia de *pull-up* de 4,7KOhmios, además, usará el pin GPIO4 aparte de la alimentación y tierra.

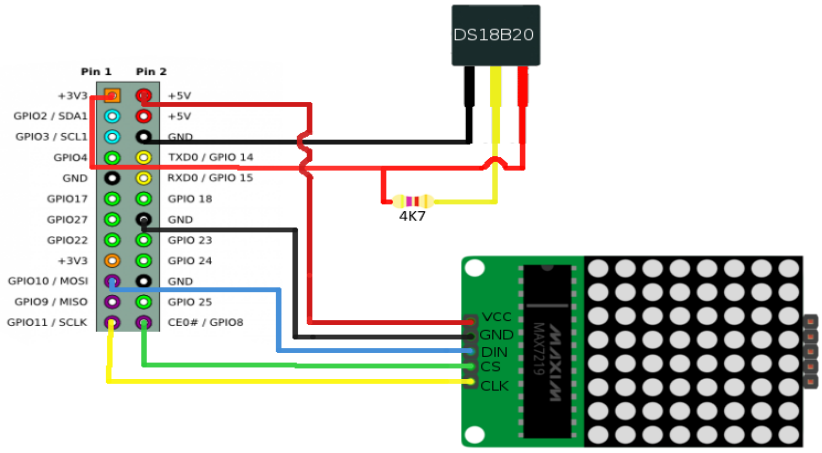


Figura 3.3 – Conexión de los periféricos

3.1.2 Pines Utilizados

Vamos a enumerar y a nombrar para qué sirven los distintos pines que hemos usado en la conexión de los dos elementos a la Raspberry Pi.

Para conectar el sensor de temperatura, hemos usado 3 pines:

- VCC – Alimentación a 3,3V.
- GND – Conexión a tierra.
- GPIO4 – Es el pin por defecto para dispositivos *1wire* [9].

Para poder usar la matriz de leds y el driver MAX7219, hemos realizado la siguiente conexión:

- VCC – Alimentación a 5V.
- GND – Conexión a tierra.
- DIN (*Data IN)* - Conexión de *Serial Peripheral Interface (SPI)*, se conecta al pin *Master Output, Slave Input (*MOSI).
- CS (*Channel Selection)* - Selección de esclavo, corresponde con el pin GPIO8 (CEO#)

- SCLK – Es el pin que marca la sincronización entre el *master* y el esclavo en *SPI*.

3.1.3 Configuración Software

Una vez que tenemos la configuración del hardware, es necesario que activemos los distintos pines que van a ser usados mediante software, ya que, por defecto, tanto *SPI* como *1Wire* vienen desactivados.

Para lograrlo, tenemos que modificar el fichero */boot/config.txt* para activarlos al iniciar el dispositivo.

```
Sudo vim /boot/config.txt
```

Añadimos las siguientes líneas al fichero

```
dtoverlay=w1-gpio #Activamos 1Wire
dtparam=spi=on    #Activamos SPI
```

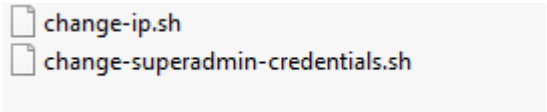
Con esto ya solo nos queda comprobar al iniciar el dispositivo que se cargan los módulos necesarios:

```
lsmod | grep w1-gpio #Comprobamos que se ha cargado el módulo 1Wire.
ls /dev/spidev0.0    #Si existe el fichero, se ha cargado el módulo.
```

3.2 Servidor WSO2 de IoT utilizado

Se ha utilizado la versión 3.3.0 del servidor IoT de WSO2, el servidor viene comprimido en formato *zip* y se puede descargar gratuitamente (https://wso2.com/iot/install/download/?type=downloader) bajo licencia Apache 2.0. El servidor viene divido en 3 partes, el núcleo Carbon está en el directorio */bin/*, el servidor de analytics se encuentra en *wso2/analytics/* y para finalizar, el bróker de MQTT está contenido en *wso2/broker/*, cada uno es un componente del servidor que se ejecuta de forma separada. También tenemos varios directorios de interés para este proyecto:

- Sample: Trae proyectos de ejemplo, como *connectedcup* que es una implementación para una cafetería, el cual me sirve de manera extensa para analizar el funcionamiento y los distintos componentes de la implementación en el servidor.
- Scripts: Varios scripts para cambiar la IP del servidor, que por defecto usa como dirección IP de escucha *localhost* y también para cambiar las credenciales del superusuario del servidor.



- Bin: Scripts de ejecutables del servidor:
  - *iot-server.sh:* Script de arranque del servidor Carbon de WSO2, véase *Anexo I*.
  - *broker.sh:* Script de arranque del bróker MQTT, véase *Anexo II*.
  - *analytics*.sh: Script de arranque del servidor de análisis de datos, Véase *Anexo III*.

3.2.1 Análisis del servidor

Por defecto, el servidor se ejecuta para la IP local y los puertos 9443 (consola de gestión de dispositivo), 9445 (DAS) y 9446 (Message Broker). Se deben ejecutar en el siguiente orden: primero el script *bin/broker.sh* después el núcleo del servidor mediante *bin/iot-server.sh* y finalmente el servidor de análisis de datos *bin/analytics.sh*.

Para poder cambiar la IP del servidor, disponemos de un script bajo el directorio *scripts* que nos permite cambiar todos los ficheros de configuración XML y JSON necesarios dentro del proyecto.

3.2.1.1 Script change-ip.sh

El script de Bash, nos pide por línea de comando los valores necesarios siguientes:

- IP para ser sobrescrita en los ficheros de configuración, es la misma IP que actualmente tiene configurado el servidor, cabe destacar que la primera vez, este valor será localhost.
- IP que queremos configurar como la del servidor a partir de la ejecución, es necesario tener en cuenta que esta IP debe estar configurada en alguna interfaz. Podemos usar un *hostname* en vez de una IP, siempre que configuremos el fichero hosts:

```
127.0.0.1    localhost.localdomain localhost
x.y.z.w      machinename.domain machinename
```

- Valores para el nuevo certificado autofirmado *SSL*:
  - Common name: IP/Hostname
  - País
  - Localización
  - Organización
  - Correo

```
buildSubject 'C' 'Country' 'C'
buildSubject 'ST' 'State' 'C'
buildSubject 'L' 'Location' 'C'
buildSubject 'O' 'Organization' 'C'
buildSubject 'OU' 'Organizational Unit' 'C'
buildSubject 'emailAddress' 'Email Address' 'C'
buildSubject 'CN' 'Common Name' 'C'
```

El script se encarga de modificar los ficheros de configuración de los distintos módulos del servidor

3.2.2 Servidor de Análisis de Datos, DAS:

El análisis de datos se refiere al módulo del servidor IoT que se dedica a la agregación o recolección de datos, análisis o manipulación y presentación visual de los resultados obtenidos de los datos.

La arquitectura del servidor de análisis de datos es como sigue, con el flujo de datos representado:

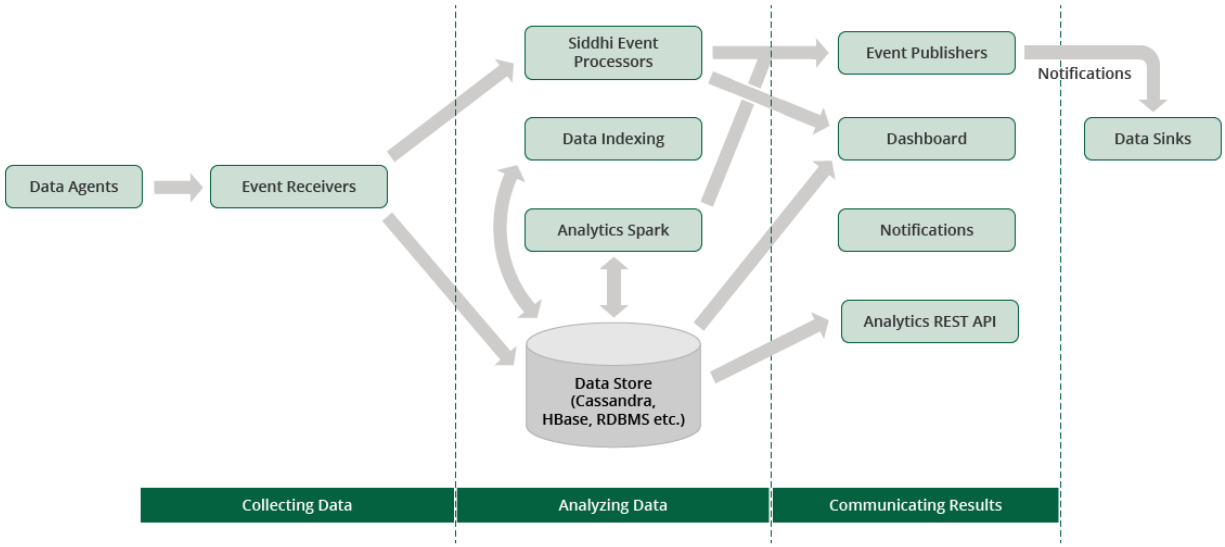


Figura 3.4 – Arquitectura DAS

El servidor DAS, se puede configurar mediante el fichero XML contenido en *wso2/analytics/conf/carbon.xml* en él vienen definidos parámetros como el puerto, parámetros de autenticación como el contenedor de certificados, tiempo máximo de un hilo para ejecutarse, véase el anexo IV.

3.2.3 Servidor bróker de mensajes

Un bróker de mensajería es el encargado de distribuir los mensajes publicados a un topic a las entidades subscritas a él, también tiene la capacidad de traducir mensajes de un sistema a otro, ya sea con varios lenguajes, mediante un medio de comunicación entre sistemas.

Permite el intercambio de mensajes de forma sencilla, se encarga de la comunicación con los agentes y de gestionar los *topics* de MQTT para la comunicación, apreciemos a continuación la arquitectura del servidor de bróker de mensajes.

Al igual que en el caso del DAS, la configuración está contenida en el fichero: *wso2/broker/conf/carbon.xml*.

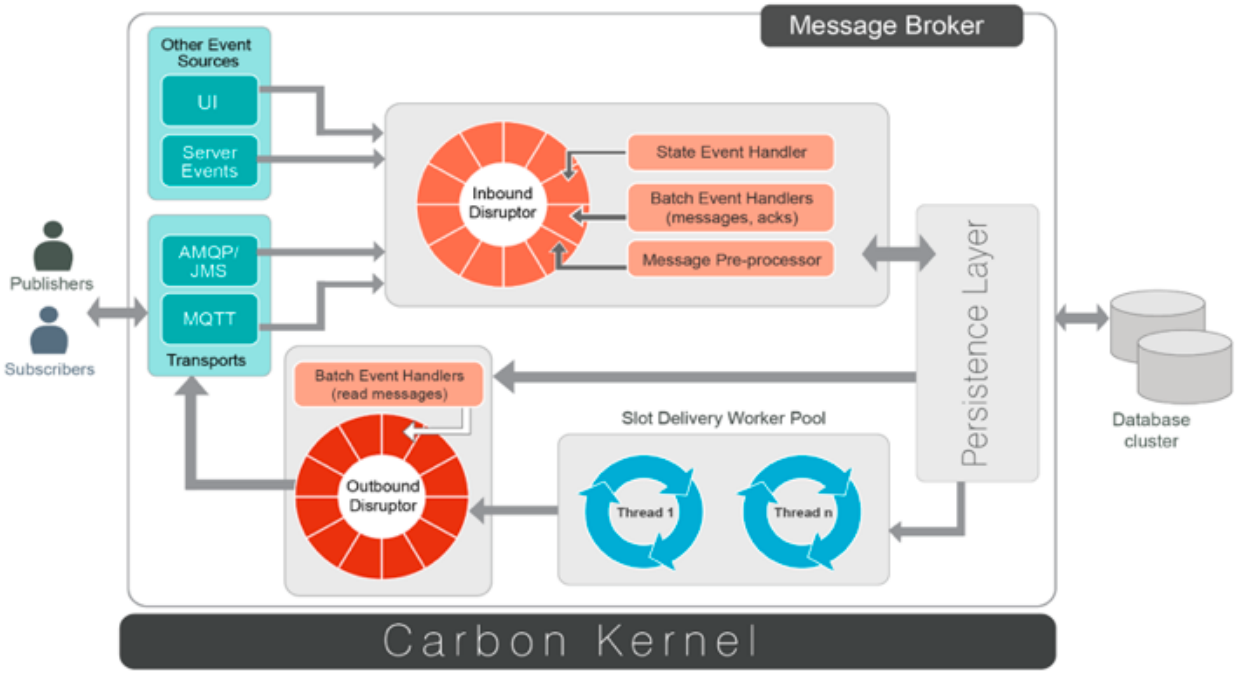


Figura 3.5 – Esquema del bróker de mensajería

3.2.4 Consola de administración de dispositivos

Es la interfaz de usuario para la gestión de dispositivos, para acceder a ella una vez se ha arrancado el servidor, hay que acceder a la dirección IP configurada que llamaremos *<DirIP>* y en concreto a la URL: https://<DirIP>:9443

La pantalla que nos recibe se puede apreciar en la siguiente figura, donde se nos requiere las credenciales de usuario.

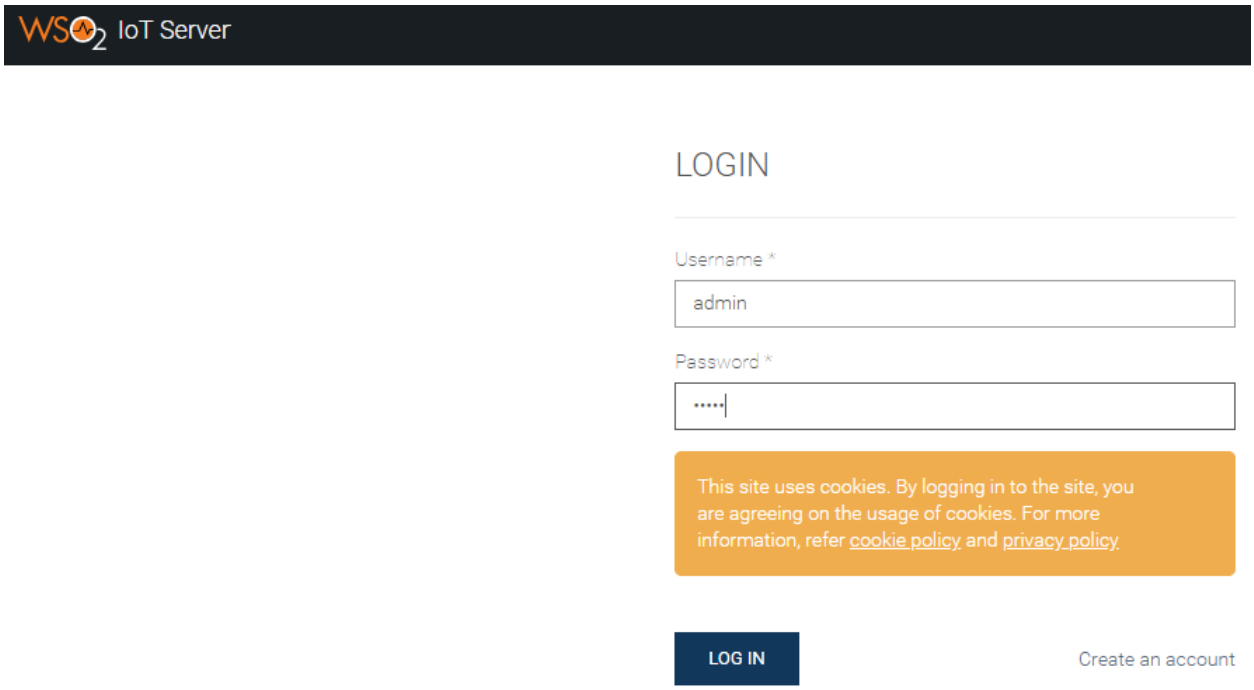


Figura 3.6 – Portal de bienvenida

Se accede introduciendo las credenciales, que por defecto son admin/admin, pero que obviamente y por motivos de seguridad deberían ser cambiadas a una contraseña con mayor seguridad.

3.2.5 Autenticación de dispositivos

Para autenticar dispositivos en el servidor, se utiliza el estándar *OAuth[18]*, que está definido para permitir flujo de datos entre dos entidades, donde una es el proveedor de un servicio y la otra la consumidora, garantizando la protección de la comunicación.

Esta autenticación, se realiza mediante tokens, estos son cadenas de caracteres generados por un servidor de autenticación (en nuestro caso, la labor de servidor de autenticación la ejerce el servidor WSO2 IoT.), para ello primero necesitamos tener unas credenciales, en la figura 3.7 podemos distinguir el esquema para la autenticación.

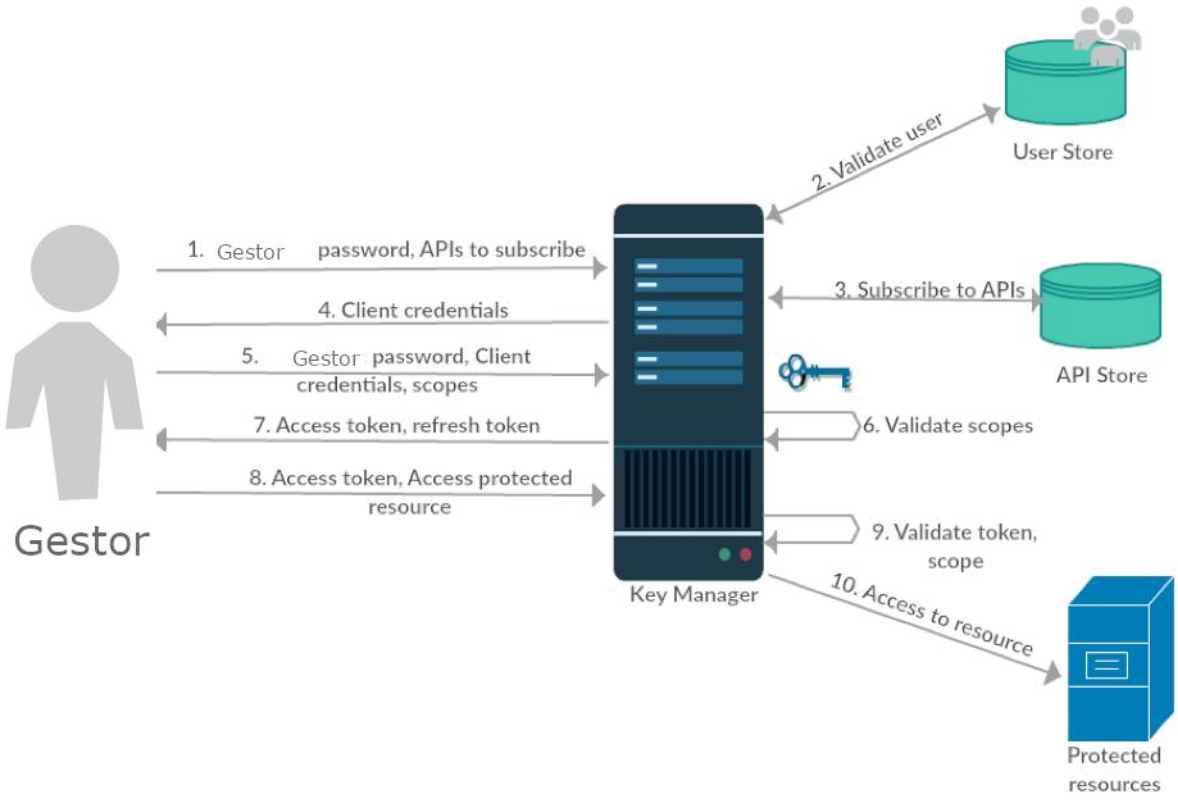


Figura 3.7 – Esquema de autenticación

Como podemos observar en la figura anterior, para generar el token de acceso necesitamos las credenciales del gestor o entidad responsable. El procedimiento obtenerlo es el siguiente:

- Hay que codificar en base64 las credenciales del gestor, para ello lo hacemos con el siguiente comando:

```
echo -n <GESTOR>:<PASSWORD> | base64
```

- A continuación, es necesario obtener el *Client ID* y el *Secret ID*, esta combinación es proporcionada por el servidor de autenticación, mediante una petición http, para ello necesitamos pasarle al servidor las credenciales codificadas del gestor y esto lo logramos usando el siguiente comando:

```
curl -k -X POST https://<IOTS_HOST>:8243/api-application-registration/register -H 'authorization: Basic <BASE 64 ENCODED GESTOR:PASSWORD>' -H 'content-type: application/json' -d '{"applicationName":"appName", "tags":["device_management"]}'
```

- Necesitamos repetir el proceso de codificación, ahora con *Client ID* y *Secret ID* proporcionados por el servidor de autenticación.

```
echo -n <Client ID>:<Secret ID> | base64
```

- Por último, tenemos que generar el token de acceso mediante una petición *http*, donde debemos indicar los *scopes* a los que solicitamos acceso, por ejemplo "perm:myRaspberry:enroll". Estos *scopes* están definidos en la API desarrollada (myRaspberryService.java). Usaremos la cabecera *http* "Authorization" para proporcionarle al servidor de autenticación el par *ClientID, SecretID* codificado. Con el siguiente comando logramos el acceso para toda la API desarrollada:

```
curl -v -k -d "grant_type=password&username=<USERNAME>&password=<PASS>&scope= perm:myRaspberry:enroll perm:devices:view perm:devices:delete" -H "Authorization: Basic <BASE 64 ENCODED CLIENT_ID:CLIENT_SECRET>" -H "Content-Type: application/x-www-form-urlencoded" https://<IOTS_HOST>:8243/token
```

3.3 GitHub del servidor IoT de WSO2

Una gran cantidad de trabajo recae en la necesidad de entender los métodos de las clases usadas en el proyecto, una vez entendidas las clases, la labor de desarrollo del plugin sobre ellas se hace una tarea menos faraónica.

En esta sección, describiremos las clases y métodos usados por las clases implementadas, su labor y su funcionalidad y el porqué de su uso, así como un análisis más profundo del código de las mismas.

3.4 Esquema de red

Para dar mayor flexibilidad al sistema, se ha tenido en cuenta tanto el uso de una red de ámbito local, como el uso de Internet, esto es posible dado que *MQTT* usa un sistema de encriptación, que nos asegura que la comunicación entre agente y servidor sea lo más segura posible, evitando que pueda ser fácilmente desencriptada por terceros.

Podemos recurrir a un sistema de DNS si queremos usar el servidor mediante nombre de dominio, lo que nos otorgaría escalabilidad al sistema.

Por otro lado, podríamos usarlo en un entorno cerrado y totalmente aislado que es lo que hemos hecho para el entorno del proyecto, haciendo uso de un router como en la figura, modelo Asus AC RT1200G+.

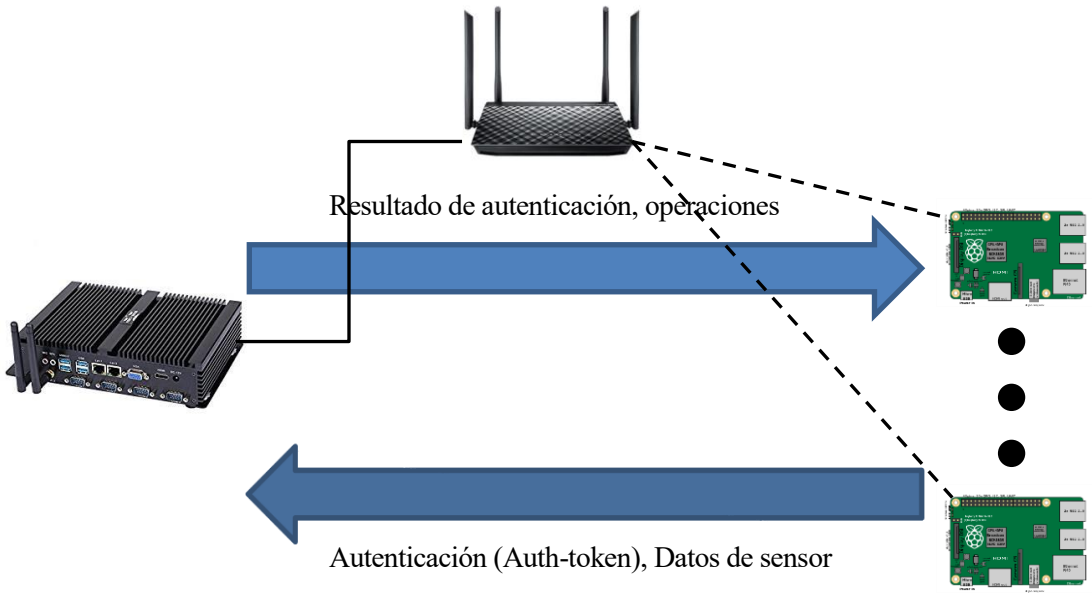


Figura 3.8 – Esquema local

Por otro lado, la máquina utilizada para desarrollo, ha sido un portátil Lenovo con las siguientes características:

- SSD 128GB
- 8GB de memoria RAM
- Intel I3-3110 @ 2,4Ghz
- HDD 500GB
- Tarjeta Wifi dual band.
- Distribución Debian Stretch

Que ha hecho las veces de barebone, durante las pruebas y desarrollo.

4 DESARROLLO DE LA SOLUCIÓN

4.1 Creación de la estructura del proyecto

El ámbito del proyecto abarca los desarrollos en el servidor y en la Raspberry. Desde el punto de vista del servidor necesitamos desarrollar el plugin de Carbon, la API y el módulo de Interfaz de Usuario y desde el lado de los dispositivos, necesitamos desarrollar el agente, tal y como vemos en la figura 4.1.

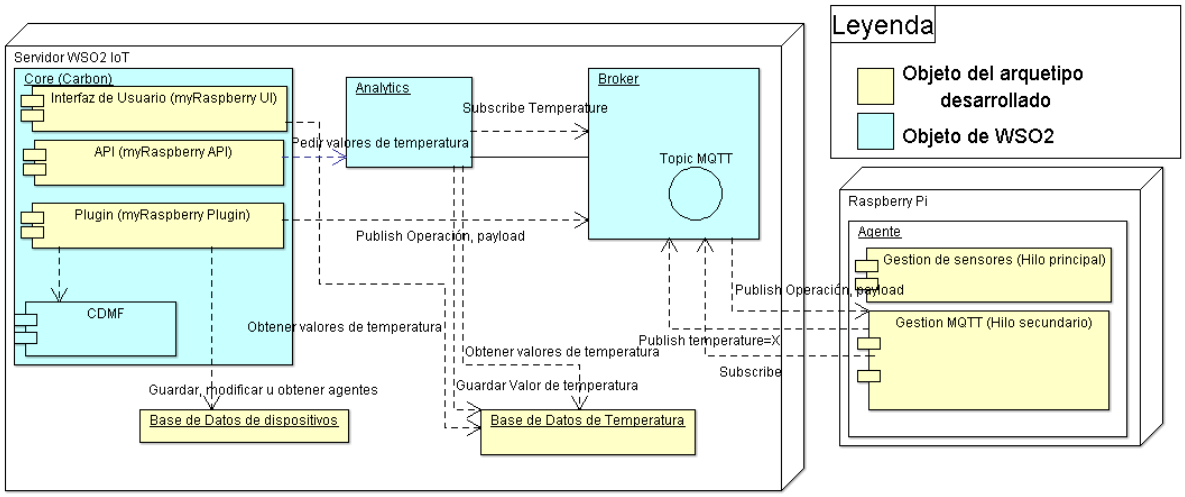


Figura 4.1 – Diagrama de despliegue del proyecto

Usaremos Maven y su utilidad de arquetipos para generar la estructura y el esqueleto del proyecto que posteriormente procederemos a ampliar y desarrollar según las necesidades del mismo, esto es posible, ya que *WSO2* define unos arquetipos con los que los desarrolladores pueden trabajar sin tener que crear desde cero el proyecto, facilitando así el desarrollo de nuevas funcionalidades y plugins de dispositivos. El diagrama de clases del proyecto, incluyendo plugin y API es el que vemos en la figura 4.2.

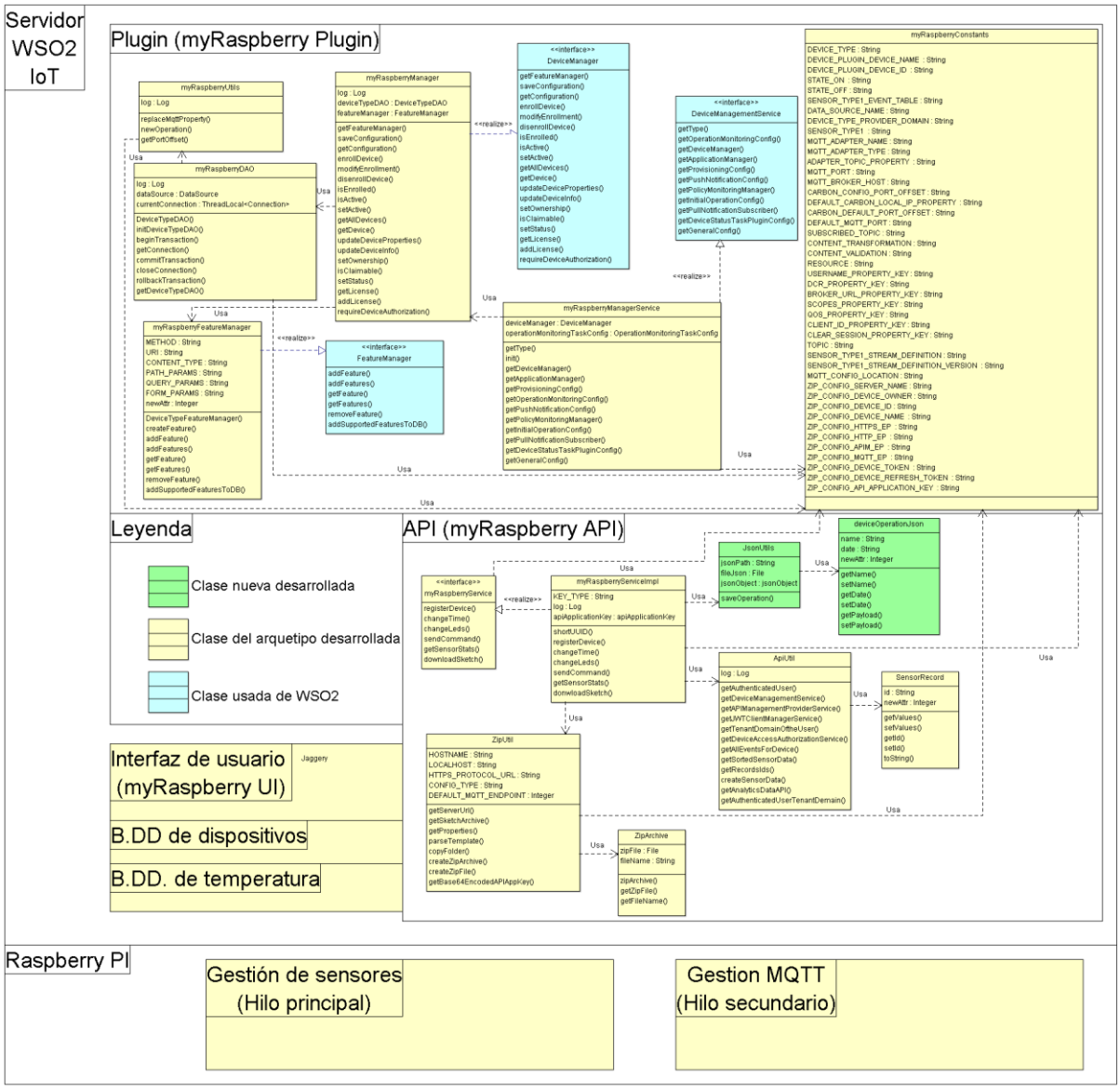


Figura 4.2 – Diagrama de clases global

Para poder acceder a los arquetipos disponibles de *WSO2* debemos primero clonar el repositorio de *git* con el siguiente comando:

```
git clone -b v1.0.0 -single-branch https://github.com/wso2/carbon-device-mgt-maven-plugin.git
```

Esto nos creará una carpeta con el repositorio de los arquetipos de *WSO2* para que podamos construirlos e instalarlos en el sistema mediante *Maven*. Para ello, tenemos que cambiar de carpeta a la nueva creada con el nombre carbon-device-mgt-plugin e instalarlo, esto lo logramos con el siguiente comando:

```
cd <Directorio donde se ha clonado>/carbon-device-mgt-plugin
mvn clean install
```

La instalación del arquetipo tiene un bug, por lo que es necesario cambiar el directorio manualmente, para ello debemos mover desde el directorio /home/<usuario>/.m2 al directorio /home/<usuario>/.m2/repository con el siguiente comando:

```
mv /home/<usuario>/.m2/archetype-catalog.xml /home/<usuario/.m2/archetype-catalog.xml
```

Con esto ya podemos generar la estructura y el esqueleto desde el que partir para desarrollar el proyecto, finalmente nos falta:

```
mvn archetype:generate -DarchetypeCatalog=local -X
```

Esto nos ejecutará *Maven* y nos pedirá que elijamos el arquetipo que queremos generar, como podemos apreciar en la siguiente figura:

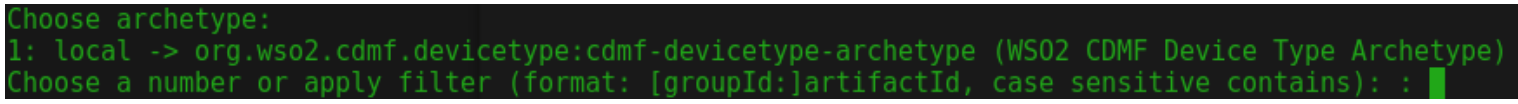


Figura 4.3 – Generación del arquetipo

Elegimos el tipo *org.wso2.cdmf.devicetype:cdmf-devicetype-archetype (WSO2 CDMF Device Type Archetype)*, se solicitarán ciertos valores necesarios para generarlos. Es importante notar que por defecto nos pedirá tres sensores obligatoriamente. Para solucionar esto, más tarde habrá que eliminar los sensores sobrantes que nos ha creado.

Los campos a dar valores para generar el arquetipo son los siguientes:

```
 Define value for property 'groupId': Nombre identificativo del proyecto
 Define value for property 'artifactId': Nombre del directorio
 Define value for property 'version' 1.0-SNAPSHOT: Versión del proyecto
 Define value for property 'package': Si no se da valor, toma el grouId
 Define value for property 'deviceType': Nombre en el servidor del tipo
de dispositivo
 Define value for property 'sensorType1': Tipo de sensor del dispositivo
 Define value for property 'sensorType2': Tipo de sensor del dispositivo
 Define value for property 'sensorType3': Tipo de sensor del dispositivo
```

Los valores escogidos son los siguientes:

```
 'groupId': sensorboard
 'artifactId': raspberry
 'version' 1.0-SNAPSHOT: 1.0-SNAPSHOT
 'package':
 'deviceType': myRaspberry
 'sensorType1': temperature
```

Con esto ya tenemos la carpeta Raspberry generada, esta contiene toda la estructura de directorios y el esqueleto de las clases con sus métodos, que vamos a analizar en apartados posteriores.

4.2 Estructura de ficheros

La estructura de directorios del proyecto está organizada en los componentes y las funcionalidades del servidor. Por un lado, tenemos el directorio *component*, que incluye los cuatro módulos principales: la API *Rest* del dispositivo, el módulo de análisis de datos, el plugin del dispositivo (implementación interna en el servidor del dispositivo *myRaspberry*) y por último la interfaz web, por otro lado, encontramos la carpeta feature; bajo esta carpeta están las definiciones del plugin de Carbon, de la base de datos, así como las operaciones de despliegue en el servidor, el agente, etc. Lo podemos observar en la siguiente figura:

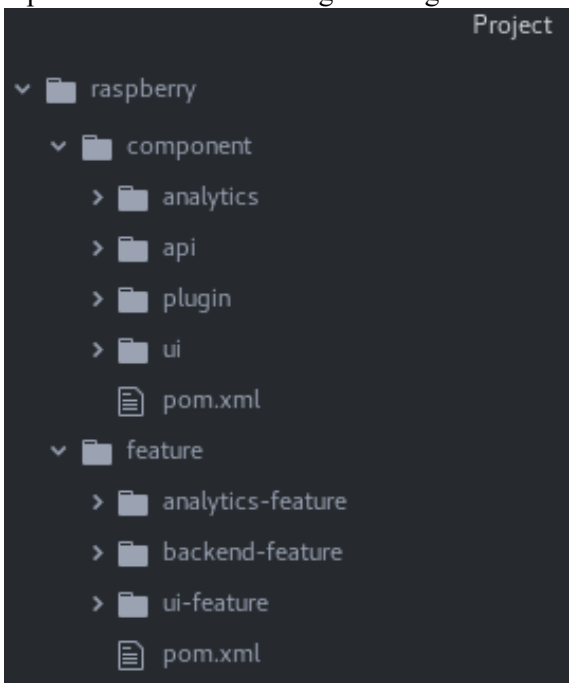


Figura 4.4 – Directorio de trabajo

Veamos a continuación para que sirve cada uno de los directorios que componen proyecto.

4.2.1 Directorio API

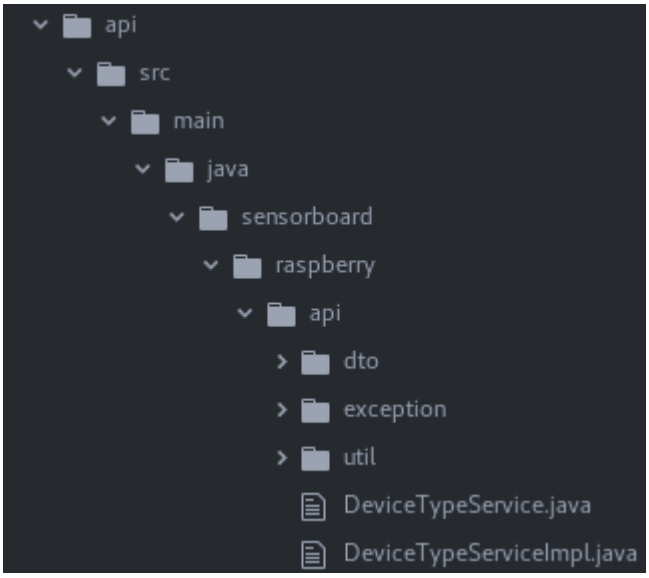


Figura 4.5 – Directorio API

Es el directorio de trabajo para la API *JAXRS*, la API está contenida en el servidor y es la capa de abstracción con la que una entidad puede interactuar para invocar acciones.

Cabe destacar que tenemos dos ficheros y tres directorios:

- dto: Objeto de transferencia de datos, definimos en él las estructuras de datos a intercambiar entre las entidades.
    - DeviceJSON.java: Definición de la clase *DeviceJSON*, contiene la clase que va a contener los datos enviados por los agentes una vez recibidos en el servidor WSO2.
    - SensorRecord.java: Definición de la clase *SensorRecord* con métodos para operar sobre los datos enviados por el agente, en el servidor.
- exception: Contiene el fichero DeviceTypeException.java que extiende a la clase *Exception*, definiendo nuevas excepciones para las operaciones.
- Util: Contiene funciones de utilidades que serán usadas por la API, como por ejemplo operaciones de obtención de datos, gestión de ficheros ZIP, etc.
- myRaspberry.java: Es una interfaz, que define la API siguiendo las recomendación y notación *swagger*, todo ello desarrollado en Java. Define las funcionalidades que va a ofrecer la API, los métodos http, la URI, así como otras características.
- myRaspberryImpl.java: La implementación de la interfaz DeviceTypeService, lo que es lo mismo que la implementación de la API.

4.2.2 Directorio Plugin

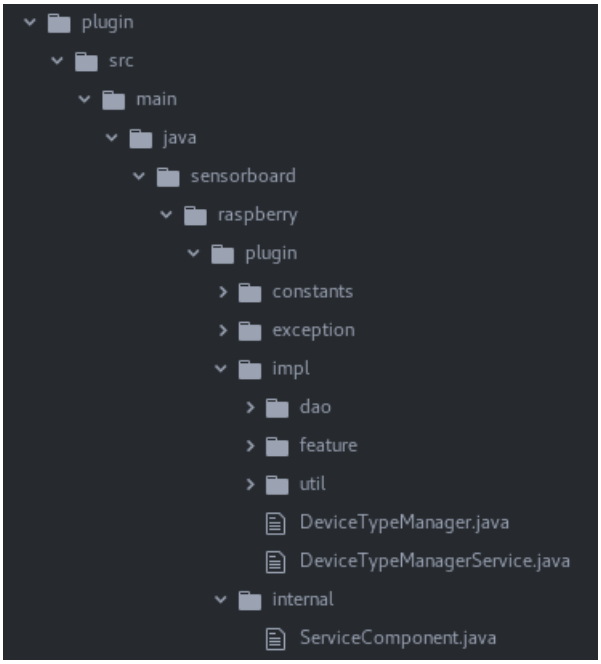


Figura 4.6 – Directorio Plugin

Es el directorio donde se implementará el plugin o paquete OSGi en *Java* para el servidor WSO2. Este paquete debe implementar la clase *DeviceManager* que es la conexión con el CDMF ya descrito en apartados anteriores.

Pasaremos a describir brevemente los directorios y ficheros contenidos en este espacio de trabajo:

- **constants**: definiciones de constantes para su uso por el *plugin*, se definen cosas como el *topic MQTT*, el nombre del tipo de dispositivo (myRaspberry), etc.
- **Impl**: Contiene el código destinado a la implementación del *plugin,* se subdivide en los siguientes directorios y ficheros:
    - **dao**: Es acrónimo de objeto de acceso a datos, es la implementación de la interfaz para operar y conectarse con la base de datos.
    - **Feature**: Implementa la clase *FeatureManager* y se encarga de gestionar las *features*, que no son más que el conjunto de operaciones que se va a poder realizar sobre el dispositivo.

- **Util**: Contiene clases de utilidades, como para inicializar el script de la base de datos u otros métodos usados por el plugin.
- **DeviceTypeManager.java**: Es la implementación de la clase *DeviceManager,* que está definida en la CDMF, esta clase inicializa y opera sobre las clases mencionadas más arriba.
- **DeviceTypeManagerService.java**: Es la implementación de la clase *DeviceManagerService* encargada de conectarse en tiempo de ejecución con OSGi.

- **Internal**: Contienen la clase *ServiceComponent*, encargada de obtener y llamar a la clase *DeviceTypeManagerService*.

4.2.3 Directorio UI

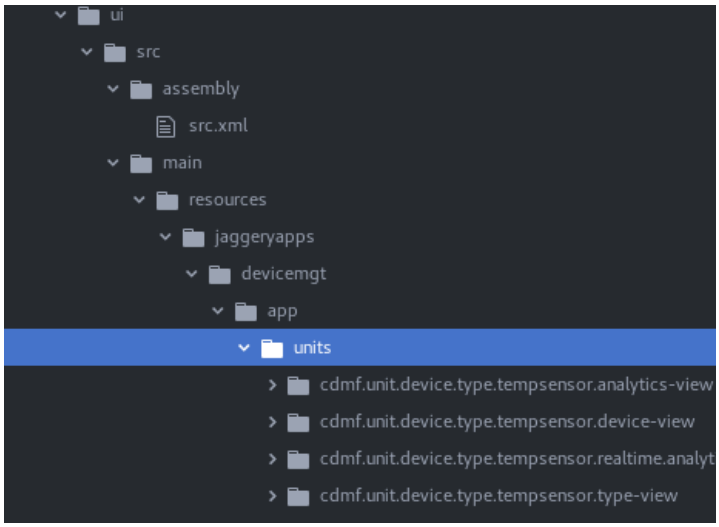


Figura 4.7 – Directorio UI

Es el directorio donde creamos las unidades de *Jaggery* de nuestro dispositivo a desarrollar, para posteriormente ser mostrados por la interfaz específica de gestión web que implementa el servidor *WSO2,* los directorios contenidos son los siguientes:

- **Assembly**: Contiene **src.xml**, que no es más que la configuración de empaquetado de las unidades *Jaggery*.
- **main/resources/jaggeryapps/devicemgt/app/units*:*** directorio de las distintas unidades de *Jaggery* que explicaremos en profundidad en apartados posteriores.

4.2.4 Directorio feature

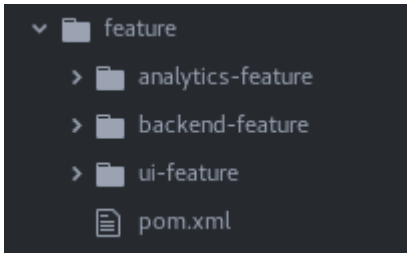


Figura 4.8 – Directorio feature

Es el directorio raíz de las *features* de Carbon, estas no son más que unidades de software reutilizable y modular. Carbon es un conjunto de *features* que a su vez son un conjunto de componentes; estos componentes son

conjuntos de paquetes OSGi, uno o varios.

En este directorio vamos a encontrar las definiciones e implementaciones de las *features*, veamos ahora los distintos directorios:

- **Analytics-feature:** Contiene el fichero **myRaspberry-datasources.xml** que define la base de datos a usar por el módulo de análisis de datos, también incluye el fichero **p2.inf** que incluye acciones para el despliegue de las funcionalidades.
- **Backend-feature**: Contiene todos los ficheros, scripts y código vario relacionado con la parte de backend. Entre ellos encontramos:
    - **Agent:** Carpeta con el código del agente, este código es el agente que será ejecutado en una o más de una Raspberry Pi, el código está incluido dentro del servidor WSO2 con el objetivo de que podamos obtener el ZIP al registrar un nuevo dispositivo.
    - **Datasources:** Una copia del fichero **myRaspberry-datasources.xml.**
    - **Dbscripts:** Scripts relacionados con la base de datos común entre dispositivos, como la creación de tablas por agente registrado, etc.
    - **Devicetypes:** contiene el fichero **myRaspberry.xml** con la definición de las características del dispositivo creado.
- **UI-feature:** Contiene el fichero **p2.inf** que describe las operaciones e instrucciones para la instalación y despliegue de los módulos de la interfaz de usuario.

4.3 Descripción del Código

4.3.1 Interfaz de programación de aplicaciones API

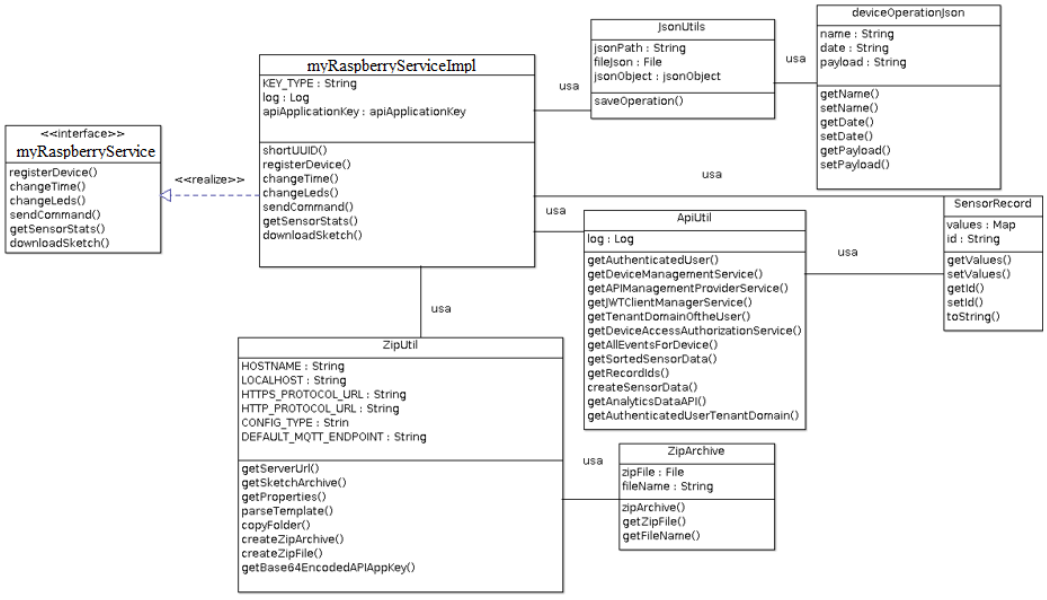


Figura 4.9 – Diagrama UML de la API

En la figura 4.9 vemos el diagrama de clases de la API, su implementación no tiene ningún tipo de restricción, tenemos total libertad para desarrollarla, por lo que lo haremos con *swagger*. Hemos hecho uso de las clases ya proporcionadas en el arquetipo, ampliándolas para implementar las operaciones y las llamadas que necesitamos. Se ha implementado también el método shortUUID que nos proporciona el string que identificará inequívocamente cada dispositivo. Mediante la clase *APIUtil* hacemos llamadas a la API del módulo Analytics que nos porporcionará los valores de temperatura almacenados.

Como ya hemos visto en apartados anteriores, la API es una capa de abstracción entre la entidad gestora y otros módulos software, dando formato y comprobando los parámetros, así como representando la salida. Especificando para el caso de WSO2, la API controla y gestiona el uso del *plugin* del dispositivo, siendo así la API el punto de entrada para operar con él, tal y como podemos observar en la siguiente figura:

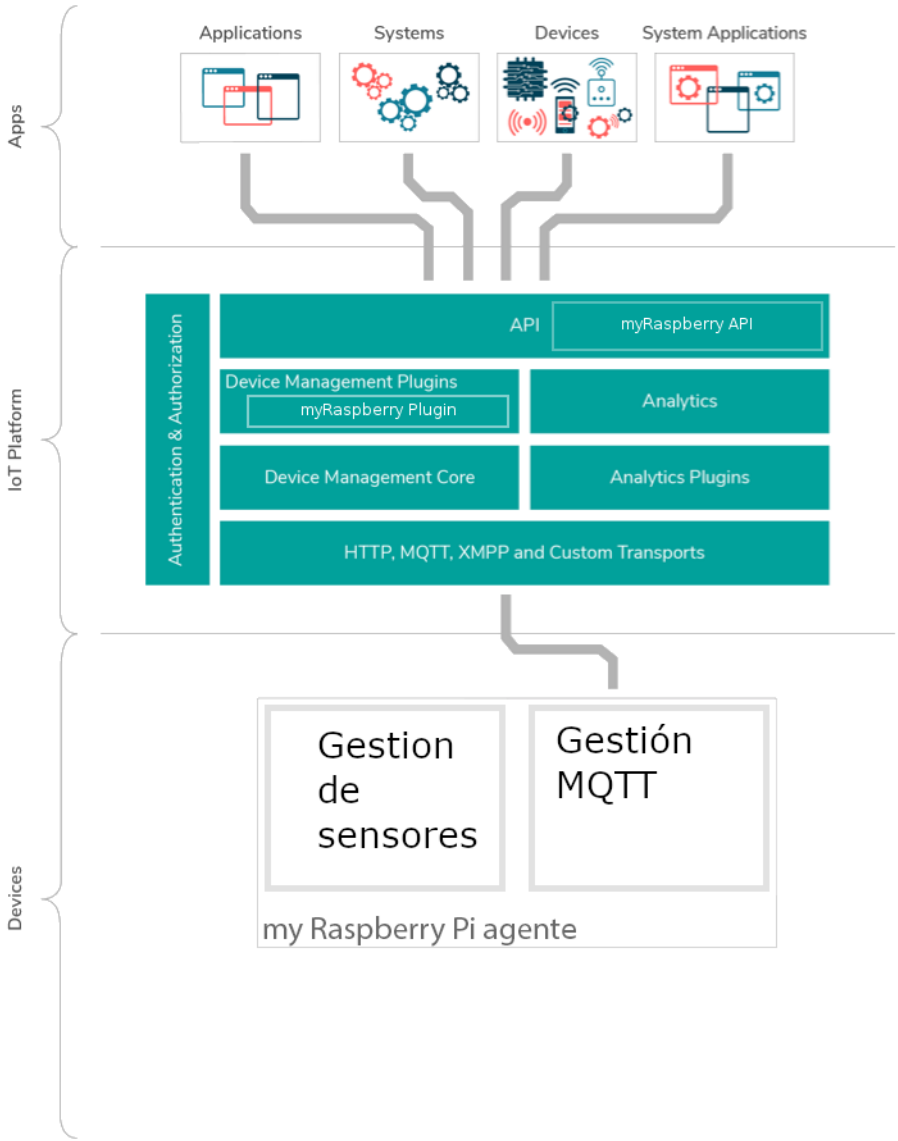


Figura 4.10 – Esquema modular del servidor WSO2 IoT

Para desarrollar la API, hacemos uso del directorio y del esqueleto ya generado gracias al arquetipo, esto nos otorga por un lado la carpeta **webapp**, en ella se encuentran distintos tipos de ficheros que no se quieren servir al usuario y que tienen alguna utilidad en el servidor, como por ejemplo ficheros de configuración, etc. Se dividen en la carpeta WEB-INF y META-INF.

Veamos el fichero de configuración **web.xml** dentro de WEB-INF:

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
         version="2.5"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         metadata-complete="true">
    <display-name>WSO2 IoT Server</display-name>
    <description>WSO2 IoT Server</description>

    <servlet>
        <servlet-name>CXFServlet</servlet-name>
        <servlet-
class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>CXFServlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
    <context-param>
        <param-name>doAuthentication</param-name>
        <param-value>true</param-value>
    </context-param>
    <context-param>
        <param-name>isSharedWithAllTenants</param-name>
        <param-value>false</param-value>
    </context-param>

    <!--publish to apim-->
    <context-param>
        <param-name>managed-api-enabled</param-name>
        <param-value>true</param-value>
    </context-param>

</web-app>
```

Código 4.1 – Fichero web.xml

Se ha generado mediante Apache CXF, que nos proporciona ayuda a la hora de desplegar y desarrollar APIs de *Front-end* como por ejemplo JAX-RS, que es el caso que nos atañe. Para ver más a fondo los ficheros *xml* contenidos en el directorio **webapp** véase el anexo V.

Tenemos también el directorio **src/main/java/sensorboard/raspberry/api** donde está el código de la API que hace uso de *swagger* y JAX-RS (API de java que proporciona métodos y clases para desarrollo de APIs Rest), las dos clases principales son las ya comentadas con anterioridad. Veamos la interfaz *myRaspberryService* en el fichero **myRaspberryService.java** la importancia de esta interfaz, reside en que usa la notación *swagger* para definir las peticiones HTTP a tratar y cómo van a ser tratadas.

myRaspberryService.java:

Como ya hemos indicado, se definen las distintas opciones y parámetros de la API, por un lado, definimos el scope, que necesitamos a la hora de generar el token de acceso, mediante la notación *@scope* y por otro, definimos cómo se va a tratar una llamada; con *@path* definimos la URI a la que se asocia.

Lo primero que debemos hacer es definir el contexto sobre el que se va a agrupar el resto de las definiciones de funcionalidades, como vemos en el código 4.2.

```
@SwaggerDefinition(
  info = @Info(
    version = "1.0.0",
    title = "myRaspberry",
```

```
    extensions = {
      @Extension(properties = {
        @ExtensionProperty(name="name",value=myRaspberryConstants.
DEVICE_TYPE),
        @ExtensionProperty(name="context",value="/"+myRaspberryConstants.
DEVICE_TYPE),
      })
    }
  ),
  tags = {
    @Tag(name  =  myRaspberryConstants.DEVICE_TYPE+",device_management",
description = "")
  }
)
@Scopes(
  scopes = {
    @Scope(
      name = "Enroll device",
      description = "",
      key = "perm:" + myRaspberryConstants.DEVICE_TYPE + ":enroll",
      permissions       =       {"/device-mgt/devices/enroll/"       +
myRaspberryConstants.DEVICE_TYPE }
    )
  }
)
```

Código 4.2 – Swagger Definition – myRaspberryService.java

La clase *myRaspberryConstants* contiene una serie de constantes que van a ser usadas por todo el código. Una de esas constantes es *DEVICE_TYPE* (véase Anexo VI) que tiene el nombre del dispositivo desarrollado (myRaspberry). Haciendo uso de este valor, tal y como vemos en el código 4.2, definimos la raíz para las peticiones a la API como la concatenación de la constante *DEVICE_TYPE* / versión. Particularizando sería: "*/myRaspberry/1.0.0"* donde "1.0.0" corresponde al campo *version* definida en el código 4.2 como la versión de la API, gracias a esto, podríamos implementar varias versiones de una API y distinguirlas.

Posteriormente, se crea la interfaz *myRaspberryService* para implementarla con la clase *myRaspberryServiceImpl*.

Los distintos recursos que va a tener disponible la API se van a implementar también con notación *Swagger* para APIs. Analicemos en el código 4.3 la implementación de uno de estos recursos, como es "*sensor*" que sigue una implementación lo suficientemente genérica para entender el resto y su valor indica el tiempo de muestreo del sensor de temperatura.

```
    /**
     * @param deviceId unique identifier for given device type instance
     * @param state    new time value
     */
    @Path("device/{deviceId}/sensor")
    @POST
    @ApiOperation(
            consumes = MediaType.APPLICATION_JSON,
            httpMethod = "PUT",
            value = "Tiempo de muestreo del sensor",
            notes = "",
            response = Response.class,
            tags = DeviceTypeConstants.DEVICE_TYPE ,
            extensions = {
                    @Extension(properties = {
```

```
                            @ExtensionProperty(name = SCOPE, value =
"perm:" + DeviceTypeConstants.DEVICE_TYPE + ":enroll")
                    })
            }
    )
    Response changeTime(@PathParam("deviceId") String deviceId,
                        @QueryParam("tiempo") int state,
                        @Context HttpServletResponse response);
```

Código 4.3 – Recurso "sensor" – myRaspberryService.java

Como podemos apreciar en el código 4.3, gracias a *@ApiOperation* estamos definiendo el tratamiento por parte de la API que va a hacer de la petición entrante *http/https*. En el caso a tratar, vemos que debe ser una petición con un método PUT, *Json* como parámetros y por último que el *@path* equivale a la URI a donde deben ser dirigidas las peticiones, en cuyo caso es *("device/{deviceId}/sensor")*.

Este esquema es el mismo para el resto de métodos de la interfaz, pero el método *downloadSketch* cambia totalmente el funcionamiento ya que no es convencional, pues devuelve un fichero zip con el agente, pero también registra el dispositivo en el servidor:

```
    @Path("/device/download")
    @GET
    @Produces("application/zip")
```

myRaspberryServiceImpl.java:

Es la clase que implementa la previamente comentada interfaz *myRaspberryService* desarrollando los métodos que introduce la interfaz para darle funcionalidad real a las peticiones de la API.

Esta implementación es el código encargado de gestionar las peticiones entrantes al servidor, que tienen como objetivo (URI) la API que vamos a crear. Se encarga de invocar las acciones necesarias y devolver el resultado deseado, entre las acciones que realiza se encuentra usar el bróker MQTT para invocar cambios en los agentes registrados en el servidor.

Analicemos el método introducido anteriormente *changeTime*:

```
public Response changeTime(@PathParam("deviceId") String deviceId,
                           @QueryParam("tiempo") int time,
                           @Context     HttpServletResponse     response)
{//throws Exception {
  try {     //Vemos si está autorizado el usuario para el dispositivo
      if
(!APIUtil.getDeviceAccessAuthorizationService().isUserAuthorized(new
          DeviceIdentifier(deviceId,  DeviceTypeConstants.DEVICE_TYPE)))
{
          return
Response.status(Response.Status.UNAUTHORIZED.getStatusCode()).build();
    }
```

Código 4.4.1 – Método ChangeTime – myRaspberryServiceImpl.java

Lo primero que hacemos en el código 4.4.1 es comprobar si la entidad que está llamando a la API está autorizado a realizar esta llamada, en caso de que no lo estuviera, se devolvería un código de error *http* 401 (Unauthorized).

```
   //Vemos si el string introducido por el usuario está vacío o no.
            if( time < 0 || time == 0 ) {
              log.error("The  requested  time  value  for  the  sensor
retrieving must be >0");
              return
Response.status(Response.Status.BAD_REQUEST.getStatusCode()).build();
            }
           //Configuremos la operación para mandarla
           Map<String, String> dynamicProperties = new HashMap<>();
           String                    publishTopic                    =
```

```
APIUtil.getAuthenticatedUserTenantDomain()
                    + "/" + DeviceTypeConstants.DEVICE_TYPE + "/command";

dynamicProperties.put(DeviceTypeConstants.ADAPTER_TOPIC_PROPERTY,
publishTopic);

            Operation commandOp = new CommandOperation();
            commandOp.setCode("change-time");
            commandOp.setType(Operation.Type.COMMAND);
            commandOp.setEnabled(true);
            commandOp.setPayLoad("timeRequest:"+Integer.toString(time));
```

Código 4.4.2 – Método changeTime – myRaspberryServiceImpl.java

En el código 4.4.2 pasamos a hacer comprobaciones de los parámetros de entrada, si el nuevo valor de tiempo es menor o igual que 0 procederemos a devolver un error *http* 400 (Bad Request), en caso contrario, seleccionamos el topic *MQTT* sobre el que vamos a enviar las peticiones a los agentes y almacenamos la operación.

```
          //Método propio que almacena los comandos
            try {
              JsonUtils.saveOperation("Cambiar         Tiempo         del
Sensor",deviceId,"Tiempo introducido:" + Integer.toString(time));
            }catch(Exception e) {
              System.out.println("ERROR" + e.toString());
            }
```

Código 4.4.3 – Método changeTime – myRaspberryServiceImpl.java

Se ha creado la clase *JsonUtils* que almacena en un *Json* un histórico de las operaciones realizadas, para poder luego mostrarlas en la interfaz de usuario.

```
            //Configuramos MQTT
            Properties props = new Properties();
            props.setProperty("mqtt.adapter.topic", publishTopic);
            commandOp.setProperties(props);
```

Dejamos configuradas las propiedades necesarias de *MQTT*.

```
List<DeviceIdentifier> deviceIdentifiers = new ArrayList<>();
            deviceIdentifiers.add(new          DeviceIdentifier(deviceId,
DeviceTypeConstants.DEVICE_TYPE));

//Aqui metemos la operación a la lista de operaciones para ser usadas, a
través de git hub podemos

//ver las distintas clases involucradas en https://github.com/wso2/carbon-
device-mgt/

APIUtil.getDeviceManagementService().addOperation(DeviceTypeConstants.DEV
ICE_TYPE, commandOp,
                    deviceIdentifiers);
            return Response.ok().build();
        } catch (DeviceAccessAuthorizationException e) {
            log.error(e.getErrorMessage(), e);
            return
Response.status(Response.Status.INTERNAL_SERVER_ERROR).build();
        } catch (OperationManagementException e) {
            String msg = "Error occurred while executing command operation
upon changing the sensor time";
```

```
            log.error(msg, e);
            return
Response.status(Response.Status.INTERNAL_SERVER_ERROR).build();
        } catch (InvalidDeviceException e) {
            String msg = "Error occurred while executing command operation
upon changing the sensor time";
            log.error(msg, e);
            return Response.status(Response.Status.BAD_REQUEST).build();
        }
    }
```

Código 4.4.4 – Método changeTime – myRaspberryServiceImpl.java

Por último, usamos la clase *DeviceManagementService*, disponible en la librería de *WSO2*, para encolar las operaciones. Hay que tener en cuenta que esta clase está implementada por *WSO2*, para tener más información sobre el proceso de encolado y almacenamiento de operaciones ha hecho falta una inversión de tiempo y esfuerzo recorriendo el repositorio en *GitHub* que contiene el código de las clases heredadas.

En el caso de que todo haya sido satisfactorio, devolvemos un código *http* 200 (*OK*), indicando que todo ha sido correcto a la entidad que haya realizado la petición a la API.

JsonUtils.java:

Esta clase usa la librería *Gson* de Google para crear un *Json* y al igual que *deviceOperationJson* ha sido desarrollada desde 0 para almacenar las distintas operaciones que el usuario realice, posteriormente este *Json* será servido y mostrado a través de la interfaz de usuario en forma de histórico de operaciones realizadas.

Esta clase usa a su vez la clase *deviceOperationJson* que define las características y métodos de la estructura de datos.

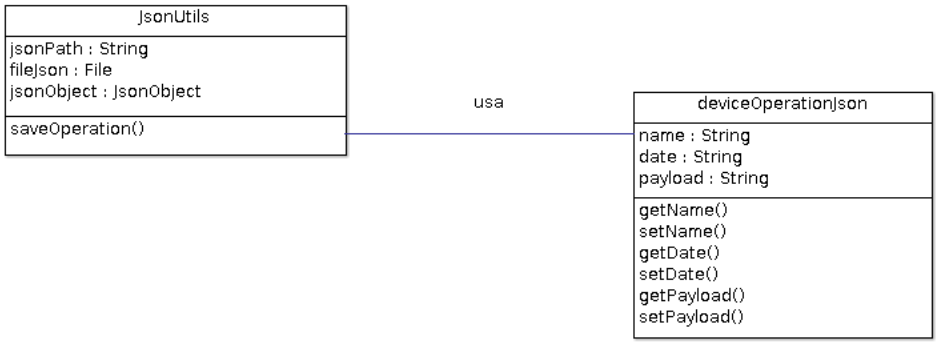


Figura 4.11 – Diagrama UML de *JsonUtils*

Veamos el código de la clase:

```
public class JsonUtils {
private static String jsonPath =

"repository/deployment/server/jaggeryapps/devicemgt/sampleData.json";
private static File fileJson = new File( jsonPath  );
private static JsonObject jsonObject = new JsonObject();
```

Código 4.5.1 – Clase jsonUtils – jsonUtils.java

En el código 4.5.1, observamos las variables de la clase, son las necesarias para crear o modificar el fichero *Json*, como por ejemplo fileJson, que es el descriptor del fichero *Json.*

```
public static void saveOperation ( String operationId, String deviceId,
String payload ){
/*
 *  Vamos a configurar la variable pendingDeviceOperation la cual va a
almacenar los datos
```

```
 *  de la última operación enviada al dispositivo identificado por deviceId
 *
 */
deviceOperationJson pendingDeviceOperation = new deviceOperationJson();
Gson gson = new Gson();
String jsonString;
pendingDeviceOperation.setName(operationId);
pendingDeviceOperation.setDate();
pendingDeviceOperation.setPayload(payload);
```

Código – 4.5.2 Clase jsonUtils – jsonUtils.java

Con el código 4.5.2 creamos una nueva instancia de la clase *deviceOperationJson* que representa a la operación a almacenar y le damos valores.

```
// Si el fichero json no existe lo inicializamos y metemos la nueva (y
unica) operación,
// si el fichero existe, tenemos que parsear las operaciones existente y
almacenar
// en la del deviceId
if (fileJson.exists())
{
  try {
  //Creamos un parser, que nos va a servir para leer el fichero JSON
existente,
  //con el vamos a leer como un jsonElement, sin determinar que tipo json
va a ser
  Path path = new File(jsonPath).toPath();
  Reader reader = Files.newBufferedReader(path, StandardCharsets.UTF_8);
  JsonParser parser = new JsonParser();
  JsonElement jsonElement = parser.parse(new FileReader( jsonPath  ));

  //Pasamos todo el contenido del fichero json (en jsonElement) a
JsonObject,
  //el cual nos da métodos para poder usarlo.
    jsonObject = jsonElement.getAsJsonObject();

    //Comprobamos si la estructura json tiene un miembro ya existente del
  //deviceId que deseamos añadir.
  if ( !jsonObject.has(deviceId) )
  {
  //Si no tiene un miembro del dispositivo que ha ejecutado la operación,
  //entonces necesitamos crear y añadir el nuevo miembro y luego añadirle
  //la operación.

    FileWriter writer = new FileWriter(jsonPath);
    //Las operaciones se guardan y se leen del fichero como JsonArray
    ArrayList<deviceOperationJson> operationsArrayList = new
                                   ArrayList<deviceOperationJson>( );
    operationsArrayList.add(pendingDeviceOperation);

    //Pasamos el array a la estructura JSON y lo añadimos al fichero leido
y
    //almacenado en jsonObject
    jsonObject.add(deviceId, gson.toJsonTree(operationsArrayList));

      //Gson nos proporciona el siguiente método para pasar de estructura
```

```
JSON
    //a un string y así poder escribirlo en el fichero
      jsonString = gson.toJson(jsonObject);
    writer.write(jsonString);
    writer.close();
    }
```

Código 4.5.3 – Clase jsonUtils – jsonUtils.java

En el código 4.5.3 comprobamos si existe el fichero *Json*, en caso de que exista, vamos a recorrerlo y leerlo para comprobar si existe ya un elemento con el id del dispositivo que ha generado la operación. En caso de que no existiera, lo crearíamos y añadiríamos a la lista.

```
  else
  {
    //Hemos comprobado que el dispositivo ya tiene una estructura creada
    //en el fichero JSON, lo cual nos evita tener que crearla de nuevo,
    //solo tenemos que seleccionar en un array el campo de JSON que tenga
      //por nombre el deviceId.
    deviceOperationJson[] operationArray =

gson.fromJson(jsonObject.get(deviceId),deviceOperationJson[].class );

    //Pasamos a un arraylist, ya que el array nos limita a un tamaño fijo.
    ArrayList<deviceOperationJson> operationsArrayList = new
                    ArrayList<deviceOperationJson>(Arrays.asList(
operationArray ));

    //Al ArrayList de todas las operaciones del dispositivo que queremos
    //le añadimos la operación nueva que queremos almacenar y
sobreescribimos
    //el campo del dispositivo en el fichero JSON.

    operationsArrayList.add(pendingDeviceOperation);
    jsonObject.add(deviceId, gson.toJsonTree(operationsArrayList));
    jsonString = gson.toJson(jsonObject);
    FileWriter writer = new FileWriter(jsonPath);
    writer.write(jsonString);
    writer.close();
  }
```

Código 4.5.3 – Clase jsonUtils - jsonUtils.java

En caso de que ya exista (código 4.5.3), necesitamos almacenar la lista de operaciones de ese dispositivo, añadirle el nuevo y por último sobreescribir en el fichero.

```
}
  catch (Exception e1)
  {
    e1.printStackTrace();
  }
  }
  //Si el fichero no existe, debemos llamar a la función que lo crea y
luego añadirle el
//dispositivo, esto ocurre solamente la primera vez, cuando se inicia todo
el sistema,
//tras enrolar un dispositivo y ejecutar una operación.
else
{
  try
  {
```

```
    fileJson.createNewFile();
    //JsonObject object = new JsonObject();
    FileWriter writer = new FileWriter(jsonPath);
    ArrayList<deviceOperationJson> operationsArrayList = new
                                   ArrayList<deviceOperationJson>( );

    operationsArrayList.add(pendingDeviceOperation);

   //object.add(deviceId, gson.toJsonTree(operationsArrayList));
   jsonObject.add(deviceId, gson.toJsonTree(operationsArrayList));

   //String jsonArray = gson.toJson(jsonObject);
   jsonString = gson.toJson(jsonObject);

   writer.write(jsonString);
   writer.close();
   }catch (Exception e) {
     e.printStackTrace();
     }
   }
}
```

Código 4.5.4 – Clase jsonUtils - jsonUtils.java

En caso de que no exista el fichero (código 4.5.4), primero procederemos a crearlo, para luego añadirle la primera operación del dispositivo que la ha ejecutado.

Finalmente, tenemos disponible una interfaz de gestión para la API del dispositivo, donde vemos las distintas llamadas posibles y la URI para invocarlas.

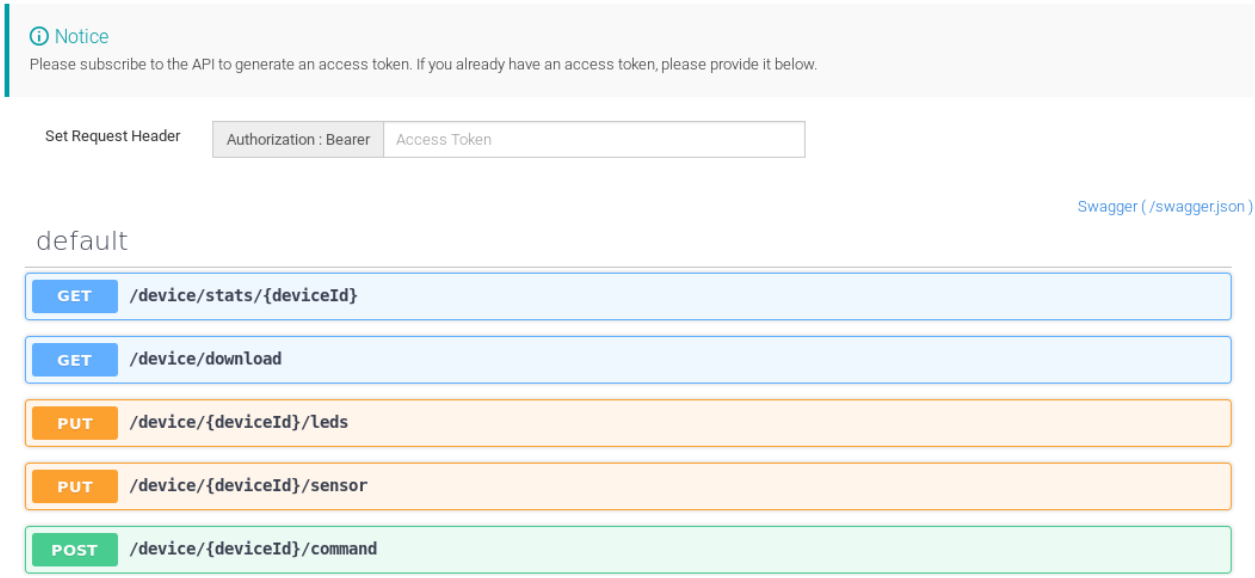


Figura 4.12 – Interfaz de gestión de la API

4.3.2 Plugin del servidor WSO2 IoT

Como ya hemos indicado con anterioridad, un plugin en nuestro caso es un paquete OSGi que se conecta con el CDMF del servidor WSO2. Permite desarrollar un nuevo tipo de dispositivo en el servidor e interconectarlo con el *framework* de dispositivo conectado (CDMF) y así tener control sobre los distintos agentes registrados (por ejemplo, permitir que se registren, borrarlos, definir funcionalidades, etc) y para gestionar los datos enviados por los agentes.

Cuando el servidor WSO2 se inicia, se registran en el CDMF todos los tipos de dispositivos o categorías de servicios que existen en el servidor gracias a OSGi. OSGi viene implementado en WSO2 para que los reconozca en tiempo de ejecución.

Para que nuestro tipo de dispositivo sea reconocido por OSGi necesitamos que implemente una interfaz OSGi. Con este objetivo hemos desarrollado la clase *myRaspberryManagerService*, esta clase define como se maneja e interacciona con el tipo de dispositivo al que hace referencia, extendiendo también a la interfaz desarrollada por WSO2 con este proposito:

org.wso2.carbon.device.mgt.common.spi.DeviceManagementService

*(*https://github.com/wso2/carbon-device-mgt/tree/master/components/device-mgt/org.wso2.carbon.device.mgt.common/src/main/java/org/wso2/carbon/device/mgt/common/spi*)*

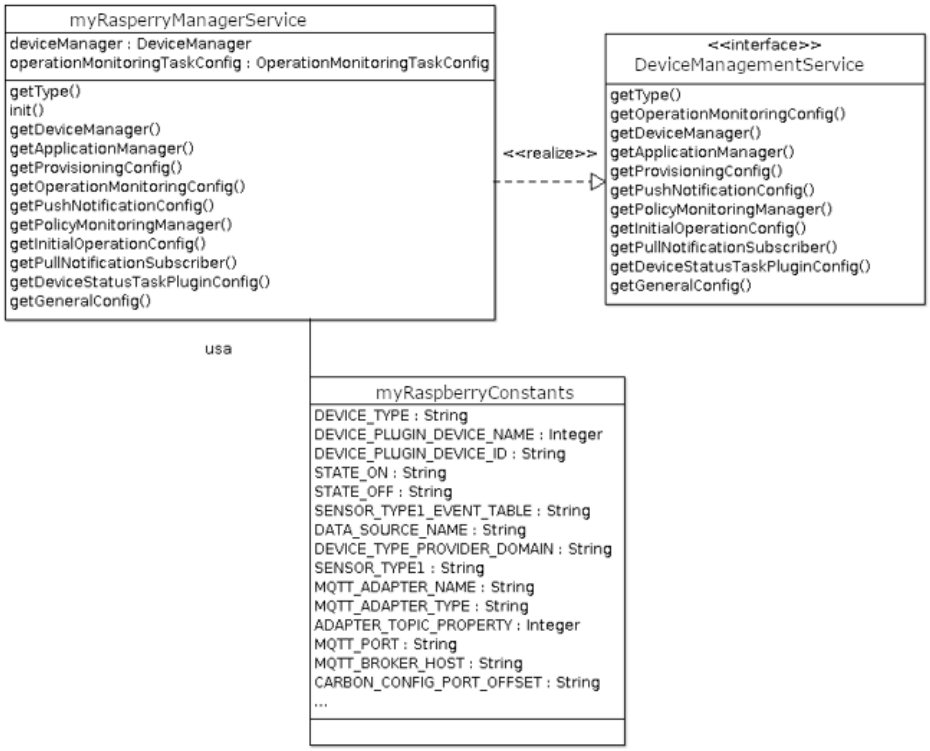


Figura 4.13 – Diagrama UML de myRaspberryManagerService

myRaspberryManagerService.java:

Analicemos a continuación los métodos más interesantes de la clase *myRaspberryManagerService:*

```
public class myRaspberryManagerService implements DeviceManagementService
{
    private DeviceManager deviceManager;
    private OperationMonitoringTaskConfig operationMonitoringTaskConfig;

    @Override
    public String getType() {
        return DeviceTypeConstants.DEVICE_TYPE;
    }
```

Código 4.6 – Método getType – myRaspberryManagerService.java

En el código 4.6 vemos el método *getType()* que sirve para devolver el tipo del dispositivo, almacenado en la clase *myRaspberryConstants* que contiene las constantes.

```
    @Override
    public void init() throws DeviceManagementException {
        this.deviceManager = new myRaspberryManager();
        this.operationMonitoringTaskConfig              =              new
```

```
OperationMonitoringTaskConfig();
    }
```

Código 4.7 – Inicializador de myRaspberryManagerService – myRaspberryManagerService.java

El método inicializador del código 4.7, nos inicializa una instancia de la clase *myRaspberryManager* desarrollada que analizaremos posteriormente, también inicializa el monitor de operaciones para realizar procedimientos de operaciones, como encolar operaciones requeridas por la API.

Esta clase gestiona el tipo de diposistivo implementado y el método inicializador es invocado por el CDMF cuando busca los tipos de dispositivos que implementan la interfaz *deviceManagementService* desarrollada por *WSO2*.

```
    @Override
    public PushNotificationConfig getPushNotificationConfig() {

        Map<String, String> properties = new HashMap<>();
        properties.put("mqttAdapterName", "raspberry_mqtt");
        properties.put("username", "admin");
        properties.put("password", "admin");
        properties.put("qos", "0");
        properties.put("clearSession", "true");
        properties.put("scopes", "");
        return new PushNotificationConfig("MQTT",false, properties);
    }
```

Código 4.8 – Método getPushNotificationConfig – myRaspberryManagerService.java

En el código 4.8 lo que apreciamos es el código encargado de crear la configuración para el *topic* de MQTT con el que nos interconectaremos con el agente en la Raspberry Pi, entre las cosas que podemos está la QoS de entrega, donde el valor 0 se refiere al ya comentado "*At most once"*.

myRaspberryManager.java:

Esta clase implementa a la clase *org.wso2.carbon.device.mgt.common.DeviceManager* desarrollada por WSO2; *myRaspberryManager* es instanciada por el gestor de servicio *myRaspberryManagerService* mencionado anteriormente y este objeto se encargará de gestionar todo el conjunto de dispositivos con acciones como:

- Registrar y borrar dispositivos o agentes.
- Activar y desactivar dispositivos o agentes.

La clase *myRaspberryManager* instancia a su vez un objeto de la clase *myRaspberryFeatureManager* que será analizado posteriormente y se encarga de gestionar las distintas funcionalidades u operaciones del tipo de dispositivo (myRaspberry) que hemos desarrollado.

Por otra parte, hacemos uso de la clase *myRaspberryDAO* (Anexo VII) para tratar con la base de datos que tiene exclusivamente este tipo de dispositivo a implementar.

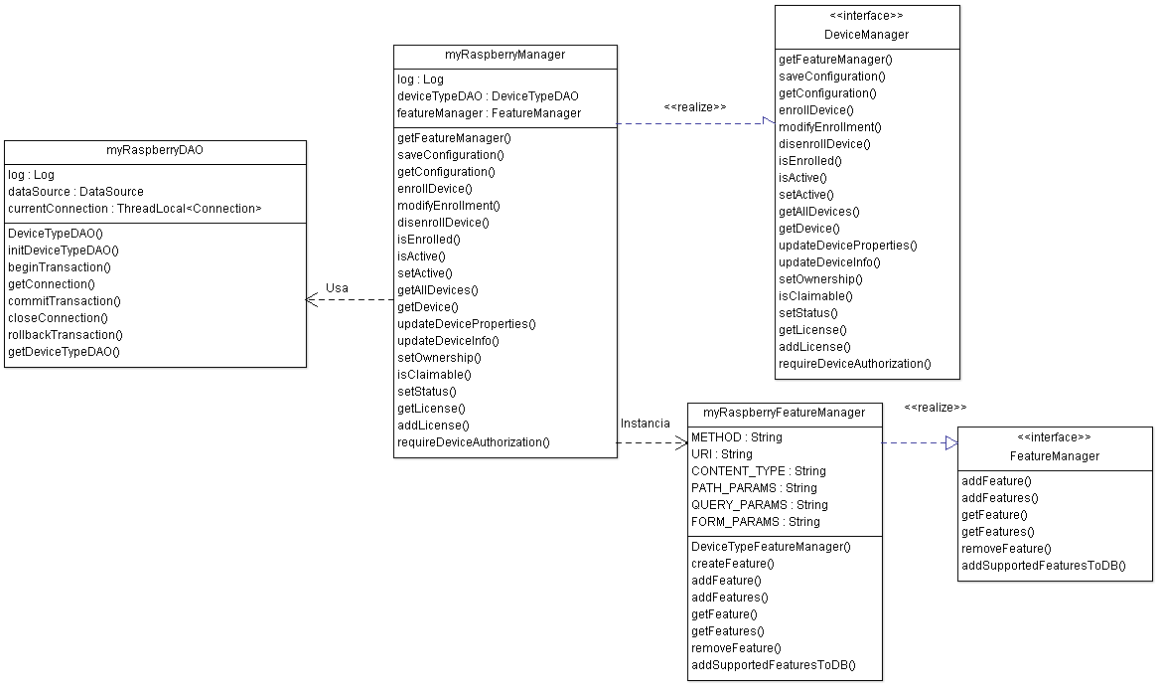


Figura 4.14 – Diagrama UML de myRaspberryManager

Veamos a continuación el código de la clase *myRaspberryManager*.

```
public class myRaspberryManager implements DeviceManager {

    private static final Log log =
LogFactory.getLog(DeviceTypeManager.class);
    private static final DeviceTypeDAO deviceTypeDAO = new
DeviceTypeDAO();
    private FeatureManager featureManager = new
DeviceTypeFeatureManager();

    @Override
    public FeatureManager getFeatureManager() {
        return featureManager;
    }

    @Override
    public boolean saveConfiguration(PlatformConfiguration
platformConfiguration) throws DeviceManagementException {
        return false;
    }

    @Override
    public PlatformConfiguration getConfiguration() throws
DeviceManagementException {
        return null;
    }

    @Override
    public boolean enrollDevice(Device device) throws
DeviceManagementException {
        boolean status;
        try {
            if (log.isDebugEnabled()) {
```

```
                log.debug("Enrolling a new raspberry device : " +
device.getDeviceIdentifier());
            }
            DeviceTypeDAO.beginTransaction();
            status = deviceTypeDAO.getDeviceTypeDAO().addDevice(device);
            DeviceTypeDAO.commitTransaction();
        } catch (DeviceMgtPluginException e) {
            try {
                DeviceTypeDAO.rollbackTransaction();
            } catch (DeviceMgtPluginException iotDAOEx) {
                log.warn("Error occurred while roll back the device enrol
transaction :" + device.toString(), iotDAOEx);
            }
            String msg = "Error while enrolling the raspberry device : "
+ device.getDeviceIdentifier();
            log.error(msg, e);
            throw new DeviceManagementException(msg, e);
        }
        return status;
    }
```

Código 4.9 – Clase myRaspberryManager – myRaspberryManager.java

En el código 4.9, el método *enrollDevice* se encarga de gestionar el registro de nuevos dispositivos en la plataforma, es invocado por la API, por el método *downloadSketch*, que a su vez es llamado por la interfaz de usuario al descargar el código del agente o por llamadas a la API, siendo esta la forma de registrar un dispositivo nuevo.

El método *enrollDevice* toma la información del dispositivo a registrar como argumento de entrada y haciendo uso del conector con la base de datos *DeviceTypeDAO* lo añade a ella, finalmente devuelve el resultado de invocar el método *addDevice* del conector.

```
    @Override
    public     boolean     modifyEnrollment(Device     device)     throws
DeviceManagementException {
        boolean status;
        try {
            if (log.isDebugEnabled()) {
                log.debug("Modifying the raspberry device enrollment
data");
            }
            DeviceTypeDAO.beginTransaction();
            status                                                    =
deviceTypeDAO.getDeviceTypeDAO().updateDevice(device);
            DeviceTypeDAO.commitTransaction();
        } catch (DeviceMgtPluginException e) {
            try {
                DeviceTypeDAO.rollbackTransaction();
            } catch (DeviceMgtPluginException iotDAOEx) {
                String msg = "Error occurred while roll back the update
device transaction :" + device.toString();
                log.warn(msg, iotDAOEx);
            }
            String msg = "Error while updating the enrollment of the
raspberry device : " +
                    device.getDeviceIdentifier();
            log.error(msg, e);
```

```
            throw new DeviceManagementException(msg, e);
        }
        return status;
    }
```

Código 4.10 – Método modifyEnrollment – myRaspberryManager.java

En el código 4.10 vemos el método *modifyEnrollment* volvemos a hacer uso de la clase *myRaspberryDAO* para obtener y modificar los valores que ya teníamos de un dispositivo registrado previamente, cambiando el valor existente por el valor que recibe como argumento de entrada.

```
@Override
    public boolean disenrollDevice(DeviceIdentifier deviceId) throws
DeviceManagementException {
        boolean status;
        try {
            if (log.isDebugEnabled()) {
                log.debug("Dis-enrolling raspberry device : " + deviceId);
            }
            DeviceTypeDAO.beginTransaction();
            status                                                    =
deviceTypeDAO.getDeviceTypeDAO().deleteDevice(deviceId.getId());
            DeviceTypeDAO.commitTransaction();
        } catch (DeviceMgtPluginException e) {
            try {
                DeviceTypeDAO.rollbackTransaction();
            } catch (DeviceMgtPluginException iotDAOEx) {
                String msg = "Error occurred while roll back the device
dis enrol transaction :" + deviceId.toString();
                log.warn(msg, iotDAOEx);
            }
            String msg = "Error while removing the raspberry device : " +
deviceId.getId();
            log.error(msg, e);
            throw new DeviceManagementException(msg, e);
        }
        return status;
    }
```

Código 4.11 – Método disenrollDevice – myRaspberryManager.java

Por último, en el código 4.11 vemos el método *disenrollDevice* es llamado cuando queremos desactivar un dispositivo registrado previamente y almacenado en la base de datos. Para ello, necesitamos hacer uso una vez más de la clase *myRaspberryDAO* y así borrar los registros que tengamos de él en la base de datos.

myRaspberryFeatureManager.java:

Por último, vamos a ver el método encargado de la gestión de las operaciones, para ello, hablaremos primero de los métodos más relevantes que tiene esta clase:

- Constructor de clase: Se ha creado para que registre mediante el método create*Feature* las funcionalidades u operaciones que deseamos que tenga este tipo de dispositivos, las cuales listamos a continuación:
    - Cambiar los leds, para encender o apagar la matriz de leds (on/off)
    - Cambiar tiempo de muestreo, permite definir el intervalo de tiempo entre muestras del sensor de temperatura
    - Mandar orden, manda una orden al dispositivo, puede ser apagar, reiniciar o un comando en Bash.

```
  public myRaspberryFeatureManager() {
        /*
```

```
        * Hemos modificado el constructor método para que solamente cree
las
        * operaciones al llamar a la función que hemos creado para que lo
        * gestione, llamada createFeature, tenemos que tener en cuenta que
las
        * operaciones deben estar definidas y configuradas (como los
iconos) en
        * type-view/private/config.json
        */
Feature deviceFeature = createFeature("change-time","tiempo",
  "Cambiar Tiempo del Sensor","Cambia el tiempo de muestreo del sensor de
temperatura");
  try {
    addFeature(deviceFeature);
  }catch (DeviceManagementException e) {
    e.printStackTrace();
  }
    deviceFeature  =  createFeature("change-leds","estado","Matriz  leds:
on/off",
  "Enciende/Apaga la matriz de leds, acepta solo las opciones on y off");
try {
  addFeature(deviceFeature);
}catch (DeviceManagementException e) {
    e.printStackTrace();
}
deviceFeature = createFeature("send-command","parametros","Mandar Orden",
  "Manda una orden de las listadas a la raspberry");
try {
  addFeature(deviceFeature);
}catch (DeviceManagementException e) {
  e.printStackTrace();
}
}
```

Código 4.12 – Clase myRaspberryFeatureManager – myRasberryFeatureManager.java

La creación de nuevas funcionalidades, queda a expensas de nuevas necesidades, siendo la implementación muy simple, tal y como vemos en el código 4.12, gracias al método *createFeature.*

```
private Feature createFeature(String feature, String queryParam,
String name, String description)
{
Feature newOperation = new Feature();
Map<String, Object> apiParams = new HashMap<>();
  List<String> pathParams = new ArrayList<>();
  List<String> queryParams = new ArrayList<>();
  List<String> formParams = new ArrayList<>();
  List<Feature.MetadataEntry> metadataEntries = new ArrayList<>();
  Feature.MetadataEntry metadataEntry = new Feature.MetadataEntry();
  newOperation.setCode(feature);
  newOperation.setName(name);
  newOperation.setDescription(description);
  apiParams.put(METHOD, "POST");
  apiParams.put(URI, "/myRaspberry/device/{deviceId}/" + feature);
  pathParams.add("deviceId");
  apiParams.put(PATH_PARAMS, pathParams);
```

```
// Si existe queryParam suministrado a la función, lo añadimos.
if(queryParam != null)
queryParams.add(queryParam);
apiParams.put(QUERY_PARAMS, queryParams);
apiParams.put(FORM_PARAMS, formParams);
metadataEntry.setId(featureCount);
metadataEntry.setValue(apiParams);
metadataEntries.add(metadataEntry);
featureCount++; //Aumentamos el contador de featureCount, para los Id
newOperation.setMetadataEntries(metadataEntries);
return newOperation;
}
```

Código 4.13 – Método createFeature – myRasberryFeatureManager.java

- Método *createFeature* (código 4.13)*:* Con este método, tomamos los parámetros de entrada que serán valores de la nueva operación o funcionalidad de nuestro tipo de dispositivo:
  - *string feature*: Nombre con el que hemos identificado la operación en la API.
  - *string queryParam:* Parámetro definido para la operación en la API.
  - string name: Nombre de la operación, tal y como será mostrada en la interfaz de usuario.
  - string description: Descripción para la interfaz de usuario, ayudará a los usuarios a entender su función.

```
@Override public boolean addFeature(Feature feature) throws
DeviceManagementException {
  features.add(feature);
  return true;
}
```

Código 4.14 – Método addFeature – myRasberryFeatureManager.java

- Método *addFeature* (Código 4.14)*:* Este método añade la operación que recibe por parámetro a la lista de operaciones, que es una variable de la clase.

```
@Override public Feature getFeature(String name) throws
DeviceManagementException {
  Feature tmpFeature = null;
//iteramos hasta encontrar el index del que debemos borrar
for (int i=0; i < features.size(); i++){
  Feature row = (Feature) features.get(i);
  if (row.getName().equals(name))
    tmpFeature = row;
}
return tmpFeature;
}
```

Código 4.15 – Método getFeature – myRasberryFeatureManager.java

- Método *getFeature* (código 4.15)*:* Este método devuelve una de las operaciones, según el nombre que se le pase por parámetro de la lista de operaciones.

4.3.3 Agente de Raspberry Pi

El agente de la Raspberry Pi es el código que se ejecuta en los dispositivos, es el encargado de gestionar la lógica de sensores, la comunicación con el servidor a través de MQTT, el envío de datos, la gestión de la matriz de leds, así como tratar e interpretar las órdenes desde el servidor.

Cabe destacar que usaremos 3 librerías para su funcionamiento, por un lado, paho.mqtt (Anexo VII) para gestionar las operaciones de MQTT de la parte del agente, por otro, la librería w1thermsensor (ver Anexo VIII) para la obtención de datos del sensor de temperatura y por último la librería max7219 (Anexo IX) para gestionar

la matriz de leds.

Podemos dividir el agente en tres partes claramente diferenciadas:

- Fichero de configuración del dispositivo: Sigue un estilo de fichero ini, que se caracterizan por organizarse según una categoría entre corchetes, como pares de la forma parámetro=valor.

  Veamos el fichero **deviceConfig.properties**, este fichero, es tomado como plantilla por la API para generar la configuración inicial de un agente a la hora de generar el fichero zip, leyendo las marcas y modificándolas en función de lo que corresponda. Por otro lado, el agente lee el fichero y se configura según los valores que tenga o los modifica si fuera necesario, como pasa con *leds-state* y *push-interval*.

  El sistema de autenticación entre agente-servidor como ya hemos visto, se hace mediante un token generado previamente, este token tiene un tiempo de vida y un token de refresco que se usa para volver a generarlo. De esta forma logramos autenticar al agente y una comunicación segura entre el agente y el servidor, siendo toda esta comunicación cifrada extremo-extremo.

```
[Device-Configurations]
owner={DEVICE_OWNER}
deviceId={DEVICE_ID}
device-name={DEVICE_NAME}
server-name={SERVER_NAME}
controller-context=/myRaspberry
device-type=myRaspberry
mqtt-ep={MQTT_EP}
https-ep={HTTPS_EP}
auth-method=token
auth-token={DEVICE_TOKEN}
refresh-token={DEVICE_REFRESH_TOKEN}
push-interval=15
application-key={API_APPLICATION_KEY}
leds-state=OFF
```

- Scripts de arranque del agente y el script del servicio: Por un lado, tenemos tres scripts en Bash con funcionalidades distintas:
  - **configure.sh**: se crea con el objetivo de configurar de forma autónoma el script de servicio agentd.sh, el cual define el arranque de servicios en los distintos niveles de ejecución. Este script toma el valor del directorio actual y lo configura en el script de servicio agentd,sh, luego copia en el lugar de destino el script del servicio y configura el rc de Linux, encargado de gestionar la lógica de scripts de servicio.

```
#!/bin/bash
PATH_DIR=`echo $PWD/$0 | sed 's/configure.sh//g'`

sed -i "s+PATH_DIR+$PATH_DIR+g" agentd.sh
sudo cp ./agentd.sh /etc/init.d/
sudo update-rc.d agentd.sh defaults
```

Código 4.16 – Fichero configure.sh

  - ***agentd.sh****:* Este *script* sigue el estándar de creación de *scripts* de servicio de *Linux*, los cuales son usados para definir la ejecución del servicio en los distintos niveles de ejecución del sistema operativo. Para ello se usa tras el *shebang* una línea como podemos ver en el código 4.17, en la que se define los niveles de ejecución de Linux en los que se arranca y aquellos en los que se para. Se ha definido el valor 5 para su arranque (cuando arranca totalmente el sistema) y para el resto, que esté parado. Esto con prioridad 90, aquellos scripts de servicio con mayor prioridad serán arrancados antes que él.

```
### BEGIN INIT INFO
# Provides:          agentd
# Required-Start:
# Required-Stop:
# Default-Start:     90 5
# Default-Stop:      20 0 1 2 3 4 6
# Short-Description: agentd initscript
# Description:       Script de servicio para el agente de WSO2 IoT
### END INIT INFO
```

Código 4.17 – Cónfiguración de rc de Linux – agentd.sh

Se han creado las funciones *start* y *stop*, para que arranquen el servicio o lo paren y así tener control sobre el agente, tal y como vemos en el código 4.18.

```
# Start the service agent
start() {
        ### Create the lock file ###
        if [ -e /var/lock/subsys/agentScript ]; then
          echo "[ERROR] The agent is already initialized"
        else
          cd PATH_DIR
          sleep 60 && sudo PATH_DIR/agentScript.sh &
          touch /var/lock/subsys/agentScript
        fi
}
# Stop the service
stop() {
        if [ -e /var/lock/subsys/agentScript ]; then
          PROCESS_PID=`ps aux | grep agentScript.sh | head -n 1 | awk
'{print $2}'`
          PYTHON_PID=`ps aux | grep agent.py | head -n 1 | awk '{print
$2}'`
          sudo kill $PROCESS_PID 2&>/dev/null
          sudo kill $PYTHON_PID 2&>/dev/null
          ### Now, delete the lock file ###
          rm -f /var/lock/subsys/agentScript
        else
          echo "[ERROR] The agent is not initialized"
        fi
}
### main logic ###
case "$1" in
  start)
        start
        ;;
  stop)
        stop
        ;;
  restart|reload|condrestart)
        stop
        start
        ;;
  *)
        echo $"Usage: $0 {start|stop|restart|reload|status}"
        exit 1
esac
exit 0
```

Código 4.18 – Funciones start y stop – agentd.sh

- **agentScript.sh**: Este script en Bash tiene toda la lógica de inicialización del agente, primero comprueba si existe la librería *paho.mqtt,* en caso de que no exista la descarga, tras esto refresca el token de acceso y finalmente copia el fichero de configuración en la carpeta del código en Python para seguidamente lanzar el script *agent.py*.

```
currentDir=$PWD
LOG_FILE="agent.log"
for f in ./deviceConfig.properties; do
    ## Check if the glob gets expanded to existing files.
    ## If not, f here will be exactly the pattern above
    ## and the exists test will evaluate to false.
    if [ -e "$f" ]; then
      CURRENT_DAY=`date +"%Y-%m-%d"`
      CURRENT_TIME=`date +"%H:%M:%S"`
                                                                    echo
"$CURRENT_DAY $CURRENT_TIME,000 INFO     Configuration field found $f" >>
$LOG_FILE
    else
      CURRENT_DAY=`date +"%Y-%m-%d"`
      CURRENT_TIME=`date +"%H:%M:%S"`
      echo          "$CURRENT_DAY          $CURRENT_TIME,000          ERROR
'deviceConfig.properties' file does not exist in current path. Exiting
installation" >> $LOG_FILE;
      exit;
    fi
    ## We can exit the loop, there is a file and it is not empty.
    break
done
```

Código 4.19 – Comprobaciones iniciales – agentScript.sh

En el código 4.19, vemos la configuración de log, así como la comprobación de que el fichero de configuración existe.

```
PAHO_DIR="./paho.mqtt.python"
if [ ! -d $PAHO_DIR ]; then
  CURRENT_DAY=`date +"%Y-%m-%d"`
  CURRENT_TIME=`date +"%H:%M:%S"`
  echo "$CURRENT_DAY $CURRENT_TIME,000 INFO     Cloning into directory the
MQTT paho git project" >> $LOG_FILE
  #install mqtt dependency if the paho directory does not exists.
  git clone https://github.com/eclipse/paho.mqtt.python.git
  cd ./paho.mqtt.python
  sudo python setup.py install
fi
cd $currentDir
```

Código 4.20 – Comprobación de paho.mqtt – agentScript.sh

En el código 4.20 se descarga la librería de eclipse paho.mqtt, que es necesaria para gestionar en el agente en Python las operaciones sobre MQTT.

```
REFRESH_TOKEN=`cat ./deviceConfig.properties | grep refresh| awk -F "="
'{print $2}'`
DEVICE_ID=`cat ./deviceConfig.properties | grep deviceId | awk -F "="
'{print $2}'`
BASIC_ENCODED=`cat ./deviceConfig.properties | grep application | awk -F
"=" '{print $2}'`
DEVICE_TYPE=`cat ./deviceConfig.properties | grep device-type | awk -F "="
```

```
'{print $2}'`
SERVER=`cat ./deviceConfig.properties | grep https-ep | awk -F "=" '{print
$2}'| awk -F "/" '{print $3}' | awk -F ":" '{print $1}'`
AUTH_TOKEN=`curl                --silent                -k                -d
"grant_type=refresh_token&refresh_token=$REFRESH_TOKEN&scope=device_type_
$DEVICE_TYPE device_$DEVICE_ID" -H "Authorization: Basic $BASIC_ENCODED"
-H           "Content-Type:           application/x-www-form-urlencoded"
https://$SERVER:9443/oauth2/token | awk -F "," '{print $1}'| awk -F ":"
'{print $2}' | sed 's/\"//g'`
sed -i "s/auth-token=.*/auth-token=$AUTH_TOKEN/g" deviceConfig.properties
```

Código 4.21 – Refresco del token – agentScript.sh

A continuación, tal y como vemos en el código 4.21, se refresca el token de autenticación en caso de que hubiera expirado.

Por último, ejecuta el agente en Python.

```
./src/agent.py  --log $LOG_FILE
```

- Agente en Python: Es realmente el agente que incorpora toda la lógica de sensores, de control de MQTT, de gestión de fichero de configuración, podemos observar su complejidad en el diagrama de secuencia siguiente:

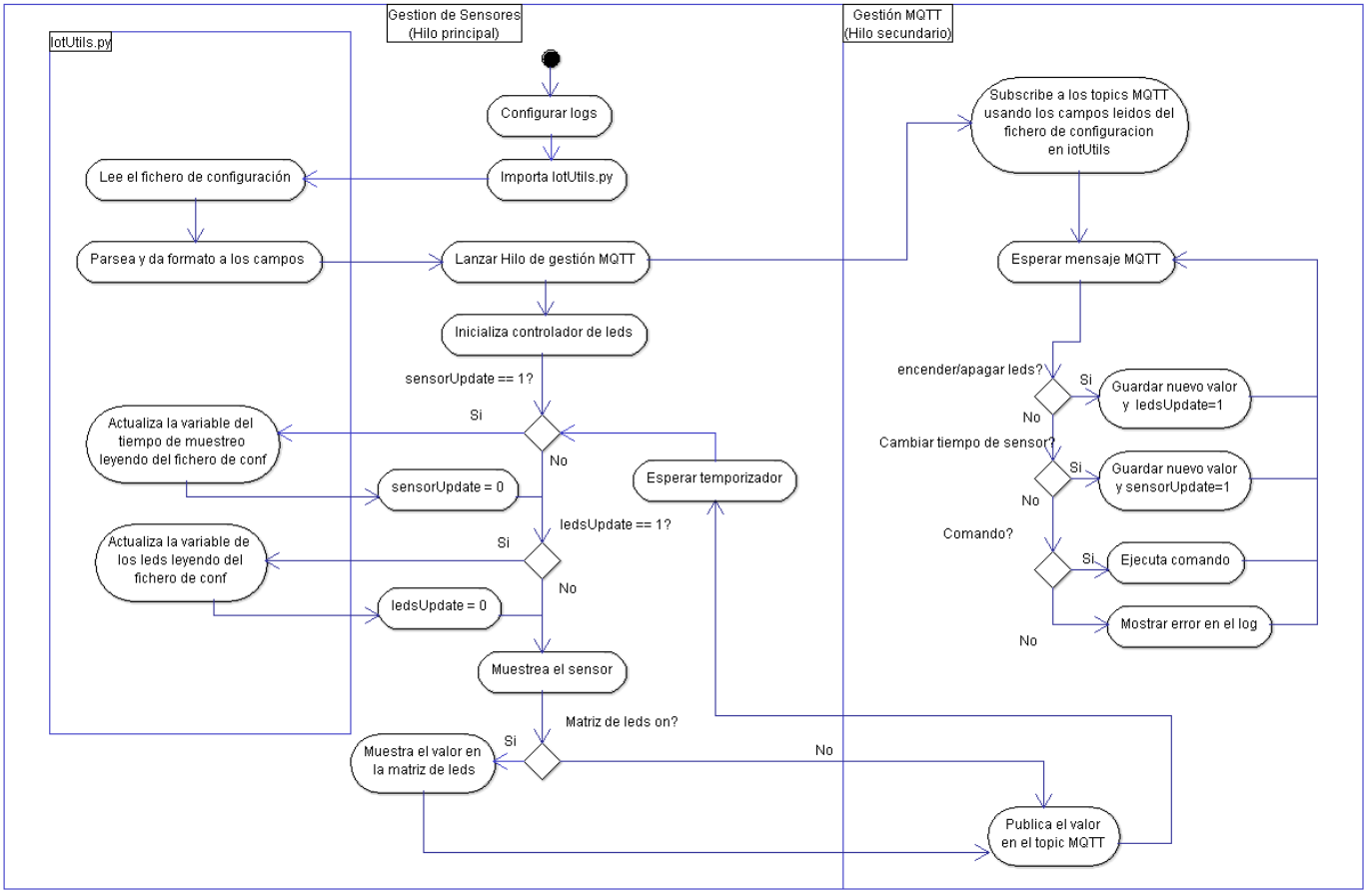


Figura 4.15 – Diagrama de Secuencia del Agente

Para facilitar la explicación, podemos dividir el agente en Python en cuatro bloques principales:

- Módulo principal: Tras realizar la inicialización del sistema de log, pasa a un bucle infinito donde se queda leyendo el sensor y mostrando los valores en la matriz si esta estuviera configurada como encendida, a la vez que actualiza el valor tiempo de muestreo si se ha recibido una actualización a través de MQTT y el estado de los leds si se hubiera mandado por MQTT la orden de cambiar su estado. Esto se logra gracias a las variables globales *ledsUpdate* y *sensorUpdate* que valen 1 cuando se ha recibido una actualización de alguno. Veamos esas subtareas en el código:

- Gestionar el sistema de log:

```
LOG_FILENAME = "agent.log"
logging_enabled = True
LOG_LEVEL = logging.INFO  # Could be e.g. "DEBUG" or "WARNING"
ledsUpdate = 0
sensorUpdate = 0
#
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~
#       A class we can use to capture stdout and sterr in the log
#
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~
class IOTLogger(object):
    def __init__(self, logger, level):
        """Needs a logger and a logger level."""
        self.logger = logger
        self.level = level

    def write(self, message):
        if message.rstrip() != "":  # Only log if there is a message (not
just a new line)
            self.logger.log(self.level, message.rstrip())
```

Código 4.22 – Clase IOTLogger – agent.py

```
def configureLogger(loggerName):
    logger = logging.getLogger(loggerName)
    logger.setLevel(LOG_LEVEL)  # Set the log level to LOG_LEVEL
    handler   =   logging.handlers.TimedRotatingFileHandler(LOG_FILENAME,
when="midnight",
                                                            backupCount=3)  #
Handler that writes to a file,
    # ~~~make new file at midnight and keep 3 backups
    formatter = logging.Formatter(
        '%(asctime)s  %(levelname)-8s  %(message)s')   #  Format  each  log
message like this
    handler.setFormatter(formatter)  # Attach the formatter to the handler
    logger.addHandler(handler)  # Attach the handler to the logger

    if (logging_enabled):
        sys.stdout = IOTLogger(logger,
                               logging.INFO)     #  Replace  stdout  with
logging to file at INFO level
        sys.stderr = IOTLogger(logger,
                               logging.ERROR)    #  Replace  stderr  with
logging to file at ERROR level
```

Código 4.23 – Función configureLogger – agent.py

- Lanzar el hilo de gestión de MQTT (código 4.23), con esto se logra que de forma asíncrona cree otro hilo que se quede gestionando el topic MQTT para en caso de que venga una orden desde el servidor, actúe en consecuencia modificando los campos que sean necesarios (*leds-state* o *push-interval*) en el fichero de configuración, que serán leídos por el bucle principal si su variable de actualización ha sido modificada. Se llama a esta función, una vez antes del bucle infinito.

```
#
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
#       This is a Thread object for listening for MQTT Messages
#
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
class ListenMQTTThread(object):
    def __init__(self):
        thread = threading.Thread(target=self.run, args=())
        thread.daemon = True  # Daemonize thread
        thread.start()  # Start the execution

    def run(self):
        global sensorUpdate;
        global ledsUpdate;
        mqttHandler.main()
```

Código 4.24 – Lanzamiento del hilo de gestión de MQTT – agent.py

- Bucle principal infinito (código 4.25), se encarga de actualizar las variables de estado del tiempo de muestreo y del estado de la matriz de leds en caso de que se hubiera modificado, leyendo los nuevos valores del fichero de configuración, lee el sensor, publica el valor en el topic MQTT y en caso de que estén activos los leds, lo muestra en la matriz. Finalmente espera el tiempo que pongamos de muestreo, antes de reiniciar el bucle.

```
def main():
    configureLogger("agent")
    ListenMQTTThread()
    device = led.matrix()
```

Instancia la matriz de leds, usando la librería max7219.

```
while True:
        try:
            if sensorUpdate == 1:
              iotUtils.PUSH_INTERVAL = iotUtils.getPushValue()
              PUSH_INTERVAL = iotUtils.PUSH_INTERVAL
              sensorUpdate = 0
            if ledsUpdate == 1:
              iotUtils.LEDS_STATE = iotUtils.getLedsValue().upper()
              LEDS_STATE = iotUtils.LEDS_STATE
              ledsUpdate=0
            currentTime = calendar.timegm(time.gmtime())
            sensorValue = iotUtils.getSensorValue()
            if( LEDS_STATE == "ON" ):
                device.show_message(str(sensorValue))
            PUSH_DATA_TO_STREAM_1                                          =
iotUtils.SENSOR_STATS_SENSOR1.format(currentTime, sensorValue)
            mqttHandler.sendSensorValue(PUSH_DATA_TO_STREAM_1)
            print   '~~~~~~~~~~~~~~~~~~~~~~~~   Publishing   Device-Data
~~~~~~~~~~~~~~~~~~~~~~~~'
            print ('PUBLISHED DATA STREAM 1: ' + PUSH_DATA_TO_STREAM_1)
            print  '~~~~~~~~~~~~~~~~~~~~~~~~  End  of  Publishing  Data
~~~~~~~~~~~~~~~~~~~~~~~~'
            time.sleep(PUSH_INTERVAL)
```

Código 4.25 – Bucle principal – agent.py

- En caso de que se interrumpa la ejecución, llama al script de servicio para pararlo.

```
        except (KeyboardInterrupt, Exception) as e:
            print   "agentStats:   Exception   in   AgentThread   (either
```

```
KeyboardInterrupt or Other)"
            print ("agentStats: " + str(e))
            os.system("sudo /etc/init.d/agentd.sh stop")
            print
'~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
            sys.exit(0)
            pass
```

- Fichero **iotUtils.py** (código 4.26)*:* Encargado de toda la lógica de sensores. Lee y almacena las variables del fichero de configuración:

```
import ConfigParser
import os
import random
from w1thermsensor import W1ThermSensor

#
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~
#       Device specific info when pushing data to server
#      Read from a file "deviceConfig.properties" in the same folder level
#
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~
configParser = ConfigParser.RawConfigParser()
configFilePath          =          os.path.join(os.path.dirname(__file__),
'./deviceConfig.properties')
configParser.read(configFilePath)

DEVICE_OWNER = configParser.get('Device-Configurations', 'owner')
DEVICE_ID = configParser.get('Device-Configurations', 'deviceId')
DEVICE_NAME = configParser.get('Device-Configurations', 'device-name')
DEVICE_TYPE = configParser.get('Device-Configurations', 'device-type')
SERVER_NAME = configParser.get('Device-Configurations', 'server-name')
MQTT_EP = configParser.get('Device-Configurations', 'mqtt-ep')
AUTH_TOKEN = configParser.get('Device-Configurations', 'auth-token')
PUSH_INTERVAL   =   float(configParser.get('Device-Configurations','push-
interval'))
LEDS_STATE = configParser.get('Device-Configurations','leds-state')
CONTROLLER_CONTEXT      =      configParser.get('Device-Configurations',
'controller-context')
DEVICE_INFO = '{"owner":"' + DEVICE_OWNER + '","deviceId":"' + DEVICE_ID
+ '",'
HTTPS_EP = configParser.get('Device-Configurations', 'https-ep')
DEVICE_DATA = '"sensorValue":"{sensorValue}"'
SENSOR_STATS_SENSOR1 = '{{"event":{{"metaData":{{"owner":"' + DEVICE_OWNER
+ '","deviceType":"' + DEVICE_TYPE \
                       +     '","deviceId":"'     +     DEVICE_ID     +
'","time":{}}},"payloadData":{{"temperature":{:.2f}}}}}}}'
```

Código 4.26 – Fichero IotUtils.py

Por otro lado, también definimos, en el código 4.27, funciones para obtener o cambiar el valor de variables como por ejemplo el estado de los leds, el tiempo de muestreo del sensor o leer el sensor, hay que tener presente que se usa la librería w1thermsensor para leer el sensor (ver Anexo VII).

```
def getPushValue():
    configParserAux = ConfigParser.RawConfigParser()
```

```
    configFilePathAux      =      os.path.join(os.path.dirname(__file__),
'./deviceConfig.properties')
    configParserAux.read(configFilePathAux)
    PUSH_INTERVAL = float(configParser.get('Device-Configurations','push-
interval'))
    return PUSH_INTERVAL
def setPushValue ( value ):
    cfgfile        =        open(os.path.join(os.path.dirname(__file__),
'./deviceConfig.properties'),'w')
    configParser.set('Device-Configurations','push-interval',value)
    configParser.write(cfgfile)
    cfgfile        =        open(os.path.join(os.path.dirname(__file__),
'../deviceConfig.properties'),'w')
    configParser.set('Device-Configurations','push-interval',value)
    configParser.write(cfgfile)
    return True
def getLedsValue():
    configParserAux = ConfigParser.RawConfigParser()
    configFilePathAux      =      os.path.join(os.path.dirname(__file__),
'./deviceConfig.properties')
    configParserAux.read(configFilePathAux)
    LEDS_STATE = configParser.get('Device-Configurations','leds-state')
    return LEDS_STATE
def setLedsValue ( value ):
    cfgfile        =        open(os.path.join(os.path.dirname(__file__),
'./deviceConfig.properties'),'w')
    configParser.set('Device-Configurations','leds-state',value)
    configParser.write(cfgfile)
    cfgfile        =        open(os.path.join(os.path.dirname(__file__),
'../deviceConfig.properties'),'w')
    configParser.set('Device-Configurations','leds-state',value)
    configParser.write(cfgfile)
    return True


def getSensorValue ():
    sensor = W1ThermSensor()
    temperature = sensor.get_temperature()
    return temperature
```

Código 4.27 – Get y Set de valores de sensor y leds – iotUtils.py

- **mqttHandler.py**: Gestión de MQTT por parte del agente, como ya hemos visto, este módulo, se ejecuta en un hilo separado, con una ejecución independiente del bucle principal, la interacción entre los mismos se realiza a través del fichero de configuración, que es modificado por uno y leído por otro (bucle principal). Cuando se ejecuta, se llama a la función *main* que veremos a en los siguientes códigos, esta función, se encarga primero de configurar las variables de MQTT (véase el código 4.28), como el topic de subscripción para recibir peticiones del servidor, o al que mandar los datos del sensor de temperatura.

```
    MQTT_ENDPOINT = iotUtils.MQTT_EP.split(":")
    MQTT_IP = MQTT_ENDPOINT[1].replace('//', '')
    MQTT_PORT = int(MQTT_ENDPOINT[2])

global TOPIC_TO_SUBSCRIBE
    TOPIC_TO_SUBSCRIBE = TANENT_DOMAIN + "/" + DEV_TYPE + "/command"
    global TOPIC_TO_PUBLISH_STREAM1
    TOPIC_TO_PUBLISH_STREAM1 = TANENT_DOMAIN + "/" + DEV_TYPE + "/" +
DEV_ID + "/temperature"
```

Código 4.28 – Configuración de MQTT – mqttHandler.py

Tras esto, en el código 4.29 se instancia el cliente MQTT de la librería *paho.mqtt* definiendo las funciones *callback* para cuando se conecta con el topic y cuando recibe un mensaje desde el topic subscrito.

```
    global mqttClient
    mqttClient.username_pw_set(iotUtils.AUTH_TOKEN, password="")
    mqttClient.on_connect = on_connect
    mqttClient.on_message = on_message
```

Código 4.29 – Instanciación y definición de callbacks de paho.mqtt – mqttHandler.py

Acto seguido se conecta con la función on_connect, tal y como vemos en el código 4.30.

```
    mqttClient.connect(MQTT_IP, MQTT_PORT,
```

```
def on_connect(mqttClient, userdata, flags, rc):
    print("MQTT_LISTENER: Connected with result code " + str(rc))
    # Subscribing in on_connect() means that if we lose the connection and
    # reconnect then subscriptions will be renewed.
    print ("MQTT_LISTENER: Subscribing with topic " + TOPIC_TO_SUBSCRIBE)
    mqttClient.subscribe(TOPIC_TO_SUBSCRIBE)
```

Código 4.30 - Subscripción al topic MQTT – mqttHandler.py

Finalmente se ejecuta un bucle infinito del cliente MQTT.

```
    mqttClient.loop_forever()
```

Cuando se recibe un mensaje, la función encargada de gestionarlo, es la definida como callback *on_message*, esta función lee el mensaje recibido, separando el método recibido de los parámetros.

```
def on_message(mqttClient, userdata, msg):
    print("MQTT_LISTENER: " + msg.topic + " " + str(msg.payload))
    #Let's split the message content for further operations
    method,parameter=str(msg.payload).split(":",1);
```

Según el método, ejecutamos una acción u otra, los métodos posibles a recibir por el servidor, son los configurados durante la creación de la API:

- *ledsRequest:* petición de encender o apagar la matriz de leds, activamos la flag para indicar al bucle principal.

```
    if method == "ledsRequest" :
        print("Received a request to turn ON/OFF the led matrix")
        iotUtils.LEDS_STATE = iotUtils.setLedsValue(parameter)
        ledsUpdate = 1
```

- *TimeRequest:* petición de cambiar el tiempo de muestreo, activamos la flag para indicar al bucle principal.

```
    elif  method == 'timeRequest':
        print("Received a request to change the sensor push time")
        iotUtils.PUSH_INTERVAL = iotUtils.setPushValue(parameter)
        sensorUpdate = 1
```

- *Reboot*: Se ha recibido una petición de reiniciar el dispositivo.

```
    elif  method == "reboot" :
        print("Received a request to reboot the device")
        os.system("sleep "+parameter)
        os.system("reboot")
```

- Shutdown: Se ha recibido una petición para apagar el dispositivo.

```
    elif  method == "shutdown" :
        print("Received a request to shutdown the device")
        os.system("sleep "+parameter)
        os.system("shutdown now")
```

- Bash: Se ha recibido un comando en Bash, esta funcionalidad es experimental y debe de tenerse cuidado al usarla, pues se tiene acceso a todo el sistema de fichero del dispositivo.

```
    elif  method == "bash":
        print("Received a bash command")
        if "_" in parameter:
            command, arguments = str( parameter ).split("_", 1)
            if "_" in arguments:
                parsedArguments = arguments.replace("_", " ")
            else:
                parsedArguments = arguments

            os.system( command + " " + parsedArguments )
        else:
            os.system( command )
```

Por último, las funciones usadas por *agent.py* para el envío de datos al servidor, hacen uso de la librería paho.mqtt y de la clase instanciada y es como sigue:

```
def on_publish(mqttClient, stream1PlayLoad):
    mqttClient.publish(TOPIC_TO_PUBLISH_STREAM1, stream1PlayLoad)
def sendSensorValue(stream1PlayLoad):
    global mqttClient
    on_publish(mqttClient, stream1PlayLoad)
```

4.3.4 Interfaz de usuario, UI

La interfaz de usuario del servidor WSO2 de IoT está construida usando el framework de interfaz de usuario unificado (UUF [19]), que permite de forma fácil y modular el desarrollo de interfaz de usuario.

UUF está desarrollado sobre Jaggery, desde el punto de vista de Carbon es una aplicación más de *Jaggery*. Para desarrollar sobre él, es necesario hacer uso de *handlebars*. La estructura de directorio es la vista con anterioridad, dentro de la carpeta app/units, que contiene las unidades de interfaz de usuario.

Cada unidad de interfaz de usuario debe contener obligatoriamente una vista *handlebars*, un fichero de configuración *json* y de manera opcional un controlador en *javascript*.

Las unidades de interfaz de usuario que tenemos son:

- Type-view: Es la unidad que se muestra cuando se accede a la lista de tipos de dispositivos, muestra una descripción del dispositivo, permite descargar el fichero zip con el código del agente (lo que también lo registra mediante la llamada a la API a *downloadSketch*).
    - Device-view.hbs: Definimos en *handlebars* la estructura de y el código de lo que será la interfaz de usuario y su representación.
    - Type-view.json: configuración, indicando la versión (1.0.0) vamos a obviarlo para el resto de vistas, ya que el contenido es igual.
    - private/config.json: Guarda la configuración de las operaciones definidas en la API, podemos definir cosas como el tipo de icono, el tipo de objeto, el valor que tienen y el nombre, veamos el código 4.31.

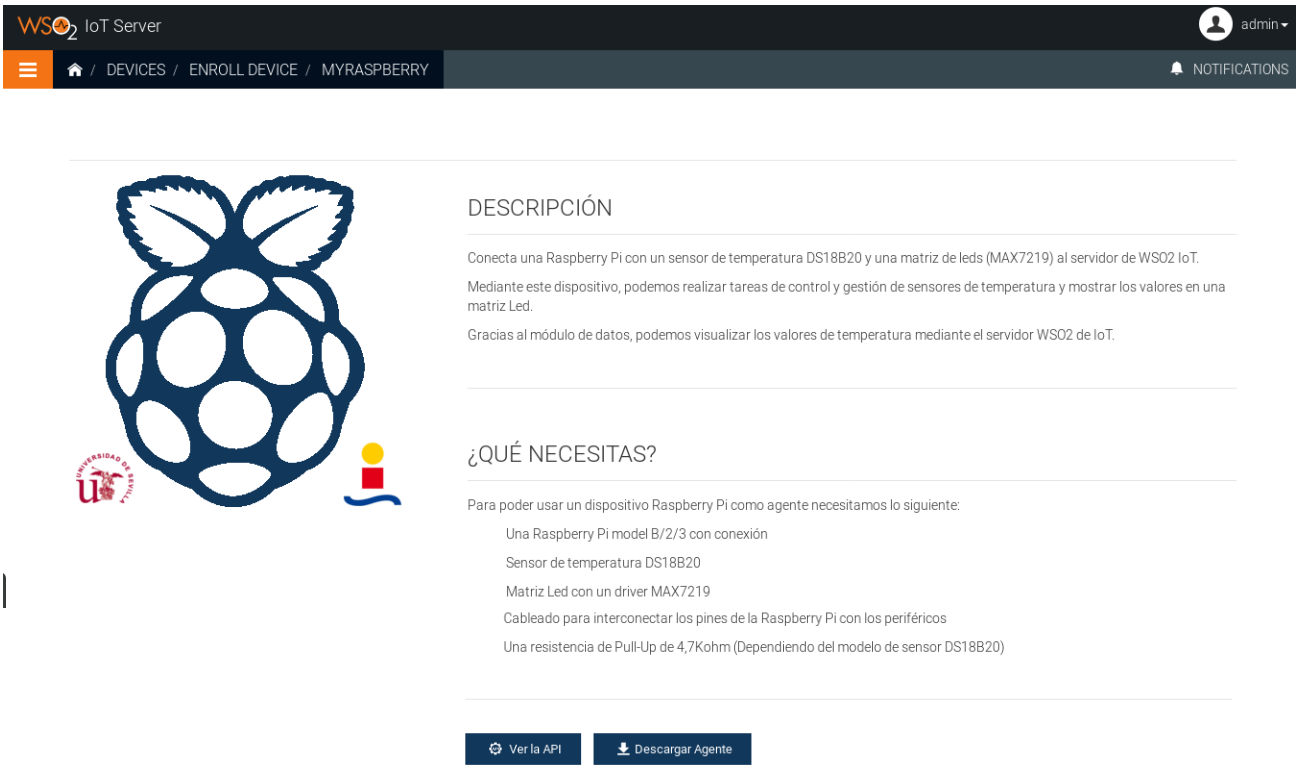
```
{
  "deviceType": {
    "label": "myRaspberry",
    "category": "iot",
    "downloadAgentUri": "myRaspberry/device/download",
```

```
    "scopes": [
      "perm:myRaspberry:enroll"
    ],
       "features":{
         "sensor": {
             "icon": "fw-settings"
          },
          "leds": {
         "icon": "fw-settings"
          },
         "send-command": {
         "icon": "fw-message",
         "formParams": [
         {
          "type": "radio",
          "id": "cmd_reboot",
          "label": "Reiniciar la Raspberry [Tiempo en Segundos]",
          "name": "orden",
          "value":"reboot"
         },
         {
           "type": "radio",
           "id": "cmd_custom",
           "label":      "Permite      mandarle      comandos      bash
[Comando_Parametro1_Parametro2_...]",
           "name": "orden",
           "value":"bash"
         },
         {
           "type": "radio",
           "id": "cmd_shutdown",
           "label": "Apagar la Raspberry [Tiempo en Segundos]",
           "name": "orden",
           "value":"shutdown"
         }
      ]
    }
  }
  }
}
```

Código 4.31 – Fichero config.json

- images: Directorio con todas las imágenes que se muestran en esta vista.
- js/download.js: Fichero *Javascript* implementando la descarga del código de agente.

Veamos el resultado de la implementación


WSO2 IoT Server
admin
DEVICES / ENROLL DEVICE / MYRASPBERRY
NOTIFICATIONS
DESCRIPCIÓN
Conecta una Raspberry Pi con un sensor de temperatura DS18B20 y una matriz de leds (MAX7219) al servidor de WSO2 IoT.
Mediante este dispositivo, podemos realizar tareas de control y gestión de sensores de temperatura y mostrar los valores en una matriz Led.
Gracias al módulo de datos, podemos visualizar los valores de temperatura mediante el servidor WSO2 de IoT.
¿QUÉ NECESITAS?
Para poder usar un dispositivo Raspberry Pi como agente necesitamos lo siguiente:
Una Raspberry Pi model B/2/3 con conexión
Sensor de temperatura DS18B20
Matriz Led con un driver MAX7219
Cableado para interconectar los pines de la Raspberry Pi con los periféricos
Una resistencia de Pull-Up de 4,7Kohm (Dependiendo del modelo de sensor DS18B20)
Ver la API
Descargar Agente
CONFIGURACIÓN
01 Conexión del sensor: Dependiendo del modelo del sensor necesitará la resistencia o no. El diagrama para el cableado del modelo acuatico es igual al modelo normal.
02 Conexión de la matriz de leds:
El pin 1 (VCC) de la matriz va conectado al pin 2 (5v).
El pin 2 (GND) de la matriz va conectado al pin 6 (GND).
El pin 3 (DIN) de la matriz va conectado al pin 19 (MOSI).
El pin 4 (CS) de la matriz va conectado al pin 24 (SPI).
El pin 5 (CLK) de la matriz va conectado al pin 23 (CLK).
03 Necesitarás una conexión a internet en la Raspberry Pi para poder pasarle por scp el fichero del agente y conectarla con el servidor.

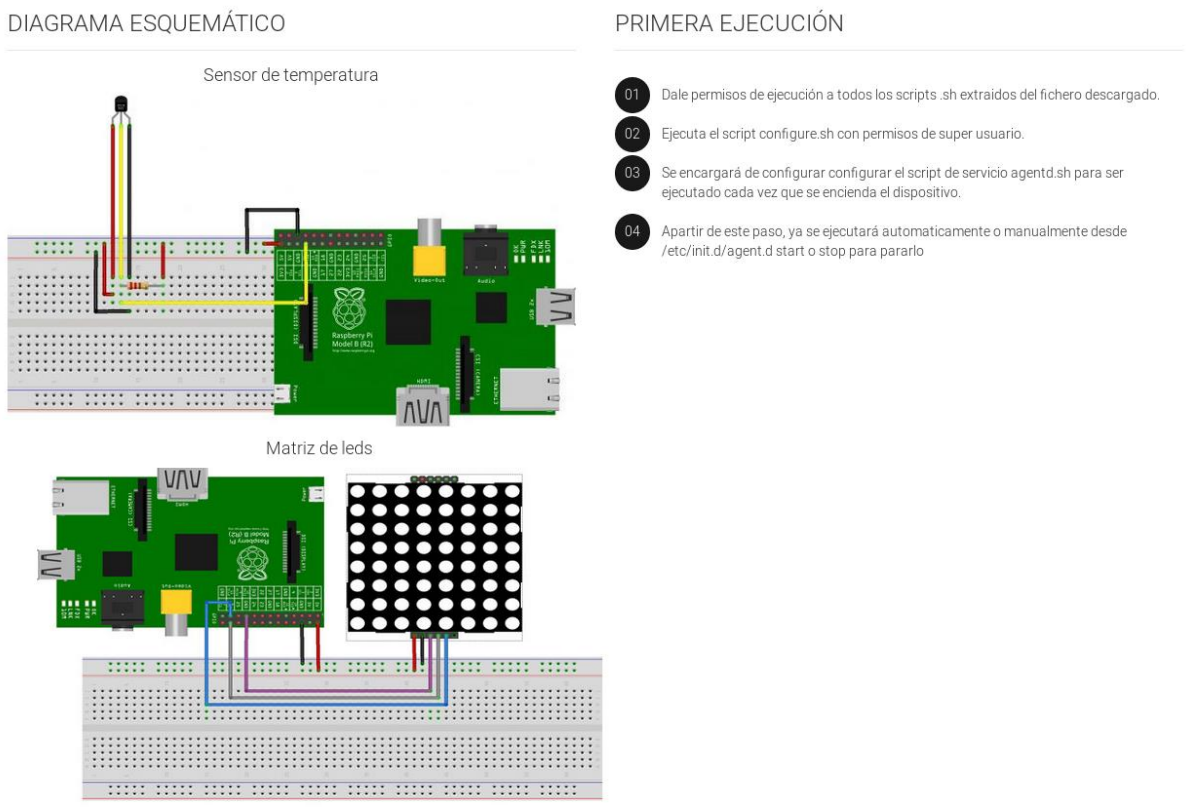


Figura 4.16 – Vista principal type-view

- Device-view: Esta es la vista que se muestra cuando accedemos a un dispositivo registrado, para ello accedemos mediante, en ella podemos realizar las distintas operaciones configuradas en pasos anteriores, tenemos el histórico de operaciones realizadas y la vista en tiempo real de los datos de sensores.
    - device-view.hbs: Implementación en handlebars de la vista.
    - device-view.js: Es el controlador de la vista, en él tenemos implementado la lectura del fichero de histórico de operaciones, tal y como vemos en el código 4.32.

```
function onRequest(context) {
  var log = new Log("device-view.js");
  var deviceType = context.uriParams.deviceType;
var deviceId = request.getParameter("id");
var autoCompleteParams = [
{"name" : "deviceId", "value" : deviceId} ];
var fileaux = new File("../sampleData.json");
if ( fileaux.isExists())
{
  var data = require("../sampleData.json");
  data = data[deviceId];
}
if (deviceType != null && deviceType != undefined && deviceId != null &&
deviceId != undefined) {
var deviceModule = require("/app/modules/business-controllers/device.js")
["deviceModule"];
var device = deviceModule.viewDevice(deviceType, deviceId);
if (device && device.status != "error") {
return     {"device":     device.content,     "autoCompleteParams"     :
```

```
autoCompleteParams, "encodedFeaturePayloads": "", "data": data};
} else {
response.sendError(404, "Device Id " + deviceId + " of type " + deviceType
+ " cannot be found!");
exit();
  }
}
}
```

Código 4.32 – UI del histórico de operaciones

Para acceder a la vista, debemos hacer click en el botón *view* de la sección *devices* del portal principal, tras esto tendremos la vista de todos los dispositivos registrados.

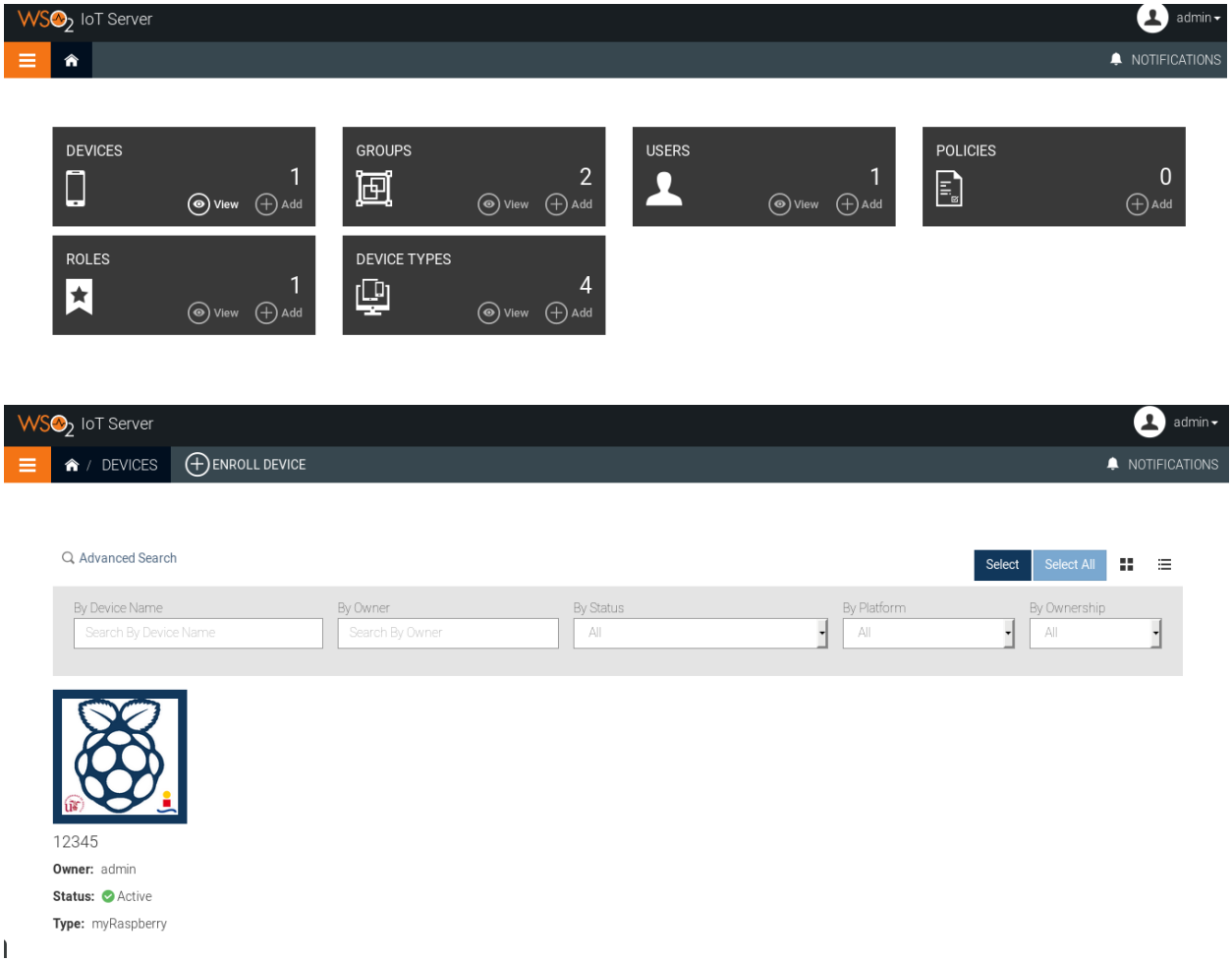


Figura 4.17 – Portal principal y sección devices

Veamos a continuación el resultado de la implementación de la vista:

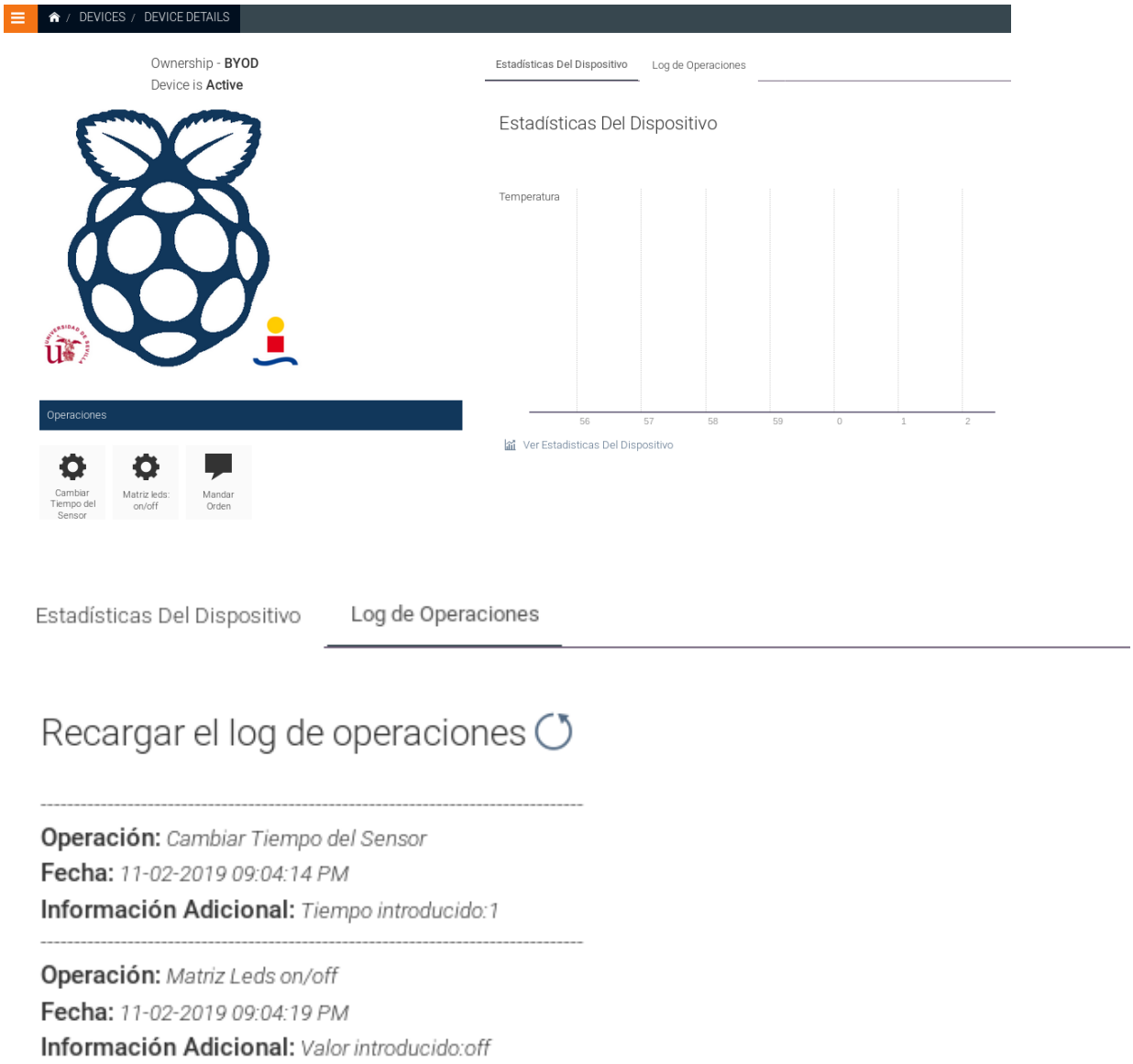


Figura 4.18 – Secciones del deviceType-view

- Analytics-view: Se llama a esta vista cuando se le da a Ver Estadísticas del Dispositivo, esta vista muestra un histórico total almacenado de los datos de sensores.
    - Analytics-view.hbs (código 4.33): Es la implementación de *handlebars* y llama a js/devicetype-graph.js que es la que se encarga de crear y mostrar la gráfica.

```
<span     id="devicetype-details"     data-devices="{{devices}}"     data-
devicename="{{device.name}}"
      data-deviceid="{{device.deviceIdentifier}}"
      data-appcontext="{{@app.context}}"></span>
<div id="chartDivSensorType1">
    <div class="chartWrapper" id="chartWrapper">
        <span id="sensorType1Title">Temperatura</span>
        <div id="sensorType1yAxis" class="custom_y_axis"></div>
        <div class="legend_container">
            <div id="smoother" title="Smoothing"></div>
            <div id="sensorType1Legend"></div>
        </div>
```

```
        <div id="chartSensorType1" class="custom_rickshaw_graph"
             data-backend-api-url= {{backendApiUri}}></div>
        <div id="sensorType1xAxis" class="custom_x_axis"></div>
        <div id="sensorType1Slider" class="custom_slider"></div>
    </div>
</div>

{{#zone "bottomJs"}}
    {{js "js/devicetype-graph.js"}}
{{/zone}}
```

Código 4.33 – Analitics-view.hbs

- js/devicetype-graph.js: Se encarga de crear la gráfica y tomar los valores a través del controlador, transformarlos para la visualización, etc, con tal de formar la gráfica y el histórico.
- Analytics-view.js (código 4.34): Como en todos los anteriores, es el controlador de la vista, gracias a él tenemos accesible los datos de sensores almacenados.

```
<span     id="devicetype-details"     data-devices="{{devices}}"     data-
devicename="{{device.name}}"
      data-deviceid="{{device.deviceIdentifier}}"
      data-appcontext="{{@app.context}}"></span>
<div id="chartDivSensorType1">
    <div class="chartWrapper" id="chartWrapper">
        <span id="sensorType1Title">Temperatura</span>
        <div id="sensorType1yAxis" class="custom_y_axis"></div>
        <div class="legend_container">
            <div id="smoother" title="Smoothing"></div>
            <div id="sensorType1Legend"></div>
        </div>
        <div id="chartSensorType1" class="custom_rickshaw_graph"
             data-backend-api-url= {{backendApiUri}}></div>
        <div id="sensorType1xAxis" class="custom_x_axis"></div>
        <div id="sensorType1Slider" class="custom_slider"></div>
    </div>
</div>

{{#zone "bottomJs"}}
    {{js "js/devicetype-graph.js"}}
{{/zone}}
```

Código 4.34 – Analytics-view.js

Al final tenemos el siguiente resultado:

Figura 4.19 – Vista de la gráfica de datos

- Realtime.analytics-view: Es la página encargada de crear y dar formato a la gráfica en tiempo real que se muestra en device-view. Volvemos a encontrarnos con la misma estructura, por un lado, la vista, el controlador y el fichero *Json* de configuración que es igual a los anteriores. También tenemos un directorio *public*, que contiene los ficheros *javascript* implementando la librería Rickshaw [20] (que proporciona un conjunto de herramientas para el diseño de gráficas) para la gráfica en tiempo real.
    - analytics-view.hbs (código 4.35): Se encarga de mostrar la gráfica generada por los ficheros *javascript* contenidos en el directorio *public*.

```
{{unit "cdmf.unit.lib.rickshaw-graph"}}
<div                  id="div-chart-sensorType1"                  data-
websocketurlStream="{{websocketEndpointForStream1}}">
    <div class="chartWrapper" id="chartWrapper">
        <divid="yAxisSensorType1" class="custom_y_axis">Temperatura</div>
        <div class="legend_container">
            <div id="smoother" title="Smoothing"></div>
            <div id="legend"></div>
        </div>
        <div id="chartSensorType1" class="custom_rickshaw_graph"></div>
        <div class="custom_x_axis">Tempo</div>
    </div>
</div>
<a class="padding-left"

href="{{@app.context}}/device/{{device.type}}/analytics?deviceId={{device
.deviceIdentifier}}
   &deviceName={{device.name}}">
    <span class="fw-stack">
        <i class="fw fw-ring fw-stack-2x"></i>
        <i class="fw fw-statistics fw-stack-1x"></i>
    </span> Ver Estadisticas Del Dispositivo
</a>
<!-- /statistics -->
{{#zone "bottomJs"}}
    {{js "js/moment.min.js"}}
    {{js "js/socket.io.min.js"}}
```

```
    {{js "js/device-stats.js"}}
{{/zone}}
```

Código 4.35 – Analytics-view.hbs, realtime.

- analytics-view.js: Es el controlador de la vista anterior, retorna el dispositivo y el socket para la comunicación de los datos del sensor.

```
function onRequest(context) {
var log = new Log("stats.js");
var carbonServer = require("carbon").server;
var device = context.unit.params.device;
var devicemgtProps = require("/app/modules/conf-reader/main.js")["conf"];
var constants = require("/app/modules/constants.js");
var websocketEndpoint = devicemgtProps["wssURL"].replace("https", "wss");
var jwtService = carbonServer.osgiService(
'org.wso2.carbon.identity.jwt.client.extension.service.JWTClientManagerSe
rvice');
var jwtClient = jwtService.getJWTClient();
var                          encodedClientKeys                              =
session.get(constants["ENCODED_TENANT_BASED_WEB_SOCKET_CLIENT_CREDENTIALS
"]);
var token = "";
var user = session.get(constants.USER_SESSION_KEY);
if (!user) {
log.error("User object was not found in the session");
throw constants.ERRORS.USER_NOT_FOUND;
} if (encodedClientKeys) {
var      tokenUtil      =      require("/app/modules/oauth/token-handler-
utils.js")["utils"];
var resp = tokenUtil.decode(encodedClientKeys).split(":");
if (user.domain == "carbon.super") {
  var    tokenPair    =    jwtClient.getAccessToken(resp[0],    resp[1],
context.user.username , "default", {});
  if (tokenPair) {
    token = tokenPair.accessToken;
  }  var websocketEndpointForStream1 = websocketEndpoint +
    "/securedwebsocket/org.wso2.iot.devices.temperature/1.0.0?"         +
"deviceId=" +
    device.deviceIdentifier    +    "&deviceType="    +    device.type    +
"&websocketToken=" +  token;
  } else {
    var    tokenPair    =    jwtClient.getAccessToken(resp[0],    resp[1],
context.user.username + "@" + user.domain, "default", {});
  if (tokenPair) {
    token = tokenPair.accessToken;
  }  var  websocketEndpointForStream1  =  websocketEndpoint  +  "/secured-
websocket/t/" +
  user.domain + "/org.wso2.iot.devices.temperature/1.0.0?" + "deviceId="
+
    device.deviceIdentifier    +    "&deviceType="    +    device.type    +
"&websocketToken=" + token;
} }
return       {"device":       device,       "websocketEndpointForStream1":
websocketEndpointForStream1};}
```

Código 4.36 – Analytics-view.js, realtime.

public/device-stats.js: La implementación de la librería *Rickshaw* mediante el uso de websockets (tecnología que permite abrir socket entre cliente y servidor sobre la tecnología de capa 7).

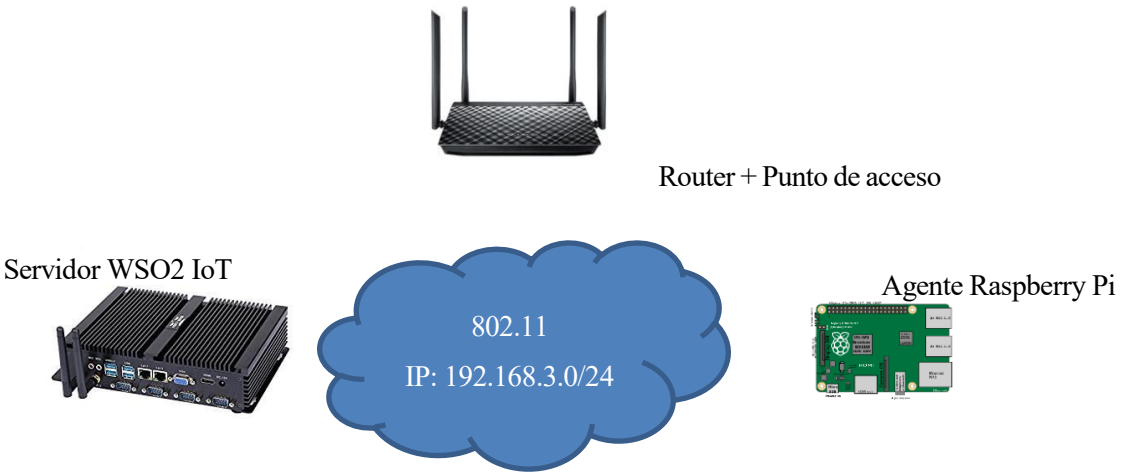
5 PRUEBAS Y VALIDACIÓN

En esta sección, se va a exponer el sistema de pruebas desarrollado con la intención de comprobar el correcto funcionamiento del sistema y todos sus componentes (API, agente, Interfaz de usuario, etc). Dado el carácter software del proyecto, es necesario invertir una gran cantidad de tiempo y esfuerzo con el objetivo de comprobar las funcionalidades del mismo y en caso de que hubiera, la corrección de errores.

Esta sección se dividirá en varios apartados, por un lado, la interfaz de usuario y por el otro, la API y el comportamiento del agente para lo que se ha desarrollado un sistema independiente para realizar las pruebas, ya que el funcionamiento de ambos está muy relacionado. Seguiremos el siguiente esquema para detallar las pruebas realizadas:

Módulo	API / Agente / Interfaz de Usuario
Objetivo	Lo que se quiere probar
Descripción	Resumen de lo que realiza la prueba
Resultado	

Para las pruebas se ha generado un entorno aislado, con el siguiente esquema de red:



5.1 Verificación de la Interfaz de Usuario

Dado el alcance limitado del proyecto, se ha decidido no desarrollar un entorno de pruebas específico para la interfaz de usuario, sin embargo, se ha hecho la verificación de forma manual y visual, dando por supuesto que el servidor WSO2 IoT es un producto lo suficientemente maduro como para no tener que probar su funcionamiento, teniéndonos que centrar en los componentes carbon desarrollados.

5.1.1 Verificación de la vista type-view

Módulo	Interfaz de Usuario
Objetivo	Botón “ver la API”
Descripción	Se comprueba el correcto funcionamiento del botón, viendo que redirecciona a la página que contiene información relativa a la API del dispositivo.
Resultado	Exitoso

Módulo	Interfaz de Usuario
Objetivo	Botón “Descargar Agente”
Descripción	Se comprueba el funcionamiento añadiendo una palabra de más de 3 caracteres y haciendo clic para descargar.
Resultado	Exitoso

Módulo	Interfaz de Usuario
Objetivo	Botón “Descargar Agente”
Descripción	Se comprueba el funcionamiento añadiendo una palabra de menos de 3 caracteres y haciendo clic para descargar, lo que debe ser erróneo.
Resultado	Exitoso

Módulo	Interfaz de Usuario
Objetivo	Botón “Descargar Agente”
Descripción	Se comprueba el funcionamiento añadiendo una palabra de más de 3 caracteres y usando la tecla intro, en vez de hacer clic.
Resultado	Exitoso

5.1.2 Verificación de la vista device-view

Módulo	Interfaz de Usuario
Objetivo	Gráfica en tiempo real

Descripción	Se ve los valores que la gráfica va mostrando después de un periodo de 30 segundos. Se compara con los valores enviados, almacenados en los logs del agente.
Resultado	Exitoso para Firefox, fallido para Chromium, no se actualiza.

Módulo	Interfaz de Usuario
Objetivo	Operación “Cambiar tiempo del sensor”
Descripción	Se da valores correctos de tiempo, al sensor, mayor que 0, positivo y numérico. Se hace clic en “Send to device”
Resultado	Exitoso

Módulo	Interfaz de Usuario
Objetivo	Operación “Cambiar tiempo del sensor”
Descripción	Se da valores incorrectos de tiempo, al sensor, negativos o letras. Se hace clic en “Send to device”
Resultado	Exitoso, no permite esos valores.

Módulo	Interfaz de Usuario
Objetivo	Operación “Cambiar tiempo del sensor”
Descripción	Se da valores correctos de tiempo, al sensor, mayor que 0, positivo y numérico. Se usa la tecla intro para enviar.
Resultado	Fallido: WSO2 IoT Server no reconoce la tecla intro como forma de enviar.

Módulo	Interfaz de Usuario
Objetivo	Operación “Matriz Leds on/off”
Descripción	Se da valores correctos (on y off) y se usa el botón “Send to device”
Resultado	Exitoso

Módulo	Interfaz de Usuario

Objetivo	Operación “Matriz Leds on/off”
Descripción	Se da valores incorrectos (distintos de on y off) y se usa el botón “Send to device”
Resultado	Exitoso, no permite esos valores

Módulo	Interfaz de Usuario
Objetivo	Operación “Matriz Leds on/off”
Descripción	Se da valores correctos (de on y off) y se usa la tecla intro
Resultado	Fallido: WSO2 IoT Server no reconoce la tecla intro como forma de enviar.

Módulo	Interfaz de Usuario
Objetivo	Operación “Mandar orden”
Descripción	Se prueba cada opción disponible
Resultado	Exitoso

Módulo	Interfaz de Usuario
Objetivo	Histórico de operaciones
Descripción	Se comprueban que todas las operaciones anteriormente realizadas aparecen.
Resultado	Exitoso

5.1.3 Verificación de la vista analytics-view

Módulo	Interfaz de Usuario
Objetivo	Histórico de datos
Descripción	Se deja el sistema encendido durante un día entero, almacenando datos y se pasa a comprobar el filtrado de datos y si los muestra correctamente.
Resultado	Exitoso

Conclusión: El servidor WSO2 IoT presenta un bug a la hora de enviar una operación usando la tecla intro, así como a la hora de mostrar la gráfica en tiempo real en ciertos exploradores como Chromium.

5.2 Verificación de la API y del agente

Dada la complejidad para verificar tanto las llamadas a la API como el funcionamiento del agente, se ha decidido crear un proyecto en el lenguaje Perl, que nos aporta una gran flexibilidad a la hora de analizar ficheros de texto, así como buscar patrones, pero el factor decisivo para decidirnos por este lenguaje es la incorporación de módulos nativos para pruebas y verificación, como sería Test::Simple[21] y Test::More[22].

El proyecto de prueba se ha diseñado y estructurado de la siguiente forma:

- Directorio config: Se incluyen ficheros de configuración, siguiendo un estilo INI ya visto anteriormente, en nuestro caso, tenemos el fichero config.ini con los siguientes valores:

```
[WSO2]
IP=
USER=
ADMIN=
[RPI]
NAME=
IP=
deviceId=
```

La sección WSO2 describe las variables relativas al servidor, como su dirección IP, su usuario y su contraseña.

La sección RPI, presenta los valores del agente, hacia el que se dirigirán las peticiones http.

- Directorio cmd: Contiene los módulos principales de las pruebas, así como el script encargado de incluir módulos necesarios y lanzar los test definidos en un fichero de extensión .t:
    - Fichero core.pl: Contiene la lógica básica de las pruebas
        - *generate_token()*: Que negocia con el servidor el token de autenticación para obtener los permisos de llamada a la API del dispositivo.
        - *check_body():* Comprueba que el Json a usar para las peticiones http tiene los campos que le corresponde según el tipo de petición.
        - *launch_test():* Es el encargado de realizar la petición http mediante *curl* y comprobar el resultado de la misma o realizar otro tipo de operaciones, como comprobaciones mediante *Bash* en la configuración del agente.
    - Fichero test.pl: Se llama con el test a realizar como único parámetro y se encarga de cargar los módulos, las definiciones de tests y por último a llamar a funciones del fichero core.pl.
- Directorio definitions: Contiene las definiciones de los tests, estas definiciones se corresponden con los parámetros que va a tener la petición http, por ejemplo, para cambiar el tiempo de muestreo del sensor, es necesario que el verbo http sea POST, el ID del agente y el nuevo valor de tiempo.
- Directorio test: En este directorio se engloban los ficheros de extensión .t que contienen la lógica de pruebas, por ejemplo, para registrar un nuevo agente en el servidor, incluye la petición correspondiente de la API y otras operaciones, como el despliegue del agente en una Raspberry PI.
- Directorio scrips: En este directorio se incluyen scripts en Bash que realizan tareas específicas, como es el caso del script reset_server.sh encargado de redesplegar el servidor para realizar las pruebas en un entorno limpio.

A continuación, veremos los resultados de las pruebas realizadas.

5.2.1 Verificación del registro de agente

En esta sección, se ha intentado comprobar el funcionamiento de la llamada a la API encargada de registrar en el CDMF un nuevo dispositivo.

Módulo	API
Objetivo	URI: /myRaspberry/1.0.0/device/download?deviceName=NOMBRE&sketchType=myRaspberry
Descripción	Se lanza la petición para registrar un nuevo dispositivo que a su vez generará un fichero .zip con el código del agente para el dispositivo
Resultado	Exitoso

Módulo	API
Objetivo	Fichero .zip con Código del agente
Descripción	Se intenta descomprimir el fichero del agente en la Raspberry PI con el objetivo de comprobar que es realmente un fichero .zip
Resultado	Exitoso

5.2.2 Verificación de la operación para cambiar tiempo del sensor

Módulo	API
Objetivo	URI: /myRaspberry/1.0.0/device/ID/sensor?tiempo=4
Descripción	Se lanza una petición a la API para cambiar el tiempo de muestreo del sensor a 4 segundos
Resultado	Exitoso

Módulo	Agente
Objetivo	Tiempo de muestreo del sensor sea 4 en el agente
Descripción	Se conecta al agente y lee el campo del fichero de configuración que contiene el tiempo de muestreo del sensor, debe ser 4 segundos.
Resultado	Exitoso

Módulo	API
Objetivo	URI: /myRaspberry/1.0.0/device/ID/sensor?tiempo=10
Descripción	Se lanza una petición a la API para cambiar el tiempo de muestreo del sensor a 10 segundos
Resultado	Exitoso

Módulo	Agente
Objetivo	Tiempo de muestreo del sensor sea 10 en el agente
Descripción	Se conecta al agente y lee el campo del fichero de configuración que contiene el tiempo de muestreo del sensor, debe ser 10 segundos.
Resultado	Exitoso

5.2.3 Verificación de la operación para cambiar el estado de la matriz de leds

Módulo	API
Objetivo	URI: /myRaspberry/1.0.0/device/ID/leds?estado=on
Descripción	Se lanza una petición a la API para cambiar el estado de la matriz de leds a on.
Resultado	Exitoso

Módulo	Agente
Objetivo	Estado de la matriz sea on en el agente
Descripción	Se conecta al agente y lee el campo del fichero de configuración que contiene el estado de la matriz de leds, debe ser on.
Resultado	Exitoso

Módulo	API
Objetivo	URI: /myRaspberry/1.0.0/device/ID/leds?estado=off
Descripción	Se lanza una petición a la API para cambiar el estado de la matriz de leds a off.
Resultado	Exitoso

Módulo	Agente
Objetivo	Estado de la matriz sea off en el agente
Descripción	Se conecta al agente y lee el campo del fichero de configuración que contiene el estado de la matriz de leds, debe ser off.
Resultado	Exitoso

6 CONCLUSIONES FINALES

En esta sección, se pretende desarrollar las conclusiones obtenidas tras la realización del proyecto junto con puntos de mejora y futuros desarrollos.

6.1 Conclusiones del Proyecto

En primer lugar, se ha logrado desarrollar un complemento para el servidor WSO2 IoT que extienda las capacidades del mismo mediante un nuevo tipo de dispositivo según nuestras necesidades. Para ello ha sido necesaria la inversión de tiempo y esfuerzo en comprender múltiples tecnologías, protocolos y lenguajes que han intervenido en el ciclo de desarrollo.

En segundo lugar, se ha logrado crear un módulo de API en Java, basándose en un framework para lograrlo. Esto ha sido a su vez costoso por la carga conceptual y la escasa documentación. Hemos logrado desarrollar llamadas a la API para un conjunto de funciones que hemos considerado necesarias, otorgándonos la capacidad de en un futuro desarrollar más llamadas si así fuera requerido.

En tercer lugar, se ha desarrollado un agente multihilos, donde un hilo atiende de forma autónoma las peticiones del servidor y el otro se encarga de la gestión de sensores y periféricos. También se ha hecho de forma que pueda desplegarse en una o más Raspberry Pi, este agente ha sido desarrollado en Python, haciendo uso de tecnologías destinadas a IoT para comunicarse con el servidor.

Como en todo proyecto software, se ha diseñado un entorno de pruebas, para garantizar el correcto funcionamiento de todo el sistema.

Todo esto se ha desarrollado teniendo en cuenta la escalabilidad y modularidad, pudiendo desplegarse en un gran abanico de sistemas y que sea reproducible en otros casos de uso.

Como resultado, se ha obtenido un sistema para la gestión de temperatura, capaz de representarla en tiempo real y con una interfaz de usuario en el servidor WSO2 IoT que permite de forma fácil e intuitiva tener un control en tiempo real y un histórico de los sensores de todos los agentes desplegados.

A pesar de lo logrado, ha resultado imposible desplegar el servidor WSO2 IoT en dispositivos Raspberry Pi por su falta de capacidades. Cabe también destacar que hay puntos de mejora en el proyecto, como es una mayor extensión de las pruebas de verificación, el número escaso de sensores, o la no generación de una documentación relativa a la API.

Se ha desarrollado las memorias de forma suficientemente ilustrativa para que futuros desarrollos puedan basarse en el trabajo aquí expuesto, buscando ser de utilidad para otros alumnos o proyectos, que lo requieran.

A nivel personal, el proyecto ha logrado que desarrolle una gran capacidad de absorción de nuevos conceptos y tecnologías, así como incentivado en mí el uso de las documentaciones existentes y toda la información disponible en la comunidad, tanto en foros de preguntas, como en el portal GitHub.

6.2 Futuros desarrollos del proyecto

A continuación, trataremos ideas y aspectos del desarrollo del proyecto, que, ya sea por falta de tiempo, alcance del proyecto u otras complicaciones no se han podido llevar a cabo.

En un primer momento se intentó englobar el proyecto en el ámbito de la salud, a pesar de estar solo centrado en la temperatura, este proyecto es fácilmente integrable con más sensores compatibles, pudiendo llegar a proporcionar gran cantidad de información al servidor WSO2 IoT, como ritmo cardíaco, temperatura corporal, concentración de oxígeno en sangre, etc. Pudiendo así llevarlo al ámbito de la monitorización de la salud.

Por otra parte, y relacionado con lo anterior, no se ha entrado a valorar otras capacidades que nos proporciona WSO2, como es el tratamiento de datos mediante los módulos de análisis. Pero sí se ha tenido en cuenta para futuros desarrollos como es la posibilidad de crear alertas y comportamientos en función de los datos obtenidos, pudiendo en nuestro caso alertarnos ante temperaturas muy altas o muy bajas y en el caso sanitario, ante valores anormales de los parámetros de salud.

Se ha comentado la capacidad que proporciona WSO2 de desarrollar un dashboard, esto personalmente considero que es de gran utilidad para este proyecto pues otorga un sumario de datos y facilidad de visualización de valores, por lo que es una de las cosas más importante a desarrollar en un futuro próximo.

Finalmente, queda por desarrollar un entorno de pruebas destinado a la verificación de las funcionalidades de la interfaz de usuario, existen frameworks destinados al desarrollo de este tipo de pruebas, que nos sería de gran utilidad.

ANEXOS

Anexo I: Script *iot-server.sh*

```
#!/bin/sh
# ----------------------------------------------------------------------
---------
#  Copyright 2005-2012 WSO2, Inc. http://www.wso2.org
#
#  Licensed under the Apache License, Version 2.0 (the "License");
#  you may not use this file except in compliance with the License.
#  You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
#  Unless required by applicable law or agreed to in writing, software
#  distributed under the License is distributed on an "AS IS" BASIS,
#   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
#  See the License for the specific language governing permissions and
#  limitations under the License.

# ----------------------------------------------------------------------
---------
# Main Script for the WSO2 Carbon Server
#
# Environment Variable Prequisites
#
#   CARBON_HOME   Home of WSO2 Carbon installation. If not set I will
try
#                   to figure it out.
#
#   JAVA_HOME     Must point at your Java Development Kit installation.
#
#    JAVA_OPTS         (Optional) Java runtime options used when the
commands
#                   is executed.
#
# NOTE: Borrowed generously from Apache Tomcat startup scripts.
# ----------------------------------------------------------------------
----------

# OS specific support.  $var _must_ be set to either true or false.
#ulimit -n 100000

cygwin=false;
darwin=false;
os400=false;
mingw=false;
case "`uname`" in
CYGWIN*) cygwin=true;;
```

```
  MINGW*) mingw=true;;
  OS400*) os400=true;;
  Darwin*) darwin=true
          if [ -z "$JAVA_VERSION" ] ; then
               JAVA_VERSION="CurrentJDK"
             else
               echo "Using Java version: $JAVA_VERSION"
             fi
             if [ -z "$JAVA_HOME" ] ; then

JAVA_HOME=/System/Library/Frameworks/JavaVM.framework/Versions/${JAVA_
VERSION}/Home
             fi
             ;;
  esac

  # resolve links - $0 may be a softlink
  PRG="$0"

  while [ -h "$PRG" ]; do
    ls=`ls -ld "$PRG"`
    link=`expr "$ls" : '.*-> \(.*\)$'`
    if expr "$link" : '.*/.*' > /dev/null; then
      PRG="$link"
    else
      PRG=`dirname "$PRG"`/"$link"
    fi
  done

  # Get standard environment variables
  PRGDIR=`dirname "$PRG"`

  # Only set CARBON_HOME if not already set
  [ -z "$CARBON_HOME" ] && CARBON_HOME=`cd "$PRGDIR/.." ; pwd`

  # Set AXIS2_HOME. Needed for One Click JAR Download
  AXIS2_HOME=$CARBON_HOME

  # For Cygwin, ensure paths are in UNIX format before anything is touched
  if $cygwin; then
    [ -n "$JAVA_HOME" ] && JAVA_HOME=`cygpath --unix "$JAVA_HOME"`
    [ -n "$CARBON_HOME" ] && CARBON_HOME=`cygpath --unix "$CARBON_HOME"`
    [ -n "$AXIS2_HOME" ] && CARBON_HOME=`cygpath --unix "$CARBON_HOME"`
  fi

  # For OS400
  if $os400; then
    # Set job priority to standard for interactive (interactive - 6) by
using
    # the interactive priority - 6, the helper threads that respond to
requests
    # will be running at the same priority as interactive jobs.
    COMMAND='chgjob job('$JOBNAME') runpty(6)'
    system $COMMAND
```

```
    # Enable multi threading
    QIBM_MULTI_THREADED=Y
    export QIBM_MULTI_THREADED
  fi

  # For Migwn, ensure paths are in UNIX format before anything is touched
  if $mingw ; then
    [ -n "$CARBON_HOME" ] &&
      CARBON_HOME="`(cd "$CARBON_HOME"; pwd)`"
    [ -n "$JAVA_HOME" ] &&
      JAVA_HOME="`(cd "$JAVA_HOME"; pwd)`"
    [ -n "$AXIS2_HOME" ] &&
      CARBON_HOME="`(cd "$CARBON_HOME"; pwd)`"
    # TODO classpath?
  fi

  if [ -z "$JAVACMD" ] ; then
    if [ -n "$JAVA_HOME"  ] ; then
      if [ -x "$JAVA_HOME/jre/sh/java" ] ; then
        # IBM's JDK on AIX uses strange locations for the executables
        JAVACMD="$JAVA_HOME/jre/sh/java"
      else
        JAVACMD="$JAVA_HOME/bin/java"
      fi
    else
      JAVACMD=java
    fi
  fi

  if [ ! -x "$JAVACMD" ] ; then
    echo "Error: JAVA_HOME is not defined correctly."
    echo " CARBON cannot execute $JAVACMD"
    exit 1
  fi

  # if JAVA_HOME is not set we're not happy
  if [ -z "$JAVA_HOME" ]; then
    echo "You must set the JAVA_HOME variable before running CARBON."
    exit 1
  fi

  if [ -e "$CARBON_HOME/wso2carbon.pid" ]; then
    PID=`cat "$CARBON_HOME"/wso2carbon.pid`
  fi

  # ----- Process the input command ----------------------------------
-----------
  args=""
  for c in $*
  do
      if [ "$c" = "--debug" ] || [ "$c" = "-debug" ] || [ "$c" = "debug"
]; then
            CMD="--debug"
            continue
      elif [ "$CMD" = "--debug" ]; then
            if [ -z "$PORT" ]; then
                  PORT=$c
```

```
            fi
     elif [ "$c" = "--stop" ] || [ "$c" = "-stop" ] || [ "$c" = "stop"
]; then
            CMD="stop"
     elif [ "$c" = "--start" ] || [ "$c" = "-start" ] || [ "$c" = "start"
]; then
            CMD="start"
     elif [ "$c" = "--version" ] || [ "$c" = "-version" ] || [ "$c" =
"version" ]; then
            CMD="version"
     elif [ "$c" = "--restart" ] || [ "$c" = "-restart" ] || [ "$c" =
"restart" ]; then
            CMD="restart"
     elif [ "$c" = "--test" ] || [ "$c" = "-test" ] || [ "$c" = "test"
]; then
            CMD="test"
     else
         args="$args $c"
     fi
  done

  if [ "$CMD" = "--debug" ]; then
    if [ "$PORT" = "" ]; then
      echo " Please specify the debug port after the --debug option"
      exit 1
    fi
    if [ -n "$JAVA_OPTS" ]; then
      echo "Warning !!!. User specified JAVA_OPTS will be ignored, once
you give the --debug option."
    fi
    CMD="RUN"
    JAVA_OPTS="-Xdebug      -Xnoagent      -Djava.compiler=NONE      -
Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=$PORT"
    echo "Please start the remote debugging client to continue..."
  elif [ "$CMD" = "start" ]; then
    if [ -e "$CARBON_HOME/wso2carbon.pid" ]; then
      if  ps -p $PID > /dev/null ; then
        echo "Process is already running"
        exit 0
      fi
    fi
    export CARBON_HOME=$CARBON_HOME
  # using nohup sh to avoid erros in solaris OS.TODO
    nohup sh $CARBON_HOME/bin/iot-server.sh $args > /dev/null 2>&1 &
    exit 0
  elif [ "$CMD" = "stop" ]; then
    export CARBON_HOME=$CARBON_HOME
    kill -term `cat $CARBON_HOME/wso2carbon.pid`
    exit 0
  elif [ "$CMD" = "restart" ]; then
    export CARBON_HOME=$CARBON_HOME
    kill -term `cat $CARBON_HOME/wso2carbon.pid`
    process_status=0
    pid=`cat $CARBON_HOME/wso2carbon.pid`
    while [ "$process_status" -eq "0" ]
```

```
    do
          sleep 1;
          ps -p$pid 2>&1 > /dev/null
          process_status=$?
    done

  # using nohup sh to avoid erros in solaris OS.TODO
    nohup sh $CARBON_HOME/bin/iot-server.sh $args > /dev/null 2>&1 &
    exit 0
  elif [ "$CMD" = "test" ]; then
      JAVACMD="exec "$JAVACMD""
  elif [ "$CMD" = "version" ]; then
    cat $CARBON_HOME/bin/version.txt
    cat $CARBON_HOME/bin/wso2carbon-version.txt
    exit 0
  fi

  # ---------- Handle the SSL Issue with proper JDK version -----------
---------
  jdk_17=`$JAVA_HOME/bin/java -version 2>&1 | grep "1.[7|8]"`
  if [ "$jdk_17" = "" ]; then
     echo " Starting WSO2 Carbon (in unsupported JDK)"
     echo " [ERROR] CARBON is supported only on JDK 1.7 and 1.8"
  fi

  CARBON_XBOOTCLASSPATH=""
  for f in "$CARBON_HOME"/wso2/lib/xboot/*.jar
  do
      if [ "$f" != "$CARBON_HOME/wso2/lib/xboot/*.jar" ];then
          CARBON_XBOOTCLASSPATH="$CARBON_XBOOTCLASSPATH":$f
      fi
  done

  JAVA_ENDORSED_DIRS="$CARBON_HOME/wso2/lib/endorsed":"$JAVA_HOME/jre/l
ib/endorsed":"$JAVA_HOME/wso2/lib/endorsed"

  CARBON_CLASSPATH=""
  if [ -e "$JAVA_HOME/lib/tools.jar" ]; then
      CARBON_CLASSPATH="$JAVA_HOME/lib/tools.jar"
  fi
  for f in "$CARBON_HOME"/bin/*.jar
  do
      if [ "$f" != "$CARBON_HOME/bin/*.jar" ];then
          CARBON_CLASSPATH="$CARBON_CLASSPATH":$f
      fi
  done
  for t in "$CARBON_HOME"/wso2/lib/commons-lang*.jar
  do
      CARBON_CLASSPATH="$CARBON_CLASSPATH":$t
  done
  # For Cygwin, switch paths to Windows format before running java
  if $cygwin; then
    JAVA_HOME=`cygpath --absolute --windows "$JAVA_HOME"`
    CARBON_HOME=`cygpath --absolute --windows "$CARBON_HOME"`
    AXIS2_HOME=`cygpath --absolute --windows "$CARBON_HOME"`
    CLASSPATH=`cygpath --path --windows "$CLASSPATH"`
    JAVA_ENDORSED_DIRS=`cygpath --path --windows "$JAVA_ENDORSED_DIRS"`
```

```
    CARBON_CLASSPATH=`cygpath --path --windows "$CARBON_CLASSPATH"`
    CARBON_XBOOTCLASSPATH=`cygpath           --path           --windows
"$CARBON_XBOOTCLASSPATH"`
  fi

  # ----- Execute The Requested Command -----------------------------
----------

  echo JAVA_HOME environment variable is set to $JAVA_HOME
  echo CARBON_HOME environment variable is set to $CARBON_HOME

  cd "$CARBON_HOME"

  TMP_DIR=$CARBON_HOME/tmp
  if [ -d "$TMP_DIR" ]; then
  rm -rf "$TMP_DIR"
  fi

  START_EXIT_STATUS=121
  status=$START_EXIT_STATUS

  #To monitor a Carbon server in remote JMX mode on linux host machines,
set the below system property.
  #    -Djava.rmi.server.hostname="your.IP.goes.here"

  while [ "$status" = "$START_EXIT_STATUS" ]
  do
      $JAVACMD \
      -Xbootclasspath/a:"$CARBON_XBOOTCLASSPATH" \
      -Xms256m -Xmx1024m -XX:MaxPermSize=512m \
      -XX:+HeapDumpOnOutOfMemoryError \
      -XX:HeapDumpPath="$CARBON_HOME/repository/logs/heap-dump.hprof" \
      $JAVA_OPTS \
      -Dcom.sun.management.jmxremote \
      -classpath "$CARBON_CLASSPATH" \
      -Djava.endorsed.dirs="$JAVA_ENDORSED_DIRS" \
      -Djava.io.tmpdir="$CARBON_HOME/wso2/tmp" \
      -Dcatalina.base="$CARBON_HOME/wso2/lib/tomcat" \
      -Dwso2.server.standalone=true \
      -Dcarbon.registry.root=/ \
      -Djava.command="$JAVACMD" \
      -Dcarbon.home="$CARBON_HOME" \
      -Dlogger.server.name="IoT-Core" \
      -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
\
      -Dcarbon.config.dir.path="$CARBON_HOME/conf" \
      -Dcarbon.components.dir.path="$CARBON_HOME/wso2/components" \
      -Dcarbon.dropins.dir.path="$CARBON_HOME/dropins" \
      -Dcarbon.external.lib.dir.path="$CARBON_HOME/lib" \
      -Dcarbon.patches.dir.path="$CARBON_HOME/patches" \
      -Dcarbon.servicepacks.dir.path="$CARBON_HOME/servicepacks" \
      -Dcarbon.internal.lib.dir.path="$CARBON_HOME/wso2/lib" \
      -Djava.util.logging.config.file="$CARBON_HOME/conf/etc/logging-
bridge.properties" \
      -Dcomponents.repo="$CARBON_HOME/wso2/components/plugins" \
```

```
     -Dconf.location="$CARBON_HOME/conf"\
     -
Dcom.atomikos.icatch.file="$CARBON_HOME/wso2/lib/transactions.properti
es" \
     -Dcom.atomikos.icatch.hide_init_file_path=true \
     -Dorg.apache.jasper.compiler.Parser.STRICT_QUOTE_ESCAPING=false \
     -Dorg.apache.jasper.runtime.BodyContentImpl.LIMIT_BUFFER=true \
     -Dcom.sun.jndi.ldap.connect.pool.authentication=simple  \
     -Dcom.sun.jndi.ldap.connect.pool.timeout=3000  \
     -Dorg.terracotta.quartz.skipUpdateCheck=true \
     -Djava.security.egd=file:/dev/./urandom \
     -Dfile.encoding=UTF8 \
     -Djava.net.preferIPv4Stack=true \
     -Dcom.ibm.cacheLocalHost=true \
     -DworkerNode=false \
     -Dorg.wso2.ignoreHostnameVerification=true \
     -Dorg.opensaml.httpclient.https.disableHostnameVerification=true
\
     -Diot.analytics.host="localhost" \
     -Diot.analytics.https.port="9445" \
     -Diot.manager.host="localhost" \
     -Diot.manager.https.port="9443" \
     -Dmqtt.broker.host="localhost" \
     -Dmqtt.broker.port="1886" \
     -Diot.core.host="localhost" \
     -Diot.core.https.port="9443" \
     -Diot.keymanager.host="localhost" \
     -Diot.keymanager.https.port="9443" \
     -Diot.gateway.host="localhost" \
     -Diot.gateway.https.port="8243" \
     -Diot.gateway.http.port="8280" \
     -Diot.gateway.carbon.https.port="9443" \
     -Diot.gateway.carbon.http.port="9763" \
     -Diot.apimpublisher.host="localhost" \
     -Diot.apimpublisher.https.port="9443" \
     -Diot.apimstore.host="localhost" \
     -Diot.apimstore.https.port="9443" \
     -Dmqtt.broker.https.port="9446" \
     -Denable-api-scopes-sharing="true" \
     org.wso2.carbon.bootstrap.Bootstrap $*
     status=$?
  done
```

Anexo II: Script *analytics.sh*

```
 #!/bin/sh
 # analytics.sh
 # ----------------------------------------------------------------------
---------
 #  Copyright 2016 WSO2, Inc. http://www.wso2.org
 #
 #  Licensed under the Apache License, Version 2.0 (the "License");
 #  you may not use this file except in compliance with the License.
 #  You may obtain a copy of the License at
 #
 #      http://www.apache.org/licenses/LICENSE-2.0
 #
 #  Unless required by applicable law or agreed to in writing, software
 #  distributed under the License is distributed on an "AS IS" BASIS,
 #   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 #  See the License for the specific language governing permissions and
 #  limitations under the License.

 cygwin=false;
 darwin=false;
 os400=false;
 mingw=false;
 case "`uname`" in
 CYGWIN*) cygwin=true;;
 MINGW*) mingw=true;;
 OS400*) os400=true;;
 Darwin*) darwin=true
         if [ -z "$JAVA_VERSION" ] ; then
              JAVA_VERSION="CurrentJDK"
            else
              echo "Using Java version: $JAVA_VERSION"
            fi
            if [ -z "$JAVA_HOME" ] ; then

JAVA_HOME=/System/Library/Frameworks/JavaVM.framework/Versions/${JAVA_
VERSION}/Home
            fi
            ;;
 esac

 # resolve links - $0 may be a softlink
 PRG="$0"

 while [ -h "$PRG" ]; do
   ls=`ls -ld "$PRG"`
   link=`expr "$ls" : '.*-> \(.*\)$'`
   if expr "$link" : '.*/.*' > /dev/null; then
     PRG="$link"
   else
     PRG=`dirname "$PRG"`/"$link"
   fi
```

```
 done

 # Get standard environment variables
 PRGDIR=`dirname "$PRG"`

 # Only set CARBON_HOME if not already set
 [ -z "$CARBON_HOME" ] && CARBON_HOME=`cd "$PRGDIR/.." ; pwd`

 ###################################################################
######
 NAME=start-analytics
 # Daemon name, where is the actual executable
 ANALYTICS_INIT_SCRIPT="$CARBON_HOME/wso2/analytics/bin/wso2server.sh"

 # If the daemon is not there, then exit.

 $ANALYTICS_INIT_SCRIPT $*
 exit;
```

Anexo III: Script *broker.sh*

```
 #!/bin/sh
 # broker.sh
 # ----------------------------------------------------------------------
---------
 #  Copyright 2016 WSO2, Inc. http://www.wso2.org
 #
 #  Licensed under the Apache License, Version 2.0 (the "License");
 #  you may not use this file except in compliance with the License.
 #  You may obtain a copy of the License at
 #
 #      http://www.apache.org/licenses/LICENSE-2.0
 #
 #  Unless required by applicable law or agreed to in writing, software
 #  distributed under the License is distributed on an "AS IS" BASIS,
 #   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 #  See the License for the specific language governing permissions and
 #  limitations under the License.

 cygwin=false;
 darwin=false;
 os400=false;
 mingw=false;
 case "`uname`" in
 CYGWIN*) cygwin=true;;
 MINGW*) mingw=true;;
 OS400*) os400=true;;
 Darwin*) darwin=true
         if [ -z "$JAVA_VERSION" ] ; then
              JAVA_VERSION="CurrentJDK"
            else
              echo "Using Java version: $JAVA_VERSION"
            fi
            if [ -z "$JAVA_HOME" ] ; then

JAVA_HOME=/System/Library/Frameworks/JavaVM.framework/Versions/${JAVA_
VERSION}/Home
            fi
            ;;
 esac

 # resolve links - $0 may be a softlink
 PRG="$0"

 while [ -h "$PRG" ]; do
   ls=`ls -ld "$PRG"`
   link=`expr "$ls" : '.*-> \(.*\)$'`
   if expr "$link" : '.*/.*' > /dev/null; then
     PRG="$link"
   else
     PRG=`dirname "$PRG"`/"$link"
   fi
```

```
 done

 # Get standard environment variables
 PRGDIR=`dirname "$PRG"`

 # Only set CARBON_HOME if not already set
 [ -z "$CARBON_HOME" ] && CARBON_HOME=`cd "$PRGDIR/.." ; pwd`

 ###################################################################
######
 NAME=start-broker
 # Daemon name, where is the actual executable
 BROKER_INIT_SCRIPT="$CARBON_HOME/wso2/broker/bin/wso2server.sh"

 # If the daemon is not there, then exit.

 $BROKER_INIT_SCRIPT $*
 exit;
```

Anexo IV: Configuración del DAS

```
 <?xml version="1.0" encoding="ISO-8859-1"?>
 <!--
   ~ Copyright (c) 2016, WSO2 Inc. (http://www.wso2.org) All Rights
Reserved.
   ~
   ~ WSO2 Inc. licenses this file to you under the Apache License,
   ~ Version 2.0 (the "License"); you may not use this file except
   ~ in compliance with the License.
   ~ You may obtain a copy of the License at
   ~
   ~   http://www.apache.org/licenses/LICENSE-2.0
   ~
   ~ Unless required by applicable law or agreed to in writing,
   ~ software distributed under the License is distributed on an
   ~ "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
   ~ KIND, either express or implied.  See the License for the
   ~ specific language governing permissions and limitations
   ~ under the License.
   -->

 <!--
     This is the main server configuration file

     ${carbon.home} represents the carbon.home system property.
     Other system properties can be specified in a similar manner.
 -->
 <Server xmlns="http://wso2.org/projects/carbon/carbon.xml">

     <!--
        Product Name
     -->
     <Name>WSO2 IoT Server</Name>

     <!--
        machine readable unique key to identify each product
     -->
     <ServerKey>IoT</ServerKey>

     <!--
        Product Version
     -->
     <Version>3.3.0</Version>

     <!--
        Host name or IP address of the machine hosting this server
        e.g. www.wso2.org, 192.168.1.10
        This is will become part of the End Point Reference of the
        services deployed on this server instance.
     -->
     <!--HostName>www.wso2.org</HostName-->

     <!--
```

```
      Host name to be used for the Carbon management console
      -->
      <!--MgtHostName>mgt.wso2.org</MgtHostName-->

      <!--
          The URL of the back end server. This is where the admin services
are hosted and
          will be used by the clients in the front end server.
          This is required only for the Front-end server. This is used
when seperating BE server from FE server
         -->
      <ServerURL>local:/${carbon.context}/services/</ServerURL>
      <!--

<ServerURL>https://${carbon.local.ip}:${carbon.management.port}${carbo
n.context}/services/</ServerURL>
      -->
       <!--
       The URL of the index page. This is where the user will be
redirected after signing in to the
       carbon server.
       -->
      <!-- IndexPageURL>/carbon/admin/index.jsp</IndexPageURL-->

      <!--
      For cApp deployment, we have to identify the roles that can be
acted by the current server.
      The following property is used for that purpose. Any number of
roles can be defined here.
      Regular expressions can be used in the role.
      Ex : <Role>.*</Role> means this server can act any role
      -->
      <ServerRoles>
          <Role>ComplexEventProcessor</Role>
          <Role>DataAnalyticsServer</Role>
          <Role>GeoDashboard</Role>
      </ServerRoles>

      <!-- uncommnet this line to subscribe to a bam instance
automatically -->
      <!--
<BamServerURL>https://bamhost:bamport/services/</BamServerURL>-->

      <!--
         The fully qualified name of the server
      -->
      <Package>org.wso2.carbon</Package>

      <!--
         Webapp context root of WSO2 Carbon management console.
      -->
      <WebContextRoot>/</WebContextRoot>

      <!--

                                                                  Prox
```

```
y context path is a useful parameter to add a proxy path when a Carbon
server is fronted by reverse proxy. In addtion
         to the proxy host and proxy port this parameter allows you add
a path component to external URLs. e.g.

                                                                    URL
of the Carbon server -> https://10.100.1.1:9443/carbon

                                                                    URL
of the reverse proxy -> https://prod.abc.com/appserver/carbon



                                                                    apps
erver - proxy context path. This specially required whenever you are
generating URLs to displace in

                                                                    Carb
on UI components.
     -->
     <!--

                                                                    <Mgt
ProxyContextPath></MgtProxyContextPath>

                                                                    <Pro
xyContextPath></ProxyContextPath>
     -->

     <!-- In-order to  get the registry http Port from the back-end when
the default http transport is not the same-->
     <!--RegistryHttpPort>9763</RegistryHttpPort-->

     <!--
     Number of items to be displayed on a management console page. This
is used at the
     backend server for pagination of various items.
     -->
     <ItemsPerPage>15</ItemsPerPage>

     <!-- The endpoint URL of the cloud instance management Web service
-->
     <!--
<InstanceMgtWSEndpoint>https://ec2.amazonaws.com/</InstanceMgtWSEndpoi
nt>-->

     <!--
        Ports used by this server
     -->
     <Ports>

         <!-- Ports offset. This entry will set the value of the ports
defined below to
          the define value + Offset.
          e.g. Offset=2 and HTTPS port=9443 will set the effective HTTPS
port to 9445
```

```
          -->
         <Offset>2</Offset>

         <!-- The JMX Ports -->
         <JMX>
             <!--The port RMI registry is exposed-->
             <RMIRegistryPort>9999</RMIRegistryPort>
             <!--The port RMI server should be exposed-->
             <RMIServerPort>11111</RMIServerPort>
         </JMX>

         <!-- Embedded LDAP server specific ports -->
         <EmbeddedLDAP>
             <!-- Port which embedded LDAP server runs -->
             <LDAPServerPort>10389</LDAPServerPort>
             <!-- Port which KDC (Kerberos Key Distribution Center)
server runs -->
             <KDCServerPort>8000</KDCServerPort>
         </EmbeddedLDAP>

                                                                    <!--
              Override datasources JNDIproviderPort defined in bps.xml
and datasources.properties files
                                                                    -->
                                                                    <!--
<JNDIProviderPort>2199</JNDIProviderPort>-->
                                                                    <!--
Override receive port of thrift based entitlement service.-->
                                                                    <Thr
iftEntitlementReceivePort>10500</ThriftEntitlementReceivePort>

         <!--
          This is the proxy port of the worker cluster. These need to
be configured in a scenario where
          manager node is not exposed through the load balancer through
which the workers are exposed
          therefore doesn't have a proxy port.
         <WorkerHttpProxyPort>80</WorkerHttpProxyPort>
         <WorkerHttpsProxyPort>443</WorkerHttpsProxyPort>
         -->

     </Ports>

     <!--
         JNDI Configuration
     -->
     <JNDI>
         <!--
              The fully qualified name of the default initial context
factory
         -->

<DefaultInitialContextFactory>org.wso2.carbon.tomcat.jndi.CarbonJavaUR
LContextFactory</DefaultInitialContextFactory>
         <!--
              The restrictions that are done to various JNDI Contexts
in a Multi-tenant environment
```

```
         -->
         <Restrictions>
             <!--
                 Contexts that will be available only to the super-
tenant
             -->
             <!-- <SuperTenantOnly>
                 <UrlContexts>
                     <UrlContext>
                         <Scheme>foo</Scheme>
                     </UrlContext>
                     <UrlContext>
                         <Scheme>bar</Scheme>
                     </UrlContext>
                 </UrlContexts>
             </SuperTenantOnly> -->
             <!--
                 Contexts that are common to all tenants
             -->
             <AllTenants>
                 <UrlContexts>
                     <UrlContext>
                         <Scheme>java</Scheme>
                     </UrlContext>
                     <!-- <UrlContext>
                         <Scheme>foo</Scheme>
                     </UrlContext> -->
                 </UrlContexts>
             </AllTenants>
             <!--
                  All other contexts not mentioned above will be
available on a per-tenant basis
                  (i.e. will not be shared among tenants)
             -->
         </Restrictions>
     </JNDI>

     <!--
         Property to determine if the server is running an a cloud
deployment environment.
         This property should only be used to determine deployment
specific details that are
         applicable only in a cloud deployment, i.e when the server
deployed *-as-a-service.
     -->
     <IsCloudDeployment>false</IsCloudDeployment>

     <!--
                                                                  Prop
erty to determine whether usage data should be collected for metering
purposes
     -->
     <EnableMetering>false</EnableMetering>
```

```
     <!-- The Max time a thread should take for execution in seconds -
->
     <MaxThreadExecutionTime>600</MaxThreadExecutionTime>

     <!--
         A flag to enable or disable Ghost Deployer. By default this is
set to false. That is
         because the Ghost Deployer works only with the HTTP/S
transports. If you are using
         other transports, don't enable Ghost Deployer.
     -->
     <GhostDeployment>
         <Enabled>false</Enabled>
     </GhostDeployment>


     <!--
         Eager loading or lazy loading is a design pattern commonly used
in computer programming which
         will initialize an object upon creation or load on-demand. In
carbon, lazy loading is used to
         load tenant when a request is received only. Similarly Eager
loading is used to enable load
         existing tenants after carbon server starts up. Using this
feature, you will be able to include
         or exclude tenants which are to be loaded when server startup.

         We can enable only one LoadingPolicy at a given time.

         1. Tenant Lazy Loading
            This is the default behaviour and enabled by default. With
this policy, tenants are not loaded at
            server startup, but loaded based on-demand (i.e when a
request is received for a tenant).
            The default tenant idle time is 30 minutes.

         2. Tenant Eager Loading
            This is by default not enabled. It can be be enabled by un-
commenting the <EagerLoading> section.
            The eager loading configurations supported are as below.
These configurations can be given as the
            value for <Include> element with eager loading.
                 (i)Load all tenants when server startup             -
*
                 (ii)Load all tenants except foo.com & bar.com       -
*,!foo.com,!bar.com
                 (iii)Load only foo.com &  bar.com to be included    -
foo.com,bar.com
     -->
     <Tenant>
         <LoadingPolicy>
             <LazyLoading>
                 <IdleTime>153722867280912</IdleTime>
             </LazyLoading>
             <!-- <EagerLoading>
                    <Include>*,!foo.com,!bar.com</Include>
             </EagerLoading>-->
```

```
         </LoadingPolicy>
      </Tenant>

      <!--
       Caching related configurations
      -->
      <Cache>
         <!-- Default cache timeout in minutes -->
         <DefaultCacheTimeout>15</DefaultCacheTimeout>
      </Cache>

      <!--
      Axis2 related configurations
      -->
      <Axis2Config>
         <!--
              Location of the Axis2 Services & Modules repository

              This can be a directory in the local file system, or a
URL.

              e.g.
              1. /home/wso2wsas/repository/ - An absolute path
              2. repository - In this case, the path is relative to
CARBON_HOME
              3. file:///home/wso2wsas/repository/
              4. http://wso2wsas/repository/
         -->

<RepositoryLocation>${carbon.home}/repository/deployment/server/</Repo
sitoryLocation>

         <!--
          Deployment update interval in seconds. This is the interval
between repository listener
          executions.
         -->
         <DeploymentUpdateInterval>15</DeploymentUpdateInterval>

         <!--
             Location of the main Axis2 configuration descriptor file,
a.k.a. axis2.xml file

             This can be a file on the local file system, or a URL

             e.g.
             1. /home/repository/axis2.xml - An absolute path
             2. conf/axis2.xml - In this case, the path is relative to
CARBON_HOME
             3. file:///home/carbon/repository/axis2.xml
             4. http://repository/conf/axis2.xml
         -->

<ConfigurationFile>${carbon.home}/conf/axis2/axis2.xml</ConfigurationF
ile>
```

```
         <!--
           ServiceGroupContextIdleTime,   which   will   be   set   in
ConfigurationContex
           for multiple clients which are going to access the same
ServiceGroupContext
           Default Value is 30 Sec.
         -->

<ServiceGroupContextIdleTime>30000</ServiceGroupContextIdleTime>

         <!--
           This repository location is used to crete the client side
configuration
           context used by the server when calling admin services.
         -->

<ClientRepositoryLocation>${carbon.home}/repository/deployment/client/
</ClientRepositoryLocation>
         <!-- This axis2 xml is used in createing the configuration
context by the FE server
          calling to BE server -->

<clientAxis2XmlLocation>${carbon.home}/conf/axis2/axis2_client.xml</cl
ientAxis2XmlLocation>
         <!-- If this parameter is set, the ?wsdl on an admin service
will not give the admin service wsdl. -->
         <HideAdminServiceWSDLs>true</HideAdminServiceWSDLs>

                                                                    <!--
WARNING-Use With Care! Uncommenting bellow parameter would expose all
AdminServices in HTTP transport.
                                                                    With
HTTP transport your credentials and data routed in public channels are
vulnerable for sniffing attacks.
                                                                    Use
bellow parameter ONLY if your communication channels are confirmed to
be secured by other means -->
         <!--HttpAdminServices>*</HttpAdminServices-->

     </Axis2Config>

     <!--
        The default user roles which will be created when the server
        is started up for the first time.
     -->
     <ServiceUserRoles>
         <Role>
             <Name>admin</Name>
             <Description>Default Administrator Role</Description>
         </Role>
         <Role>
             <Name>user</Name>
             <Description>Default User Role</Description>
         </Role>
     </ServiceUserRoles>
```

```
      <!--
        Enable following config to allow Emails as usernames.
      -->
      <!--EnableEmailUserName>true</EnableEmailUserName-->

      <!--
        Security configurations
      -->
      <Security>
          <!--
              KeyStore which will be used for encrypting/decrypting
passwords
              and other sensitive information.
          -->
          <KeyStore>
              <!-- Keystore file location-->

<Location>${carbon.home}/repository/resources/security/wso2carbon.jks<
/Location>
              <!-- Keystore type (JKS/PKCS12 etc.)-->
              <Type>JKS</Type>
              <!-- Keystore password-->
              <Password>wso2carbon</Password>
              <!-- Private Key alias-->
              <KeyAlias>wso2carbon</KeyAlias>
              <!-- Private Key password-->
              <KeyPassword>wso2carbon</KeyPassword>
          </KeyStore>

          <!--
              System wide trust-store which is used to maintain the
certificates of all
              the trusted parties.
          -->
          <TrustStore>
              <!-- trust-store file location -->

<Location>${carbon.home}/repository/resources/security/client-
truststore.jks</Location>
              <!-- trust-store type (JKS/PKCS12 etc.) -->
              <Type>JKS</Type>
              <!-- trust-store password -->
              <Password>wso2carbon</Password>
          </TrustStore>

          <!--
              The Authenticator configuration to be used at the JVM
level. We extend the
              java.net.Authenticator to make it possible to authenticate
to given servers and
              proxies.
          -->
          <NetworkAuthenticatorConfig>
              <!--
```

```
                Below is a sample configuration for a single
authenticator. Please note that
                all child elements are mandatory. Not having some child
elements would lead to
                exceptions at runtime.
            -->
            <!-- <Credential> -->
                <!--
                    the pattern that would match a subset of URLs for
which this authenticator
                    would be used
                -->
                <!-- <Pattern>regularExpression</Pattern> -->
                <!--
                    the type of this authenticator. Allowed values are:
                    1. server
                    2. proxy
                -->
                <!-- <Type>proxy</Type> -->
                <!-- the username used to log in to server/proxy -->
                <!-- <Username>username</Username> -->
                <!-- the password used to log in to server/proxy -->
                <!-- <Password>password</Password> -->
            <!-- </Credential> -->
        </NetworkAuthenticatorConfig>

        <!--
         The Tomcat realm to be used for hosted Web applications.
Allowed values are;
         1. UserManager
         2. Memory

         If this is set to 'UserManager', the realm will pick users &
roles from the system's
         WSO2 User Manager. If it is set to 'memory', the realm will
pick users & roles from
         CARBON_HOME/repository/conf/tomcat/tomcat-users.xml
        -->
        <TomcatRealm>UserManager</TomcatRealm>

                                                                <!--
Option to disable storing of tokens issued by STS-->
                                                                <Dis
ableTokenStore>false</DisableTokenStore>

        <!--
         Security token store class name. If this is not set, default
class will be
         org.wso2.carbon.security.util.SecurityTokenStore
        -->
        <!--
TokenStoreClassName>org.wso2.carbon.identity.sts.store.DBTokenStore</T
okenStoreClassName-->

        <XSSPreventionConfig>
            <Enabled>true</Enabled>
            <Rule>allow</Rule>
```

```
            <Patterns>
                <!--Pattern></Pattern-->
            </Patterns>
        </XSSPreventionConfig>

        <!-- Configurations to avoid Cross Site Request Forgery
vulnerabilities -->
        <CSRFPreventionConfig>
            <!-- CSRFPreventionFilter configurations that adopts
Synchronizer Token Pattern -->
            <CSRFPreventionFilter>
                <!-- Set below to true to enable the
CSRFPreventionFilter -->
                <Enabled>false</Enabled>
                <!-- Url Pattern to skip application of CSRF
protection-->

<SkipUrlPattern>(.*)(/images|/css|/js|/docs)(.*)</SkipUrlPattern>
            </CSRFPreventionFilter>
        </CSRFPreventionConfig>

        <!-- Configuration to enable or disable CR and LF sanitization
filter-->
        <CRLFPreventionConfig>
            <!--Set below to true to enable the CRLFPreventionFilter-
->
            <Enabled>true</Enabled>
        </CRLFPreventionConfig>
    </Security>

    <!--
       The temporary work directory
    -->
    <WorkDirectory>${carbon.home}/tmp/work</WorkDirectory>

    <!--
       House-keeping configuration
    -->
    <HouseKeeping>

        <!--
           true  - Start House-keeping thread on server startup
           false - Do not start House-keeping thread on server startup.
                   The user will run it manually as and when he wishes.
        -->
        <AutoStart>true</AutoStart>

        <!--
           The interval in *minutes*, between house-keeping runs
        -->
        <Interval>10</Interval>

        <!--
          The maximum time in *minutes*, temp files are allowed to live
```

```
            in the system. Files/directories which were modified more
than
            "MaxTempFileLifetime" minutes ago will be removed by the
            house-keeping task
          -->
          <MaxTempFileLifetime>30</MaxTempFileLifetime>
      </HouseKeeping>

      <!--
         Configuration for handling different types of file upload &
other file uploading related
         config parameters.
         To map all actions to a particular FileUploadExecutor, use
         <Action>*</Action>
      -->
      <FileUploadConfig>
          <!--
             The total file upload size limit in MB
          -->
          <TotalFileSizeLimit>100</TotalFileSizeLimit>

          <Mapping>
              <Actions>
                  <Action>keystore</Action>
                  <Action>certificate</Action>
                  <Action>*</Action>
              </Actions>

<Class>org.wso2.carbon.ui.transports.fileupload.AnyFileUploadExecutor<
/Class>
          </Mapping>

          <Mapping>
              <Actions>
                  <Action>jarZip</Action>
              </Actions>

<Class>org.wso2.carbon.ui.transports.fileupload.JarZipUploadExecutor</
Class>
          </Mapping>
          <Mapping>
              <Actions>
                  <Action>dbs</Action>
              </Actions>

<Class>org.wso2.carbon.ui.transports.fileupload.DBSFileUploadExecutor<
/Class>
          </Mapping>
          <Mapping>
              <Actions>
                  <Action>tools</Action>
              </Actions>

<Class>org.wso2.carbon.ui.transports.fileupload.ToolsFileUploadExecuto
r</Class>
          </Mapping>
          <Mapping>
```

```
            <Actions>
                <Action>toolsAny</Action>
            </Actions>

<Class>org.wso2.carbon.ui.transports.fileupload.ToolsAnyFileUploadExec
utor</Class>
         </Mapping>
     </FileUploadConfig>

     <!-- FileNameRegEx is used to validate the file input/upload/write-
out names.
     e.g.
      <FileNameRegEx>^(?!(?:CON|PRN|AUX|NUL|COM[1-9]|LPT[1-
9])(?:\.[^.])?$)[^&lt;&gt:"/\\|?*\x00-\x1F][^&lt;&gt:"/\\|?*\x00-\x1F\
.]$</FileNameRegEx>
     -->
     <!--<FileNameRegEx></FileNameRegEx>-->

     <!--
        Processors which process special HTTP GET requests such as
?wsdl, ?policy etc.

        In order to plug in a processor to handle a special request,
simply add an entry to this
        section.

        The value of the Item element is the first parameter in the
query string(e.g. ?wsdl)
        which needs special processing

        The value of the Class element is a class which implements
        org.wso2.carbon.transport.HttpGetRequestProcessor
     -->
     <HttpGetRequestProcessors>
         <Processor>
             <Item>info</Item>

<Class>org.wso2.carbon.core.transports.util.InfoProcessor</Class>
         </Processor>
         <Processor>
             <Item>wsdl</Item>

<Class>org.wso2.carbon.core.transports.util.Wsdl11Processor</Class>
         </Processor>
         <Processor>
             <Item>wsdl2</Item>

<Class>org.wso2.carbon.core.transports.util.Wsdl20Processor</Class>
         </Processor>
         <Processor>
             <Item>xsd</Item>

<Class>org.wso2.carbon.core.transports.util.XsdProcessor</Class>
         </Processor>
     </HttpGetRequestProcessors>
```

```
     <!-- Deployment Synchronizer Configuration. t Enabled value to true
when running with "svn based" dep sync.
                                                                      In
master nodes you need to set both AutoCommit and AutoCheckout to true
                                                                     and
in  worker nodes set only AutoCheckout to true.
     -->
     <DeploymentSynchronizer>
         <Enabled>false</Enabled>
         <AutoCommit>false</AutoCommit>
         <AutoCheckout>true</AutoCheckout>
         <RepositoryType>svn</RepositoryType>
         <SvnUrl>http://svnrepo.example.com/repos/</SvnUrl>
         <SvnUser>username</SvnUser>
         <SvnPassword>password</SvnPassword>
         <SvnUrlAppendTenantId>true</SvnUrlAppendTenantId>
     </DeploymentSynchronizer>

     <!--  Deployment  Synchronizer  Configuration.  Uncomment  the
following section when running with "registry based" dep sync.
         In  master  nodes  you  need  to  set  both  AutoCommit  and
AutoCheckout to true
         and in  worker nodes set only AutoCheckout to true.
     -->
     <!--<DeploymentSynchronizer>
         <Enabled>true</Enabled>
         <AutoCommit>false</AutoCommit>
         <AutoCheckout>true</AutoCheckout>
     </DeploymentSynchronizer>-->

     <!-- Mediation persistence configurations. Only valid if mediation
features are available i.e. ESB -->
     <!--<MediationConfig>
         <LoadFromRegistry>false</LoadFromRegistry>
         <SaveToFile>false</SaveToFile>
         <Persistence>enabled</Persistence>
         <RegistryPersistence>enabled</RegistryPersistence>
     </MediationConfig>-->

     <!--
     Server intializing code, specified as implementation classes of
org.wso2.carbon.core.ServerInitializer.
     This code will be run when the Carbon server is initialized
     -->
     <ServerInitializers>
         <!--<Initializer></Initializer>-->
     </ServerInitializers>

     <!--
     Indicates whether the Carbon Servlet is required by the system,
and whether it should be
     registered
     -->

<RequireCarbonServlet>${require.carbon.servlet}</RequireCarbonServlet>
```

```
     <!--
     Carbon H2 OSGI Configuration
     By default non of the servers start.
         name="web" - Start the web server with the H2 Console
         name="webPort" - The port (default: 8082)
         name="webAllowOthers" - Allow other computers to connect
         name="webSSL" - Use encrypted (HTTPS) connections
         name="tcp" - Start the TCP server
         name="tcpPort" - The port (default: 9092)
         name="tcpAllowOthers" - Allow other computers to connect
         name="tcpSSL" - Use encrypted (SSL) connections
         name="pg" - Start the PG server
         name="pgPort"  - The port (default: 5435)
         name="pgAllowOthers"  - Allow other computers to connect
         name="trace" - Print additional trace information; for all
servers
         name="baseDir" - The base directory for H2 databases; for all
servers
     -->
     <!--H2DatabaseConfiguration>
         <property name="web" />
         <property name="webPort">8082</property>
         <property name="webAllowOthers" />
         <property name="webSSL" />
         <property name="tcp" />
         <property name="tcpPort">9092</property>
         <property name="tcpAllowOthers" />
         <property name="tcpSSL" />
         <property name="pg" />
         <property name="pgPort">5435</property>
         <property name="pgAllowOthers" />
         <property name="trace" />
         <property name="baseDir">${carbon.home}</property>
     </H2DatabaseConfiguration-->
     <!--Disabling statistics reporter by default-->
     <StatisticsReporterDisabled>true</StatisticsReporterDisabled>

     <!-- Enable accessing Admin Console via HTTP -->
     <!-- EnableHTTPAdminConsole>true</EnableHTTPAdminConsole -->

     <!--
        Default Feature Repository of WSO2 Carbon.
     -->
     <FeatureRepository>
         <RepositoryName>default repository</RepositoryName>
         <RepositoryURL>http://product-
dist.wso2.com/p2/carbon/releases/wilkes/</RepositoryURL>
     </FeatureRepository>

     <!--
                                                                  Conf
igure API Management
    -->
    <APIManagement>
```

```
                                                                        <!--
Uses the embedded API Manager by default. If you want to use an external
                                                                        API
Manager instance to manage APIs, configure below  externalAPIManager-->

                                                                        <Ena
bled>true</Enabled>

                                                                        <!--
Uncomment and configure API Gateway and
                                                                        Publ
isher URLs to use external API Manager instance-->

                                                                        <!--
ExternalAPIManager>



                                                                        <API
GatewayURL>http://localhost:8281</APIGatewayURL>

                                                                        <API
PublisherURL>http://localhost:8281/publisher</APIPublisherURL>

                                                                        </Ex
ternalAPIManager-->

                                                                        <Loa
dAPIContextsInServerStartup>true</LoadAPIContextsInServerStartup>
    </APIManagement>
  </Server>
```

Anexo V: Ficheros de META-INF y WEB-INF

permissions.xml

```
<PermissionConfiguration>
    <APIVersion></APIVersion>
    <!-- Device related APIs -->
    <Permission>
        <name>Get device</name>
        <path>/device-mgt/myRaspberry/user</path>
        <url>/device/*</url>
        <method>GET</method>
        <scope>myRaspberry_user</scope>
    </Permission>
    <Permission>
        <name>Remove device</name>
        <path>/device-mgt/myRaspberry/user</path>
        <url>/device/*</url>
        <method>DELETE</method>
        <scope>myRaspberry_user</scope>
    </Permission>
    <Permission>
        <name>Download device</name>
        <path>/device-mgt/myRaspberry/user</path>
        <url>/device/download</url>
```

```
        <method>GET</method>
        <scope>myRaspberry_user</scope>
    </Permission>
    <Permission>
        <name>Update device</name>
        <path>/device-mgt/myRaspberry/user</path>
        <url>/device/*</url>
        <method>POST</method>
        <scope>myRaspberry_user</scope>
    </Permission>
    <Permission>
        <name>Get Devices</name>
        <path>/device-mgt/myRaspberry/user</path>
        <url>/device</url>
        <method>GET</method>
        <scope>myRaspberry_user</scope>
    </Permission>
    <Permission>
        <name>Register Device</name>
        <path>/device-mgt/myRaspberry/user</path>
        <url>/device/register</url>
        <method>POST</method>
        <scope>myRaspberry_device</scope>
    </Permission>
    <Permission>
        <name>Control Sensor</name>
        <path>/device-mgt/myRaspberry/user</path>
        <url>/device/*/change-status</url>
        <method>POST</method>
        <scope>myRaspberry_user</scope>
    </Permission>
    <Permission>
        <name>Get Stats</name>
        <path>/device-mgt/myRaspberry/user</path>
        <url>/device/stats/*</url>
        <method>GET</method>
        <scope>myRaspberry_device</scope>
    </Permission>
</PermissionConfiguration>
```

webapp-classloading.xml

```
<Classloading xmlns="http://wso2.org/projects/as/classloading">

    <!-- Parent-first or child-first. Default behaviour is child-first.--
>
    <ParentFirst>false</ParentFirst>
```

```
    <!--
                                                                    Defa
ult environments that contains provides to all the webapps. This can be
overridden by individual webapps by specifing required environments
                                                                    Tomc
at environment is the default and every webapps gets it even if they didn't
specify it.
                                                                    e.g.
If a webapps requires CXF, they will get both Tomcat and CXF.
     -->
    <Environments>CXF,Carbon</Environments>
</Classloading>
```

cxf-servlet.xml

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxrs="http://cxf.apache.org/jaxrs"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
         http://cxf.apache.org/jaxrs
http://cxf.apache.org/schemas/jaxrs.xsd">
    <jaxrs:server id="myRaspberry" address="/">
        <jaxrs:serviceBeans>
            <bean id="DeviceTypeService"

class="sensorboard.raspberry.api.DeviceTypeServiceImpl">
            </bean>
        </jaxrs:serviceBeans>
        <jaxrs:providers>
            <bean
class="org.codehaus.jackson.jaxrs.JacksonJsonProvider"/>
        </jaxrs:providers>
    </jaxrs:server>
</beans>
```

web.xml

```
p://www.w3.org/2001/XMLSchema-instance"
         version="2.5"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         metadata-complete="true">
    <display-name>WSO2 IoT Server</display-name>
    <description>WSO2 IoT Server</description>

    <servlet>
        <servlet-name>CXFServlet</servlet-name>
        <servlet-
class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>CXFServlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
    <context-param>
```

```
        <param-name>doAuthentication</param-name>
        <param-value>true</param-value>
    </context-param>
    <context-param>
        <param-name>isSharedWithAllTenants</param-name>
        <param-value>false</param-value>
    </context-param>

    <!--publish to apim-->
    <context-param>
        <param-name>managed-api-enabled</param-name>
        <param-value>true</param-value>
    </context-param>
</web-app>
```

Anexo VI: Clase DeviceTypeConstants.java

Define las constantes que serán usadas en el código.

```
public class DeviceTypeConstants {
    public final static String DEVICE_TYPE = "myRaspberry";
    public final static String DEVICE_PLUGIN_DEVICE_NAME = "DEVICE_NAME";
    public    final    static    String    DEVICE_PLUGIN_DEVICE_ID    =
"myRaspberry_DEVICE_ID";
    public final static String STATE_ON = "ON";
    public final static String STATE_OFF = "OFF";

    //sensor events summerized table name
    public    static    final    String    SENSOR_TYPE1_EVENT_TABLE    =
"ORG_WSO2_IOT_DEVICES_temperature";
    public static final String DATA_SOURCE_NAME = "jdbc/myRaspberryDM_DB";
    public   final   static   String   DEVICE_TYPE_PROVIDER_DOMAIN   =
"carbon.super";
    public final static String SENSOR_TYPE1 = "temperature";

    //mqtt tranport related constants
    public static final String MQTT_ADAPTER_NAME = "temperature_mqtt";
    public static final String MQTT_ADAPTER_TYPE = "oauth-mqtt";
    public static final String ADAPTER_TOPIC_PROPERTY = "topic";
    public static final String MQTT_PORT = "\\{mqtt.broker.port\\}";
    public     static     final     String     MQTT_BROKER_HOST     =
"\\{mqtt.broker.host\\}";
    public static final String CARBON_CONFIG_PORT_OFFSET = "Ports.Offset";
    public  static  final  String  DEFAULT_CARBON_LOCAL_IP_PROPERTY  =
"carbon.local.ip";
    public static final int CARBON_DEFAULT_PORT_OFFSET = 0;
    public static final int DEFAULT_MQTT_PORT = 1883;
    public     static     final     String     SUBSCRIBED_TOPIC     =
"carbon.super/myRaspberry/+/temperature";
    public    static    final    String    CONTENT_TRANSFORMATION    =
"contentTransformer";
    public static final String CONTENT_VALIDATION = "contentValidator";
    public static final String RESOURCE = "resource";
```

```
    public static final String USERNAME_PROPERTY_KEY = "username";
    public static final String DCR_PROPERTY_KEY = "dcrUrl";
    public static final String BROKER_URL_PROPERTY_KEY = "url";
    public static final String SCOPES_PROPERTY_KEY = "scopes";
    public static final String QOS_PROPERTY_KEY = "qos";
    public static final String CLIENT_ID_PROPERTY_KEY = "qos";
    public    static    final    String    CLEAR_SESSION_PROPERTY_KEY    =
"clearSession";
    public static final String TOPIC = "topic";

    public   final   static   String   SENSOR_TYPE1_STREAM_DEFINITION   =
"org.wso2.iot.devices.temperature";
    public final static String SENSOR_TYPE1_STREAM_DEFINITION_VERSION =
"1.0.0";

    public     static     final     String     MQTT_CONFIG_LOCATION     =
CarbonUtils.getEtcCarbonConfigDirPath() + File.separator
            + "mqtt.properties";

    //agent config file related constants.
    public static final String ZIP_CONFIG_SERVER_NAME = "SERVER_NAME";
    public static final String ZIP_CONFIG_DEVICE_OWNER = "DEVICE_OWNER";
    public static final String ZIP_CONFIG_DEVICE_ID = "DEVICE_ID";
    public static final String ZIP_CONFIG_DEVICE_NAME = "DEVICE_NAME";
    public static final String ZIP_CONFIG_HTTPS_EP = "HTTPS_EP";
    public static final String ZIP_CONFIG_HTTP_EP = "HTTP_EP";
    public static final String ZIP_CONFIG_APIM_EP = "APIM_EP";
    public static final String ZIP_CONFIG_MQTT_EP = "MQTT_EP";
    public static final String ZIP_CONFIG_DEVICE_TOKEN = "DEVICE_TOKEN";
    public   static   final   String   ZIP_CONFIG_DEVICE_REFRESH_TOKEN   =
"DEVICE_REFRESH_TOKEN";
    public   static   final   String   ZIP_CONFIG_API_APPLICATION_KEY   =
"API_APPLICATION_KEY";
}
```

Anexo VII: Librería paho.mqtt

Esta librería de *Eclipse* [11] implementa un cliente *MQTT* versión 3.1 y 3.1.1. Permite a aplicaciones en Python el hacer uso de un bróker de comunicación para publicar y escuchar de *topics*. Debido a su complejidad, no analizaremos el código entero, solo su funcionamiento y las partes del mismo más importante para el proyecto.

Define funciones de ayuda, para que esto se realice de forma fácil y sencilla, analicemos la librería:

```
mqttClient = mqtt.Client(client_id='myRaspberry_client')
```

Constructor de la clase, permite los siguientes parámetros:

- client_id: Un identificador de cliente cuando se conecta con el bróker. Si no está definido, se genera de forma aleatoria, pero el siguiente parámetro debe estar puesto a *true*.
- clean_session: Define la forma de tratar la información del cliente cuando se desconecta. Si está definido como *true* se borrará toda la información, en caso de que esté a *false* no se eliminarán los datos del cliente.
- userdata: Datos de usuario que se le pasa a las funciones *callback*.
- protocol: La versión de *MQTT* a usar, permite el uso de MQTTv31 o MQTTv311.

- transport: El método de transporte, que por defecto es TCP, pero puede usar websockets.

Entre las funciones que implementa, tiene funciones de uso de websockets, que no es el caso de este trabajo de fin de grado, analizaremos las que vamos a usar:

username_pw_set(): llamado antes de la función subscribe(), define la autenticación con el *topic* que en nuestro caso será mediante *token*.

subscribe(topic): Suscribe el objeto a un topic MQTT identificado por una cadena.

on_connect(client, userdata,flgas,rc): Se usa para definer una función de callback que será llamada cuando el bróker responda al intento de conexión.

on_message(client, userdata, message): Con este método logramos definir una función que será el callback ante un evento de recibir un mensaje en el topic.

loop_forever(): Genera un bucle infinito, que en caso de recibir un mensaje, ejecutará la función definida con el método on_message().

Anexo VIII: Librería w1thermsensor

Da soporte para leer sensores que funcionan desde el pin w1 del GPIO de Raspberry Pi. Los dispositivos que soporta son los siguientes:

- DS18B20
- DS1822
- DS18S20
- DS28EA00
- DS1825/MAX31850K

Tiene distintas formas de instalación, mediante *pip:* con la ventaja de ser multi-distribución:

```
pip install w1thermsensor
```

Desde Raspbian con apt-get:

```
sudo apt-get install python3-w1thermsensor
```

Se usa de una forma muy simple y sencilla importando el módulo en el programa en *Python:*

```
from w1thermsensor import W1ThermSensor
```

Tras esto, tenemos que instanciar un objeto:

```
sensor=W1ThermSensor()
```

y posteriormente obtenemos los datos, por defecto nos lo dará en grados Celsius.

```
temperatura = sensor.get_temperature()
```

- Múltiples sensores

Los sensores pueden ser inicializados según su ID, lo que hace que podamos operar con múltiples sensores, para ello, primero debemos saber que ID tiene, eso lo logramos con el método *get_available_sensors().*

Podemos también discriminar según el tipo de sensor, por ejemplo:

```
get_available_sensors([W1ThermSensor.THERM_SENSOR_DS18B20])
```

Por último, podemos cambiar la precisión del sensor, para ello tenemos el método *set_precision().*

Anexo IX: Librería Luma para Max7219

Es una librería en Python que proporciona una interfaz para una matriz de leds con un controlador Max7219, mediante SPI.

Para inicializar el dispositivo, primero importamos el paquete:

```
import max7219.led as led
```

luego, asignamos a una variable:

```
device = led.matrix()
```

y para finalizar, para mostrar un mensaje en la matriz:

```
device.show_message("mensaje")
```

REFERENCIAS

[1] Massachusetts Institute of Technology. http://www.mit.edu/about

[2] Fundación Raspberry Pi. https://www.raspberrypi.org

[3] WSO2, 2005. https://wso2.com

[4] Handlebars. https://handlebarsjs.com

[5] Jaggery. http://jaggeryjs.org

[6] Barebone https://es.wikipedia.org/wiki/Barebone

[7] MQTT. http://mqtt.org

[8] Swagger. https://swagger.io

[9] Atom. https://atom.io

[10] Vim. https://www.vim.org

[11] Eclipse. https://www.eclipse.org

[12] Guake. https://github.com/guake

[13] Oh-My-Zsh!. https://ohmyz.sh

[14] Git. https://git-scm.com

[15] Github. https://github.com

[16] Powerline. https://github.com/powerline/fonts

[17] Apache. https://www.apache.org

[18] OAuth. https://oauht.net

[19] UUF. https://www.openhub.net/p/carbon-uuf

[20] Rickshaw. https://github.com/shutterstock/rickshaw

[21] Test::Simple: https://perldoc.perl.org/Test/Simple.html

[22] Test::More: https://perldoc.perl.org/Test/More.html

GLOSARIO

IoT: *Internet of Things* – Internet de las cosas

REST: *Representational State Transfer* – Transferencia de estado representacional

API: *Application Programing Interface* – Interfaz de programación de aplicaciones

IDE: *Integrated Development Environment* – Entorno de Desarrollo Integrado

M2M: *Machine to machine* – De maquina a maquina

UUF: *Unified User Interface Framework* – Framework de interfaz de usuario unificada

CDMF: *Connected Device Managemenet Framworkd* – Framework de gestion de dispositivo conectado

BIBLIOGRAFÍA

Documentación MQTT: *http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html*

Notación *Swagger*: *https://github.com/swagger-api/swagger-core/wiki/annotations*