UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

PROYECTO FIN DE CARRERA
Ingeniería de Telecomunicación

Diseño, desarrollo e implementación de un protocolo de comunicaciones entre sensores en red y computadores

Luis Pulido García-Duarte

JULIO 2016

Diseño, desarrollo e implementación de un protocolo de comunicaciones entre sensores en red y computadores

AUTOR: Luis Pulido García-Duarte
TUTOR: Francisco de Borja Rodríguez Ortiz

Grupo de Neurocomputación Biológica (GNB)
Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio de 2016

Resumen

En esta memoria se presenta un sistema que permite exponer y presentar los datos capturados por múltiples sensores, y las variables implicadas en el proceso de toma de muestras, para poder ser consumidos por un cliente en red. El envío de las muestras tomadas y las variables implicadas se realiza a un servidor externo y distinto al implicado en la toma de muestras facilitando que los datos sean accesibles desde cualquier punto que tenga acceso a dicha red. Además, permite el envío de la configuración de los parámetros de una prueba a realizar sobre el sensor por parte del cliente en red, creando un protocolo de comunicación entre el cliente y el sensor. Todo esto se realiza mediante una plataforma software que actúa de pasarela entre los sensores y los clientes en red. El correcto funcionamiento del sistema se comprueba mediante pruebas con múltiples sensores, así como con pruebas de escalabilidad.

Una vez se dispone de una muestra capturada por el sensor y acondicionada, el dispositivo que dota de conectividad al sensor procede a generar un paquete HTTP con la información de la muestra para su envío a través de la red. Dicho paquete contiene el valor digital de la muestra en formato JSON que será interpretado por la plataforma para su persistencia en la base de datos. El envío de la muestra se realiza mediante el protocolo HTTP a la plataforma encargada de su recepción. Una vez los datos están almacenados son accesibles para cualquier cliente que se encuentre en red.

La utilización de este sistema permite la creación de un catálogo de sensores y modelos de pruebas que se pueden ejecutar, generando así un inventario de la información y las pruebas ejecutadas en el sistema. Además, permite añadir nuevos sensores al sistema de una manera sencilla. Es una plataforma versátil y escalable, totalmente configurable por parte del usuario, que otorga una interfaz para la configuración inicial de las variables que están presentes en el modelo de análisis que se desea ejecutar. Además, el sistema ofrece la visualización y monitorización de los datos enviados por el sensor, en tiempo real (*online*) y *offline*, en una interfaz amigable y responsiva.

En resumen, esta plataforma facilita el desarrollo de la comunicación con sensores activos a través de una red, deslocalizando el sensor del equipo de investigación. La lectura de la muestra de estos sensores activos varía dependiendo de la configuración que se le aplique. De esta manera, se crea un protocolo para la comunicación entre un cliente del sistema y el sensor bajo el protocolo HTTP. La plataforma permite por tanto que la ubicación del sensor y la monitorización de éste estén en distintos lugares, consiguiendo, por ejemplo, que diferentes grupos de investigación puedan trabajar sobre el mismo espectro de muestras previamente almacenadas por la plataforma.

Palabras clave

sensor, nariz electrónica, protocolo, HTTP, escalabilidad, base de datos, ordenador de placa reducida, API REST, Cassandra, Spring, Django, Java, Python, Cloud, microservicios, Raspberry Pi, SQLite.

Abstract

This M.Sc. Thesis presents a system that exposes the data captured by multiple sensors, as well as the variables involved in the sampling process, to be consumed by a network client. The samples and the related variables are then sent to an external server. This server and the one used for sampling are not the same so that the data is accessible from any other computer in the network. Furthermore, it allows any client in the network to send the configuration parameters of a sensor's test, which creates a communication protocol between the client and the sensor. The functionality explained before is provided by a software platform which behaves like a bridge between the sensors and the users. The correct performance of the system is checked by testing it with multiple sensors, as well as scalability tests.

Once the sampling is done and parsed, the device which connects the sensor to the network generates a HTTP package with the sample information, which is ready to be sent over the network. The package has a JSON format, that is then interpreted by the software platform to persist it in the database. A HTTP protocol is used to send the sample from the sensor to the platform. Once the data has been persisted it is accessible to any client in the network.

This system allows the creation of a catalogue of sensors and test models that can be executed. And so it generates an inventory of the information and the tests that have been executed in the system. Moreover, new sensors can be easily added to the system. Hence it is a versatile and scalable platform, which is completely configurable by the user. It acts as an interface for the initial configuration of the variables in the model that will be executed. Furthermore the system allows real time and offline visualization and monitoring of the data sent by the sensor. The interface developed is user friendly and responsive.

Summing up, this platform eases the communication with active sensors in a network, making the sensor independent from the research team. Depending on the configuration applied to the active sensor, the data obtained from the sampling might vary. Hence, a protocol for the communication between a network client and a sensor is created using the HTTP protocol. The platform allows the monitoring of the sensor to be done from a different location. One of the main advantages of this approach is that several research teams can work on the same data at the same time.

Keywords

sensor, electronic nose, protocol, HTTP, scalability, databases, single board computer, API REST, Cassandra, Spring, Django, Java, Python, Cloud, microservices, Raspberry Pi, SQLite.

Agradecimientos

En primer lugar, me gustaría dar las gracias a mi tutor, Francisco de Borja Rodríguez, por brindarme la oportunidad de realizar este proyecto. También a todos aquellos profesores que han dedicado parte de su tiempo a formarme, tanto a nivel académico como personal.

No puedo olvidarme de todos los amigos que me han apoyado durante mi vida, desde el colegio hasta hoy, por esos buenos ratos y esas horas “muertas” en el chino que tanto hemos aprovechado y disfrutado, en especial a Manu y Berty porque juntos llegamos a la UAM. También acordarme de todas las personas que me han acompañado durante esta etapa de mi vida. Gracias por el apoyo, las risas, los buenos momentos y por ayudarme a ponerme nuevas metas.

Quiero dar las gracias a todos los compañeros de universidad que me han acompañado a lo largo de la carrera, por esos ratos en los laboratorios y esos buenos momentos fuera de la escuela que nos han hecho crecer como personas.

Agradecer a mis excompañeros de trabajo su apoyo, por transmitirme las ganas para acabar este proyecto y ofrecerme su ayuda en todo momento para lo que necesitara. En especial a Borja, por orientarme y aconsejarme en este escrito, y sin duda a Sara por acompañarme y guiarme a lo largo de toda la memoria, por esas críticas y esos enfados que han dado su fruto, sin ti no hubiera sido posible.

También agradecer a mi compañera de viajes todas esas horas en la carretera en las que no me preocupaba de nada más que disfrutar y mejorar. A ti por enseñarme otra visión del mundo, por brillar y enseñarme a disfrutar de otra manera. A Tinta por esos paseos de madrugada que me ayudaban a reflexionar y afrontar los problemas con otra perspectiva y por esas horas de compañía.

Por último me gustaría dar las gracias a mi familia, eternos luchadores. Gracias por inculcarme los valores de esfuerzo, lucha y superación. A mis padres por demostrar que hace falta mucho más para acabar con ellos. A mis hermanas por su valor y su esfuerzo, por enseñarme que todo sacrificio tiene su recompensa. También dar las gracias a los nuevos integrantes de la familia por aportar felicidad a la familia.

Esta memoria va dedicada a todos aquellos que en algún momento dijeron: “A ver si acabas ya el proyecto”, pues bien, aquí está el resultado.

A todos ¡Gracias!

Luis

Índice

Índice de tablas ..... **XII**

Índice de figuras ..... **XII**

1. Introducción **1**

1.1. Marco del proyecto y motivación **3**

1.2. Definición del problema **4**

1.3. Objetivos **5**

1.4. Estructura de la memoria **6**

2. Estado del arte **7**

2.1. Hardware en el IoT **9**

2.1.1. Sensores 9

2.1.1.1. Tipos de sensores básicos 10

2.1.1.2. Narices electrónicas 11

2.1.1.3. Interfaces de acceso a sensores 13

2.1.2. Conectividad 14

2.1.2.1. Estándares de comunicación en el IoT 15

2.1.2.2. Ordenadores de placa reducida 19

2.1.2.3. Placas de desarrollo 21

2.1.2.4. Plataformas *ad hoc* 22

2.1.3. Soluciones comerciales 23

2.2. Comunicación y gestión de datos en el Cloud **25**

2.2.1 Computación en la nube (Cloud) 25

2.2.2. Protocolos de comunicación máquina a máquina 26

2.2.3. Servicios web 30

2.2.3.1. Arquitectura de microservicios 30

2.2.3.2. SOAP 31

2.2.3.3. REST 32

2.2.4. Bases de datos 33

2.2.4.1. Bases de datos relacionales 33

2.2.4.2. Bases de datos no relacionales 34

2.2.5. Soluciones Cloud para el IoT 35

2.3. Soluciones completas **38**

3. Tecnologías utilizadas en el proyecto **41**

3.1. Capa de acceso a datos **43**

3.1.1. Python 43

3.1.1.1. Django 44

3.1.2. Java 44

3.1.2.1. Spring 45

3.1.3. Nginx 45

3.2. Capa de presentación **46**

3.2.1. HTML 46

3.2.2. CSS 46

3.2.2.1. Bootstrap 47

3.2.3. JavaScript 47

3.2.3.1. AngularJS 47
4. Diseño y desarrollo 49
4.1. Diseño 51
4.1.1. Arquitectura general del sistema 51
4.1.1.1. Requisitos de los servicios 53
4.1.1.2. Funcionalidad global 54
4.1.2. Comunicación del sensor 55
4.1.2.1. Funcionalidades del dispositivo 55
4.1.2.2. Tecnologías 56
4.1.3. Servicio de gestión de los resultados de sensores 57
4.1.3.1. Funcionalidad del servicio 57
4.1.3.2. Estructura de la información almacenada 58
4.2.3.3. Tecnologías 58
4.1.4. Servicio de gestión del catálogo de la aplicación 59
4.1.4.1. Funcionalidades del servicio 59
4.1.4.2. Estructura de la información almacenada 60
4.1.4.3. Tecnologías 60
4.1.5. Visualización del sistema 61
4.1.5.1. Interfaz de administración del catálogo 61
4.1.5.2. Interfaz de administración de las pruebas 63
4.2. Desarrollo 65
4.2.1. Dispositivo interfaz entre la nube y los sensores 66
4.2.1.1. Recepción de la configuración mediante API REST 66
4.2.1.1.1. Servicio 67
4.2.1.1.2. Controladores 67
4.2.1.2. Programa de captura de datos y envío de información a la nube 69
4.2.1.2.1. Interacción con el sensor 70
4.2.1.2.2. Envío de datos por protocolo HTTP 71
4.2.2. Servicio de gestión de los resultados de los sensores 73
4.2.2.1. Entidad de la base de datos 75
4.2.2.2. Repositorio 76
4.2.2.3. Controladores 77
4.2.3. Servicio de gestión del catálogo de la aplicación 78
4.2.3.1. Diagrama de clases 79
4.2.3.2. Serializadores 81
4.2.3.3. Controladores 83
4.2.3.4. Enrutador 84
4.2.4. Visualización del sistema 85
4.2.4.1. Interfaz de administración del catálogo 85
4.2.4.2. Interfaz de administración de las pruebas 86
4.2.4.2.1. Módulos de la interfaz 87
4.2.5. Integración de nuevos programas en el sistema 89
5. Pruebas y resultados 91
5.1. Captura de datos mediante circuitos RC 94
5.1.1. Potenciómetro 98
5.1.2. Fotorresistencia 101
5.2. Captura de datos mediante conversor analógico-digital 103
5.2.1. Potenciómetro 104

5.2.2. Sensor de temperatura ... 106
5.2.3. Fotorresistencia ... 108
5.2.4. Sensor de odorantes ... 109
5.3. Prueba de carga ... 113
5.3.1. Servicio de gestión de los resultados ... 115
5.3.2. Servicio de gestión del catálogo ... 116
6. Conclusiones y trabajo futuro ... 117
6.1. Conclusiones ... 119
6.2. Trabajo futuro ... 121
Glosario ... 123
Referencias ... 125
Anexo A. Despliegue de los servicios del sistema ... 129
Anexo B. Código API REST recepción de la configuración ... 133
Anexo C. Código API REST gestión de los resultados de los sensores ... 145
Anexo D. Código API REST gestión del catálogo del sistema ... 159
Anexo E. Código interfaz de administración de las pruebas. ... 175
Anexo F. Códigos de las pruebas ... 203
Anexo G. Presupuesto ... 217
Anexo H. Pliego de condiciones ... 219

Índice de tablas

Tabla 2.1: Tecnologías, velocidad de transmisión, tipo de cable y distancia máxima de Ethernet. 15
Tabla 4.1: Elementos que componen el catálogo de la aplicación. 53
Tabla 4.2: Esquema de la base de datos Cassandra para el servicio gestión de resultados. 76
Tabla 5.1: Resumen de las pruebas realizadas. 93
Tabla 5.2: Conexiones del ADC MCP3008 con los pines GPIO de la Raspberry Pi B+. 103

Índice de figuras

Figura 2.1: Proceso de análisis en una nariz electrónica. 11
Figura 4.1: Arquitectura general del sistema. 52
Figura 4.2: Flujo de creación y ejecución de una prueba. 54
Figura 4.3: Representación del envío de datos a la plataforma mediante una llamada HTTP POST de un programa embebido en el dispositivo interfaz. 55
Figura 4.4: Representación de la recepción de los parámetros de un modelo por parte del dispositivo interfaz gracias a una API REST. También se representa la recepción de la petición para detener una prueba. 56
Figura 4.5: Representación de la recepción de los datos mediante una llamada HTTP POST por parte de la API REST 57
Figura 4.6: Representación del envío de datos y la descarga de ficheros mediante llamadas HTTP GET por parte de la API REST. 58
Figura 4.7: Representación de las peticiones necesarias para realizar el CRUD de los recursos. 59
Figura 4.8: Wireframe de la visualización de la UI para la gestión del catálogo de los recursos del sistema. 62
Figura 4.9: Wireframe de la visualización de la UI para la gestión de los resultados de los sensores. 63
Figura 4.10: Módulos del sistema con las tecnologías y representación de la comunicación entre los módulos. 65
Figura 4.11: Flujo de la comunicación entre los servicios en el lanzamiento de una prueba. 66
Figura 4.12: Flujo de la ejecución de un programa por parte de la API REST. 69
Figura 4.13: Comunicación interna de la API REST entre los controladores y la base de datos. 75
Figura 4.14: Diagrama entidad-relación de la base de datos de catálogo de la aplicación 80
Figura 4.15: Flujo de la creación o detención de una prueba. 83
Figura 4.16: Interfaz gráfica generada por Swagger para la gestión del catálogo del sistema. 86

Figura 4.17: Página principal de la interfaz gráfica de usuario para la gestión de las pruebas. 87
Figura 5.1: Circuito RC 94
Figura 5.2: Creación de un modelo en la interfaz de usuario destinada a la gestión del catálogo. 95
Figura 5.3: Creación del parámetro "sensor" en la UI destinada a la gestión del catálogo 96
Figura 5.4: Creación del parámetro "tiempo de envío" en la UI destinada a la gestión del catálogo. 96
Figura 5.5: Diseño e implementación de un circuito RC con un potenciómetro. 98
Figura 5.6: Creación del dispositivo "Raspberry Pi" en la UI destinada a la gestión del catálogo. 98
Figura 5.7: Creación del parámetro "tiempo de envío" en la UI destinada a la gestión de catálogo. 99
Figura 5.8: Elección del dispositivo para ejecutar una prueba en la UI para gestión de las pruebas. 99
Figura 5.9: Elección del modelo a ejecutar en una prueba mediante la UI destinada a la gestión de las pruebas. 100
Figura 5.10: Elección de los parámetros del modelo a ejecutar en una prueba mediante la UI destinada a la gestión de las pruebas. 100
Figura 5.11: Visualización de los resultados de una prueba de un potenciómetro en un circuito RC. 101
Figura 5.12: Diseño e implementación de un circuito RC con un sensor LDR. 101
Figura 5.13: Visualización de los resultados de una prueba de un sensor LDR en un circuito RC. 102
Figura 5.14: Diseño e implementación de un potenciómetro con un ADC modelo MCP3008. 104
Figura 5.15: Asignación de los parámetros a distintos modelos mediante la UI de gestión del catálogo. 105
Figura 5.16: Visualización de los resultados de una prueba de un potenciómetro con un ADC en la UI de gestión de las pruebas. 105
Figura 5.17: Diseño e implementación de un sensor LM335 con un ADC modelo MCP3008. 106
Figura 5.18: Creación mediante la UI de un parámetro de tipo sensor para representar el valor de una variable. 106
Figura 5.19: Visualización de los resultados de una prueba de un sensor LM335 con un ADC en la UI destinada a la gestión de las pruebas. 107
Figura 5.20: Visualización de la variable que representa el valor de un sensor LM335 en grados centígrados. 107
Figura 5.21: Diseño e implementación de un sensor LDR con un ADC modelo MCP3008 y un LED conectado a un pin GPIO. 108
Figura 5.22: Visualización de la salida de un sensor LDR y de la variable que representa si un LED está encendido. 109
Figura 5.23: Circuito adaptado para la conversión de la señal de salida de un sensor TGS2600 mediante un ADC. 110
Figura 5.24: Implementación del circuito diseñado en la Figura 4.23. 110
Figura 5.25: Visualización de las variables *slope, intercept, Rvalue, Pvalue* y *stderr* en una prueba con un sensor TGS2600 a una temperatura ambiente de 31ºC. 111

Figura 5.26: Visualización de la respuesta de un sensor TGS2600 a una temperatura ambiente de 31ºC. 112
Figura 5.27: Visualización de la temperatura aplicada a un sensor TGS2600 ajustada por regresión lineal. 112
Figura 5.28: Interfaz gráfica de JMeter. Configuración del número de peticiones a realizar.. 114
Figura 5.29: Petición POST a la ruta "/api/save/results" desde la interfaz gráfica de Jmeter. 114
Figura 5.30: Prueba de carga del servicio de gestión de los resultados de los sensores. ..115
Figura 5.31: Prueba de carga del servicio de gestión del catálogo del sistema. 116

1. Introducción

1.1. Marco del proyecto y motivación

Las nuevas tecnologías están facilitando una revolución tecnológica: la revolución de la sensorización total [7]. La industria de los sensores ha emergido en las últimas décadas y ha experimentado un desarrollo espectacular. Esta revolución deriva de la incorporación masiva de sensores y dispositivos electrónicos repartidos por el entorno capaces de procesar enormes cantidades de datos. El descenso de precios, el bajo consumo y la capacidad de conexión han dado paso a las redes de sensores distribuidos.

La industria tecnológica contribuye a esta sensorización revolucionaria y cada vez son más las empresas y grupos de investigación que trabajan en el desarrollo de este tipo de dispositivos. En este sentido, existen numerosos sistemas implantados en diferentes sectores que, bajo este paradigma, permiten por ejemplo la detección de fugas en gaseoductos [56] por parte de compañías petroquímicas, el estudio del hábitat de ciertas especies [43] o la detección y control de tropas enemigas en la industria militar, entre otras muchas aplicaciones.

Esta sensorización masiva se engloba dentro del paradigma planteado por Mark Weiser: la computación ubicua [64]. La computación ubicua se basa en repartir la capacidad computacional a todo el entorno mediante la distribución de pequeños dispositivos. Una red de diminutos dispositivos distribuidos es capaz de realizar pequeñas tareas y filtrar toda la información para enviar los datos a una estación central a través de la red. De esta manera, cualquier elemento se podría dotar de inteligencia y crear una red en la que estén interconectados. Este fenómeno es conocido hoy en día como el Internet de las cosas (en inglés, Internet of Things, IoT) [16][58].

El término IoT es un concepto que denota la conexión digital avanzada de dispositivos, sistemas y servicios con Internet que va más allá del tradicional concepto de máquina a máquina (M2M) y cubre una amplia variedad de protocolos, dominios y aplicaciones. Se calcula que todo ser humano está rodeado de entre 1000 y 5000 objetos [62] y algunas empresas predicen que en 2020 habrá en el mundo en torno a 30 millones de dispositivos conectados a Internet [22][39], confirmando así la revolución tecnológica promovida por el IoT.

Basándose en esta idea Kris Pister, en el año 2002, acuñó el término "*Smartdust*" [17], que sintetizaba en una única palabra los conceptos de sensores diminutos distribuidos por grandes áreas con comunicación sin hilos. Una red inalámbrica de sensores consta de nodos sensores distribuidos espacialmente. Cada nodo sensor es capaz de realizar de forma independiente algunas tareas de detección, procesamiento y análisis. Además, los nodos sensores se pueden comunicar entre sí con el fin de enviar su información detectada a una central de procesamiento o realizar algún tipo de coordinación local.

Por tanto, es importante tener un sistema versátil que facilite la comunicación con los sensores, permitiendo la fusión de datos entre ellos. Esta fusión de datos permite obtener nueva información ya que la información cruzada muestra nuevas dimensiones de los datos, creando así una fusión sensorial. Además, estos datos deben ser almacenados para su monitorización y visualización, facilitando así el estudio de los sensores.

1.2. Definición del problema

La muestra capturada por un sensor ha de ser analizada para estudiar los datos que se poseen de ésta o actuar en consecuencia. Sin embargo, no todos los sensores poseen la capacidad de analizar o actuar sobre los datos que detectan, necesitando enviar la información a un dispositivo para que realice dicha función.

La ubicación de los sensores suele ser estática, y en algunos casos, de difícil acceso a los datos tomados, como por ejemplo en gaseoductos. Igualmente, en grandes áreas dispersas donde se encuentran muchos sensores repartidos, adquiere especial importancia el acceso remoto a los sensores, como en el caso de sensores distribuidos por el entorno para el estudio del hábitat de ciertas especies.

Además, los sensores por sí mismos no pueden enviar las mediciones registradas a una plataforma externa, por lo que en ocasiones se requiere algún tipo de dispositivo intermedio que proporcione este servicio, consiguiendo que exista una comunicación centralizada entre los sensores y los clientes de la red. El envío de la información capturada por los sensores a un controlador central permite por tanto que se pueda monitorizar la medición tomada y analizar el comportamiento de las muestras registradas.

Por otra parte, los sensores necesitan una configuración inicial para comenzar a tomar muestras. Esta configuración depende del tipo de sensor, llegando a ser crucial en la medición de la variable de algunos sensores. Según la configuración del sensor existen sensores pasivos, si la configuración es estática, y sensores activos, si es dinámica. En la configuración dinámica se cambian ciertos parámetros según la condición de la muestra tomada o del entorno que le rodea, pudiendo incluso adaptarse según la medida tomada por otro sensor.

La visualización de las muestras tomadas por los sensores permite poder fusionar los datos de distintos sensores de una manera sencilla, facilitando el análisis de la fusión sensorial. Así, se pueden distinguir y evaluar nueva información proveniente de las muestras rápidamente.

La monitorización y observación de los datos de una manera versátil implica la necesidad de persistir la información. Adquiere especial importancia el envío de los datos a una plataforma externa que facilite el estudio y análisis de los datos de forma remota. Los datos recogidos por los sensores deben ser accesibles para ser examinados y utilizados posteriormente en diversos estudios. Estas características son fundamentales en análisis complejos con sensores sensibles y dinámicos cuyo comportamiento varía según se configuren ciertos parámetros involucrados en la toma de muestras. Además, como primera fase de análisis es útil la visualización y monitorización de los datos, y para conseguir mayor rendimiento y agilidad en el estudio de la información, esta visualización debe ser en tiempo real.

1.3. Objetivos

El presente Proyecto Fin de Carrera tiene como principal objetivo diseñar una solución de acceso remoto a múltiples sensores y en tiempo real. Al proporcionar acceso remoto se consigue que la ubicación del sensor y del usuario interesado en la información extraída por el sensor sean independientes, y si además este acceso es en tiempo real, se agiliza el proceso de análisis y estudio de las muestras del sensor.

Para que el sensor sea accesible de forma remota es necesario un dispositivo capaz de conectarse a la red. Al disponer de una interfaz capaz de realizar la conexión, se tiene acceso al envío de los datos de los sensores a través de la red y se posibilita la recepción de la información para la configuración del sensor. Para ello, se estudiarán distintas formas de comunicación del sensor con la red, y en la medida de lo posible, se escogerá la solución más universal que se ajuste a la resolución del problema a través de protocolos de comunicación estándar.

El acceso a los datos capturados por el sensor también se ha de realizar a través de un dispositivo. Este dispositivo ha de permitir la comunicación y el control de todo tipo de sensores. Para ello se estudiarán los sensores más representativos y se diseñará una solución que permita hacer de interfaz con el mayor número de opciones posible. Este dispositivo otorgará la capacidad de tratar y analizar los datos de las muestras tomadas.

Debe existir una plataforma capaz de realizar tanto la conexión con el dispositivo encargado de enviar los datos del sensor como la recepción y almacenamiento de los datos para su análisis y monitorización. Los datos estarán disponibles para su utilización por cualquier cliente de la red. Se realizará un estudio de las distintas posibilidades de persistencia de datos eligiendo la solución que otorgue mayor rendimiento y escalabilidad para un número elevado de sensores.

Para completar el sistema se diseñarán y desarrollarán todas aquellas herramientas que faciliten el acceso a los datos, así como la plataforma necesaria para mantener la información estructurada y accesible. Las herramientas dispondrán de interfaces de usuario que simplifiquen el acceso a la información al mismo tiempo que permitan la visualización gráfica de los datos en tiempo real y su descarga rápida y sencilla.

En resumen, para poder cumplir con el objetivo principal de acceder en remoto a múltiples sensores es necesario cumplir también los siguientes objetivos:

- Añadir un dispositivo que se comunique con el sensor y envíe los datos por la red.
- Crear una plataforma para el almacenamiento de los datos de los sensores.
- Desarrollar las herramientas necesarias para catalogar la información y monitorizar los datos de los sensores, permitiendo además la incorporación de nuevos sensores.

El sistema diseñado ha de ser versátil y reutilizable con el fin de abaratar costes y facilitar la incorporación de nuevos modelos de análisis, sensores u otras características. Proporcionará así mismo la infraestructura necesaria para reutilizar ciertas soluciones según el problema planteado con distintos tipos de sensores. Estas particularidades permitirán por tanto que el sistema sea utilizado en el ámbito de la investigación.

1.4. Estructura de la memoria

Una vez expuesto el contexto en el que se enmarca el proyecto, explicados los problemas latentes en este marco y planteados los objetivos, el presente trabajo se estructura en los siguientes capítulos:

- **Estado del arte**: Este capítulo se divide en distintos bloques. En el primero de ellos se analiza el hardware empleado en el IoT, esto es, los sensores y la comunicación con ellos, y las posibilidades de conectividad existentes. En el segundo, se analiza el *Cloud*, la comunicación con él y la gestión de datos y arquitecturas empleadas. El capítulo finaliza con el análisis de soluciones comerciales relacionadas con el contexto analizado.

- **Diseño y desarrollo**: A lo largo de este capítulo se analiza el diseño necesario para cumplir con los objetivos deseados, y a continuación, se explica el desarrollo llevado a cabo cumpliendo el diseño especificado.

- **Tecnologías utilizadas**: En este capítulo se recogen las tecnologías empleadas para la creación de los servicios ofrecidos en el *Cloud* y las utilizadas para la creación de la interfaz gráfica de usuario para la monitorización y creación de las pruebas.

- **Pruebas y resultados**: El análisis del correcto funcionamiento del sistema se lleva a cabo mediante diversas pruebas expuestas en este capítulo, analizando los resultados y comprobando que los objetivos del proyecto se cumplen.

- **Conclusiones y trabajo futuro**: El último capítulo de la memoria expone las conclusiones extraídas tras la realización del presente trabajo. También se explican los posibles trabajos futuros que dan continuidad al proyecto.

2. Estado del arte

Este capítulo comienza con el análisis del hardware utilizado en el Internet de las Cosas, necesario para acceder a los datos de los sensores al mismo tiempo que los dotan de conectividad, convirtiéndolos en sensores inteligentes.

También, se estudia el fenómeno llamado *Cloud*, el cual permite almacenar la información en plataformas en Internet. Para ello, se exponen los distintos protocolos y estándares de comunicación con la “nube”, así como la estructura de ésta.

El capítulo finaliza con el análisis de grandes compañías que han invertido recursos en crear sistemas completos, abarcando desde el hardware hasta los servicios *online*, para ofrecer productos destinados al IoT.

2.1. Hardware en el IoT

En este apartado se explican las tecnologías y dispositivos encargados de la toma de muestras y el envío de ellas a la plataforma. Para ello se emplean sensores capaces de transformar la magnitud a tomar en una variable conocida. Además, es necesario disponer de una interfaz dedicada a acceder al dato capturado por el sensor.

Por otro lado, para poder enviar el dato es necesario utilizar un dispositivo con conectividad a Internet que sea capaz de mandar la información a la plataforma externa. Este dispositivo también tiene como finalidad el poder comunicarse con la interfaz encargada de recoger el dato para enviarlo por la red.

Para concluir, se expondrán distintas empresas que ofrecen soluciones comerciales que aúnan todas estas tecnologías en su producto.

2.1.1. Sensores

Un sensor es un objeto capaz de detectar magnitudes físicas o químicas, llamadas variables de instrumentación, y transformarlas en otra magnitud, normalmente una variable eléctrica, para que se pueda cuantificar y manipular. Un sensor es un dispositivo que convierte una forma de energía en otra, aprovechando una de sus propiedades con el fin de adaptar la señal que mide para que la pueda interpretar otro dispositivo.

La variación de la magnitud que se obtiene en el sensor es un valor directo de la variable a medir, pero los sistemas de control no trabajan con estas señales, sino con tensión o intensidad por lo que, con ayuda de un transductor, se asociada a una variación de voltaje o intensidad. El transductor, por tanto, suele incluir al sensor. Normalmente estos dispositivos se realizan con componentes pasivos que varían su magnitud en función de alguna variable y otros componentes activos.

Los sensores son utilizados con multitud de fines y en diversas industrias: automotriz, robótica, industria aeroespacial, medicina, industria de manufactura, alimentaria, medioambiental, etc.

En cuanto a las variables de instrumentación, éstas pueden ser por ejemplo: intensidad lumínica, temperatura, distancia, aceleración, inclinación, presión, desplazamiento, fuerza, torsión, humedad, movimiento, pH, concentraciones químicas, etc. Según estas variables existen distintos tipos de sensores. Estos sensores se pueden clasificar por grupos, siendo los más importantes los sensores de luminosidad, de temperatura, de proximidad (ultrasonidos o infrarrojos), de velocidad y aceleración, de humedad, de presión y fuerza, sensores de turbidez y sensores de movimiento.

2.1.1.1. Tipos de sensores básicos

Un tipo de clasificación de sensores muy básico es diferenciarlos entre pasivos o activos. Los sensores activos generan la señal de salida sin la necesidad de una fuente de alimentación externa, mientras que los pasivos sí requieren de esta alimentación para poder efectuar su función.

Sin embargo, lo usual es clasificarlos según la magnitud física a medir. A continuación se muestran los sensores más representativos:

- **Sensores de luz**: Estos sensores se caracterizan por transformar la luminosidad a una variable eléctrica. Existen distintos tipos:
    - Fotorresistencias (LDR): La resistencia varía con la intensidad luminosa que recibe. Los fotones activan los electrones del material semiconductor, disminuyendo la resistencia, por tanto, la resistencia varía negativamente con la luminosidad. Están fabricados con semiconductores en zigzag sobre sustrato cerámico. Estos sensores serán utilizados en este proyecto.
    - Fotodiodos: Es un diodo semiconductor, construido con una unión PN, que está expuesto a la luz a través de una cobertura cristalina y es especialmente sensible a la incidencia de la luz visible o infrarroja. La construcción está orientada a lograr que esta sensibilidad sea máxima.
    - Fototransistores: Están compuestos por el mismo material semiconductor que un transistor normal. Se diferencian en la cápsula, que posee una ventana o es totalmente transparente, para dejar que la luz incida sobre la pastilla semiconductora y produzca el efecto fotoeléctrico.
- **Sensores de temperatura**: Estos sensores presentan salidas lineales respecto a la temperatura y son estables y rápidas. Estas características conllevan a la utilización de estos sensores en las pruebas del proyecto y demostrar así su funcionamiento. Existen distintos tipos:
    - Termistor: Es un resistor cuyo valor varía en función de la temperatura. Existen dos clases de termistores: NTC (Coeficiente de Temperatura Negativo), que es una resistencia variable cuyo valor se decrementa a medida que aumenta la temperatura; y PTC (Coeficiente de Temperatura Positivo), cuyo valor de resistencia eléctrica aumenta cuando aumenta la temperatura.
    - RTD: Estos sensores consisten en una fina película de resistencia variable con la temperatura y se utilizan para medir temperaturas por contacto o inmersión, y en especial para un rango de temperaturas elevadas. El funcionamiento se basa en el hecho de que en un metal, cuando sube la temperatura, aumenta la resistencia eléctrica.
    - Termocuplas: Está formado por la unión de dos piezas de metales diferentes. La unión de los metales genera un voltaje muy pequeño, que varía con la temperatura. Su valor está en el orden de los milivoltios y aumenta en proporción con la temperatura.
- **Sensores de humedad**: Existen sensores de humedad capacitivos y resistivos, más simples, y algunos integrados con diferentes niveles de complejidad y prestaciones.

- **Sensores de proximidad**: Pueden ser de tipo lineal (detectores de desplazamiento) o de tipo conmutador (la conmutación entre dos estados indica una posición particular). Hay dos tipos de detectores de proximidad muy utilizados en la industria: inductivos, se basan en el fenómeno de amortiguamiento que se produce en un campo magnético a causa de las corrientes inducidas (corrientes de Foucault) en materiales situados en las cercanías; y capacitivos, detectan las variaciones de la capacidad parásita que se origina entre el detector propiamente dicho y el objeto cuya distancia se desea medir.
- **Sensores infrarrojos**: Es un dispositivo electrónico capaz de medir la radiación electromagnética infrarroja de los cuerpos en su campo de visión. Todos los cuerpos reflejan una cierta cantidad de radiación.
- **Sensores de presión**: Existen multitud de sensores de presión, la mayoría están orientados a medir la presión de un fluido sobre una membrana.
- **Sensores de fuerza**: La aplicación de una fuerza al área activa de detección del sensor se traduce en un cambio en la resistencia eléctrica del elemento sensor en función inversamente proporcional a la fuerza aplicada.
- **Sensores químicos**: La función de estos sensores es dar lugar a una magnitud física, la cual pueda ser capturada por el hardware de adquisición. Dicha magnitud debe reflejar en menor o mayor medida la exposición de los sensores a la muestra olorosa. Están compuestos por un receptor que se encarga de reconocer selectivamente a la especie química a detectar y un transductor que se encarga de convertir la señal química a eléctrica. Estos sensores se emplean en narices electrónicas.

2.1.1.2. Narices electrónicas

Una nariz electrónica es un sistema electrónico con capacidad analítica cuya finalidad es detectar los compuestos orgánicos volátiles que forman parte de una muestra olorosa, pudiendo de esa forma, reconocerla o discriminarla dentro de un conjunto de sustancias olorosas. Su objetivo es relacionar el aroma que se percibe con una respuesta que, tras ser almacenada en la memoria, servirá como modelo en sucesivos análisis. Las narices electrónicas tratan de imitar, de una manera simplificada, el principio del sistema olfativo de los mamíferos.

De esta manera, una nariz electrónica no es más que un sensor químico, o conjunto de sensores, especializado en la captación de odorantes junto con un procesador que analiza la señal recibida a través de una programa quimiométrico de reconocimiento de olores individuales o complejos.

El proceso de análisis de un odorante está representado en la Figura 2.1 y detallado a continuación:

Figura 2.1: Proceso de análisis en una nariz electrónica.

Una nariz electrónica, por tanto, está compuesta de los siguientes bloques bien definidos:

- El primer bloques es un bloque de transducción, cuya finalidad es transformar el odorante a una señal eléctrica analógica y cuyo elemento fundamental es un conjunto de sensores químicos o de gas. Generalmente no son específicos y reaccionan ante un espectro relativamente grande de compuestos, es decir, no han sido diseñados para reconocer ningún compuesto concreto sino que, cuanto mayor sea el número de compuestos ante los que pueden reaccionar, teóricamente mayor es el número de ámbitos de aplicación. Este conjunto suele estar formado por un número determinado de sensores. El número de sensores del conjunto, así como la tecnología empleada para implementar los sensores, influye de forma importante en las prestaciones de la aplicación.
- Un segundo bloque de adquisición de señal y conversión a un formato digital, por norma general un conversor analógico-digital, así como componentes electrónicos para el acondicionamiento de la señal analógica entregada por el conjunto y su tratamiento por el conversor.
- Un último bloque de procesado que es el encargado de analizar la señal recibida y determinar el odorante. Por norma general, el procesado de esta señal se encuentra dentro del ámbito del Aprendizaje Automático (*Machine Learning*). Cuando se trata de narices electrónicas, este aprendizaje es denominado Aprendizaje Olfativo Automático.

Los sensores involucrados en la adquisición de los odorantes y la transformación de las concentraciones químicas en señales eléctricas se pueden clasificar de la siguiente manera:

- Quimioresistores. Se basan en la medida de los cambios de conductividad del material al entrar en contacto con un gas.
- Quimiocapacitores. El sensor es un condensador cuyo dieléctrico es un polímero poroso que varía su capacitancia cuando las moléculas gaseosas a analizar son absorbidas.
- Quimiosensores potenciométricos. Varían su umbral de disparo ante la presencia de ciertos gases.
- Quimiosensores gravimétricos. Miden la variación de la frecuencia de resonancia del conjunto en función de los gases que se adhieren a su superficie o bien la variación de la velocidad de propagación del sonido en un sustrato.
- Quimiosensores térmicos “calorimétricos”. Fundamentalmente de tipo piroeléctrico. Son adecuados para detectar gases combustibles.
- Quimiosensores amperimétricos. Miden la corriente eléctrica que pasa por una celda electroquímica. Su coste es reducido y su sensibilidad elevada.

En este proyecto se usarán narices electrónicas para demostrar el funcionamiento del sistema con sensores activos. Los sensores utilizados serán el modelo TGS2600 [57].

2.1.1.3. Interfaces de acceso a sensores

El valor de la muestra tomada por el sensor es accesible gracias al transductor, encargado de transformar y acondicionar el dato tomado en un valor de intensidad o voltaje. De esta manera, se puede acondicionar la señal para que un dispositivo pueda leer el resultado. Este valor es analógico, por lo que debe ser convertido a un valor digital capaz de ser entendido por el procesador.

La toma de la muestra se puede realizar directamente del pin de salida del sensor o mediante algún protocolo de comunicación con él, dependiendo del tipo de sensor.

Para establecer la comunicación directa con el sensor existen dispositivos que actúan de interfaz y son capaces de realizar la lectura de la señal analógica directamente, por un pin de E/S de propósito general (GPIO), y convertirla a un formato digital, de tal forma que el dato pueda ser manipulado en este dispositivo. Además, la comunicación a través de estos GPIO permite la configuración de los parámetros del sensor, tales como la alimentación. También es posible realizar la conversión analógico-digital fuera de este dispositivo. De este modo, se puede recoger la salida del conversor directamente por el dispositivo y manipular el dato.

Por otro lado, existen sensores con un chip encargado de acondicionar la señal y convertirla a digital. Esta señal se transmite por un bus de datos siguiendo algún protocolo de envío de datos. Estos protocolos se basan en multiplexación de división temporal (*Serial*) y los datos son enviados bit a bit y se reconstruyen por medio de registros o rutinas. El bus de datos está formado por pocos conductores y su ancho de banda depende de la frecuencia.

Existen distintos buses de datos para comunicar un controlador y un periférico, en este caso el sensor. Entre los protocolos y buses más usados con sensores caben destacar los siguientes:

- ***Universal Asynchronous Receiver-Transmitter*** **(UART)** [60]. El Transmisor-Receptor Asíncrono Universal, es una interfaz que controla los puertos y dispositivos serie.
  El canal de comunicación es una sola vía, en el que un extremo toma bytes de datos y transmite los bits individuales de forma secuencial y el otro extremo reensambla los bits en bytes completos. Los datos transferidos son bytes de 8 bits. Cada byte se envía en el cable de bits por bits, precedido por un bit de inicio, seguido por un bit de paridad opcional y uno o dos bits de parada. Por lo tanto, un byte de información está representado de 10 a 12 bits.
  Las velocidades más comunes de transmisión de los bits son 9600, 19200, 38400, 115200 baudios[1]. Por tanto, para realizar la comunicación es necesario saber la velocidad de transmisión, el tipo de paridad, el número de bits de parada que espera y los niveles de tensión utilizados para designar a estado alto y bajo.

[1] Representa el número de símbolos por segundo en un medio de transmisión digital. Cada símbolo puede codificar 1 o más bits, dependiendo del esquema de modulación.

- ***Serial Peripheral Interface* (SPI)** [21]. El bus de Interfaz de Periféricos Serie es un estándar de comunicaciones usado principalmente para la transferencia de información entre circuitos integrados en equipos electrónicos. El bus SPI es un estándar para controlar casi cualquier dispositivo electrónico digital que acepte un flujo de bits serie regulado por un reloj.
  El SPI es un protocolo síncrono que trabaja en modo full-duplex para recibir y transmitir información, permitiendo que varios dispositivos puedan comunicarse entre sí al mismo tiempo utilizando canales o líneas diferentes en el mismo cable. Al ser un protocolo síncrono el sistema cuenta con una línea adicional a la de datos encargada de llevar el proceso de sincronismo.
  Incluye una línea de reloj, dato entrante, dato saliente y un pin de *chip select,* que conecta o desconecta la operación del dispositivo con el que desea comunicarse. De esta forma, este estándar permite multiplexar las líneas de reloj.
  Dentro de este protocolo se define un maestro, que será aquel dispositivo encargado de transmitir información a sus esclavos. Los esclavos serán aquellos dispositivos que se encarguen de recibir y enviar información al maestro. El maestro envía la señal de reloj, y tras cada pulso de reloj envía un bit al esclavo y recibe un bit de éste.

- ***Inter-Integrated Circuit* (I^2C)** [20]. El bus I^2C es un estándar que facilita la comunicación entre microcontroladores, memorias y otros dispositivos con cierto nivel de "inteligencia"; sólo requiere de dos líneas de señal y un común o masa. I^2C permite el intercambio de información entre muchos dispositivos a una velocidad aceptable.
  El I²C está diseñado como un bus maestro-esclavo. La transferencia de datos es siempre inicializada por un maestro. Es posible tener varios maestros (Multimaster-Mode). En el modo multimaestro se pueden comunicar dos maestros entre ellos mismos, de modo que uno de ellos trabaja como esclavo.
  La comunicación de datos del bus I^2C es en serie y sincrónica. Una de las señales del bus marca el tiempo (pulsos de reloj) y la otra se utiliza para intercambiar datos. El maestro y el esclavo envían datos por el mismo cable que es controlado por el maestro creando la señal de reloj.
  No se utiliza selección de esclavo, sino direccionamiento. El primer byte enviado por el maestro se forma de 7 bits para la dirección y un bit de lectura/escritura, indicando si el próximo byte vendrá desde el maestro o el esclavo. Tras cada byte recibido se envía una confirmación con el noveno pulso de reloj. Si el maestro quiere recibir datos sólo genera pulsos de reloj. El esclavo tiene que cuidar que el próximo bit esté listo cuando la señal de reloj es dada.

2.1.2. Conectividad

El acceso a los datos de los sensores se ha de realizar mediante un dispositivo que dote de conectividad a los sensores y envíe el dato a través de Internet. Además, es necesario utilizar algún estándar para la comunicación y crear un protocolo entre los distintos

dispositivos. Por tanto, a lo largo de este apartado se exponen los distintos estándares existentes para la comunicación entre dispositivos, así como distintos tipos de dispositivos que pueden ser utilizados para dotar de conectividad a los sensores.

2.1.2.1. Estándares de comunicación en el IoT

A continuación se detallan los principales estándares de comunicación empleados en el Internet de las Cosas, tanto cableados como inalámbricos. Estos protocolos son: Ethernet, WiFi, BlueTooth, ZigBee, LoRaWAN, SIGFOX.

Ethernet

Ethernet es un estándar de redes de área local para computadores con acceso al medio por detección de la onda portadora y con detección de colisiones (CSMA/CD). Mediante CSMA/CD, es posible que los dispositivos escuchen la red para determinar si el canal y los recursos se encuentran libres para realizar la conexión sin colisionar con otros paquetes, ya que todos los dispositivos pueden transmitir paquetes en cualquier momento. Ethernet define las características de cableado y señalización de nivel físico y los formatos de tramas de datos del nivel de enlace de datos del modelo OSI.

Ethernet se tomó como base para la redacción del estándar internacional IEEE 802.3 [10][29], siendo usualmente tomados como sinónimos. Se diferencian en uno de los campos de la trama de datos. Sin embargo, las tramas Ethernet e IEEE 802.3 pueden coexistir en la misma red.

Los datos que se envían o reciben mediante este estándar están fragmentados en paquetes y se envían a través de un método denominado “Conmutación de paquetes”. Un paquete está formado por la cabecera, dirección del dispositivo en la red a quién va destinado y qué dispositivo de la red lo está enviando. Además contiene datos de control y otras informaciones relativas al mismo como la cantidad de datos que transporta. Los paquetes se envían a todos los dispositivos que conforman la red y cada dispositivo se encarga de recoger los paquetes que van destinado a ellos.

Dentro de Ethernet existen distintas tecnologías que ofrecen distintas características tales como la velocidad de transmisión, la distancia máxima y el tipo de cable, como queda reflejado en la Tabla 2.1.

Tabla 2.1: Tecnologías, velocidad de transmisión, tipo de cable y distancia máxima de Ethernet.

Tecnología	Velocidad de transmisión	Tipo de cable	Distancia máxima
10Base2	10 Mbit/s	Coaxial	185 m
10BaseT	10 Mbit/s	Par Trenzado	100 m
10BaseF	10 Mbit/s	Fibra óptica	2000 m
100BaseT4	100 Mbit/s	Par Trenzado (categoría 3UTP)	100 m

100BaseTX	100 Mbit/s	Par Trenzado (categoría 5UTP)	100 m
100BaseFX	100 Mbit/s	Fibra óptica	2000 m
1000BaseT	1000 Mbit/s	4 pares trenzado (categoría 5e ó 6UTP)	100 m
1000BaseSX	1000 Mbit/s	Fibra óptica (multimodo)	550 m
1000BaseLX	1000 Mbit/s	Fibra óptica (monomodo)	5000 m

Wi-Fi

El Wi-Fi es un mecanismo de conexión de dispositivos electrónicos de forma inalámbrica. Los dispositivos habilitados con Wi-Fi pueden conectarse a internet a través de un punto de acceso de red inalámbrica. Wi-Fi es una marca de la Wi-Fi Alliance, la organización comercial que adopta, prueba y certifica que los equipos cumplen con los estándares 802.11 [4] relacionados a redes inalámbricas de área local.

La norma IEEE 802.11 fue diseñada para sustituir el equivalente a las capas físicas y MAC de la norma 802.3 (Ethernet). Esto quiere decir que en lo único que se diferencia una red wifi de una red Ethernet es en cómo se transmiten las tramas o paquetes de datos; el resto es idéntico. Por tanto, una red local inalámbrica 802.11 es completamente compatible con todos los servicios de las redes locales (LAN) de cable Ethernet.

Los estándares IEEE 802.11b, IEEE 802.11g/e, IEEE 802.11n trabajan en la banda de 2,4 GHz, con una velocidad de hasta 11 Mbit/s, 54 Mbit/s y 300 Mbit/s, respectivamente. El estándar IEEE 802.11ac, conocido como WIFI 5, opera en la banda de 5 GHz. Su alcance es algo menor que el de los estándares que trabajan a 2,4 GHz (aproximadamente un 10%), debido a que la frecuencia es mayor, pero esta banda está menos saturada dado que hay menos tecnologías usándola. El estándar mejora las tasas de transferencia hasta 433 Mbit/s, consiguiendo teóricamente tasas de 1.3 Gbit/s empleando 3 antenas.

Dentro de los dispositivos wifi se diferencian dos grupos: los dispositivos de distribución o de red y los dispositivos terminales. Los dispositivos terminales son tarjetas receptoras para conectarse vía wifi a un distribuidor. Dentro de los distribuidores destacan los puntos de acceso, permiten conectar dispositivos de forma inalámbrica a una red existente; los repetidores inalámbricos, equipos que se utilizan para extender la cobertura de una red inalámbrica existente para ampliar la cobertura; los enrutadores inalámbricos, dispositivos encargados de interconectar redes e incluyen un punto de acceso. Dicho punto de acceso tiene un alcance de unos veinte metros en interiores, llegando a unos 100-150 metros en exteriores.

BlueTooth

BlueTooth [18] es una especificación industrial para Redes Inalámbricas de Área Personal (*Wireless Personal Area Network,* WPAN) que posibilita la transmisión de voz y datos entre

diferentes dispositivos mediante un enlace por radiofrecuencia en la banda de los 2,4 GHz. Facilita las comunicaciones entre equipos móviles. Ofrece la posibilidad de crear pequeñas redes inalámbricas y facilitar la sincronización de datos entre equipos personales.

Los dispositivos que incorporan este protocolo pueden comunicarse entre sí cuando se encuentran dentro de su alcance. Estos dispositivos se clasifican como "Clase 1", "Clase 2" o "Clase 3" en referencia a su potencia de transmisión de 100 mW, 2.5 mW, 1 mW respectivamente, con un alcance de 100, 10 y 1 metros.

Existen distintas versiones de BlueTooth, siendo la actual la 4.0. Esta versión tiene retrocompatibilidad con las versiones anteriores. El BlueTooth de baja energía (BLE) es un subconjunto de Bluetooth v4.0 con una pila de protocolo completamente nueva para desarrollar rápidamente enlaces sencillos. Está dirigido a aplicaciones de muy baja potencia. Esta especificación ofrece una velocidad de emisión y transferencia de datos de 32 Mbit/s

ZigBee

ZigBee es el nombre de la especificación de un conjunto de protocolos de alto nivel de comunicación inalámbrica para su utilización con radiodifusión digital de bajo consumo, basada en el estándar IEEE 802.15.4 [4] de redes WPAN. Su objetivo son las aplicaciones que requieren comunicaciones seguras con baja tasa de envío de datos y maximización de la vida útil de sus baterías, siendo su principal objetivo ahorrar energía. Es una tecnología de fácil integración y permite tres topologías: estrella, el coordinador se sitúa en el centro; árbol, el coordinador será la raíz del árbol; malla, al menos uno de los nodos tendrá más de dos conexiones. Esta última permite que si un nodo del camino falla y se cae, pueda seguir la comunicación entre todos los demás nodos debido a que se rehacen todos los caminos. La gestión de los caminos es tarea del coordinador.

ZigBee se basa en el nivel físico y el control de acceso al medio (MAC) definidos en la versión de 2003 del estándar IEEE 802.15.4.

ZigBee utiliza la banda ISM, reservada internacionalmente para uso no comercial de radiofrecuencia electromagnética para usos industriales, científicos y médicos, en concreto, 868 MHz en Europa, 915 en Estados Unidos y 2,4 GHz en todo el mundo. Generalmente el diseño de los dispositivos está centrada en la banda de 2,4 GHz, por ser libre en todo el mundo. El desarrollo de la tecnología se centra en la sencillez y el bajo costo. Un nodo ZigBee completo requiere en teoría cerca del 10% del hardware de un nodo Bluetooth o Wi-Fi típico, siendo el tamaño del código cerca del 50% del tamaño del de Bluetooth.

LoRaWAN

LoRaWAN [51] es una especificación para redes de baja potencia y área amplia (*Low Power Wide Area Network*, LPWAN) diseñada específicamente para dispositivos de bajo consumo de alimentación.

Las principales características del estándar son: las conexiones bidireccionales seguras, bajo consumo de energía, largo alcance de comunicación, la velocidad de transmisión de datos es baja al igual que la frecuencia de transmisión y provee servicios de localización. Permite la interconexión entre objetos inteligentes sin la necesidad de instalaciones locales complejas y además, otorga amplia libertad para instalar su propia red. Estas características se ajustan a los requerimientos del Internet de las Cosas.

La arquitectura de red típica es una red de Redes en Estrella. La primera estrella está formada por los dispositivos finales y las puertas de enlace. La conexión es inalámbrica de un solo salto, usando tecnología RF LoRa[2] o FSK[3], formando así una red en estrella. La segunda estrella está formada por las puertas de enlace y un servidor de red central. Las puertas de enlace se conectan al servidor de red central por medio de conexiones IP estándar, formando así una red en estrella. Las puertas de enlaces son un puente transparente entre los dispositivos finales y el servidor de red central.

Las comunicaciones entre los dispositivos y el servidor de red, son generalmente unidireccionales o bidireccionales, pero el estándar también soporta multidifusión, permitiendo la distribución de mensajes en masa.

La comunicación entre dispositivos finales y las puertas de enlace se hacen en diferentes canales de frecuencias y a distintas velocidades de datos. Gracias a la tecnología de espectro ensanchado, las comunicaciones a distintas velocidades de datos no interfieren con otras comunicaciones a distinta velocidad. La selección de la velocidad de datos es un compromiso entre la distancia de alcance, y la duración y consumo de energía del mensaje.

Las velocidades de datos se encuentran en el rango de 0.3 kbps a 50 kbps. Para maximizar en forma conjunta la duración de la batería de los dispositivos finales y la capacidad de la red, el servidor central LoRaWAN maneja la velocidad de datos para cada dispositivo en forma individual, por medio de un esquema adaptativo de velocidad de datos (ADR, *Adaptive Data Rate*).

SIGFOX

SIGFOX [53] es una red de telecomunicaciones mundial para el Internet de las Cosas pensada para comunicaciones de baja velocidad que permite reducir los precios y el consumo de energía para los dispositivos conectados. El objetivo principal de SIGFOX consiste en permitir que todo objeto pueda conectarse a Internet, de forma asequible, sin importar la ubicación y con un consumo bajo de batería. Esta eficiencia energética permite a los usuarios habilitar dispositivos conectados capaces de durar años con una batería estándar.

[2] **RF LoRa** son módulos de radiofrecuencia aptos para el protocolo usado en las redes LoRaWAN.

[3] La **modulación por desplazamiento de frecuencia o FSK** es una técnica de modulación para la transmisión digital de información utilizando dos o más frecuencias diferentes para cada símbolo. La señal moduladora solo varía entre dos valores de tensión discretos formando un tren de pulsos.

SIGFOX proporciona una red mundial para objetos conectados altamente escalable, los dispositivos se conectan a Internet sin costes suplementarios ligados a la situación geográfica y sin configuración de red para una ubicación específica. Esta solución de conectividad mundial es administrada por SIGFOX gracias a su programa de colaboración con los operadores de redes celulares, conectando los ecosistemas locales a las redes mundiales. El protocolo SIGFOX es compatible con la mayoría de transceptores existentes.

La infraestructura de antenas y de estaciones de base SIGFOX es totalmente independiente de las redes existentes. Esta red está instalada en países como España, Francia, Holanda, Bélgica, Luxemburgo, Portugal, EEUU y se irá instalando en 60 países durante los próximos cinco años.

SIGFOX utiliza una modulación UNB (Ultra Narrow Band), basada en una tecnología de radio, para conectar dispositivos a su red mundial. La utilización de la UNB es esencial para suministrar una red escalable, con alta capacidad, evolutiva, de muy bajo consumo energético y fácil de implantar. La red SIGFOX opera en bandas ISM y ofrece una gran resistencia a las interferencias.

La red de SIGFOX está diseñada para pequeños mensajes espaciados temporalmente y no es apropiada para grandes usos de ancho de banda. La plataforma SIGFOX permite una comunicación bidireccional, desde y hacia el dispositivo. No hay negociación entre el dispositivo y la estación de recepción. La comunicación se inicia siempre desde el dispositivo, simplemente emite en una banda de frecuencia disponible y la señal es detectada por las estaciones base más cercanas, decodificando y transmitiendo al *Cloud* SIGFOX. Los mensajes después son transmitidos a la aplicación del usuario y son accesibles a través de la API de SIGFOX.

Además, SIGFOX ofrece una interfaz de aplicación web para gestionar los dispositivos y los datos, así como APIs web estándares para automatizar la gestión de los dispositivos e implementar la integración de datos. Las APIs se basan en las peticiones HTTPS REST con formato JSON. También es posible usar el protocolo MQTT.

El máximo de mensajes que se pueden enviar por día es de 140. El tamaño máximo de cada mensaje es 12 bytes. El protocolo ya incluye un *timestamp* y una única identificación de dispositivo, por lo que los 12 bytes son de datos reales de carga.

Esta solución no es óptima en la realización de este proyecto ya que no está desplegada a nivel mundial, perdiendo así la facultad de universalidad buscada. Además, no puede ser desplegada en una red local o privada, por lo que no es útil para pequeños grupos de investigación que deseen utilizar el sistema en laboratorios.

2.1.2.2. Ordenadores de placa reducida

Un ordenador de placa reducida (en inglés: *Single Board Computer* o SBC) es una computadora completa en un sólo circuito. El diseño se centra en un solo microprocesador

con la RAM, E/S y todas las demás características de un computador funcional en una sola tarjeta, que suele ser de tamaño reducido, y que tiene todo lo necesario en la placa base.
Hay plataformas de hardware libre que ofrecen productos comerciales con un hardware muy potente en este tipo de placas a un precio muy asequible. Gracias a su bajo coste y sus infinitas posibilidades se han convertido en una herramienta perfecta para el mundo del IoT y los sensores.

Dentro de estas plataformas cabe destacar Raspberry Pi, Banana Pi y BeagleBoard, las cuales se detallan a continuación:

- **Raspberry Pi** [49]. Fabricado en Reino Unido por la fundación Raspberry Pi, es un dispositivo originalmente ideado para promover la educación de ciencias de la computación en las escuelas. Sin embargo, debido al interés despertado, se ha convertido en un producto comercial utilizado en distintos ámbitos y para multitud de tareas. En 2006 comenzaron los primeros diseños de la Raspberry pero hasta seis años más tarde no presentaron un producto comercial y puesto a la venta, el modelo A. Desde entonces han comercializado cuatro modelos más cuyas diferencias se basan en procesadores más potentes y mayor capacidad RAM.
  Raspberry Pi es un ordenador completo de placa reducida, una plataforma que permite instalar un sistema operativo y comenzar a funcionar como si de un ordenador se tratase. Esto es, es un ordenador convencional con la capacidad de un ordenador tradicional pero a pequeña escala, simplificado y con las principales funciones condensadas en una pequeña placa.
  El dispositivo ofrece conexión Ethernet, entradas USB, pines E/S de propósito general (GPIO) y salida HDMI. Estos GPIO permiten usar protocolos como I^2C, SPI o UART. Además, permiten realizar modulación por ancho de pulso (PWM) y con las entradas USB se puede dotar al dispositivo de conexión WiFi gracias a tarjetas de este tipo con conexión USB.
  El software es completamente libre, siendo el sistema operativo oficial Raspbian, una adaptación de Debian para Raspberry, aunque da soporte y permite descargar otros sistemas operativos como RISC OS 5, Arch Linux ARM (derivado de Arch Linux) y Pidora (derivado de Fedora). También existen otros sistemas operativos que se pueden utilizar pero no se pueden descargar desde la plataforma oficial, como es el caso de una versión de Windows 10.

- **Banana Pi** [3] es otro ordenador de placa reducida fabricada por Shenzhen LeMaker Technology Co., Ltd. El diseño está fuertemente influenciado por Raspberry Pi y es compatible con un gran número de periféricos diseñados para ella.
  Gracias a este diseño tan similar, la mayoría de software creado para Raspberry Pi se puede ejecutar en Banana Pi con poca o ninguna modificación. Se puede emplear como sistema operativo NetBSD, Android, Ubuntu, Debian, Arch Linux y Bananian Linux. Raspbian también puede ser ejecutado, aunque no es muy recomendable.
  Banana Pi Incluye conexión Ethernet, WiFi, BlueTooth (Banana Pi M3), entradas USB, pines GPIO que permiten usar protocolos como I^2C, SPI o UART o configurar PWM. También incluye una salida HDMI.

- **BeagleBoard** [37] es una un ordenador de placa reducida de hardware libre y de bajo consumo producido por Texas Instruments. Al igual que Raspberry, la BeagleBoard fue desarrollada con fines educativos para ser usada en colegios y que promoviera el estudio de ciencias de la computación.
  El software es de código abierto y el sistema operativo empleado es Debian, pero se pueden usar otros como Angstrom, Ubuntu o Android.
  Entre los distintos productos destaca la BeagleBone y la BeagleBone Black, una versión mejorada de BeagleBone, una alternativa de bajo coste con un procesador potente y pines de E/S con altas prestaciones que hacen que sea una placa perfecta para la comunicación con sensores y diversos dispositivos electrónicos.
  Dispone de entrada Ethernet, salida HDMI, con los pines de E/S permite comunicación I^2C, SPI, UART y bus CAN, y además, permite modular la salida de estos pines con PWM.

En este proyecto se usarán dispositivos de este tipo ya que ofrecen las características de un computador en un tamaño reducido y de bajo coste. Concretamente se usará una Raspberry Pi gracias a la posibilidad de manipular la salida de los pines digitales mediante PWM, a su bajo coste respecto a sus rivales y a la gran comunidad de desarrolladores existente.

2.1.2.3. Placas de desarrollo

Una placa de desarrollo consiste en un microcontrolador, una placa y un pequeño sistema de procesamiento. Estos sistemas proporcionan conjuntos de pines E/S digitales y analógicos que pueden interactuar con diversas tarjetas de expansión y otros circuitos y llevar la información por los puertos al microcontrolador, pieza encargada de procesar esta información.

Estas placas están pensadas para fomentar y facilitar el uso de la electrónica. La ventaja de estas placas es que proporcionan todo el hardware necesario para arrancar un proyecto y proporcionan un ambiente de desarrollo software integrado (IDE) para programar con un lenguaje propio, lo que asegura la compatibilidad del hardware con el software y facilita el desarrollo. De esta manera, acercan el hardware a desarrolladores de software.

Dentro de esta categoría cabe citar los siguientes soluciones:

- **Arduino** [63]**.** El primer Arduino se introdujo en 2005 con el objetivo de proporcionar acceso a los microprocesadores embebidos, de forma sencilla y a bajo coste, tanto para principiantes como para profesionales. Así, se pueden crear proyectos de diseño interactivo y dispositivos que interactúan con su entorno mediante sensores y actuadores. Arduino permite la creación de prototipos electrónicos rápidos y baratos. De esta manera los problemas de hardware difíciles se convierten en problemas de software mucho más simples.

Arduino se basa en una placa con un microcontrolador, y además de ofrecer pines de E/S, cuenta con interfaces de comunicación serie, incluyendo Universal Serial Bus (USB) en algunos modelos, para la carga de programas de computadoras personales.
El poder de Arduino reside en la facilidad para la programación de los microcontroladores. Para ello Arduino proporciona un entorno de desarrollo integrado (IDE) basado en un lenguaje de programación llamado *Processing*, que también es compatible con los lenguajes *C* y *C++*.
Además existen diversos modelos de Arduino que están dotados de conectividad Ethernet o módulos que se pueden conectar a la placa para proporcionar conexión WiFi o Bluetooth.

- **Wiring** [66]**.** El hardware Wiring es un pequeño circuito que incluye un microcontrolador y un grupo de conectores. El hardware de Wiring puede ser usado de inmediato, provee un puerto USB el cual puede ser directamente conectado al computador para cargar programas, sin ajustes especiales o la necesidad de conexiones para empezar a trabajar con él.
  Hay tres escenarios básicos de prototipado para los cuales Wiring ha sido diseñado: objetos o espacios interactivos autónomos sin la necesidad de estar conectado a un computador; objetos o espacios interactivos conectados a un computador anfitrión; o interconectar objetos o espacios comunicando múltiples dispositivos de hardware.
  El microcontrolador puede ser programado en el entorno de programación Wiring para controlar toda clase de sensores y actuadores y para comunicar el hardware Wiring con un computador anfitrión para otros propósitos, como enviar datos desde o hacia la tarjeta a una aplicación corriendo en un PC.

- **Netduino** [41]**.** La placa de desarrollo Netduino también dispone de un microcontrolador y pines E/S. Es una placa basada en un ARM Córtex construida alrededor del microcontrolador STMicro STM32Fx de 32 bits. Además se pueden conectar la mayoría de los *shields* (pequeñas piezas de hardware con distinta funcionalidad que se pueden añadir al dispositivo) de Arduino existentes.
  Existen distintas tarjetas Netduino disponibles y algunos modelos incluyen conexión Ethernet o WiFi. El sistema operativo es el NET Micro Framework, siendo las tarjetas programables en C#, directamente desde Microsoft Visual C# Express 2010.

2.1.2.4. Plataformas *ad hoc*

La electrónica necesaria para poder enviar el dato de un sensor a la red puede ser diseñada de manera específica para cada sensor. Para ello hay que realizar todo el estudio de conexiones y hardware necesario para realizar la tarea deseada, lo que llevaba un elevado coste.

La plataforma está compuesta por un microcontrolador, generalmente PIC. El PIC es una Unidad Completa de Microcontrolador (MCU) con procesador incorporado, memoria y E/S programables, cuya programación se realiza con lenguaje C de bajo nivel. Este

microcontrolador está conectado al sensor y se encarga de enviar la señal a otro controlador encargado de procesar la señal y enviarla por la red. Además, es necesario realizar todas las conexiones necesarias con los elementos que envían los datos por la red y con los sensores, elevando el coste del sistema.

2.1.3. Soluciones comerciales

A continuación se exponen distintas soluciones comerciales encargadas de ofrecer dispositivos para la comunicación con el sensor y el envío de los datos a través de la red.. Para ello es necesario disponer primero de una interfaz de comunicación entre el sensor y el dispositivo encargado del envío.

- ***SmartEveryThing*** [27] es una placa de desarrollo fabricada por Arrow Electronics destinada para el uso del Internet de las Cosas. Es compatible con Arduino y viene con una gran variedad de puertos y sensores de E/S de fábrica, además tiene una tecnología de conectividad inalámbrica de alta eficiencia energética.
  La placa *SmartEverything* incluye un sensor de proximidad y de luz ambiental; un sensor digital capacitivo para las mediciones de humedad y temperatura; un acelerómetro 3D de nueve ejes, un sensor combinado de giroscopio 3D y magnetómetro 3D y un sensor de presión basado en MEMS (*MicroElectroMechanical Systems*).
  En cuanto a la conectividad, dispone de una antena de 868 MHz, un módulo NFC (Near Field Communication) y un módulo Bluetooth Low Energy (BLE). Además, cuenta con SIGFOX, una tecnología eficiente con gran alcance de transmisión que utiliza la tecnología UNB basada en la radio tecnología, ofreciendo bajas velocidades de transferencia de datos de 10 a 1.000 bits por segundo.
  Ofrece la posibilidad de utilizar distintas placas de Arduino para poder usar otros dispositivos y sensores.

- ***Visible Things*** [61] es una plataforma desarrollada por *Avnet Memec - Silica* para el desarrollo de sistemas y aplicaciones para el IoT. Está destinada a mercados industriales siendo una plataforma de evaluación y desarrollo altamente flexible e integral.
  La plataforma ofrece hardware testado y seguro además de un software integrado para conectar los sensores inteligentes y dispositivos embebidos a través de soluciones de puerta de enlace o de redes de área extensa de baja potencia (*Low-Power Wide-Area-Networks*, LPWAN), a través de las aplicaciones en la nube y software empresarial.
  Ofrecen tres kits iniciales que sirven como referencia para el diseño, cada uno gestionado por microcontroladores basados en *ARM Córtex*.
  El primer kit está compuesto por dos placas. La primera se trata de un sensor inteligente que incluye Bluetooth y conectividad inteligente. Esta placa incluye sensores de movimiento, temperatura, humedad, luz y proximidad. La segunda placa se trata de una placa que cumple las funciones de puerta de enlace (*Gateway*), que gestiona toda la conectividad necesaria para el servicio en la nube a través de WiFi.

Además, esta última placa ofrece de una opción de expansión, un módulo periférico GSM con opciones de conector SIM y SIM embebida para permitir la conectividad celular a los servicios de software *cloud enterprise*.
Los otros dos kits están diseñados para su uso con las redes SIGFOX y LoRaWAN, e incluyen respectivos módulos sensores SIGFOX y LoRaWAN. Cada placa incorpora sensores de movimiento y luz adicionales. En este caso los mensajes se pueden intercambiar con servicios en la nube sin necesidad de comunicarse a través de una puerta de enlace.
Además, ofrecen una aplicación móvil disponible para iOS y Android que proporciona acceso a la configuración de los sensores y es compatible con los servicios en la nube. La aplicación también proporciona una guía de inicio rápido para que sea fácil conectar y configurar el sistema.

- **Libelium** [35] se constituyó en noviembre de 2006 como *Spin Off* de la Universidad de Zaragoza tras detectar la necesidad de desarrollar tecnología capaz de monitorizar de manera inalámbrica cualquier tipo de parámetro ambiental. Esta empresa se dedica al diseño, desarrollo y fabricación de hardware específico para la implementación de redes sensoriales inalámbricas utilizadas en la monitorización de parámetros ambientales.
  Libelium ofrece un producto llamado *Waspmote*. Se trata de una plataforma de desarrollo, con su propio IDE, a la que se pueden conectar distintas placas que contienen los sensores, por ejemplo: placas con sensores de gases, entre los que se encuentran sensores de monóxido de carbono, dióxido de carbono, oxígeno, metano, hidrógeno, amoniaco, isobutano, etanol, tolueno, sulfuro de hidrógeno, dióxido de nitrógeno, ozono, hidrocarburos, además de sensores de temperatura, humedad y presión atmosférica; placas destinadas a medir parámetros relacionados con la calidad del agua, como por ejemplo, pH, el potencial de oxidación-reducción, el oxígeno disuelto, la conductividad, la temperatura y la turbiedad además de iones presentes en el agua; placas destinadas a control de eventos que incorporan sensores de presión, vibración, impacto, efecto Hall, inclinación, temperatura, presencia de líquidos, fugas de agua, nivel líquido, luminosidad y presencia. Ofrecen una placa que incluye un tubo de Geiger para medir radiaciones.
  Además, disponen de placas que se integran en la plataforma para dotarla de conectividad. De esta manera ofrecen conexión por ZigBee, LoRaWAN, Lora, SIGFOX, WiFi, BlueTooth, 6LoWPAN y comunicación móvil, 3G / GPRS / GSM.
  También disponen de una solución comercial, Waspmote Plug&Sense, compuesta por los productos anteriores para tomar muestras medioambientales, muestras del agua, sensores orientados a seguridad o soluciones para *smart cities.*
  Para crear una red de sensores y dotar de conectividad ofrecen otro producto llamado Meshlium. Tiene su propia base de datos y permite la conexión a internet para enviar los datos a la nube.

2.2. Comunicación y gestión de datos en el Cloud

En este apartado se abordan las tecnologías necesarias para gestionar y almacenar los datos en la nube. Además, se exponen distintas alternativas para la arquitectura de los servicios ofrecidos en la nube. De esta manera se podrá construir una plataforma software en la nube para cumplir con los objetivos

También se explican distintos protocolos para comunicar la nube con los dispositivos que deseen interactuar con ella. Estos protocolos son necesarios para poder enviar datos por la red de una manera eficiente y fiable.

Por último, se describen distintas empresas del sector destinadas a ofrecer este tipo de servicios orientados al Internet de las Cosas.

2.2.1 Computación en la nube (Cloud)

La computación en la nube (*cloud computing* [45]) es un paradigma que permite ofrecer servicios de computación a través de una red, que usualmente es Internet, en el que la información se procesa y se almacena de manera permanente en servidores de Internet y se envía a los clientes.

En este tipo de computación todo lo que puede ofrecer un sistema informático se ofrece como servicio, de modo que los usuarios puedan acceder a los servicios disponibles "en la nube de Internet". La computación en la nube son servidores desde Internet encargados de atender las peticiones en cualquier momento. Se puede tener acceso a su información o servicio, mediante una conexión a Internet desde cualquier dispositivo móvil o fijo ubicado en cualquier lugar. De esta manera, el acceso a los datos es instantáneo en todo momento sin importar la ubicación del cliente.

Este paradigma también permite aprovechar mejor los recursos de los dispositivos cliente, ya que el procesamiento de los datos y el almacenamiento de éstos se realiza en la nube. El concepto de nube permite aumentar el número de servicios basados en la red. Se pueden ofrecer, de forma más rápida y eficiente, un mayor número de servicios.

En la nube se ofrecen distintas soluciones tecnológicas que se pueden resumir en tres modalidades: software como servicio (*Software As A Service*, SaaS), Plataforma como Servicio (PaaS) e Infraestructura como Servicio (IaaS).

El software como servicio (SaaS) es un modelo de distribución de software en el que las aplicaciones están alojadas por una compañía o proveedor de servicio y puestas a disposición de los usuarios a través de una red, generalmente Internet. Plataforma como servicio (PaaS) es un conjunto de utilitarios para abastecer al usuario de sistemas operativos y servicios asociados a través de Internet sin necesidad de descargas o instalación alguna. Infraestructura como Servicio (IaaS) se refiere a la tercerización de los

equipos utilizados para apoyar las operaciones, incluido el almacenamiento, hardware, servidores y componentes de red.

El concepto de nube se suele aplicar a cualquier modalidad que ofrezca los servicios, lo que permite que los servicios puedan estar alojados en algún proveedor de internet, en servidores privados o incluso en una máquina en un entorno local.

2.2.2. Protocolos de comunicación máquina a máquina

Es necesario establecer un protocolo de comunicación en el envío de información para la persistencia de datos y la configuración. A continuación, se muestran los protocolos más utilizados en el Internet de las Cosas.

Hypertext Transfer Protocol (HTTP)

HTTP [46] es el protocolo de comunicación que permite las transferencias de información en la web. HTTP define la sintaxis y la semántica que utilizan los elementos de software de la arquitectura web (cliente, servidor, proxy) para comunicarse. HTTP es un protocolo sin estado, es decir, no guarda ninguna información sobre conexiones anteriores.

Desde el punto de vista de las comunicaciones, está soportado sobre los servicios de conexión TCP/IP[4]. Un proceso servidor escucha en un puerto de comunicaciones TCP (por defecto, el 80), y espera las solicitudes de conexión de los clientes web. Una vez que se establece la conexión, el protocolo TCP se encarga de mantener la comunicación y garantizar un intercambio de datos libre de errores.

HTTP se basa en operaciones de solicitud/respuesta. Un cliente establece una conexión con un servidor y envía un mensaje con los datos de la solicitud. El servidor responde con un mensaje similar, que contiene el estado de la operación y su posible resultado. Todas las operaciones pueden adjuntar un objeto o recurso sobre el que actúan. Cada objeto es conocido por su URL. Los mensajes HTTP son en texto plano lo que lo hace más legible, fácil de depurar y los mensajes son más largos.

HTTP define una serie predefinida de métodos de petición que pueden utilizarse, pero tiene flexibilidad para ir añadiendo nuevos métodos para así añadir nuevas funcionalidades. Cada método indica la acción que desea que se efectúe sobre el recurso identificado. Lo que este recurso representa depende de la aplicación del servidor.

[4] TCP (**Transmission Control Protocol**) es la capa intermedia entre el protocolo de internet (IP) y la aplicación. TCP que asegura que los datos que emite el cliente son recibidos por el servidor sin errores y en el mismo orden que fueron emitidos. Es un protocolo orientado a la conexión, ya que el cliente y el servidor deben anunciarse y aceptar la conexión antes de comenzar a transmitir los datos a ese usuario que debe recibirlos.

Los métodos más importantes son los siguientes:

- HEAD: Solicita una respuesta idéntica correspondiente a una petición GET pero sin el cuerpo.
- GET: Pide el recurso especificado.
- POST: Envía los datos para que sean procesados por el recurso identificado. Los datos van incluidos en el cuerpo de la petición. Con este método se puede crear un nuevo recurso, actualizar algún recurso existente o ambas cosas.
- PUT: Envía los datos en el cuerpo para actualizar un recurso especificado.
- DELETE: Borra el recurso especificado.
- CONNECT: Se utiliza para saber si se tiene acceso a un host, no necesariamente la petición llega al servidor, este método se utiliza principalmente para saber si un proxy da acceso a un host bajo condiciones especiales.

La universalidad que ofrece este protocolo, basado sobre conexión TCP/IP, y las operaciones de solicitud/respuesta conllevan a la utilización de este protocolo para la comunicación desarrollada en este proyecto. Además, los servicios REST (Apartado 2.2.3.3) basados en este protocolo ofrecen una gran escalabilidad, por lo que se convierte en la mejor opción para desarrollar un *Cloud* preparado para soportar peticiones de múltiples sensores.

Constrained Application Protocol (CoAP)

El Protocolo de aplicación restringida es un protocolo de capa de aplicación de software destinado a ser utilizado en dispositivos electrónicos muy simples con recursos limitados que les permite comunicarse de forma interactiva a través de Internet. CoAP [48] está diseñado para trasladar el modelo HTTP a dispositivos y redes restrictivas. El diseño del protocolo traduce fácilmente a HTTP la integración simplificada con la web. De esta manera, cumple los requisitos de bajo coste, simplicidad y es compatible con multidifusión.

El modelo de interacción cliente-servidor es similar al de HTTP con la diferencia que CoAP realiza estas interacciones de forma asíncrona por medio del protocolo de transporte UDP[5], siendo los paquetes mucho más pequeños. Soporta los métodos GET, PUT, POST y DELETE.

CoAP se basa en el intercambio de mensajes asíncronos entre dos nodos, un nodo actuando como cliente envía una o más peticiones sobre uno o más recursos alojados en un determinado servidor que atiende la petición. El servidor responde a la petición indicando si la petición recibida es exitosa o no. Las peticiones se envían mediante mensajes

[5] **User Datagram Protocol** es un protocolo del nivel de transporte basado en el intercambio de datagramas. Permite el envío de datagramas a través de la red sin que se haya establecido previamente una conexión. El datagrama incorpora suficiente información de direccionamiento en su cabecera. Tampoco tiene confirmación ni control de flujo, por lo que los paquetes pueden adelantarse unos a otros; y tampoco se sabe si ha llegado correctamente, ya que no hay confirmación de entrega o recepción.

Confirmable (CON) o No-Confirmable (NON) y ejecuta un método sobre un recurso, el cual viene identificado por una ruta contenida en el campo *URI-Path* del paquete CoAP.

WebSocket

WebSocket [47] es una tecnología que proporciona un canal de comunicación bidireccional y full-duplex sobre un único *socket*[6] TCP. Está diseñada para ser implementada en navegadores y servidores web, pero puede utilizarse por cualquier aplicación cliente-servidor. El uso de esta tecnología proporciona una funcionalidad similar a la apertura de varias conexiones en distintos puertos pero multiplexando diferentes servicios WebSocket sobre un único puerto TCP (a costa de una pequeña sobrecarga del protocolo).

Para establecer una conexión WebSocket, el cliente manda una petición de negociación WebSocket, y el servidor manda una respuesta de negociación WebSocket. Esta negociación puede parecerse a la negociación HTTP, pero no es así. Permite al servidor interpretar parte de la petición de negociación como HTTP y entonces cambiar a WebSocket.

Una vez establecida, las tramas WebSocket de datos pueden empezar a enviarse en ambos sentidos entre el cliente y el servidor en modo *full-duplex*. Las tramas de texto pueden ser enviadas en modo *full-duplex* también, en ambas direcciones al mismo tiempo. La información se segmenta en tramas de 2 bytes.

La implementación de cliente del protocolo WebSocket intenta detectar si el agente de usuario está configurado para utilizar un proxy a la hora de conectar a un host y puerto remoto, y si es así, utiliza el método HTTP CONNECT para establecer un túnel persistente.
Implementa una negociación compatible con HTTP para que los servidores HTTP puedan compartir sus puertos HTTP y HTTPS por defecto (80 y 443) con una pasarela o servidor WebSocket. El protocolo WebSocket define un prefijo “ws://” y “wss://” para indicar una conexión WebSocket y WebSocket *Secure*, respectivamente. Algunos servidores proxy no interfieren en la conexión y funcionan perfectamente con WebSocket; otros afectan al correcto funcionamiento de WebSocket, provocando que la conexión falle.

Message Queue Telemetry Transport (MQTT)

Es un protocolo ligero usado para la comunicación máquina a máquina (M2M) en el *Internet of Things* ya que los clientes son pequeños y utiliza el ancho de banda de red de forma eficiente, facilitando ser utilizado en la mayoría de los dispositivos empotrados con pocos recursos. La cabecera de longitud fija tiene sólo 2 bytes de longitud y se minimizan los intercambios de protocolo para reducir el tráfico en la red. Se ejecuta sobre TCP/IP, que proporciona conectividad de red básica. No depende en modo alguno del contenido del mensaje.

[6] ***Socket*** es un concepto abstracto por el cual dos programas pueden intercambiar cualquier flujo de datos de manera fiable y ordenada.

El protocolo MQTT [15] da soporte a la entrega asegurada y a transferencias 'dispara y olvida'. Es un protocolo de publicación/suscripción por lo que la entrega de mensajes es independiente de la aplicación. La entrega desacoplada libera a una aplicación de tener que estar conectada a un servidor y esperando mensajes. El modelo de interacción es como en el correo electrónico, pero optimizado para la programación de aplicaciones.

La entrega de mensajes puede ser de tres tipos: como máximo una vez, los mensajes se entregan en base a la carga de la red, se puede producir pérdida de mensajes; al menos una vez, se asegura que los mensajes llegan, pero se pueden producir duplicados; exactamente una vez, se asegura que los mensajes llegan exactamente una sola vez. Además, dispone de una función que notifica a los suscriptores si se produce una desconexión de un cliente de un servidor MQTT.

La arquitectura de MQTT sigue una topología de estrella, con un nodo central que hace de servidor o "*broker*" con una capacidad de hasta 10.000 clientes. El *broker* es el encargado de gestionar la red y de transmitir los mensajes, para mantener activo el canal, los clientes mandan periódicamente un paquete (PINGREQ) y esperan la respuesta del broker (PINGRESP). La comunicación puede ser cifrada entre otras muchas opciones.

La comunicación se basa en unos "*topics*" (temas), que el cliente que publica el mensaje crea y los nodos que deseen recibirlo deben suscribirse a él. La comunicación puede ser de uno a uno, o de uno a muchos. Un "*topic*" se representa mediante una cadena y tiene una estructura jerárquica. Cada jerarquía se separa con '/'. De esta forma un nodo puede suscribirse a un "*topic*" concreto o a varios.

Advanced Message Queuing Protocol (AMQP)

El estándar AMQP es un protocolo de comunicaciones abierto en la capa de aplicaciones. Está diseñado como un servicio estándar altamente disponible para el intercambio de mensajes de misión crítica y para hacer posible el intercambio de mensajes empresariales entre diferentes plataformas. Las características que definen al protocolo AMQP [1] son la orientación a mensajes, encolamiento, enrutamiento (tanto punto a punto como publicación/subscripción), exactitud y seguridad.

AMQP determina el comportamiento tanto del servidor que provee los mensajes como del cliente de la mensajería. De esta manera, las implementaciones de diferentes proveedores son verdaderamente interoperables. Cualquier programa que pueda crear e interpretar mensajes conforme a este formato de datos puede interoperar con cualquier otra herramienta que cumpla con este protocolo, independientemente del lenguaje de implementación.

La arquitectura del protocolo está formada por un cliente productor, el encargado de enviar el mensaje, el consumidor, el cliente que quiere recibir el mensaje, y el "*broker*", entidad intermediaria entre el productor y el consumidor. El *broker* a su vez está formado por distintas entidades: *exchange*, es la entidad que toma los mensajes enviados al *broker* por el productor y los enruta a una o más colas. Existen cuatro tipos básicos de *exchanges*, los cuales difieren únicamente del algoritmo que utilizan para determinar qué cola debe de

recibir los mensajes; "*binding*", es la entidad que determina las reglas a ser utilizadas por los *exchanges*, estas reglas son aplicadas entre otras cosas para determinar cómo enrutar un mensaje a una cola, su misión es ligar un *exchange* a un cola; cola, es la encargada de almacenar el mensaje que será consumido por el consumidor.

Los consumidores reciben mensajes suscribiéndose a la cola que contiene el mensaje.
Estos suscriptores pueden ver entre los mensajes sin consumirlos, dejándolos en la cola para que puedan ser vistos. Una vez consumido el mensaje se elimina de la cola.

2.2.3. Servicios web

El término **servicios web** designa una tecnología que utiliza un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones. Esta comunicación es independiente de la plataforma y del lenguaje de programación empleado. Un servicio web es una interfaz de software que describe un conjunto de operaciones. Distintas aplicaciones de software desarrolladas en lenguajes de programación diferentes, y ejecutadas sobre cualquier plataforma, pueden utilizar los servicios web para intercambiar datos en redes de ordenadores como Internet. La interoperabilidad se consigue mediante la adopción de estándares abiertos.

Dentro de los servicios web existen ciertos estándares. Los más destacados son REST y SOAP, que serán detallados en los siguientes apartados.

Un grupo de servicios web que interactúa de esa forma define la aplicación de un servicio web específico en una Arquitectura Orientada a Servicios (SOA). Dentro de estas arquitecturas cabe destacar la arquitectura orientada a microservicios. Esta arquitectura se caracteriza por crear servicios más pequeños e independientes y se detalla en el siguiente apartado.

2.2.3.1. Arquitectura de microservicios

Un sistema basado en **microservicios** es aquel que distribuye toda su organización de forma vertical. La información solicitada puede ser consultada a un servicio específico independiente en recursos y es capaz de responder la solicitud.

Esta arquitectura contrasta con la arquitectura monolítica que se organiza de manera horizontal, es decir, la información pasa por toda la infraestructura de la aplicación. Aunque partes de la aplicación no dependan de otras están encapsuladas dentro del mismo sistema horizontal.

El uso de microservicios proporciona ventajas frente a la arquitectura monolítica, por ejemplo, un fallo en un servicio no afecta al resto de servicios, en una arquitectura monolítica el fallo de una sección afecta a la toda aplicación. En el uso de microservicios la

escalabilidad no está sujeta a ninguna de las partes, como son independientes es posible agrupar y distribuir los servicios en varios servidores como sea necesario.

La arquitectura de microservicios ofrece modularidad. Un servicio puede usarse o no dependiendo de las necesidades. Además, gracias a la facilidad de escalabilidad que presentan estas arquitecturas se puede alojar cada servicio en una máquina distinta con las especificaciones que mejor se ajusten al servicio, abaratando costes y proporcionando una mejora en el rendimiento general del sistema.

2.2.3.2. SOAP

SOAP [33] (Simple Object Access Protocol) es un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML. Por tanto, SOAP permite la comunicación entre aplicaciones a través de mensajería en una dirección sin estado, es independiente de la plataforma y del lenguaje y está basado en XML.

Este protocolo deriva del protocolo XML-RPC [34]. SOAP fue creado principalmente por Microsoft e IBM y actualmente está bajo la supervisión de la W3C.

SOAP permite cualquier modelo de programación y es independiente de la plataforma. Además, puede ser utilizado sobre cualquier protocolo de transporte, como por ejemplo HTTP, garantizando que cualquier cliente con un navegador estándar pueda conectarse con a servidor remoto.

Los mensajes SOAP son transmisiones unidireccionales desde un emisor a un receptor. Se suelen combinar mensajes para implementar patrones, como petición/respuesta. Un mensaje SOAP es un documento XML con una estructura detallada de la siguiente manera:

- *Envelope*: es la parte que identifica al mensaje SOAP como tal y es obligatoria.
- *Header*: permite enviar información relativa a como debe ser procesado el mensaje. Es una herramienta para que los mensajes puedan ser enviados de la forma más conveniente para las aplicaciones.
- *Body*: contiene la información relativa a la llamada y la respuesta.
- *Fault*: bloque que contiene información relativa a errores que se hayan producido durante el procesado del mensaje y el envío.

Los mensajes deben ser procesados de la siguiente manera:

1. Identificación de las partes del mensaje SOAP dirigidas a la aplicación.
2. Aceptar y procesar las partes anteriores adecuadamente. Si no se pueden procesar se descarta el mensaje.
3. Si la aplicación SOAP no es el destino final, eliminar las partes del primer paso y reenviar el mensaje.

2.2.3.3. REST

La Transferencia de Estado Representacional (*Representational State Transfer*, REST [19]) es un estilo de arquitectura software para sistemas hipermedia distribuidos que se apoya en el estándar HTTP.

REST permite crear servicios y aplicaciones que pueden ser usadas por cualquier dispositivo o cliente que entienda HTTP, convirtiéndose en un proceso más simple y convencional que otras alternativas como SOAP y XML-RPC. En la actualidad, REST se utiliza para describir cualquier interfaz entre sistemas que utilicen HTTP, como por ejemplo aplicaciones web, siendo así el tipo de arquitectura más natural y estándar para crear APIs para servicios orientados a Internet.

REST es un protocolo cliente/servidor **sin estado**, de manera que nunca se debe guardar el estado en el servidor. Toda la información necesaria que se requiere para solicitar un recurso debe estar contenida en el mensaje de la petición HTTP del cliente. Sin embargo, en la práctica, muchas aplicaciones basadas en HTTP utilizan cookies y otros mecanismos para mantener el estado de la sesión. Al no guardar estado, REST permite **escalar mejor** sin preocuparse de temas como el almacenamiento de variables de sesión e incluso se pueden emplear distintas tecnologías para servir determinadas partes o recursos de una misma API [75].

El modelo Richardson Maturity Model [76] define tres niveles de calidad para aplicar REST en el desarrollo de una aplicación web y más concretamente una API. Estos niveles son:

1. Uso correcto de URIs. En un sistema REST, cada recurso es direccionable únicamente a través de su URI
2. Uso correcto de HTTP. Un conjunto de operaciones bien definidas que se aplican a todos los recursos de información: HTTP en sí define un conjunto pequeño de operaciones, las más importantes son POST,GET, PUT y DELETE. Con frecuencia estas operaciones se equiparan a las operaciones CRUD en bases de datos que se requieren para la persistencia de datos, aunque POST no encaja exactamente en este esquema.
3. El uso de hipermedios, tanto para la información de la aplicación como para las transiciones de estado de la aplicación: la representación de este estado en un sistema REST son típicamente HTML o XML. Como resultado de esto, es posible navegar de un recurso REST a muchos otros, simplemente siguiendo enlaces sin requerir el uso de registros u otra infraestructura adicional.

Este proyecto se basa en este estándar debido a las ventajas que ofrece en la escalabilidad, la utilización del protocolo universal HTTP, el bajo consumo de recursos y la flexibilidad que ofrece en la elección de las tecnologías.

2.2.4. Bases de datos

Una base de datos es un banco de información que contiene datos relativos a diversas temáticas y categorizados de distinta manera, pero que comparten entre sí algún tipo de vínculo o relación que busca ordenarlos y clasificarlos en conjunto.

Desde el punto de vista informático, la base de datos es un sistema formado por un conjunto de datos almacenados en discos que permiten el acceso directo a ellos y un conjunto de programas que manipulen ese conjunto de datos.

Existen programas denominados sistemas gestores de bases de datos (*Database Management System*, DBMS), que permiten almacenar y posteriormente acceder a los datos de forma rápida y estructurada.

Se diferencian dos tipos de gestores de bases de datos según las relaciones existentes en el contenido: relacionales y no relacionales.

2.2.4.1. Bases de datos relacionales

La base de datos relacional (BDR) es un tipo de base de datos (BD) que cumple con el modelo relacional. Permite establecer interconexiones o relaciones entre los datos (que están almacenados en tablas), y a través de dichas conexiones relacionar los datos de ambas tablas.

Tras ser postulados sus fundamentos en 1970 por Edgar Frank Codd, de los laboratorios IBM en San José, California, no tardó en consolidarse como un nuevo paradigma en los modelos de base de datos. Su idea fundamental es el uso de "relaciones".

En este modelo, el lugar y la forma en que se almacenen los datos no tienen relevancia lo que hace que tenga una curva de aprendizaje muy pequeña y sea de fácil manejo. La información puede ser recuperada o almacenada mediante "consultas" que ofrecen una amplia flexibilidad y poder para administrar la información.

El lenguaje más habitual para construir las consultas a bases de datos relacionales es SQL, *Structured Query Language* o Lenguaje Estructurado de Consultas, un estándar implementado por los principales motores o sistemas de gestión de bases de datos relacionales.

Dentro de los gestores de bases de datos destacan los siguientes:

- **MySQL** [40]: Base de datos *open source* desarrollada por Oracle. Entre sus características destacan: un amplio subconjunto del lenguaje SQL, disponibilidad en gran cantidad de plataformas y sistemas, la posibilidad de seleccionar distintos mecanismos de almacenamiento, replicación de los datos, búsqueda e indexación de los campos de texto. Está optimizada para consultas sencillas.

- **PostgreSQL** [44]: Es un Sistema de gestión de bases de datos relacional orientado a objetos y de libre distribución. Está desarrollada por PostgreSQL Global Development Group (PGDG). Al ser una base de datos relacional las características son similares a las de MySQL cuya ventaja radica en que es capaz de unir cantidades relativamente grandes de tablas eficientemente.

- **SQLite** [54]: Es un sistema de gestión de bases de datos relacional contenida en una relativamente pequeña biblioteca escrita en C. SQLite es de dominio público creado por D. Richard Hipp. Se caracteriza por tener una rápida configuración, sólo es necesario incluir una librería en la aplicación, está embebida, no es necesario pensar en conexiones, no es escalable y no mejora el rendimiento. Trabaja muy bien con sitios de tráfico medio-bajo y es ideal para pequeñas aplicaciones.

2.2.4.2. Bases de datos no relacionales

Son bases de datos cuyos sistemas de gestión de bases de datos difieren del modelo clásico del sistema de gestión de bases de datos relacionales (RDBMS) en aspectos importantes, el más destacado es que no usan SQL como el principal lenguaje de consultas, por lo que se conocen como **NoSQL**, no requieren estructuras fijas como tablas, normalmente no soportan operaciones JOIN, ni garantizan completamente ACID (atomicidad, consistencia, aislamiento y durabilidad), y habitualmente escalan bien horizontalmente. Los sistemas NoSQL se denominan a veces "no sólo SQL" para subrayar el hecho de que también pueden soportar lenguajes de consulta de tipo SQL.

Las bases de datos NoSQL se clasifican según su forma de almacenar los datos, y comprenden categorías como clave-valor, las implementaciones de *BigTable*, bases de datos documentales, y bases de datos orientadas a grafos.

El uso de estas bases de datos crecieron gracias a que grandes compañías de Internet tenían la necesidad de proporcionar información procesada a partir de grandes volúmenes de datos y que las bases de datos relacionales dedicaban mucho tiempo al proceso.

En ese sentido, a menudo, las bases de datos NoSQL están altamente optimizadas para las operaciones recuperar y agregar, y normalmente no ofrecen mucho más que la funcionalidad de almacenar los registros, por ejemplo el almacenamiento clave-valor. La pérdida de flexibilidad en tiempo de ejecución, comparado con los sistemas SQL clásicos, se ve compensada por ganancias significativas en escalabilidad y rendimiento cuando se trata con ciertos modelos de datos.

Entre las bases de datos no relacionales destacan las siguientes:

- **Redis** [50]: Escrita en *C*, es un motor de base de datos en memoria, basado en el almacenamiento en tablas de *hashes* (clave/valor) pero que opcionalmente puede ser usada como una base de datos durable o persistente. Su uso está indicado

cuando los datos cambian rápidamente y con un tamaño de base de datos previsible, por ejemplo, el análisis en tiempo real.

- **MongoDB** [38]: Escrita en *C++*, es una base de datos de código abierto orientada a documentos. Su uso es conveniente si se necesitan consultas dinámicas. Es útil, por ejemplo, para cuando se tiene una base de datos relacional pero el número de columnas es variable.

- **Cassandra** [11]: Está basada en un modelo de almacenamiento de «clave-valor», de código abierto que está escrita en *Java*. Permite grandes volúmenes de datos en forma distribuida. Su objetivo principal es la escalabilidad lineal y la disponibilidad. La arquitectura distribuida de Cassandra está basada en una serie de nodos iguales que se comunican con un protocolo P2P con lo que la redundancia es máxima. Está indicada para almacenar grandes volúmenes de datos, como los provenientes de sensores, por ello será utilizada en este proyecto. Además, tiene una interfaz denominada CQL que es muy parecida a SQL, lo que hace que adaptarse a esta base de datos sea muy sencillo.

2.2.5. Soluciones Cloud para el IoT

En la actualidad existen distintos proyectos dedicados a almacenar y tratar datos de los objetos del Internet de la Cosas. Todos estos proyectos se basan en la nube y la comunicación es a través de Internet tratándose de plataformas externas al dominio propio.

Dentro de estos proyectos cabe destacar distintos productos comerciales de pago que ofrecen algunas empresas, como por ejemplo:

- **thethings.iO** [30]: Es una *startup*[7] que ayuda a empresas de hardware a conectar sus dispositivos a Internet de forma rápida y fácil para que estos dispositivos, a su vez, puedan interactuar con otras “cosas conectadas”. De esta manera, diferentes dispositivos de diferentes fabricantes son capaz de interoperar entre ellos.
  La conexión entre los dispositivos se realiza mediante una API, de esta manera aceptan todo tipo de hardware y no fuerzan a utilizar un tipo en particular.
  Para poder almacenar información primero hay que crear el objeto desde una plataforma web que ofrecen. Una vez activado el dispositivo se pueden enviar a la nube los datos a almacenar en forma clave-valor. Ofrecen soporte para los protocolos HTTP, WebSocket, MQTT, CoAP y UDP.
  La monitorización de los dispositivos se realiza mediante una página web en la que se pueden crear paneles para mostrar la información que se quiera según el tipo de “cosa” que se haya creado previamente. También permiten monitorizar en tiempo real. Ofrecen ciertos tipos de analíticas relacionadas con el número de peticiones realizadas y el gasto que conllevan.

[7] **startup** es un término utilizado actualmente en el mundo empresarial para empresas emergentes apoyadas en la tecnología, las cuales buscan arrancar, emprender o montar un nuevo negocio.

- **Xively** [67]: Es en un servicio *online* desarrollado específicamente para el Internet de las Cosas. La plataforma permite publicar los datos recogidos por distintos sensores, como pueden ser sensores de humedad, temperatura, gases, luminosidad o radiación, mediante gráficas en tiempo real y *widgets*[8]. La plataforma es un conjunto de servicios que proporcionan los principales bloques de construcción para que los clientes los puedan usar. Cada servicio o bloque es el encargado de gestionar una parte de la arquitectura.
  Los dispositivos del IoT se comunican con la plataforma a través del protocolo de mensajería MQTT, enviando los datos para que se almacenen en la nube. Proporcionan librerías escritas para arquitecturas ARM, Arduino y lenguajes de programación C, Java, Python y Ruby.
  En cuanto a la gestión de organizaciones y dispositivos, así como el acceso a los datos almacenados, se realiza a través de una API REST en HTTP. De esta manera se accede a la gestión de identidades y al servicio *Blueprint*, encargado de mantener un catálogo de dispositivos y organizaciones.

- **Carriots** [72]: es una Plataforma como Servicio diseñada para proyectos del Internet de las Cosas y de Máquina a Máquina. Permite recopilar los datos de los objetos conectados, almacenar los datos, crear aplicaciones que consuman esos datos en pocas líneas de código y permite la integración con sistemas informáticos externos. Carriots ofrece un entorno de desarrollo, APIs y alojamiento para el desarrollo de proyectos de IoT.
  Está destinado a cualquier dispositivo que disponga de conexión a Internet para poder enviar los datos. Para ello, se expone una API REST encargada de recibir los datos, en formato XML o JSON, y los almacena. Permite el envío a través del protocolo HTTP o de MQTT.
  Desde la plataforma web ofrecen la posibilidad de activar o desactivar la recepción de datos, comprobar el estado de los dispositivos y ejecutar lógica para crear alarmas cuando algún evento suceda.
  El acceso a la información se realiza a través de la API REST. Para ello ofrecen un SDK escrito en *Groovy* para realizar la conexión y facilitar el desarrollo de la aplicación que consuma estos datos. También ofrecen desde su plataforma web la posibilidad de descargar los datos en formato CSV, JSON o XML.
  Carriots dispone de integración nativa con *InitalState* para construir un panel de visualización de los datos de una forma muy sencilla.

- **ThingWorx** [59] es una unidad de negocio de la empresa *PTC*, una compañía de software global. ThingWorx es una plataforma tecnológica diseñada desde el principio para el Internet de los Cosas. Simplifica la creación y despliegue de aplicaciones para productos conectados e inteligentes, ofreciendo a los desarrolladores las herramientas necesarias para conectar, crear, analizar, experimentar y colaborar sobre sus "cosas".
  Disponen de unos kits para facilitar el inicio del desarrollo que permiten ser personalizados. Para ello disponen de un SDK desarrollado en *C* y en *Java* para ser

[8] Un **widget** es una pequeña aplicación o programa, usualmente presentado en archivos o ficheros pequeños que son ejecutados por un motor de widgets o Widget Engine.

utilizados en los dispositivos de Intel y en la Raspberry Pi. De esta manera, proporcionan librerías para tomar los datos y enviarlos a la nube. También proveen una solución para ser utilizada en ordenadores convencionales y teléfonos móviles para poder enviar y visualizar los datos.
La consulta de datos se realiza a través de una plataforma web en la que se pueden configurar los paneles de visualización y crear sentencias y alertas.

En el ámbito de la investigación existe un proyecto desarrollado por *Carnegie Mellon University* (CMU) llamado ESDR (*Environmental Sensor Data Repository*) [8]. ESDR es un repositorio de datos *open source*, destinado al almacenamiento y recuperación de datos de series de tiempo ambientales. Los datos almacenados deben tener una marca temporal y un valor. ESDR se encarga de almacenar y proporcionar maneras de recuperar la información de forma rápida y segura. Los datos se almacenan en un almacén de datos desarrollado por CMU, de código abierto, que ofrece inserciones de datos de manera rápida. También permite la recuperación de datos de manera rápida, lo que permite hacer visualizarlos. El sitio web ESDR proporciona una interfaz API REST para el almacén de datos, por lo que es fácil leer o escribir datos. Los metadatos se almacenan en una base de datos MySQL.

ESDR es un repositorio de datos que permite la visualización no sólo para el CMU, sino también para cualquier persona que desee almacenar los datos y tener herramientas para visualizarlos fácilmente.

Este proyecto está fuertemente influenciado por Xively, de tal manera que muchos de sus términos se parecen a los de Xively. El catálogo de ESDR consta de clientes, usuarios, productos, dispositivos, *feeds* (instalaciones particulares de dispositivos), canales y *tiles* (pequeños fragmento de los datos de un canal). La creación de este catálogo se realiza mediante llamadas HTTP a la API REST. Para poder realizar estas llamadas hay que utilizar algún cliente que ofrezca este servicio.

El proyecto presentado en esta memoria pretende ofrecer un servicio para mantener un catálogo, al igual que estas soluciones, además de almacenar los datos provenientes de los sensores. También ofrecerá herramientas para la recuperación y visualización de los datos y la creación del catálogo mediante interfaces gráficas de usuario.

A diferencia de las soluciones comerciales, este proyecto es gratuito y de dominio público. Además de las opciones ofrecidas por las soluciones descritas a lo largo de este apartado, este proyecto permite la comunicación directa con el sensor, ofreciendo así un sistema práctico, versátil y funcional en el ámbito de la investigación, permitiendo el análisis y el estudio del comportamiento de los sensores.

2.3. Soluciones completas

El mundo del Internet de las Cosas está en auge y lo demuestra el interés suscitado por grandes compañías que han comercializado soluciones completas destinadas a este ecosistema.

Estas soluciones ofrecen desde el hardware para comunicar los sensores, pasando por el software necesario para programar estos dispositivos, hasta una nube privada en la que poder gestionar, monitorizar y analizar los datos.

A continuación se muestran dos ejemplos de grandes multinacionales que ofrecen productos de estas características:

- **Samsung Artik** [26]: La multinacional Samsung dispone de un producto que combina el hardware y el software para ayudar a fabricar rápida y fácilmente dispositivos que se conectan a Internet. Este producto consta de distintas partes.
  Por un lado Artik, se trata de un procesador que tiene tres variantes: Artik 1, Artik 5 y Artik 10, cada uno con más poder de procesamiento y con habilidades más complejas que el anterior. Estos dispositivos se pueden usar desde monitores hasta centrales de hogar inteligente. De esta manera ofrecen un dispositivo con capacidad para actuar con sensores y dotar de conectividad a éstos, desde el BlueTooth que incorpora Artik 1 hasta WiFi o ZigBee incorporado en Artik 10.
  Estos microprocesadores son compatibles con Arduino, lo que facilita que pueda ser programado con el IDE de Arduino y se beneficie de una comunidad que es extraordinariamente activa en este ámbito.
  Por otro lado, ofrece servicios en la nube denominado Artik Cloud Service, que consta de un conjunto de herramientas creadas para ayudar a que los desarrolladores dispongan de los medios para recolectar, almacenar y analizar los datos provenientes de un amplio número de sensores de todo tipo.
  Dentro de Artik Cloud Services están soportados un amplio número de sensores y se proporcionan *plugins* para gestionar diversas plataformas hardware, programas, y sistemas operativos. Es un servicio de hardware agnóstico ya que la comunicación de mensajes entre los dispositivos y la nube se realiza a través de una API REST y de WebSocket así como protocolos MQTT y CoAP. De esta manera, cualquier dispositivo con posibilidad para realizar y gestionar este tipo de comunicación es capaz de establecer la conexión y enviar los datos. También facilita ejecutar tareas o enviar alertas cuando ocurre algún evento.
  El servicio permite la visualización de datos y ofrece herramientas de gestión de dispositivos, capacidades de análisis y kits de desarrollo de software. Dentro del conjunto de herramientas cabe citar la inclusión de desarrollo, basado en web, de módulos Arduino, así como Temboo que facilita la parte de conectividad en la nube y generación automática de código de forma muy simple y con soporte de un amplio número de lenguajes, aplicaciones y sensores.

- **IBM Watson** [28]: La multinacional *IBM* dispone de un ecosistema creado para el Internet de las Cosas que permite la gestión y conectividad de dispositivos, el almacenamiento de los datos en la nube, análisis en tiempo real y servicios cognitivos.
  Para ello, disponen de una dispositivo desarrollado por la empresa *ARM MBED* que consiste en un kit de iniciación compuesto por una placa de desarrollo con conexión Ethernet. Dispone de acelerómetros, un sensor de temperatura y facilidad para comunicar el microcontrolador con Arduino.
  Además, ofrecen una plataforma en la nube que permite el almacenamiento de datos y la visualización de éstos creando paneles de administración para monitorizar los datos. Para ello, es necesario el envío de los datos desde los dispositivos a la nube. Este envío se puede realizar a través de protocolo HTTP o MQTT gracias a la API que expone la nube para la recepción de paquetes. Ofrecen SDKs para el lenguajes como *Python, Java* o *C* para facilitar la conexión entre los dispositivos y la nube.
  Los servicios ofrecidos disponibles se encuentran dentro de su plataforma *Bluemix* que está conectada con *IBM Watson Internet of Things,* lo que permite utilizar herramientas como la visualización en tiempo real de los datos o el análisis de éstos usando herramientas que lanzan alertas según las condiciones previamente fijadas. Además, el panel de visualización es configurable para mostrar las analíticas deseadas.

Estas soluciones son de pago y no son tan flexibles a la hora de desarrollar soluciones personalizadas. Esto hace que no sean interesantes para investigadores que persiguen la agilidad a la hora de manipular los servicios de manera que se puedan adaptar a un problema concreto.

3. Tecnologías utilizadas en el proyecto

En este capítulo se exponen y detallan las tecnologías utilizadas en el desarrollo del proyecto. Toda la comunicación se realiza mediante el protocolo HTTP (Apartado 2.2.3) gracias a la utilización de API REST (Apartado 2.2.3.3), por lo que es necesario separar las tecnologías en dos partes: tecnologías utilizadas para el desarrollo de la nube y tecnologías empleadas en el desarrollo de la interfaces gráficas de usuario (GUI).

Las tecnologías empleadas en la nube conforman la capa de acceso a datos y son:

- Java, con el uso del *framework* Spring para la creación de API REST.
- Python, empleando el *framework* Django en la creación de API REST.
- Bases de datos: Cassandra y SQLite (Apartado 2.2.4).
- Servidor Nginx para ofrecer los servicios de la nube.

La capa de presentación de los datos muestra las interfaces gráficas. Las tecnologías utilizadas son:

- Swagger para facilitar la creación de llamadas HTTP a las API REST mediante una interfaz gráfica (Apartado 4.1.5.1).
- HTML, CSS y JavaScript, esta última tecnología empleando el *framework* AngularJS para el desarrollo de la GUI que permita la monitorización de los datos.

3.1. Capa de acceso a datos

En este apartado se presentan todas las tecnologías empleadas para el desarrollo de los servicios ofrecidos en la nube. Como se muestra en el Apartado 4.1 el diseño del sistema está modulado en distintos servicios, por lo que se proceden a explicar los lenguajes de programación y los *frameworks* empleados en cada servicio. El apartado finaliza con el servidor utilizado para ofrecer estos servicios.

Las bases de datos empleadas son **Cassandra** (Apartado 2.2.4.2) y **SQLite** (Apartado 2.2.4.1). El diseño y la utilización de cada tecnología será explicada en el capítulo 4: Diseño y desarrollo.

3.1.1. Python

Python [65] es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible. Python está administrado por la Python Software Foundation y posee licencia de código abierto. Se trata de un lenguaje de programación multiparadigma ya que soporta:

- Orientación a objetos: usa objetos en sus interacciones para diseñar aplicaciones y programas informáticos.
- Programación imperativa: describe detalladamente un conjunto de instrucciones que le indican al computador cómo realizar una tarea.
- Programación funcional: está basado en la utilización de funciones aritméticas que no maneja datos mutables o de estado.

Python es un lenguaje interpretado, lo que significa que no se necesita compilar el código fuente para poder ejecutarlo. Además, usa tipado dinámico lo que permite que una misma variable pueda tomar valores de distinto tipo en distintos momentos. Python es multiplataforma.

Python ofrece multitud de librerías desarrolladas para los paradigmas más comunes, como la realización de conexiones HTTP y el envío de paquetes.

Este lenguaje será empleado en el desarrollo de los códigos a ejecutar dentro del dispositivo interfaz encargado de capturar las muestras de los sensores y enviar estos valores a la nube, como se muestra en el Apartado 4.2.1.2.

Además, Python se usará para desarrollar la API REST (Apartado 2.2.3.3) encargada de gestionar los elementos y programas que conforman el catálogo. Para desarrollar esta API REST se emplea el *framework* Django que se explica a continuación.

3.1.1.1. Django

Django [13] es un *framework* de desarrollo web de código abierto, escrito en Python, que respeta el patrón de diseño conocido como modelo–vista–controlador. Su finalidad es facilitar la creación de sitios web complejos. Django pone énfasis en el re-uso, la conectividad y extensibilidad de componentes, el desarrollo rápido y el principio "No te repitas". Django tiene una comunidad activa en internet que facilita muchas piezas de código para no tener que desarrollarlas. Django, para facilitar el desarrollo rápido, utiliza SQLite por defecto, a no ser que se indique otra base de datos, por lo que no hace falta instalar software adicional y agiliza el desarrollo. Además, posee conectores para las bases de datos relacionales más comunes como MySQL y PostgreSQL.

Gracias a la comunidad existen *frameworks* que ayudan a Django a ser más potente para el desarrollo. En este proyecto se utilizará Django Rest Framework [14] para la creación de la API REST. Este framework ofrece un conjunto de funciones y librerías para crear Web APIs de una manera muy rápida y sencilla, de manera que el enrutado, serializadores y demás componentes necesarios para procesar llamadas HTTP se ofrecen como librerías de Django Rest Framework, agilizando así el desarrollo.

3.1.2. Java

Java [73] es un lenguaje de programación de propósito general, concurrente, orientado a objetos. Este lenguaje fue desarrollado por James Gosling de Sun Microsystems, que posteriormente fue adquirida por Oracle. Java se diseñó para ser compilado una vez y que pudiera ser ejecutado en cualquier dispositivo gracias a la máquina virtual Java (JVM).

La sintaxis de Java deriva en gran parte de los lenguajes de programación C y C++, pero tiene menos utilidades de bajo nivel que ellos.

Las aplicaciones de Java son compiladas a *bytecode*, una clase Java, que puede ejecutarse en cualquier máquina virtual Java sin importar la arquitectura de la computadora en la que se ejecuta. De esta manera, se consigue cumplir la filosofía WORA ("*write once, run anywhere*") con lo que el desarrollador sólo ha de preocuparse de desarrollar la aplicación y la JVM se encargará de ejecutarla en cualquier computadora.

Esta tecnología será empleada en el desarrollo de la API REST encargada de recibir los parámetros de los programas en el dispositivo interfaz, como se explica en el Apartado 4.2.1.1. También se utiliza Java para la creación del servicio en la nube encargado de almacenar los datos provenientes de los sensores mediante una API REST. Estas APIs se desarrollan con el framework Spring, explicado a continuación.

3.1.2.1. Spring

Spring Framework [70] es una plataforma que proporciona una infraestructura que actúa de soporte para desarrollar aplicaciones Java. Spring maneja toda la infraestructura de manera que el desarrollador sólo se ha de centrar en desarrollar la aplicación.

Spring es bastante complejo y por ello está dividido en distintos módulos. De esta manera, no es necesario utilizar todos los módulos en un proyecto. En este proyecto se usarán los módulos de Spring MVC para la parte web, el módulo de acceso a la base de datos llamado Spring Data.

Además, Spring proporciona un módulo llamado Spring Boot que ofrece toda la configuración básica para poder desarrollar un proyecto Spring. Además, Spring Boot genera una aplicación Java autónoma, esto es, Spring se encarga de embeber dentro de la aplicación un servidor capaz de desplegar aplicaciones Java con un servidor HTTP.

Spring se caracteriza por el uso de etiquetas en el código que se encargan de generar el código indicado mediante la etiqueta en la compilación. Además, Spring permite modificar toda la configuración de una manera muy sencilla, pudiendo así ajustar el proyecto a las necesidades de la aplicación.

3.1.3. Nginx

Nginx [69] es un servidor web o HTTP ligero, de alto rendimiento que además permite ser utilizado como servidor proxy. Un servidor web es un programa informático capaz de ejecutar programas del lado del servidor, así como, establecer conexiones bidireccionales con cualquier cliente que se conecte a dicho servidor, generando respuestas dinámicas. Es decir, Nginx permite alojar sitios web o aplicaciones web en cualquier servidor remoto y poder compartirlos en Internet.

Nginx es software libre y de código abierto. Al ser un servidor ligero ofrece una gran velocidad, lo que permite ofrecer aplicaciones a mayor velocidad convirtiendo a Nginx en un servidor de alto rendimiento. Además, Nginx también permite actúa como servidor de archivos estáticos.

Este servidor se utiliza en este proyecto para ofrecer los servicios desarrollados de una manera rápida. Además de exponer las API REST, la descarga del código de la capa de presentación también se realiza gracias a este servidor, ya que son archivos estáticos.

3.2. Capa de presentación

En este apartado se presentan las tecnologías utilizadas en el desarrollo de la interfaz gráfica de usuario creada para la visualización de los datos y la configuración de las pruebas. Esta interfaz está detallada en el Apartado 4.1.5.2.

3.2.1. HTML

HTML (HyperText Markup Language) [25] hace referencia al lenguaje utilizado en la elaboración de páginas web. Es un estándar que sirve de referencia en la elaboración de páginas web. Define una estructura básica y un código para la definición de contenido de una página web. Es el estándar que se ha impuesto en la visualización de páginas web y es el que todos los navegadores actuales han adoptado.

Con el uso de este lenguaje, la página web contiene solamente texto y recae en el navegador web interpretar el código para unir todos los elementos y visualizar la página final. El texto representado con HTML hace una referencia a la ubicación del elemento y el navegador se encarga de insertar el elemento.

Este lenguaje está mantenido por el World Wide Web Consortium (W3C), organización dedicada a la estandarización de casi todas las tecnologías ligadas a la web, sobre todo en lo referente a su escritura e interpretación. Se considera el lenguaje web más importante siendo su invención crucial en la aparición, desarrollo y expansión de la World Wide Web (WWW).

3.2.2. CSS

CSS (Cascading Style Sheets, Hoja de estilo en cascada) [9] es un lenguaje usado para definir y crear la presentación de un documento estructurado escrito en HTML. Este lenguaje pretende separar la estructura de un documento de su presentación. La información de estilo puede ser definida en un documento separado o en el mismo documento HTML.

El W3C es el encargado de formular la especificación de las hojas de estilo que sirven de estándar a los navegadores.

CSS ofrece ventajas como son el control centralizado de la presentación de un sitio web y optimiza el ancho de banda de la conexión, ya que pueden definirse los mismos estilos para distintos elementos utilizando así el mismo archivo CSS para distintos documentos.

3.2.2.1. Bootstrap

Bootstrap [5] es un conjunto de herramientas de código abierto para el diseño de sitios y aplicaciones web. Ofrece plantillas de diseño con elementos de diseño basados en HTML y CSS.

Bootstrap fue desarrollado por Twitter para fomentar la consistencia a través de herramientas internas acelerando así el desarrollo de los proyectos. En agosto de 2011 Twitter liberó el código. Además, Bootstrap se centra en el desarrollo responsivo, de manera que cualquier aplicación o sitio web pueda ser visualizado en cualquier dispositivo.

Este proyecto pretende poder mostrarse en cualquier dispositivo, por lo tanto usará esta librería en el desarrollo de las interfaces de usuario.

3.2.3. JavaScript

JavaScript [31] es un lenguaje de programación interpretado, orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico. JavaScript se diseñó con una sintaxis similar a C, aunque adopta nombres y convenciones del lenguaje de programación Java. Sin embargo, Java y JavaScript tienen semánticas y propósitos diferentes.

La utilización de este lenguaje es principalmente en el lado del cliente, utilizado en páginas web HTML, implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas. Todos los navegadores modernos interpretan el código JavaScript integrado en las páginas web. JavaScript se interpreta en el navegador al mismo tiempo que se descargan las sentencias junto con el código HTML.

También existe una forma de JavaScript del lado del servidor.

3.2.3.1. AngularJS

AngularJS [2], es un framework de JavaScript de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página (SPA). Una aplicación de una única página ofrece una mejor experiencia de usuario ya que son páginas dinámicas que no necesitan ser recargadas. Todo el código se descarga una única vez y es el código JavaScript el encargado de comunicarse con el servidor para descargar los datos que precise la página.

Angular se basa en patrón modelo-vista-controlador de manera que separa las vistas, los archivos HTML, de la lógica de la aplicación, archivos JavaScript. Estos archivos JavaScript contienen directivas y etiquetas que se comunican con la vista. De esta manera, el

controlador puede actualizar la vista mediante "*data binding*". Además, esta técnica es recíproca, por lo que si la vista modifica una variable el controlador recibe este cambio.

AngularJS, al tratarse de un *framework* para crear SPAs, el controlador y el modelo se mantienen en el navegador del cliente, generando nuevas vistas sin interacción alguna con el servidor.

El empleo de AngularJS conlleva el uso de ciertas herramientas para facilitar el desarrollo. Estas herramientas son:

- **Bower** [6]: gestor de paquetes de JavaScript utilizado para instalar librerías en el proyecto.
- **Grunt** [23]: herramienta para compilar el proyecto y gestionar y automatizar tareas.
- **Yeoman** [71]: generador de código para crear un proyecto desde el inicio definiendo una estructura para la aplicación.

Este *framework* se utilizará para crear la interfaz de usuario para la gestión y visualización de las pruebas realizadas sobre los sensores, ya que como se describe en el Apartado 4.1.5.2. correspondiente al diseño, se pretende conseguir una mejor experiencia de usuario en la monitorización de los sensores.

4. Diseño y desarrollo

En este capítulo se abordará una descripción del proceso, estructura y desarrollo seguido en la elaboración de la plataforma para la comunicación y monitorización de sensores. En primera instancia se detallan los requisitos y la arquitectura que debe cumplir la plataforma para después adentrarse en el desarrollo detallado de cada pieza que compone dicha arquitectura.

4.1. Diseño

En este apartado queda reflejado el diseño propuesto para desarrollar el sistema completo. Por un lado, hay que realizar la comunicación entre el sensor y la plataforma encargada de almacenar los datos. Por otro, se presenta el planteamiento para poder llevar a cabo la gestión de los elementos implicados en el proceso de comunicación entre la plataforma y el sensor, así como las interfaces gráficas necesarias para poder desplegar una prueba y monitorizar los valores devueltos por esta prueba en tiempo real.

El diseño consta de varios módulos con distinta funcionalidad, por lo que primero se presenta la arquitectura completa y a continuación, el diseño en profundidad de cada módulo por separado.

Todas las tecnologías y los dispositivos empleados son de libre distribución, cumpliendo así con la filosofía de dominio público del proyecto.

4.1.1. Arquitectura general del sistema

El diseño planteado para la arquitectura presentada en este proyecto se basa en **microservicios** (Apartado 2.2.3.1) debido a la modularidad que ofrece este tipo de arquitectura. Las funcionalidades se separan en distintos servicios según su uso y los requisitos necesarios para poder desarrollar y cumplir con su objetivo.

La principal característica de los microservicios es la facilidad y la velocidad de añadir nuevos módulos, ya que un módulo es un servicio que se comunica con el resto. Si un servicio deja de funcionar o no se usa no afecta al resto de funcionalidades, lo que proporciona mayor consistencia e independencia en el desarrollo de la plataforma. Por ello, un servicio puede usarse o no dependiendo de las necesidades. Además, gracias a la facilidad para la escalabilidad horizontal que presentan estas arquitecturas, se puede alojar cada servicio en una máquina distinta con las especificaciones que mejor se ajusten al servicio, abaratando costes y proporcionando una mejora en el rendimiento general del sistema.

La arquitectura general del sistema se refleja en la Figura 4.1. En ella se pueden diferenciar las tres partes de la arquitectura:

- El dispositivo que actúa de interfaz entre el sensor y la nube.
- La plataforma software (la nube), cuyas tareas son almacenar los datos del sensor y gestionar el registro de sensores, dispositivos y modelos de análisis que se pueden ejecutar en el dispositivo, creando así un catálogo de los elementos y programas del sistema.
- Interfaz gráfica de usuario para visualizar los resultados de las pruebas y administrar el catálogo de componentes y pruebas.

La comunicación entre todas estas partes se realiza mediante protocolo HTTP (Apartado 2.2.3) debido a la estandarización y a la velocidad en las comunicaciones que presenta este protocolo. Por tanto, los servicios deben ofrecerse mediante API REST (Apartado 2.2.3.3) para establecer la conexión HTTP y crear el protocolo de comunicación. El empleo de API REST contribuye y facilita la escalabilidad gracias a que es un protocolo que no guarda el estado de la comunicación. Toda la información necesaria para comprender la petición está contenida en el mensaje HTTP, por lo que se pueden responder todas las operaciones y escalar el servicio.

La modularidad que ofrece el enfoque de microservicios permite la creación de dos servicios distintos para el desarrollo de la nube, pudiendo ser los servicios explotados de distintas formas y aplicar la configuración y las tecnologías que mejor se ajusten a los requisitos de cada uno de ellos.

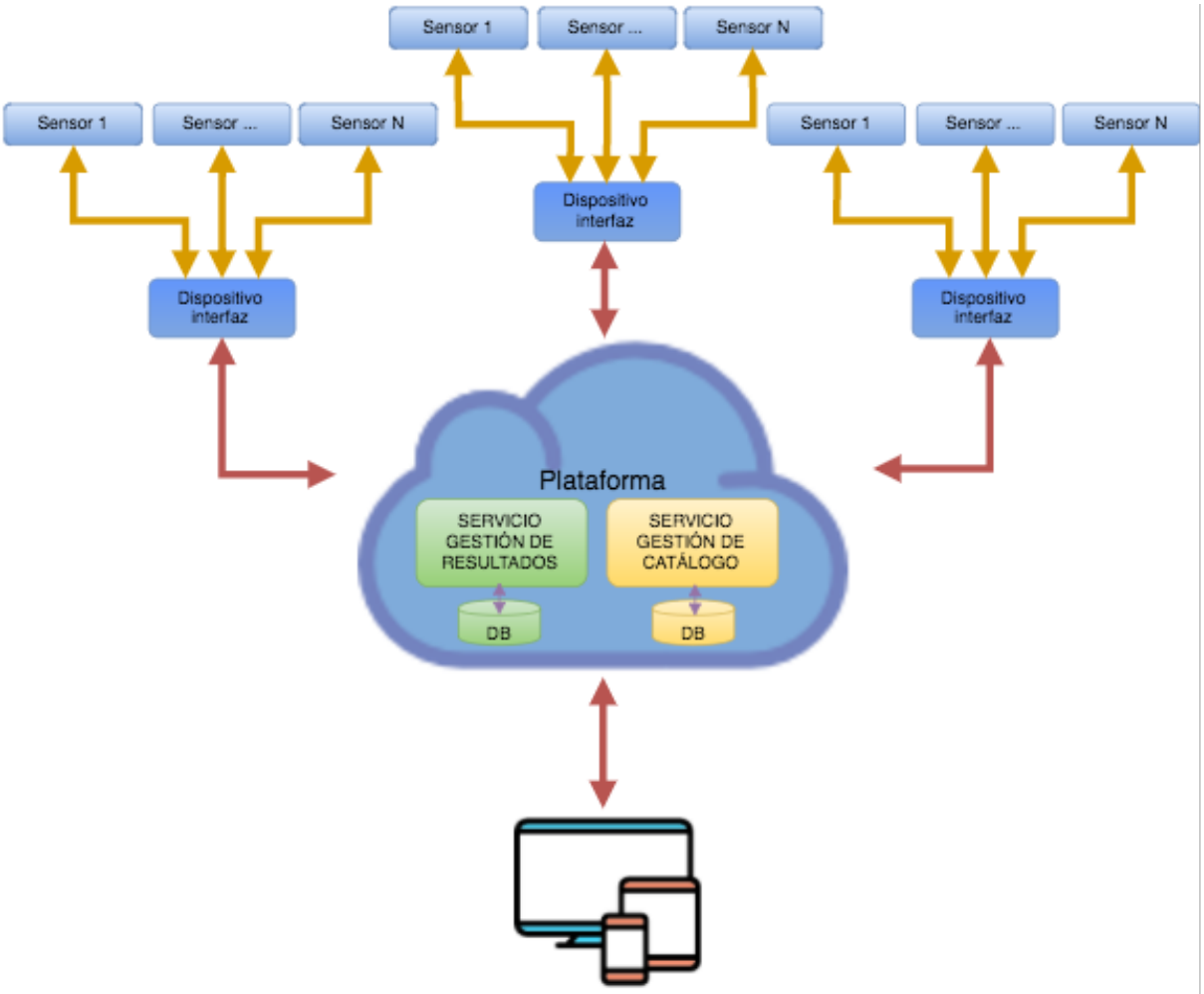

Figura 4.1: Arquitectura general del sistema. La comunicación HTTP se representa con flechas de color rojo. Las flechas de color amarillo representan la comunicación propia entre el dispositivo y los sensores.

4.1.1.1. Requisitos de los servicios

El **dispositivo que actúa como interfaz** debe enviar los datos capturados por los sensores a la plataforma, a través de Internet, por tanto, debe ser capaz de capturar los valores tomados por el sensor y empaquetarlos para ser enviados mediante el protocolo HTTP. Además, debe exponer una API REST capaz de recibir paquetes HTTP con el contenido de la configuración necesaria para arrancar el modelo que se desea ejecutar en la adquisición de los datos.

El **servicio de gestión de los resultados** de los sensores debe ser capaz de establecer comunicación HTTP con el dispositivo que actúa de interfaz entre el sensor y la nube mediante una API REST. Esta API REST es la encargada de recibir los datos con el contenido del valor del sensor y almacenarlos. También debe poder devolver la información persistida, de manera que los datos puedan ser consumidos y monitorizados en tiempo real.

El **servicio que gestiona el catálogo** de los elementos del sistema (Tabla 4.1) también debe exponer una API REST para poder procesar peticiones HTTP y realizar las operaciones de alta, baja, modificación y consulta (*Create-Read-Update-Delete*, CRUD) sobre cada componente del catálogo. La existencia de este servicio simplifica la creación de pruebas, ya que ofrece un inventario de los dispositivos que ejercen de interfaz entre la plataforma y los sensores, los propios sensores y los modelos disponibles en los dispositivos para interactuar con el sensor y enviar sus datos, facilitando al usuario la información necesaria para configurar las pruebas. Este servicio, al crear una prueba para ejecutar un modelo que interaccione con el sensor, ha de poder enviar además la configuración al dispositivo que inicie el programa asociado a ese modelo.

Tabla 4.1: Elementos que componen el catálogo de la aplicación.

Elementos del catálogo
Sensores
Dispositivos interfaz
Representación de los programas de manejo de sensores
Ejecuciones de los programas realizadas sobre sensores

La **interfaz gráfica de usuario (GUI)** se divide en dos partes: la primera debe proporcionar una interfaz para la administración del catálogo, esto es, realizar el CRUD de los elementos de la base de datos, y la segunda, debe ofrecer una interfaz amigable para crear las pruebas con la configuración de los parámetros del programa a ejecutar y visualizar los resultados de la prueba. La separación de las interfaces permite la creación de dos roles: el administrador de los elementos del sistema (catálogo) y el usuario encargado de realizar las pruebas.

4.1.1.2. Funcionalidad global

La comunicación del usuario con el sensor se realiza mediante la nube, ofreciendo una interfaz gráfica amigable para llevar a cabo la configuración y ejecución de distintos programas destinados a la captura y análisis de los datos de los sensores. La interfaz se comunica con el servicio que gestiona el catálogo obteniendo los dispositivos en los que se pueden lanzar los programas y configura los parámetros necesarios para ejecutar una prueba. La configuración se envía a este servicio, el cual se comunica con el dispositivo y envía la configuración para ejecutar la prueba. El programa lanzado que realiza la prueba se encarga de mandar el valor capturado por el sensor al servicio de almacenamiento de resultados. Por otro lado, la interfaz consume de este segundo servicio los datos para así poder monitorizar los sensores y visualizar los resultados de las pruebas. Este flujo de comunicaciones queda reflejado en la Figura 4.2.

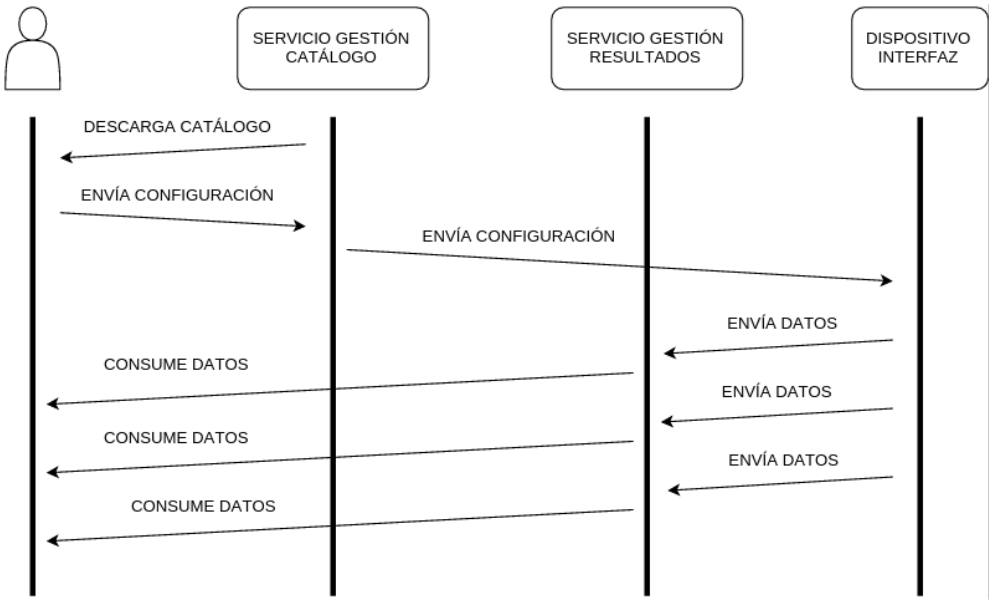

Figura 4.2: Flujo de creación y ejecución de una prueba.

La existencia de distintos servicios implica seguir un orden lógico y secuencial a la hora de crear el catálogo y el programa que se ejecuta dentro de cada dispositivo. Esto es importante, ya que para lanzar un programa que interaccione con el sensor, es necesario disponer del identificador del modelo y la estructura de los parámetros del modelo del programa. Por lo tanto, el orden a la hora de desarrollar el programa es el siguiente:

1. Crear el modelo en el catálogo y obtener el identificador del modelo.
2. Añadir los parámetros correspondientes al modelo en el catálogo.
3. Crear la entidad necesaria que se corresponde con el modelo creado en el catálogo, en la API REST embebida dentro del dispositivo.
4. Crear el *endpoint* que lanza el programa para capturar y enviar datos del sensor. El *endpoint* está identificado por el ID devuelto al crear el modelo en el catálogo.

Estos pasos están detallados en el Apartado 4.2.5, correspondiente al desarrollo.

4.1.2. Comunicación del sensor

El sensor por sí sólo carece de conexión a Internet para establecer conexiones HTTP necesarias en el envío de los datos capturados y la recepción de la configuración. Para solventar este problema, se ha de usar algún dispositivo capaz de enviar los datos a la plataforma software, así como de establecer la comunicación con el sensor y capturar los datos.

Del mismo modo, este dispositivo ha de poder comunicarse con cualquier tipo de sensor, ya sea por la lectura directa del sensor o por el acceso a través de algún bus de datos como por ejemplo, SPI o I^2C (Apartado 2.1.1.3). Este dispositivo ha de continuar la filosofía del proyecto en la utilización de código y plataformas de libre distribución. El diseño propuesto otorga universalidad en el uso de sensores y permite el control sobre el dispositivo de interfaz entre la plataforma software y el sensor o conjunto de sensores.

4.1.2.1. Funcionalidades del dispositivo

El envío de datos tomados por los sensores se realiza a través del dispositivo interfaz mediante el protocolo HTTP con una petición POST, cuyo cuerpo contiene la estructura detallada en el Apartado 4.1.3.2. El envío se realiza con un programa que se ejecuta en el dispositivo capaz de comunicarse con el sensor y procesar las peticiones HTTP. La plataforma recibe y almacena los datos pertenecientes a cada sensor y a cada ejecución de un modelo de análisis determinado. Este flujo queda reflejado en la Figura 4.3.

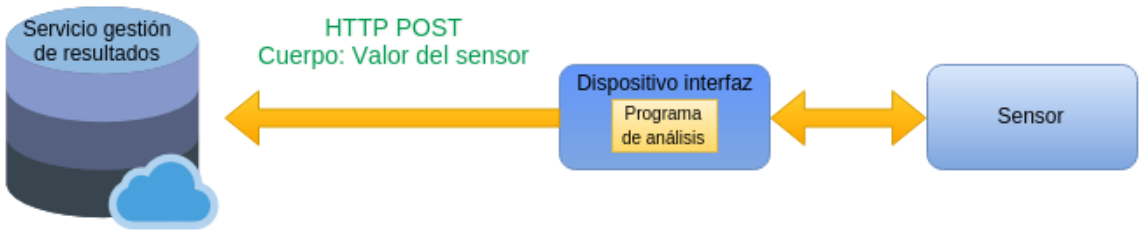

Figura 4.3: Representación del envío de datos a la plataforma mediante una llamada HTTP POST de un programa embebido en el dispositivo interfaz.

La tasa del envío de paquetes con la información perteneciente al valor del sensor viene determinado por el modelo a ejecutar y/o la configuración determinada por el usuario que ejecuta el test.

La recepción de los parámetros de configuración de un modelo por parte del dispositivo se realiza mediante protocolo HTTP. El dispositivo debe exponer una API REST que permita la comunicación con la plataforma. Este flujo queda reflejado en la Figura 4.4.

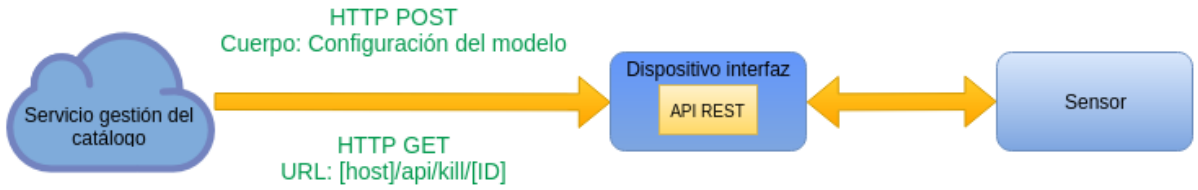

Figura 4.4: Representación de la recepción de los parámetros de un modelo por parte del dispositivo interfaz gracias a una API REST. También se representa la recepción de la petición para detener una prueba.

Además de poder recibir la configuración para ejecutar una prueba sobre un modelo, es necesario poder detener la prueba de forma remota. Para esto, el dispositivo debe aceptar una petición tipo GET con el identificador del proceso del programa que ejecuta la prueba y poder pararlo (Figura 4.4). De esta manera, se cancela el envío de datos a la plataforma y se detiene el programa que ejecuta el modelo correspondiente al identificador recibido.

4.1.2.2. Tecnologías

Los requisitos del dispositivo conllevan a la elección de una **Raspberry Pi** (Apartado 2.1.2.2), ya que se trata de un dispositivo *open source* y dispone de un bus de datos SPI, I^2C y UART, así como de pines de propósito general capaces de comunicarse con el sensor directamente y su precio es reducido en comparación a sus competidores. Además, al tratarse de un ordenador de placa reducida permite ejecutar distintos programas, por lo que se pueden realizar múltiples análisis al mismo tiempo utilizando el mismo dispositivo y conectar distintos sensores.

El lenguaje utilizado en los **programas** que ejecutan los modelos para la captura de datos de los sensores, las operaciones que se deseen realizar con esos datos en el dispositivo y el envío de los valores capturados a la plataforma será **Python** (Apartado 3.1.1). Python es un lenguaje interpretado con una sintaxis muy simple, por lo que la curva de aprendizaje es muy progresiva y permite crear programas completos en pocas líneas. Estas características favorecen la elección de Python frente a Java, ya que Java necesita crear un proyecto completo y más complejo y las librerías desarrolladas para tomar las muestras de los pines GPIO o la utilización de buses de datos comunes no están tan avanzadas.

En cambio, la tecnología empleada en la creación de la **API REST** para procesar las peticiones HTTP es **Java** (Apartado 3.1.2) con el uso del framework **Spring** (Apartado 3.1.2.1), ya que no precisa de una base de datos para crear una API REST, frente a otros lenguajes como Python con el framework de Django. Spring es un framework que agiliza la creación de proyectos Java, puesto que dispone de una configuración básica predeterminada que facilita el desarrollo, pudiendo modificar esta configuración de manera sencilla, y dispone de módulos que facilitan la creación de este tipo de servicios.

Con el uso de Spring Boot se generan aplicaciones autónomas, esto es, Spring se encarga de embeber dentro de esta aplicación un servidor **Apache Tomcat**, un contenedor para desplegar aplicaciones Java con un servidor HTTP. Además, la configuración del servidor puede ser modificada mediante un archivo externo de configuración. Al tratarse de una

aplicación autónoma, no es necesario instalar ningún servidor dentro del dispositivo para poder recibir las peticiones, con lo que se consigue empaquetar todo lo necesario para la ejecución de un programa en un mismo fichero. Este es el principal motivo de la elección de Java frente a otras tecnologías como pueden ser Python o Rubi.

El empleo de estos lenguajes de programación permite que los servicios sean exportados a cualquier dispositivo capaz de ejecutarlos, proveyendo así una solución completa para la comunicación con la plataforma y con el sensor.

4.1.3. Servicio de gestión de los resultados de sensores

La plataforma software debe recibir y almacenar los datos enviados por el sensor. Por ello debe contener un servicio específico que exponga una API REST capaz de recibir estas peticiones y almacenarlas en la base de datos. Esta base de datos debe a su vez almacenar la información de una manera rápida para no colapsar la recepción de los datos.

4.1.3.1. Funcionalidad del servicio

La API REST expuesta debe poder realizar las siguientes tareas para satisfacer los requisitos:

- Recepción de los datos mediante una llamada POST cuyo cuerpo del mensaje contiene el valor y la identificación del sensor, así como la fecha y la prueba que ha generado ese valor, y guardar esta información en base de datos. Este flujo queda reflejado en la Figura 4.5.

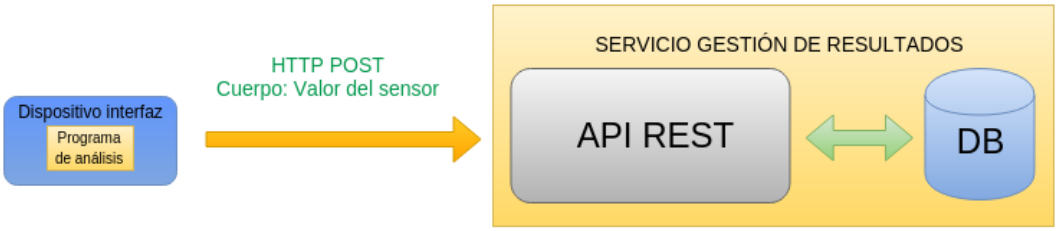

Figura 4.5: Representación de la recepción de los datos mediante una llamada HTTP POST por parte de la API REST

- Devolver el conjunto de los datos almacenados de manera paginada, de manera que se devuelve una lista de elementos de un tamaño determinado. El consumo de datos es a través del método GET (Figura 4.6).

- Descarga de todos los datos en formato CSV pertenecientes a una prueba mediante una petición GET (Figura 4.6).

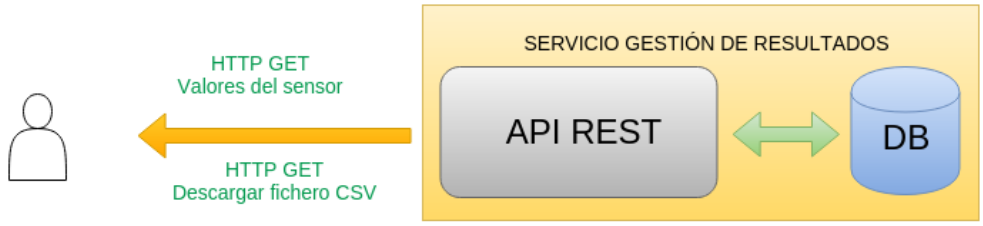

Figura 4.6: Representación del envío de datos y la descarga de ficheros mediante llamadas HTTP GET por parte de la API REST.

4.1.3.2. Estructura de la información almacenada

La información enviada a la plataforma por parte de los sensores y que se almacena en el servicio debe contener la siguiente información:

- Identificador de la prueba realizada
- Fecha de la recepción
- Identificador de la placa que contiene el sensor
- Identificador del sensor al que pertenecen los datos
- Valor aplicado al sensor
- Valor devuelto por el sensor
- Fecha de la toma del valor asociado

La base de datos consiste en una única tabla que debe almacenar dicha información.

4.2.3.3. Tecnologías

La base de datos escogida es **Cassandra** (Apartado 2.2.4.2) debido a que está indicada para almacenar grandes volúmenes de datos, como los provenientes de sensores. Además, su tipología de híbrido entre un modelo clave-valor y una base de datos tabular (orientada a columnas) permite la creación de una súper tabla en la que almacenar todos los datos y buscarlos por el identificador de la prueba, otorgando mayor velocidad. Además, al tratarse de una base de datos no relacional permite la ingesta masiva de información sin tener que crear relaciones, lo que proporciona un aumento de velocidad en las transacciones.

La API REST, al igual que la API REST embebida en el dispositivo que ejerce de interfaz entre el sensor y la plataforma, está escrita en **Java** (Apartado 3.1.2) usando el framework de **Spring** (Apartado 3.1.2.1) debido a que existe un módulo muy optimizado llamado “Spring Data Cassandra” que facilita y aumenta el rendimiento de la conexión con la base de datos, frente a otras tecnologías como Python que no disponen de librerías optimizadas para esta base de datos. Este módulo ofrece un conector creado para la comunicación con la base de datos Cassandra y proporciona una capa de abstracción en el acceso a la capa de persistencia sin tener que ocuparse de la transaccionalidad de las operaciones con la base de datos. Además, con la utilización del módulo “Spring Data REST” se crea, de una manera muy sencilla y con pocas líneas de código, la configuración y el API REST

necesario para la comunicación HTTP entre el servicio y los clientes. El uso de este módulo permite así mismo publicar métodos de la capa de persistencia como recursos REST. Todo esto facilita el desarrollo y agiliza el proceso de creación de los servicios frente a otras tecnologías, como por ejemplo Python, que no disponen de conectores optimizados para la base de datos Cassandra, lo que reduce la capacidad de procesamiento del servicio.

Nuevamente, el uso de Spring Boot genera una aplicación autocontenida con un servidor **Apache Tomcat**, por lo que este servicio no requiere mayor configuración para ser desplegado.

4.1.4. Servicio de gestión del catálogo de la aplicación

La plataforma debe gestionar las pruebas realizadas, los sensores y los dispositivos utilizados para dotar de conectividad a los sensores, al mismo tiempo que debe administrar la información del modelo de análisis utilizado para las pruebas. Con todo esto, la información del sistema queda inventariada y catalogada para disponer de los datos relacionados para un análisis posterior.

4.1.4.1. Funcionalidades del servicio

La API REST expuesta tiene que poder realizar las siguientes tareas para satisfacer los requisitos:

- Aceptar la información de cada elemento del catálogo y almacenarla en la base de datos. En este catálogo, al crear un test, la plataforma se encargará de comunicar a cada dispositivo qué modelo ha de ejecutar y la configuración a aplicar para ese modelo.
- Devolver dicha información cuando sea solicitada por cada cliente.
- Permitir actualizar la información de cada elemento.
- Eliminar los datos cuando sea requerido.

Toda esta funcionalidad queda reflejada en la Figura 4.7.

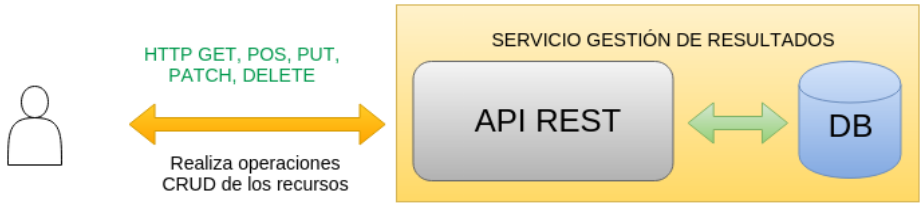

Figura 4.7: Representación de las peticiones necesarias para realizar el CRUD de los recursos.

4.1.4.2. Estructura de la información almacenada

Atendiendo a las especificaciones de la Tabla 4.1, el catálogo de los componentes de la aplicación y de los análisis debe estar compuesto por los siguientes elementos:

- **Board**: objeto que representa al dispositivo encargado de realizar la comunicación entre el sensor y la plataforma.
- **Sensor**: representación de cada sensor contenido en un “board”.
- **Model**: conjunto de parámetros que recibe la placa para poder ejecutar el programa encargado de la toma de muestras e interacción con el sensor.
- **Test**: representación de las ejecuciones que se lanzan de cada modelo sobre cada “board”.

Adicionalmente, el modelo necesita otra entidad que contenga los parámetros de dicho modelo. El test por su parte, al estar relacionado con el modelo, también necesita otra entidad que refleje los valores configurados para cada modelo. De esta manera existen también:

- **Parameters**: representación de los parámetros de cada modelo necesarios para los programas encargados de interactuar con el sensor o con el conjunto de sensores, dependiendo del análisis a realizar. A su vez, estos parámetros pueden tener parámetros relacionados. Por ejemplo, un parámetro de tipo sensor puede tener como parámetro relacionado la temperatura que se aplica al sensor, o puede haber otro parámetro declarado que sea la temperatura del sensor y sea el programa encargado del análisis el que se ocupe de hacer esta relación. La existencia de esta relación o no dependerá del programador y de cómo quiera que se visualice el modelo de cara al usuario encargado de ejecutar las pruebas.

- **TestParameters**: valor tomado por cada parámetro del modelo en el análisis ejecutado para cada prueba.

4.1.4.3. Tecnologías

La estructura de la información a almacenar en este servicio está relacionada, por tanto la **base de datos** debe ser relacional. La información guardada será un pequeño catálogo de dispositivos y pruebas, por lo que la carga de procesamiento de este servicio será baja, al igual que el tráfico soportado, no siendo necesario escalar el servicio. Estos motivos conllevan a la elección de **SQLite** (Apartado 2.2.4.1) como base de datos, ya que se ajusta perfectamente a los requisitos necesarios para el servicio. Además, esta base de datos se caracteriza por tener una configuración rápida, ofreciendo ventajas en el desarrollo frente a otras bases de datos relacionales como MySQL, que requieren de una configuración mayor y consumen más recursos.

En cuanto a la API REST encargada de procesar las peticiones y comunicarse con la base de datos, esta vez se desarrolla con el lenguaje **Python** (Apartado 3.1.1) y más concretamente con el framework **Django** (Apartado 3.1.1.1), ya que facilita el desarrollo rápido integrando SQLite por lo que no es necesario instalar software adicional y agiliza el desarrollo, en lugar de Java, que precisa proyectos más complejos y mayor código para conectar con la base de datos. Como se explica en el Apartado 3.1.1.1, Django es un framework de desarrollo web de código abierto, escrito en Python, que respeta el patrón de diseño conocido como modelo–vista–controlador. La meta fundamental de Django es facilitar la creación de sitios web complejos y el principio "No te repitas" (DRY, del inglés *Don't Repeat Yourself*), facilitando el desarrollo rápido y la reutilización de componentes.

La creación de API REST en Django se consigue de una manera muy rápida y sencilla utilizando un proyecto de libre acceso llamado "**Django Rest Framework**" que recoge un conjunto de funciones y librerías para crear *Web* APIs, facilitando el desarrollo y automatizando el enrutado, serializadores y demás componentes necesarios para procesar llamadas HTTP.

Las aplicaciones web escritas en Python precisan de un servidor WSGI[9] HTTP para exponer sus servicios, por lo que es necesario el uso de un servidor llamado Gunicorn que consume pocos recursos y es muy rápido.

4.1.5. Visualización del sistema

El sistema se completa con dos interfaces gráficas web que otorgan la versatilidad necesaria en herramientas de este tipo y proporcionan una solución completa para poder realizar pruebas e interactuar con sensores:

- La primera interfaz está concebida para administrar el catálogo y el inventario del sistema.
- La segunda se centra más en la experiencia de usuario permitiendo la ejecución de las pruebas y la visualización de los resultados.

Con la creación de dos interfaces diferenciadas se distinguen dos roles distintos: el administrador del sistema y el usuario que ejecuta las pruebas.

4.1.5.1. Interfaz de administración del catálogo

La interfaz gráfica debe permitir la administración del catálogo y así tener una base de datos con la información de cada sensor, el dispositivo que comunica al sensor con la plataforma y los modelos de análisis. Esta interfaz tiene que permitir realizar todas las operaciones CRUD sobre cada entidad facilitando la gestión del catálogo.

[9] **Web Server Gateway Interface**, es una especificación para interfaces simples y universales entre servidores web y aplicaciones webs o frameworks para el lenguaje de programación Python.

La interfaz debe ser amigable y usable de modo que la creación del catálogo no sea un escollo en el desarrollo de las pruebas. Además, gracias a esta interfaz se ofrece una pequeña documentación de la API REST del servicio que administra el catálogo en formato web para cualquier usuario que desee utilizar la plataforma, y también facilita al desarrollador que desee usar el servicio la estructura de rutas disponibles para cada recurso en la aplicación. En la Figura 4.8 se muestra el diseño esperado de la interfaz.

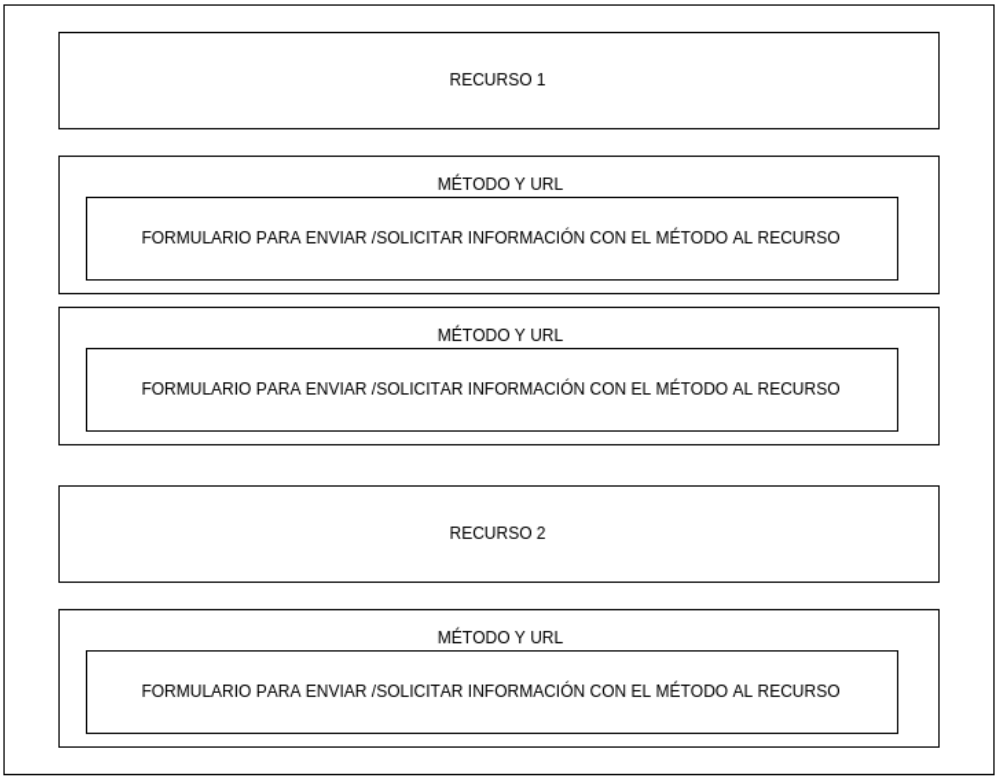

Figura 4.8: Wireframe de la visualización de la UI para la gestión del catálogo de los recursos del sistema.

La realización de esta interfaz se desarrolla con **Swagger**. Swagger es una tecnología de código abierto y proporciona un conjunto de herramientas para la programación de interfaces de desarrollo de aplicaciones. Al disponer esta interfaz, la creación del catálogo por parte del administrador del sistema se realiza de una manera muy sencilla y se minimizan los errores a la hora de insertar el contenido en esta base de datos. La interfaz se lleva a cabo utilizando la herramienta Swagger UI, la cual proporciona una colección de archivos HTML, JavaScript y CSS que generan de forma dinámica, tanto la documentación, como la interfaz para enviar peticiones HTTP al recurso especificado. Gracias a esta herramienta y la facilidad de integración con Django (tecnología utilizada para la creación del servicio de catálogo) mediante la librería “Django REST Swagger”, Swagger se impone ante otras tecnologías, como por ejemplo RAML, que sólo disponen de la interfaz para la especificación de la API REST.

4.1.5.2. Interfaz de administración de las pruebas

La creación de pruebas se realizará mediante otra interfaz de usuario, ya que requiere el envío de los parámetros del modelo, la información referente al sensor y el dispositivo en el que se quiere ejecutar la prueba de forma conjunta. Esta interfaz a su vez permitirá consultar la información de cada modelo y de cada dispositivo visualizando tanto los parámetros requeridos de cada modelo, como los sensores disponibles para cada dispositivo.

La visualización de los dispositivos y de los modelos queda reflejada en la Figura 4.9, en la cual la información se presenta como una lista de objetos: cajas en las que se muestra una información reducida de cada elemento y que, al ser accionada, abre un diálogo mostrando el detalle completo de la información.

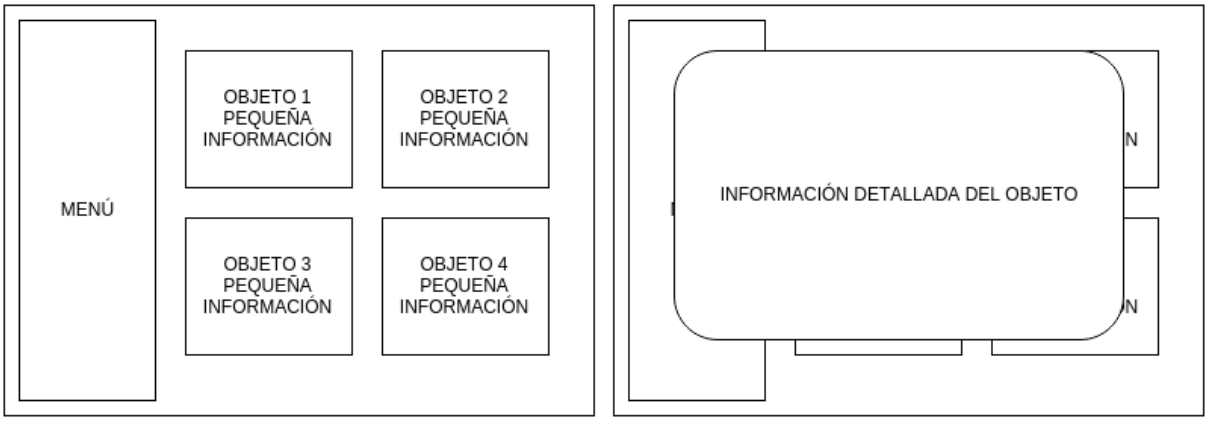

Figura 4.9: Wireframe de la visualización de la UI para la gestión de los resultados de los sensores. La imagen de la izquierda representa la visualización de objetos. La imagen de la derecha representa el detalle mostrado al presionar sobre un objeto.

La creación de una prueba conlleva una serie de pasos en el envío de la petición a la plataforma para ser procesada: primero es necesario seleccionar el dispositivo en el que se quiere ejecutar la prueba, a continuación se elige el modelo que se desea ejecutar, y por último, se han de completar todos los campos requeridos por el modelo, así como el sensor o los sensores implicados en él. Estos sensores pertenecen al dispositivo seleccionado en el primer paso.

La interfaz de usuario proporciona la visualización de las pruebas para la monitorización de los datos enviados por el sensor. Por ello, debe implementar un gráfico en el que se seleccionen las variables a mostrar y el número de muestras de estas variables que se desea ver facilitando el estudio del comportamiento de las variables. Esta visualización debe contemplar el caso de tiempo real y solicitar los datos periódicamente a la plataforma para las pruebas que se están ejecutando en el momento de la visualización.

El diseño de esta interfaz se contempla como una aplicación de página única (SPA), ya que se proporciona una aplicación web con una experiencia de usuario más fluida asemejándose a una aplicación de escritorio. En una SPA la página no tiene que ser

recargada de nuevo en ningún punto del proceso ni se transfiere a otra página. El código HTML, JavaScript y CSS se carga de una vez, lo que libera al servidor de peticiones y las aplicaciones se ejecutan en el cliente. Los recursos necesarios se cargan dinámicamente como lo requiera la página según las acciones del usuario. En contraposición a la programación tradicional, donde se lleva a cabo todo el proceso en el lado del servidor, los conceptos como controlador e interacción de modelo no producen nuevas vistas de HTML desde el servidor, sino en el lado cliente disponiendo de todas las vistas.

La tecnología escogida para crear esta SPA es AngularJS. Como se especifica en el Apartado 3.2.3.1, AngularJS es una librería del lado del cliente que proporciona plantillas que se basan en la tecnología de interfaz de usuario (UI) bidireccional (*data binding*). El "*data binding*" es una forma automática de actualizar la vista cuando el modelo mostrado sufre algún cambio, así como actualizar el modelo cuando la vista es modificada. La plantilla HTML está compilada en el navegador, esto es, se crea un HTML puro que el navegador interpreta y muestra según la interacción con el usuario. Dentro del framework AngularJS, el controlador y el modelo se mantienen en el navegador del cliente, generando nuevas vistas sin interacción alguna con el servidor. Esta tecnología se impone frente a otras, como Backbone.js, gracias al "*data binding*" proporcionado, la estructuración del código y el uso del modelo vista-controlador, el cual separa la lógica de negocio y las vistas, otorgando mayor control sobre el código. Además, hay que añadir que el uso de AngularJS favorece el uso de peticiones AJAX[10] y dispone de una gran comunidad y documentación, facilitando el aprendizaje y desarrollo de aplicaciones.

[10] ***Asynchronous JavaScript And XML***, es una técnica de desarrollo web para crear aplicaciones interactivas que se ejecutan en el cliente mientras se mantiene la comunicación asíncrona con el servidor en segundo plano. De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, mejorando la interactividad, velocidad y usabilidad en las aplicaciones.

4.2. Desarrollo

En este apartado se muestra el desarrollo de cada módulo que compone el sistema (Figura 4.10). En primer lugar se muestra el desarrollo llevado a cabo para poder comunicar el sensor con la plataforma software encargada de almacenar los datos del sensor. A continuación se detallan los servicios ofrecidos por dicha plataforma para almacenar los datos y gestionar el catálogo de los componentes del sistema otorgando consistencia y manteniendo el control sobre el sistema. En última instancia se muestra el desarrollo llevado a cabo para ofrecer las herramientas de visualización de los resultados y administración del sistema. El despliegue de los servicios en un servidor puede consultarse en el Anexo A.

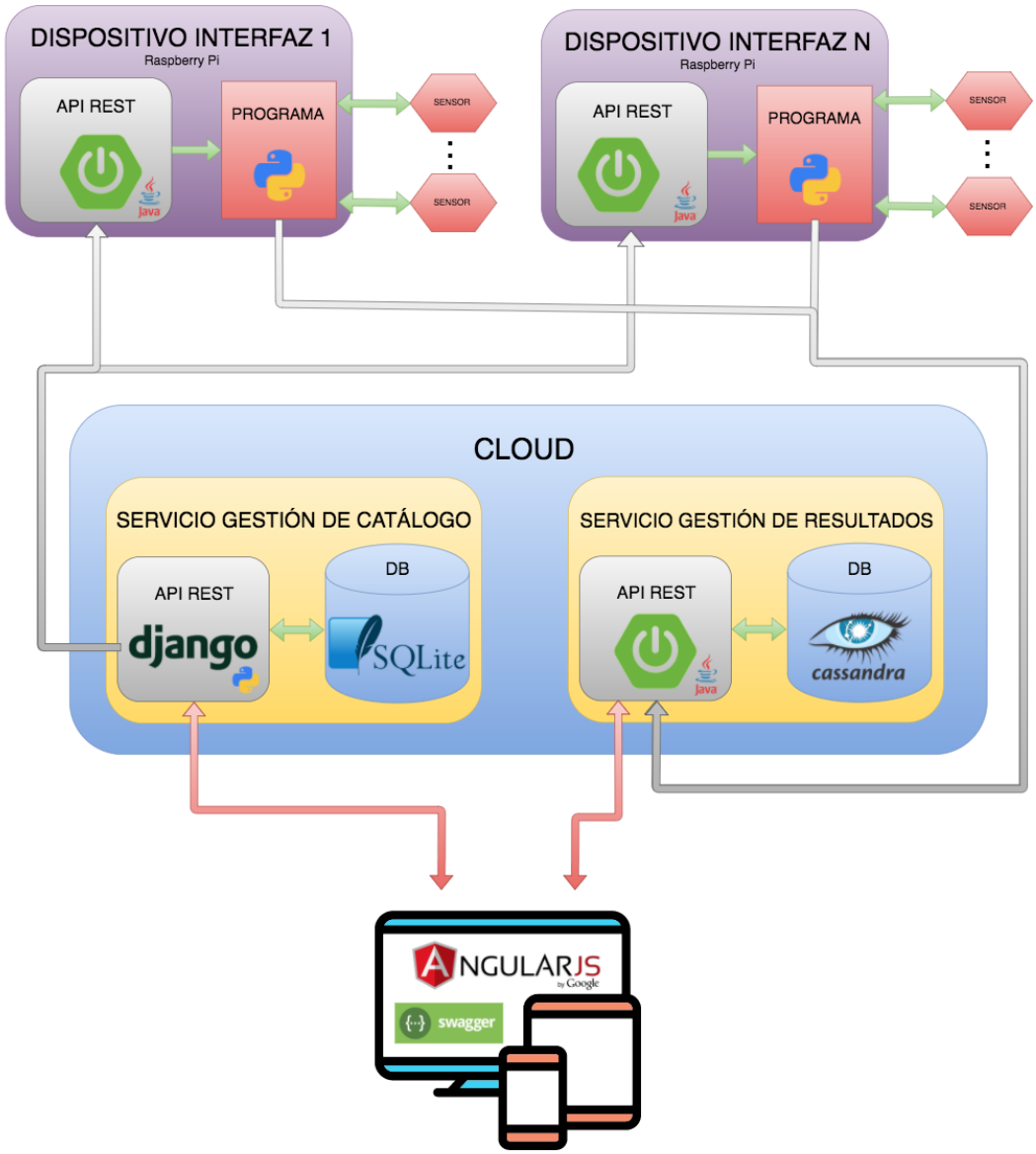

Figura 4.10: Módulos del sistema con las tecnologías y representación de la comunicación entre los módulos.

4.2.1. Dispositivo interfaz entre la nube y los sensores

El dispositivo debe ser capaz de recibir peticiones HTTP con la configuración que se desea aplicar al modelo encargado de comunicarse con el sensor y enviar los datos a la plataforma. Por ello, se divide en dos partes: la primera es la API REST encargada de recibir la petición y lanzar el programa, y la segunda, es el programa que recoge los datos del sensor, realiza los análisis oportunos según el modelo y envía la información a la plataforma en red encargada de almacenarla. En la Figura 4.11 se muestra el flujo de la comunicación entre los servicios en la ejecución de una prueba.

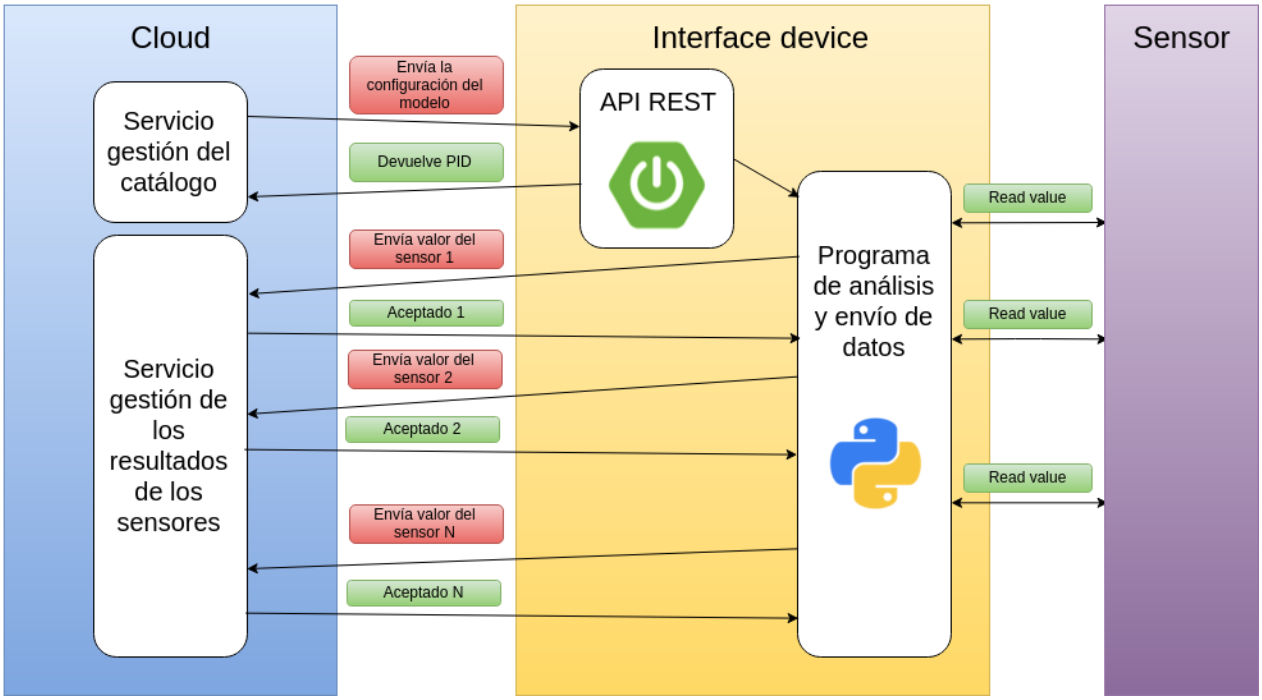

Figura 4.11: Flujo de la comunicación entre los servicios en el lanzamiento de una prueba. El dispositivo Interfaz empleado es una Raspberry PI.

4.2.1.1. Recepción de la configuración mediante API REST

La API REST está desarrollada en Java con el framework Spring (Apartado 3.2.1.1), por lo que se crea un proyecto **Maven** y se añaden ciertas dependencias en el archivo "pom.xml". Maven es una herramienta para la gestión y construcción de proyectos Java. El archivo POM (*Project Object Model*) sirve para describir el proyecto, el orden de construcción de los elementos e incluir las dependencias de otros módulos y componentes externos. Maven realiza la gestión de librerías, teniendo en cuenta las dependencias transitivas, y permite ciertas tareas como la compilación de código y su empaquetado. Todo el código del servicio se puede consultar en el Anexo B.

Las dependencias necesarias para realizar este servicio se reflejan en el archivo “pom.xml” y son las siguientes:

- El módulo “spring-boot-starter-parent”: es el padre de todos los proyectos de Spring Boot y proporciona una configuración completa para comenzar un proyecto.
- El módulo “spring-boot-starter-web”: necesario para comenzar a usar Spring Boot, permitiendo además que la aplicación sea autocontenida.
- La dependencia de “spring-data-rest-webmvc”: módulo que se encarga de crear toda la configuración básica y proporcionar las librerías necesarias para que el proyecto pueda exponer una API REST para procesar las peticiones HTTP recibidas.

El archivo principal es “Application.java” y en él se encuentra el método principal que lanza la aplicación y carga un archivo de configuración. En este archivo se encuentra la ruta absoluta de los programas de análisis que serán ejecutados en el dispositivo cuando se lanza un modelo y almacena dicha ruta en una propiedad global.

Toda la configuración se carga gracias a la etiqueta “@SpringBootApplication” que recorre el proyecto completo para añadir los controladores y componentes de la aplicación haciéndolos accesibles.

La estructura de la API REST consta de dos partes diferenciadas: servicio y controladores, los cuales se explican a continuación.

4.2.1.1.1. Servicio

El servicio se encarga de actuar como interfaz entre los controladores y el sistema operativo (SO), ejecutando comandos propios del SO mediante la clase Runtime de Java. Este servicio tiene dos funcionalidades:

- Se encarga de recibir los datos del modelo y lanzar un comando para ejecutar el programa que configura el sensor y recibe los datos de éste. Cada método corresponde a un modelo que genera el comando necesario para ejecutar el programa y enviarle los parámetros. La ejecución del comando se realiza mediante la función “executeCommand” y devuelve el identificador del proceso lanzado.
- Recibe el identificador del proceso y se ocupa de ejecutar el comando necesario para detener el proceso de envío de datos.

4.2.1.1.2. Controladores

Los *endpoints* encargados de realizar la lógica de la API REST se gestionan mediante controladores. Los controladores enrutan las peticiones a los métodos que están marcados como controlador mediante la URL que los identifica. Gracias al *framework* Spring, esta propiedad se especifica mediante la etiqueta “@RestController”. Para indicar la ruta principal de este controlador se utiliza la etiqueta “@RequestMapping(‘url-de-ejemplo’)”. Dentro de la clase se especifican los métodos que tendrá este controlador también con la

etiqueta “@RequestMapping”. En este caso, en el contenido hay que añadir el verbo HTTP que acepta (ver sección 2.2.2), pudiendo añadir a la URL del padre otra URL que identifica a este método. Además, se añade el código HTTP de respuesta que va en la cabecera mediante “@ResponseStatus(‘Estado HTTP’).

En el caso de la API REST que contiene el dispositivo existen dos controladores: el encargado de procesar las peticiones para lanzar los programas de análisis y el que detiene estos programas.

El lanzamiento de un programa se hace mediante el controlador “ApiModelController.java”. En este controlador se ubican todos los métodos encargados de recibir y procesar la petición recibida y lanzar el programa deseado. A continuación se muestra un ejemplo de un método que realiza esta función:

```
Nota: ‘X’ representa el identificador del modelo.
@RestController
@RequestMapping("/api/model")
public class ApiModelController {

    private static final Logger LOGGER = LoggerFactory.getLogger(ApiModelController.class);
    private final ApiService service;
    @Autowired
    ApiModelController(ApiService service) {
        this.service = service;
    }

    @RequestMapping(value="/[ID]", method = RequestMethod.POST, consumes = {"application/json"},
             produces = "application/json")
    @ResponseStatus(HttpStatus.ACCEPTED)
    public @ResponseBody int create(@RequestBody @Valid ModelXDTO modelEntry) {
        LOGGER.info("Processing a new model entry with information: {}", modelEntry);
        int pid = service.launchModelX(modelEntry.getAttribute1(), ... , modelEntry.getAttributeN() );
        LOGGER.info("Process launched with pid {}", pid);
        return pid;
    }
}
```

La ruta principal es “/api/model” y dentro se declara el método “create()” que acepta un método POST y cuya ruta añade el identificador único del modelo que se desea ejecutar. Este identificador se corresponde con el identificador que crea el servicio de gestión del catálogo cuando se añade un modelo a la base de datos.

El método *create* recibe como parámetro el objeto “ModelXDTO”. Este modelo es una clase que contiene la estructura del modelo creado en el servicio que gestiona el catálogo del sistema. De esta manera, y gracias a las etiquetas “@Valid” y “@RequestBody”, se comprueba que el JSON recibido en el cuerpo de la petición POST es válido y se transforma a la clase “ModelXDTO”, utilizada en el método para pasar los argumentos al servicio encargado de lanzar el programa y devolver el identificador del proceso como respuesta. El identificador del proceso lanzado se utiliza para poder parar dicho proceso.

El flujo de datos dentro de la API REST para gestionar el lanzamiento de una prueba queda reflejado en la Figura 4.12.

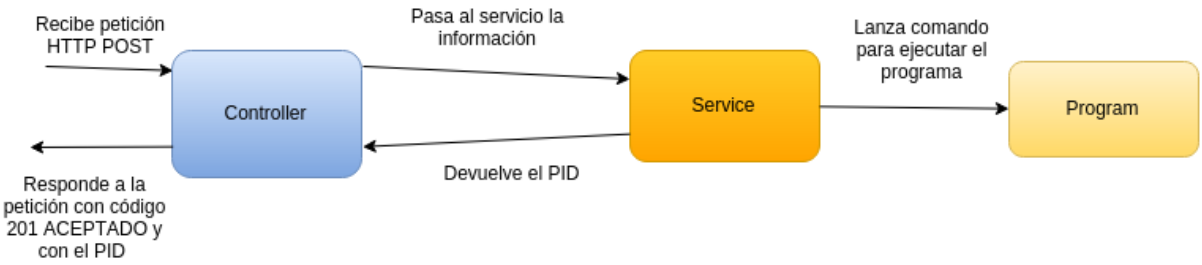

Figura 4.12: Flujo de la ejecución de un programa por parte de la API REST.

Por otro lado, es necesario poder parar una prueba debido a la existencia de pruebas de duración indeterminada o porque los resultados durante la monitorización no son los esperados. Por tanto, el controlador encargado de detener la prueba se encuentra en "ApiKillProcessController.java". Este controlador recibe en la URL el identificador del proceso que se desea detener. El identificador se corresponde con el que se devuelve al lanzar un programa. Con este identificador se ejecuta el método del servicio encargado de parar el proceso.

4.2.1.2. Programa de captura de datos y envío de información a la nube

En este apartado se muestra la lógica del programa encargado de capturar los datos del sensor y enviarlos a la nube. Este programa está adaptado al desarrollo implementado en una Raspberry Pi, dispositivo utilizado en las pruebas.

El programa lanzado por la API REST responsable de interactuar con el sensor y enviar los datos de éste a la plataforma está escrito en Python. Este programa recibe los parámetros por argumento según se los transmite el comando ejecutado por la API REST. Para ello, recibe cada argumento y lo convierte a la variable deseada como se muestra en el siguiente ejemplo:

```
if __name__ == "__main__":
    # Parameters from input
    board_id = str(sys.argv[1])
    test_id = str(sys.argv[2])
    sensor_id = int(sys.argv[3])
```

La comunicación con el sensor comienza tras comprobar los parámetros recibidos. Cada modelo ejecuta distintos análisis de los datos y los envía a la plataforma mediante llamadas HTTP. La tasa de envío de los datos depende de cada modelo y las necesidades de almacenamiento para su monitorización.

4.2.1.2.1. Interacción con el sensor

La comunicación con el sensor se realiza directamente mediante los pines GPIO del dispositivo Raspberry Pi y existen dos funcionalidades básicas en la comunicación:

- Controlar el valor del voltaje que se suministra en el pin *input* del sensor (generalmente el pin de alimentación). Esta característica es muy útil en sensores activos.
- Recoger el valor devuelto por el sensor.

Para realizar estas funciones de comunicación es necesario utilizar el módulo **RPi.GPIO**. El sistema operativo Raspbian ya incluye por defecto este módulo, por lo que se procede a importar la librería al programa de Python de la siguiente manera:

```
import RPi.GPIO as GPIO
```

A continuación, se declara el sistema de numeración que se utiliza para referenciar a los pines. Existen dos modos de numeración:

- BOARD: hace referencia a la numeración física de los pines en la tarjeta.
  `GPIO.setmode(GPIO.BOARD)`

- BCM: utiliza la numeración de pines del procesador Broadcom de la Raspberry.
  `GPIO.setmode(GPIO.BCM)`

Además, hay que declarar los pines que se usarán para el sensor indicando el tipo de pin, entrada o salida, según corresponda:

```
GPIO.setup(sensor_pin_output, GPIO.IN)
GPIO.setup(sensor_pin_input, GPIO.OUT)
```

El control del voltaje que se aplica a la salida en el pin de la Raspberry Pi, y por consiguiente en el pin de entrada (generalmente el de alimentación) del sensor, se controla mediante modulación PWM. En esta modulación existen dos parámetros importantes y se ajustan de la siguiente manera:

- Frecuencia: número de veces por segundo que se genera el pulso en hercios (Hz).
  `sensor_input = GPIO.PWM(sensor_pin_input, frecuencia)`

- Ciclo de trabajo (*Duty Cycle, D*): porcentaje del tiempo que la señal está en estado activo entre pulsos.
  `sensor_input.start(D)`
  Modificación del ciclo de trabajo: `sensor_input.ChangeDutyCycle(D')`

La lectura del sensor depende del protocolo utilizado para comunicarse con él. Al utilizar la Raspberry Pi se ofrece una solución universal para todo tipos de sensores. A continuación se explican las distintas maneras de capturar los datos según el tipo de sensor.

Los pines GPIO de la Raspberry son digitales, por lo que la comunicación con sensores analógicos no es directa y es necesario emplear alguna técnica para la toma de valores. Una adaptación muy sencilla para la lectura de sensores analógicos resistivos es aplicar circuitos RC[11], de manera que midiendo el tiempo de carga del capacitor se puede obtener la resistencia equivalente del sensor y el valor arrojado por él. Para utilizar esta aproximación es necesario seguir los siguientes pasos:

- Descargar el condensador: para ello es necesario poner el pin de la Raspberry como salida y en estado bajo.

```
GPIO.setup(PiPin, GPIO.OUT)
GPIO.output(PiPin, GPIO.LOW)
time.sleep(0.1)
```

- Activar el pin como entrada para proceder a leer el valor del pin.

```
GPIO.setup(PiPin, GPIO.IN)
```

- Contar el número de veces que se ejecuta el bucle hasta que se carga el capacitor, obteniendo así la medida del sensor.

```
while (GPIO.input(PiPin) == GPIO.LOW):
   medida += 1
```

Otra alternativa para sensores analógicos es usar un conversor analógico-digital externo y leer la señal de éste mediante el protocolo SPI. Esta configuración también se aplica a todos los sensores digitales que dispongan esta interfaz. Para ello es necesario utilizar la librería **spidev** que facilita la comunicación SPI y aplicar los siguientes pasos:

- Habilitar el bus SPI.

```
spi = spidev.SpiDev()
spi.open(0,0)
```

- Leer el valor del bus. En este ejemplo se muestra la lectura de un conversor de 8 canales y 10 bits de resolución:

```
adc = spi.xfer2([1,(8+canal)<<4,0])
dato = ((adc[1]&3) << 8) + adc[2]
```

- El cálculo del valor del voltaje se realiza de la siguiente manera:

```
voltaje = (adc * Vref) / (2 resolución)
```

La lectura de sensores digitales es directa y en el caso de sensores que utilicen algún bus de datos, la lectura es inmediata a través de los buses que proporciona la Raspberry Pi.

4.2.1.2.2. Envío de datos por protocolo HTTP

La librería utilizada para el envío de paquetes HTTP en Python es **requests**. Esta librería permite que la integración con servicios web sea transparente para el programador y elimina la necesidad de agregar parámetros a las URLs o transformar la información a formularios

[11] Un circuito RC es un circuito compuesto de resistencias y condensadores alimentados por una fuente eléctrica.

para hacer peticiones POST. Además, requests incluye a su vez la librería *urllib3* que realiza automáticamente la reutilización de keep-alive[12] y la conexión HTTP.

La información que contiene el valor aplicado al sensor y el recibido por éste se tiene que corresponder con la esperada por la plataforma. Además de estos datos, debe contener la información de la prueba que genera el resultado, el identificador del dispositivo que posee el sensor, el identificador del propio sensor y la fecha de la captura de los datos. Para ello, se crea un objeto JSON con esta información y se ejecuta una función en un hilo separado, de manera que no se bloquea la ejecución del programa:

```
    test_time = current_milli_time()
    # Send the info to api-results in background
    info = {
        "testId": test_id,
        "boardId": board_id,
        "sensorId": sensor_id,
        "inputValue": input_value,
        "outputValue": output_value,
        "testTime": test_time
    }
    Thread(target=send_value, args=(info, )).start()
```

Existen dos funciones específicas para establecer la comunicación con cada servicio: una encargada de enviar los datos del sensor y otra para actualizar la información de la prueba.

La función encargada de enviar la información al servicio responsable de almacenar los datos recibe dicha información y, mediante la librería *requests*, envía el paquete generado con una llamada HTTP POST. Si la llamada se realiza con éxito, termina el hilo, en caso contrario, se envía el mensaje de error al servicio catálogo indicando el motivo de error y finalizando el programa. Esta función se muestra a continuación:

```
def send_value(_info):
    api_results_uri = "http://dev-api-results.pfc.com/api/save/results"
    r = None
    try:
        r = requests.post(api_results_uri, json=_info)
    except requests.exceptions.RequestException as e:
        sys.stderr.write("send_value: " + "\n")
        error = "ERROR: send_value: " + str(e)
        Thread(target=update_test_info, args=(test_id, {"status": error},)).start()
        os._exit(os.EX_UNAVAILABLE)
    if r.status_code != 201:
        sys.stderr.write("send_value: Fail to create result entry: \n" + json.dumps(_info,
sort_keys=True, indent=4) + "\n")
        error = "ERROR: send_value: Fail to create result entry: \n" + json.dumps(_info,
sort_keys=True, indent=4)
        Thread(target=update_test_info, args=(test_id, {"status": error},)).start()
        os._exit(os.EX_DATAERR)
```

[12] Una conexión **HTTP/1.1 con keepalive** permite realizar más de una petición por la misma conexión evitando establecer una conexión nueva para cada parte de la web.

Adicionalmente, existe otra función para enviar actualizaciones del estado de la prueba al servicio de catálogo que almacena la información del test realizado. Esta función recibe el identificador de la prueba, necesario en el acceso al recurso mediante la URL, y la información que se quiere actualizar del test. Típicamente esta información es el estado del test o el progreso, por lo que la llamada que se realiza es de tipo PATCH, ya que se trata de una actualización parcial del recurso. En caso de ocurrir algún error con la petición, el programa se detendrá de manera que no se pierda la información. A continuación se detalla el código de la función:

```
def update_test_info(_test_id, _info):
    api_catalog_uri = "http://dev-api-catalog.pfc.com/test/"+_test_id+'/'
    headers = {'Content-Type': 'application/json'}

    r = None
    try:
        r = requests.patch(api_catalog_uri, json=_info, headers=headers)
    except requests.exceptions.RequestException as e:
        sys.stderr.write("update_test_info: " + "\n")
        error = "ERROR: update_test_info: " + str(e)
        Thread(target=update_test_info, args=(test_id, {"status": error},)).start()
        os._exit(os.EX_UNAVAILABLE)

    if r.status_code != 200:
        sys.stderr.write("update_test_info: Fail to update test entry: \n" + json.dumps(_info,
sort_keys=True, indent=4) + "\n")
        error = "ERROR: update_test_info: Fail to update test entry: \n" + json.dumps(_info,
sort_keys=True, indent=4)
        Thread(target=update_test_info, args=(test_id, {"status": error},)).start()
        os._exit(os.EX_DATAERR)
```

Esta función se debe lanzar en un hilo separado para no bloquear la ejecución del programa de la siguiente manera:

```
Thread(target=update_test_info, args=(test_id, {"progress": 100, "status": "finished", "pid": None,
"finished_on": finish}, )).start()
```

4.2.2. Servicio de gestión de los resultados de los sensores

La plataforma software alojada en la nube tiene un servicio único dedicado a la gestión de los datos enviados de los sensores. Este servicio se ofrece mediante una API REST y está destinado a recibir peticiones con la información de los sensores, devolver los resultados a los clientes que los soliciten, pudiendo estos clientes monitorizar los datos, y generar un fichero con los resultados de una prueba para su descarga. La base de datos destinada a este servicio es **Cassandra** (Apartado 2.2.4.2) ya que se trata de una base de datos concebida para la ingesta masiva de datos, como en este caso, de la información proveniente de múltiples sensores. El código completo de este servicio se encuentra en el Anexo C.

El servicio nuevamente ha sido creado con Maven, ya que se trata de un proyecto Java con el *framework* Spring Boot. Las dependencias y descripción del proyecto se encuentran en el fichero “pom.xml” y se detallan a continuación:

- El módulo padre es “spring-boot-starter-parent”, que proporciona toda la configuración básica para empezar el proyecto.
- El módulo “spring-boot-starter-web” permite crear una aplicación autocontenida y es necesario para usar Spring Boot.
- El módulo “spring-data-rest-webmvc” es el encargado de crear toda la configuración básica y proporcionar las librerías necesarias para que el proyecto pueda exponer una API REST y procesar las peticiones HTTP.
- El módulo “spring-data-cassandra” provee una capa de abstracción para acceder a la capa de persistencia, en este caso Cassandra.
- La librería “Super CSV” facilita la creación y manipulación de archivos CSV de una manera sencilla, abstrayendo al programador de realizar la tarea.

De manera similar a la API del dispositivo, la aplicación se lanza desde el archivo principal “Application.java”. En este archivo se realizan todas las configuraciones necesarias para arrancar la aplicación mediante la etiqueta “@SpringBootApplication”, que contiene toda la configuración básica y escanea el proyecto para añadir todos los componentes. En este caso es necesario habilitar la configuración para establecer la comunicación con la base de datos Cassandra. Esto se hace con la etiqueta “@EnableCassandraRepositories”.

La configuración de Cassandra está en la clase “AbstractCassandraConfiguration”, que es propia del módulo de Spring para Cassandra, por lo que hay que extender esta clase para cargar el esquema y la tabla en la base de datos con la estructura deseada. La tabla se encuentra en la ruta de recursos en el archivo “SCHEMA.cql” y se muestra a continuación:

```
CREATE TABLE IF NOT EXISTS results (
    test_id text,
    test_time bigint,
    board_id text,
    sensor_id text,
    input_value double,
    output_value double,
    event_time bigint,
    PRIMARY KEY (test_id, sensor_id, test_time)
)
WITH comment='' AND read_repair_chance = 2.0 AND CLUSTERING ORDER BY (sensor_id ASC, test_time DESC);
```

El archivo “SCHEMA.cql” se usa para crear la tabla al arrancar la aplicación si fuera necesario. Además, hay que especificar el *keyspace* donde se almacena esta tabla, así como la configuración del punto de acceso y el puerto de la base de datos. Esta configuración se encuentra en el archivo “cassandra.properties”.

La API REST es capaz de procesar peticiones y recibir los datos de los sensores, devolver los datos a clientes y generar ficheros con los datos perteneciente a una prueba en concreto. Todo esto se realiza con distintos controladores que se comunican con un repositorio que tiene acceso a la base de datos.

En la Figura 4.13 se muestra el detalle de la comunicación interna de la API REST que está explicada en los siguientes apartados.

Figura 4.13: Comunicación interna de la API REST entre los controladores y la base de datos.

4.2.2.1. Entidad de la base de datos

Cassandra es una base de datos con una estructura de súper tabla en la que la información se almacena por filas y no puede existir una fila con la misma clave primaria. La clave primaria en Cassandra se usa para identificar una fila y buscar la información únicamente por las variables que se encuentran en ella, lo que otorga la velocidad en las transacciones de la base de datos. Para solventar el problema de almacenar para un mismo test los resultados de cada sensor, la clave primaria está compuesta por:

1. El identificador de la prueba.
2. El identificador del sensor que genera esta entrada.
3. La fecha en la que el sensor generó la muestra.

De esta manera, se pueden insertar todos los datos generados por los sensores y realizar las búsquedas por:

- El test ejecutado
- El test ejecutado y el identificador del sensor
- El identificador completo de la muestra, esto es, el conjunto de la clave primaria.

El esquema de la base de datos está compuesto por una única tabla, la cual se muestra en la Tabla 4.2.

Tabla 4.2: Esquema de la base de datos Cassandra para el servicio gestión de resultados.

Campo	Descripción
test_id	Identificador de la prueba realizada
sensor_id	Identificador del sensor al que pertenecen los datos
test_time	Fecha de la toma del valor asociado
board_id	Identificador de la placa que contiene el sensor
input_value	Valor aplicado al sensor
output_value	Valor devuelto por el sensor
event_time	Fecha de la recepción

El objeto necesario para la comunicación con la base de datos debe tener la misma estructura que la tabla creada, por tanto, se crea la clase “Results.java” que contiene las variables que se corresponden con la tabla. Esta clase, a su vez, extiende de la clase “ResultsKey.java” que posee la clave primaria en Cassandra.

Por otro lado, los controladores no pueden utilizar la misma clase que usa la base de datos. La comunicación entre procesos se realiza por tanto mediante interfaces, creando Objetos de Transferencia de Datos (DTO) para transportar los datos entre procesos. Con la creación de un objeto de este tipo, la comunicación entre la API REST y el cliente se realiza con la misma estructura de objeto y se convierte de una manera eficiente. La clase que se corresponde con el esquema de la base de datos y con la estructura esperada por la API REST es “ResultsDTO.java”.

4.2.2.2. Repositorio de los sensores

El repositorio provee una interfaz a los controladores para acceder a la base de datos de manera transparente para éstos. El repositorio extiende de una clase preparada en el módulo de Spring para la comunicación con Cassandra denominada “CassandraRepository”. De esta manera, se obtienen las principales transacciones con la base de datos (CRUD), pudiendo añadir otras nuevas. A continuación se detallan las consultas añadidas, necesarias para el funcionamiento del servicio:

1. Solicitar los resultados de una prueba y un sensor en concreto.

```
@Query("SELECT * FROM results WHERE test_id=?0 AND sensor_id=?1 LIMIT ?2 ")
public List<Results> findByTestIdAndSensorId(@Param("test_id") String testId,
                @Param("sensor_id") String sensorId,
                @Param("size") int size);
```

2. Consultar en la base de datos un número concreto de resultados de un sensor para una prueba con una fecha específica.
    a. Resultados posteriores a la fecha.

```
@Query("SELECT * FROM results WHERE test_id=?0 AND sensor_id=?1 AND test_time > ?2
```

```
ORDER BY  sensor_id DESC LIMIT ?3")
public List<Results> findByTestIdAndSensorIdNextPage(
        @Param("test_id") String testId,
        @Param("sensor_id") String sensorId,
        @Param("event_time") long eventTime,
        @Param("size") int size);
```

b. Resultados anteriores a la fecha.

```
@Query("SELECT * FROM results WHERE test_id=?0 AND sensor_id=?1 AND test_time < ?2
LIMIT ?3")
public List<Results> findByTestIdAndSensorIdPreviousPage(
                @Param("test_id") String testId,
        @Param("sensor_id") String sensorId,
        @Param("event_time") long eventTime,
        @Param("size") int size);
```

3. Reclamar los resultados de todos los sensores de una prueba.

```
@Query("SELECT * FROM results WHERE test_id=?0")
public List<Results> findByTestId(@Param("test_id") String testId);
```

4.2.2.3. Controladores

El almacenamiento de la información enviada se gestiona mediante el controlador "ApiSaveResultsController.java". Este controlador gestiona la URL "api/save/results" y se encarga de recibir una petición POST cuyo cuerpo tiene que seguir la estructura de "ResultsDTO.java", estructura que puede ser consultada en el Anexo C. El controlador se encarga de comprobar que están presentes todos los campos y se comunica con el repositorio para almacenar el objeto en la base de datos. Si todo es correcto, se resuelve la petición con el código de *HTTP 201 Creado*, en caso contrario, se devuelve un código de error.

La recuperación de la información de un sensor se realiza mediante el endpoint "api/results/[testID]/sensor/[sensorID]". Este endpoint se encuentra en el controlador "ApiResultsController.java" y se encarga de procesar las peticiones GET a esta ruta. El controlador se comunica con el repositorio para pedir la información almacenada de un sensor en una prueba y devolverla al cliente. Estos datos son devueltos paginados para agilizar las peticiones HTTP. El tamaño por defecto de página es 10, siendo parametrizable en la petición GET, asignándole al parámetro "*size*" el tamaño de página deseado.

Al estar paginada la respuesta devuelve, además de los datos paginados, las rutas para la consulta de los datos posteriores a la fecha y los anteriores según el tamaño de página especificado para el sensor y test consultados. Este mismo controlador se encarga de gestionar las peticiones de páginas siguientes y previas. Para ello, en la petición GET deben existir los parámetros "*page*", que especifica si se requiere la página siguiente o la anterior a la fecha, y el parámetro "*date*", que indica la fecha desde la que se desea comenzar la consulta.

Por último, para la descarga de los datos de una prueba existe un controlador bajo la ruta "api/downloadCSV/" llamado "CSVFileDownloadController.java" (Anexo C). En este controlador existe un método que acepta peticiones GET con el identificador del test del

cual se quieren descargar los datos. Para ello, el controlador solicita al repositorio los datos de la prueba y, mediante la librería Super CSV, transforma la respuesta del repositorio a un archivo CSV que se devuelve al cliente que lo ha solicitado. Este archivo se genera bajo el nombre “test_[ID]_[DATE].csv”, siendo [ID] el identificador de la prueba solicitada y [DATE] la fecha en milisegundos de la petición.

4.2.3. Servicio de gestión del catálogo de la aplicación

La gestión del sistema se realiza mediante un servicio en el que se almacenan los distintos dispositivos interfaz entre los sensores y la plataforma, así como los sensores y los modelos que se pueden ejecutar en cada dispositivo. De esta manera, se obtiene un catálogo con el cual se pueden ejecutar pruebas sobre un sensor con un modelo concreto, completando así toda la información del sistema. Este servicio proporciona al usuario encargado de ejecutar las pruebas la información necesaria para configurarlas. Además, permite establecer la relación entre los datos almacenados en el servicio de gestión de resultados para su estudio y visualización.

La tecnología escogida para este módulo es Django, que utiliza el lenguaje de programación Python, por lo que es necesario disponer de **Virtualenv,** una herramienta para crear entornos de Python aislados, es decir, entornos donde las librerías o las versiones de Python no interfieren con los usados en la máquina por defecto.

El archivo en el que se especifica toda la configuración del proyecto es “settings.py” y en él se definen las aplicaciones que usa este proyecto. En este caso, una aplicación es un proyecto externo (módulos de Django y Django REST Framework) o módulos propios del proyecto, como el módulo correspondiente a la API REST. El código de la API REST se puede consultar en el Anexo D.

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'catalog',
    'rest_framework_swagger',
    'corsheaders',)
```

Además de las aplicaciones citadas se añade un módulo para evitar el CORS[13], evitando que si el dominio es distinto al del servicio rechace las peticiones HTTP, y el módulo de Swagger, el cual será explicado más adelante en el apartado 3.2.4.1 referente a la visualización del sistema.

[13] **Cross-Origin Resource Sharing**, es un mecanismo que permite restringir la solicitud de recursos desde una página web a otro dominio externo al dominio desde el que se originó el recurso.

La base de datos utilizada en el proyecto es SQLite y también se define en este archivo de la siguiente manera:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Django REST automatiza la creación de servicios REST de manera que necesita pocos recursos para crear el servicio. Éstos son:

- El modelo de la base de datos.
- Los serializadores que convierten los datos recuperados de la base de datos a objetos Python.
- Controladores, cuya misión es renderizar la respuesta a las peticiones.
- El enrutador que indica al proyecto las rutas de la API.

4.2.3.1. Diagrama de clases

El modelo de la base de datos se crea en el archivo “models.py”. Con este archivo, además de crear la base de datos, Django se encarga de comprobar los campos de un modelo cuando se hace una transacción con la base de datos. Las relaciones entre entidades se crean en este mismo fichero.

Como se explica en el diseño de este servicio (Apartado 4.1.4.2), las entidades necesarias en este servicio son: el dispositivo que dota de conectividad al sensor, los sensores, los modelos de análisis y las pruebas realizadas.

Las relaciones entre entidades son las siguientes:

- Un dispositivo puede tener muchos sensores, pero un sensor sólo puede pertenecer a un dispositivo, ya que son objetos físicos.
- Los modelos están compuestos por los parámetros del programa que se ejecuta en los dispositivos, por tanto, un modelo tiene muchos parámetros. Estos parámetros son magnitudes o entidades generales, como por ejemplo la temperatura, por lo que un parámetro puede estar presente en varios modelos. Además, un parámetro puede estar relacionado con varios parámetros a su vez.
- La relación entre una prueba y sus parámetros es idéntica a la de los modelos, ya que una prueba ejecuta un modelo, y por ende, son necesarios los parámetros de dicho modelo.

Los requisitos del catálogo implican la creación de ciertas entidades y relaciones que se muestran en el diagrama entidad-relación de la Figura 4.14.

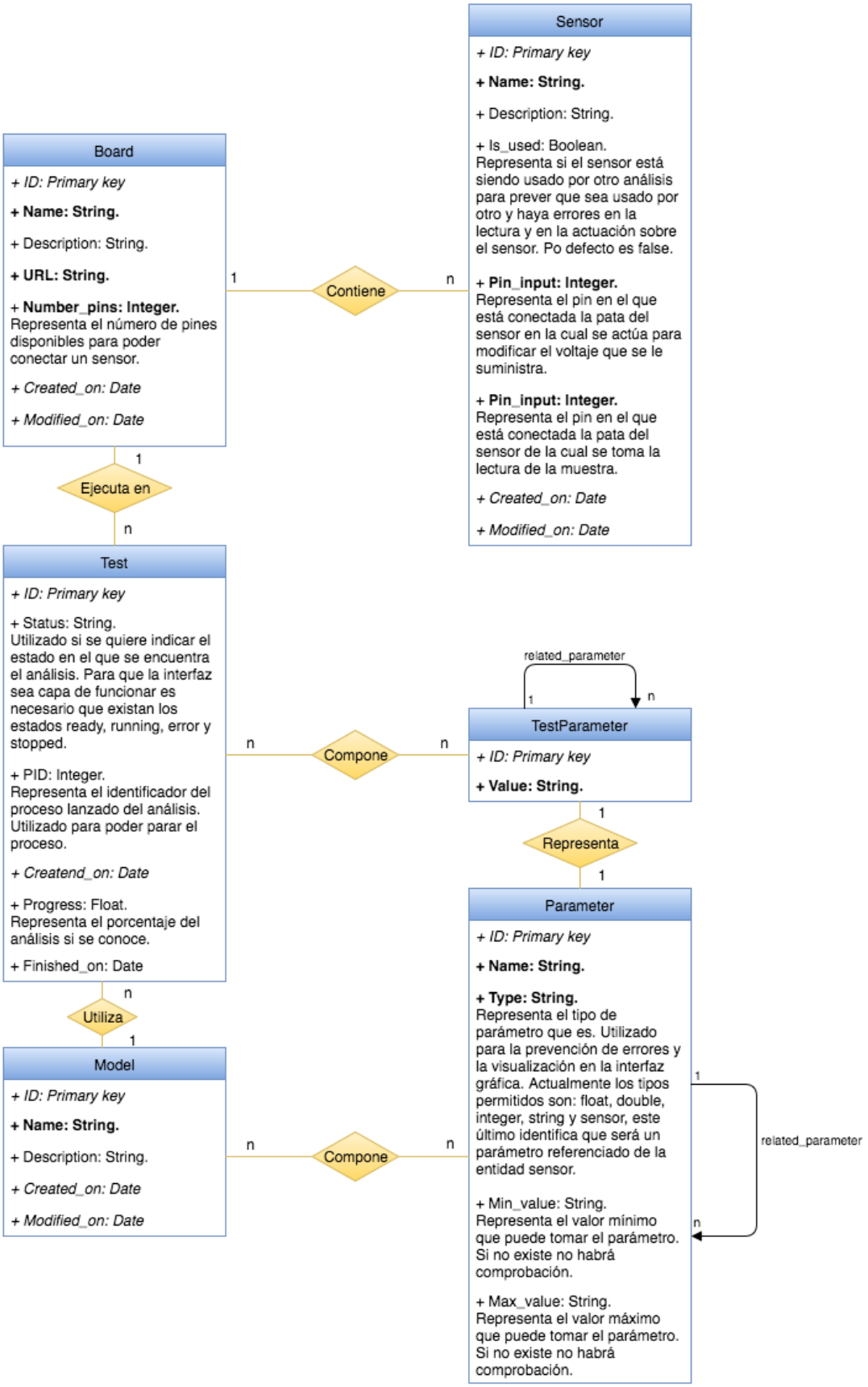

Figura 4.14: Diagrama entidad-relación de la base de datos de catálogo de la aplicación. En negrita se resaltan los campos obligatorios y en cursiva se representan los campos que se crean automáticamente, el resto de campos son opcionales.

Las relaciones “uno a muchos” se declaran dentro de la clase correspondiente a la relación “muchos” mediante el uso de claves foráneas. En el caso de la relación entre “*board*” y “*sensor*” es necesario crear la clase *Board* y la clase *Sensor*, con los parámetros de cada clase, y además en *Sensor,* declarar una variable llamada *board,* que es una clave foránea de la clase *Board* que representa la relación. Como dicha relación puede o no existir, se declara como “*True*” el atributo que indica si puede ser nula esta variable. También hay que declarar el nombre que Django utilizará para esta relación. La misma lógica se aplica en el caso de *Test* y *TestParameters*.

En el caso de las relaciones “muchos a muchos” existe otro valor que se asigna a la variable llamado *ManyToManyField*. Este es el caso de la relación entre *Model* y *Parameters*, en el que hay que declarar una variable llamada *model* dentro de la clase *Parameters* del tipo *ManyToManyField* que representa la relación entre ambas clases. De igual manera, hay que declarar un atributo que designe el nombre de la relación.

La relación existente entre parámetros se crea indicando dentro de la clase *Parameters* una variable tipo *ManyToManyField* “*self*” para indicar que es una relación de la propia clase. A continuación se muestra la definición de una clase, en este caso *Parameter.*

```
class Parameter(models.Model):
    model = models.ManyToManyField(Model, related_name='parameters', blank=True)
    name = models.TextField()
    type = models.TextField()
    min_value = models.TextField(blank=True, null=True)
    max_value = models.TextField(blank=True, null=True)
    related_parameters = models.ManyToManyField('self', blank=True)
```

4.2.3.2. Serializadores

Los datos que se envían a través de las peticiones HTTP hacia la API REST deben ser traducidos al modelo de datos. Esta traducción se realiza mediante serializadores, que transforman las peticiones HTTP a objetos JSON. Los serializadores se encuentran en el archivo “serializers.py”. Con el uso de Django REST framework, los serializadores se crean por defecto importando el *ModelSerializer* de esta librería, indicando a qué modelo corresponde y las variables a tratar de ese modelo. En el caso de las relaciones “uno a muchos” hay que indicar también el serializador de la variable que contiene el conjunto de la relación, como por ejemplo, en la relación entre *Board* y *Sensor*. Para ello, dentro del serializador de *Board* hay que indicar que la variable *sensors* tiene que añadir el serializador de *Sensor* y que sólo realice esta acción cuando sean lecturas. A continuación se muestra el detalle de los serializadores correspondientes a *Sensor* y *Board*.

```
class SensorSerializer(ModelSerializer):
    class Meta:
        model = Sensor


class BoardSerializer(ModelSerializer):
    sensors = SensorSerializer(many=True, read_only=True)

    class Meta:
        model = Board
        fields = ('id', 'name', 'description', 'uri', 'number_pins', 'created_on',
'modified_on', 'sensors')
```

En el caso de los parámetros, tanto del *Test* como del modelo, es necesario crear distintos serializadores: un serializador para poder crear la relación entre parámetros, otro para añadir el valor del parámetro, y por último, el serializador que utiliza el modelo o el test para representar el objeto completo.

El serializador del test debe sobrescribir el método “create” ya que es el serializador que se utiliza para la creación de un test. El test recibe en la misma llamada los parámetros, por lo que es necesario separar cada objeto correspondiente al modelo de la base de datos. A continuación se muestra el detalle del serializador del *Test*.

```
class TestSerializer(ModelSerializer):
    parameters = TestParameterSerializer(many=True)

    class Meta:
        model = Test
        fields = ('id', 'created_on', 'finished_on', 'model', 'board', 'status',
'progress', 'pid', 'parameters')

    def create(self, validated_data):
        parameters = validated_data.pop('parameters')
        test = Test.objects.create(**validated_data)
        for parameter in parameters:
            related_parameters = parameter.pop('related_parameters')
            parameter = TestParameter.objects.create(test=test, **parameter)
            for related_parameter in related_parameters:
                related_parameter = TestParameter.objects.create(**related_parameter)
                parameter.related_parameters.add(related_parameter)
        return test
```

4.2.3.3. Controladores

La creación de los controladores se realiza en el archivo "*views*.py". Las operaciones a realizar en este servicio con todas las entidades de la base de datos son de tipo CRUD, por lo que se utiliza la funcionalidad *ModelViewSet* de Django REST Framework. Esta clase permite la creación automática de los endpoints indicando de dónde extraer la información (*queryset*) y qué serializador utilizar como traductor (*serializer_class*). Además, se pueden sobrescribir los métodos encargados de las operaciones CRUD para comprobar parámetros y evitar inserciones erróneas en la base de datos, pudiendo devolver los códigos de error oportunos.

En el caso de la creación de un test (Figura 4.15), éste conlleva una serie de pasos adicionales, ya que una vez recibida la petición POST para crear el test y haber validado los parámetros, este servicio se tiene que comunicar con el dispositivo que se ha seleccionado para lanzar la prueba y enviarle la información. Si todo funciona correctamente, la prueba se persiste, en caso contrario, se devuelve un código de error indicando el problema.

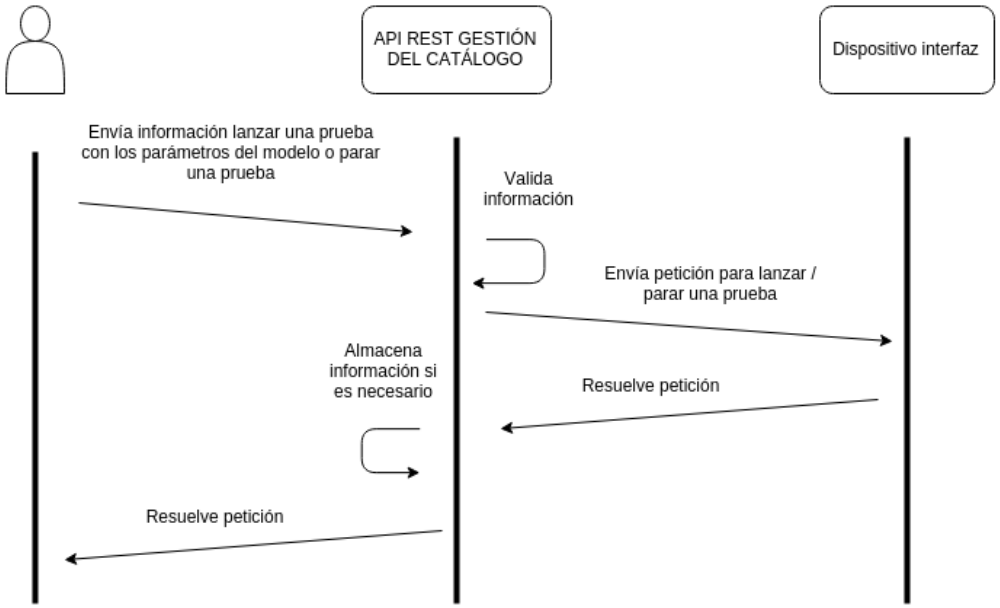

Figura 4.15: Flujo de la creación o detención de una prueba.

Además de realizar operaciones CRUD, el *endpoint* de test acepta una petición de tipo GET a la URL "/stop" (Figura 4.15). Esta función se comunica con el dispositivo y le notifica que quiere detener una prueba. Una vez recibida la respuesta por parte del dispositivo, actualiza la información del test y resuelve la petición devolviendo un código de respuesta *HTTP 200 OK*. La creación de esta ruta se realiza utilizando la etiqueta "@detail_route" e indicando el verbo HTTP que recibe esta ruta. El nombre de la ruta viene determinado por el nombre que recibe el método indicado a continuación de la etiqueta.

4.2.3.4. Enrutador

La conexión de las URLs de la API REST y del *ViewSet* creado se realiza en el archivo "urls.py". Para ello se utiliza un *SimpleRouter* propio de Django REST al que se le añaden los *ViewSets* creados en el controlador "views.py" y se registran dentro de las URLs de la API REST. Además, la ruta para las pruebas se declara como una ruta interna para que no se muestre en la interfaz de la administración del catálogo (apartado 4.2.4.1). A continuación se detalla el código empleado para la creación de las rutas dentro de la API REST.

```
router.register(r'board', BoardViewSet)
router.register(r'sensor', SensorViewSet)
router.register(r'model', ModelViewSet)
router.register(r'parameter', ParameterViewSet)
router_test = SimpleRouter()
router_test.register(r'test', TestViewSet)


internal_apis = patterns('', url(r'^', include(router_test.urls)),)


urlpatterns = [
   url(r'^docs/', include('rest_framework_swagger.urls')),
   url(r'^admin/', include(admin.site.urls)),
   url(r'^', include(router.urls))
]


urlpatterns = urlpatterns +
       patterns('', url(r'^', include(internal_apis,namespace="internal_apis")),)
```

De esta manera Django genera automáticamente las rutas para poder realizar las operaciones CRUD sobre las entidades del catálogo (ver Tabla 4.1 y Apartado 4.1.4.2). Estas rutas son:

- /board
- /board/[ID]
- /sensor
- /sensor/[ID]
- /model
- /model/[ID]
- /parameter
- /parameter/[ID]
- /test
- /test/[ID]
- /test/[ID]/stop

4.2.4. Visualización del sistema

La herramienta se completa con dos interfaces gráficas que ayudan al usuario a crear el catálogo y a consumir los datos de los sensores. De esta manera, se consigue que el sistema desarrollado ofrezca herramientas para la monitorización de los sensores y facilita el trabajo de administración de las pruebas realizadas.

4.2.4.1. Interfaz de administración del catálogo

Esta interfaz proporciona una herramienta web con la que poder administrar la información alojada en el servicio que gestiona el catálogo. La interfaz está pensada para ser utilizada por el administrador del catálogo, por lo que se ofrece la posibilidad de realizar peticiones tipo CRUD desde una herramienta web.

La instalación de Swagger, tecnología empleada para generar la interfaz, dentro de la aplicación de gestión de catálogo se realiza en pocos pasos. Swagger, una vez instalado, se encarga de recorrer el proyecto en busca de los *endpoints* expuestos y crea la interfaz para ser consumida por el cliente. Para instalar este programa es necesario realizarlo a través de un sistema de gestión de paquetes para Python. Este paquete se llama **pip** y sirve para instalar y administrar paquetes de software escritos en Python.

Los pasos a seguir para incluir este proyecto dentro de la aplicación son:

1. Instalar el proyecto con pip. En la línea de comandos insertar:
```
pip install django-rest-swagger
```

2. Añadir en el archivo “settings.py” del proyecto “api_catalog” la dependencia del proyecto “rest_framework_swagger”:
```
INSTALLED_APPS = (
        …,
    'rest_framework_swagger',
 )
```

3. En el archirvo “urls.py” indicar la ruta deseada para añadir las vistas generadas por swagger para poder ser consumidas por el cliente.
```
urlpatterns = [    …,
        url(r'^docs/', include('rest_framework_swagger.urls')),
        …]
```

Siguiendo estos pasos, Swagger automáticamente recorre el proyecto y ofrece una vista con los endpoints disponibles.

En la figura 4.16 se puede observar la interfaz de Swagger UI generada para este proyecto. En ella se pueden observar los recursos con las operaciones que se pueden realizar, cumpliendo así con el diseño especificado.

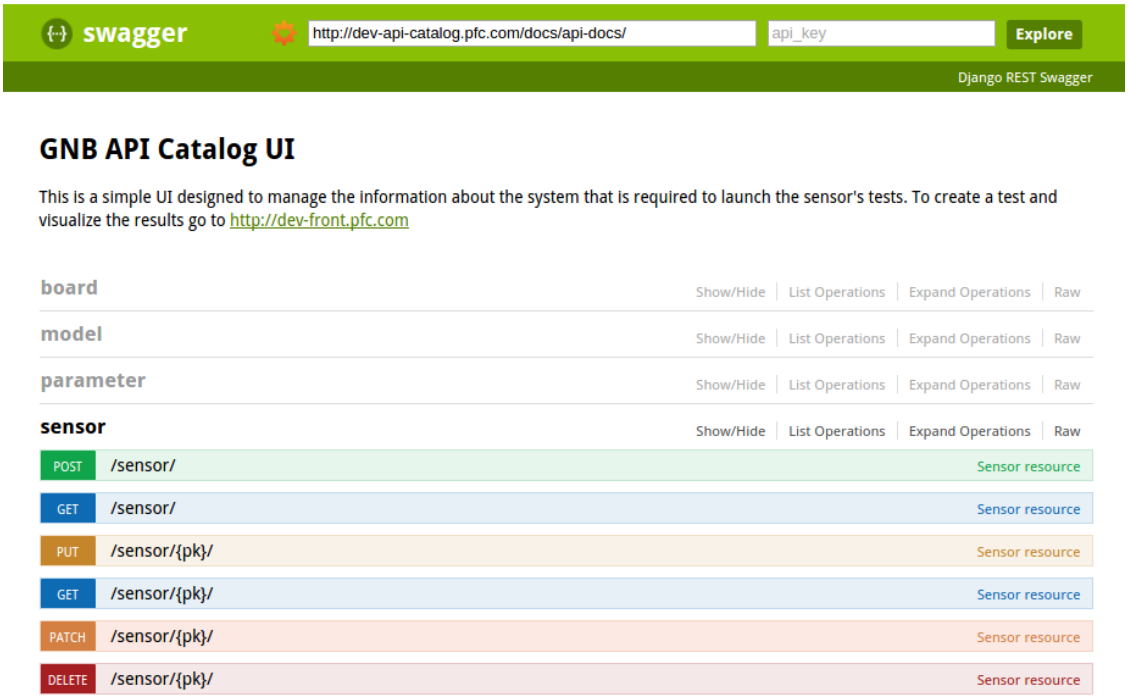

Figura 4.16: Interfaz gráfica generada por Swagger para la gestión del catálogo del sistema.

4.2.4.2. Interfaz de administración de las pruebas

La creación de pruebas y la visualización de los resultados de cada una de ellas se realizan en una interfaz gráfica distinta a la interfaz de gestión del catálogo. Esta interfaz está separada de la administración del catálogo para ofrecer una interfaz amigable y responsiva, de manera que pueda ser consultada desde cualquier tamaño de pantalla. De esta manera, se separa el rol de administrador de la base de datos del sistema y el rol de usuario que ejecuta las pruebas. Todo el código puede ser consultado en el Anexo E.

La adaptabilidad de tamaño de la página web se consigue gracias a la utilización de la librería **Bootstrap** (Apartado 3.2.2.1). Esta librería es un conjunto de archivos CSS y JavaScript que provee clases que adaptan la vista al tipo de pantalla en la que se está mostrando. La plantilla utilizada para la interfaz es una plantilla gratuita llamada “SB Admin 2” [52] que se basa en Bootstrap para ofrecer una interfaz amigable y con aspecto de panel de administración.

El framework principal de la aplicación es AngularJS, ya que permite crear aplicaciones web que se ejecutan en el explorador del cliente ofreciendo mejor experiencia de usuario y aliviando la carga de trabajo por parte del servidor.

El framework de AngularJS se basa en el patrón vista-controlador, separando la lógica de negocio de la vista que se presenta al usuario. Por este motivo, existe un directorio llamado “*scripts*” donde se encuentran todos los archivos JavaScript, incluidos los controladores, y otro directorio llamado “views”, donde se ubican los archivos HTML con las vistas asociadas a cada controlador.

El archivo principal de la aplicación es “app.js”. En este archivo se declaran todos los módulos creados para la aplicación y librerías externas que se usan en la aplicación. Además, con la librería “**ngRoute**” propia de AngularJS se especifican todas las rutas de la aplicación, indicando el controlador y la vista asociados a cada ruta. También se crean las factorías para almacenar el objeto del dispositivo y del modelo seleccionado cuando hay una transición de las vistas y poder recuperarlo en la siguiente vista. Por último, se declaran los servicios que serán llamados para transformar un número a texto, y viceversa, y el servicio que serializa los formularios a formato JSON para ser enviados por la red. La vista asociada a este JavaScript, y por tanto la vista principal, es “*index*.html”. En esta vista se carga en menú lateral de la aplicación y se va intercambiando la vista mostrada según la ruta indicada en la etiqueta HTML “*div*”, la cual contiene el atributo “*ng-view*”, que indica a AngularJS la sección en la que se tienen que mostrar las vistas seleccionadas por “*ngRoute*”.

4.2.4.2.1. Módulos de la interfaz

La información mostrada en la herramienta web se divide en distintos módulos según la selección del menú lateral. En la figura 4.17 se muestra el detalle de la página principal.

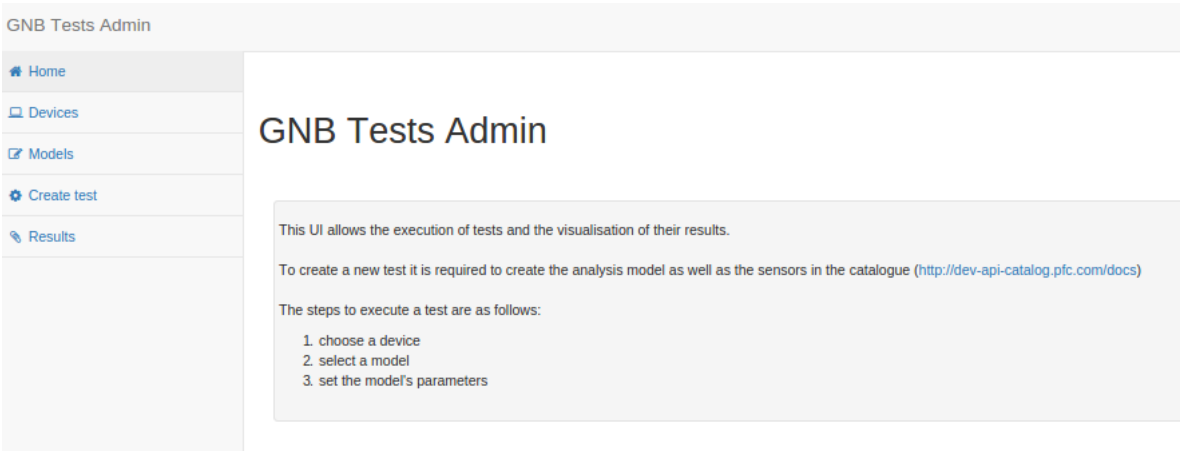

Figura 4.17: Página principal de la interfaz gráfica de usuario para la gestión de las pruebas. En el margen izquierdo se puede observar el menú lateral.

Boards

La información de todos los dispositivos de la plataforma se muestra en la sección del menú llamada “Boards”. Esta vista contiene una pequeña información de cada dispositivo, nombre y descripción, pudiendo ampliar la información de cada dispositivo desplegando una pantalla emergente al seleccionar un dispositivo. Esta vista se encuentra en “boards.html”. El controlador asociado a esta vista se encarga de consultar la información de los dispositivos mediante llamadas HTTP al servicio catálogo. Además, como esta información se devuelve paginada se gestionan las consultas para mostrar todos los dispositivos.

Models

Los modelos de análisis creados en el servicio catálogo se muestran del mismo modo que los dispositivos que ejercen de interfaz con los sensores. De igual manera, la vista se encuentra en el archivo "models.html" y el controlador asociado en "models.js". En este caso, la información solicitada al servicio corresponde a los modelos, mostrando la información de nombre y descripción del modelo, pudiendo desplegar toda la información de éste.

Create tests

La creación de un test se realiza en la sección "*create test*". Los pasos que hay que realizar para crear un test son los siguientes:

1. Seleccionar el dispositivo en el que se quiere ejecutar el modelo. Esta vista es "create-test-board.html" y se muestran los dispositivos disponibles. El controlador asociado a la vista es "create-test-board.js" y se encarga de pedir los dispositivos al servicio catálogo, y cuando se selecciona un dispositivo, redirige la vista a la selección del modelo.
2. Elegir el modelo que se quiere ejecutar. Para ello, se muestra la vista "create-test-model-html". Una vez seleccionado, se redirige a la vista para poder rellenar los campos necesarios para ejecutar el modelo. El controlador "create-test-model.js" se encarga de hacer esta redirección pasando a la nueva vista el modelo y el dispositivo seleccionado.
3. Rellenar los campos para el modelo seleccionado. La vista "create-test-parameters.html" se genera automáticamente según los parámetros del modelo, mostrando la información de forma clara y sencilla. El controlador "create-test-parameters.js" se encarga de generar el JSON que se envía, mediante una petición HTTP POST con la información referente al modelo, al servicio catálogo para ejecutar la prueba. Cuando recibe la respuesta del servicio, el controlador redirige la vista para mostrar el detalle del test ejecutado y poder monitorizar los resultados.

Results

El conjunto de todos los test se muestran en la pestaña "Results" del menú. Esta vista se encuentra en el archivo "results.html" y consta de una tabla en la que se presentan las pruebas realizadas, pudiendo ir al detalle de cada prueba. El controlador "results.js" se encarga de realizar las peticiones HTTP al servicio catálogo para poder consultar estas pruebas.

Por último, el detalle de una prueba se muestra en la vista "results-id.html". Se trata de una vista en la que se presenta la información de los parámetros aplicados en el test y un cuadro gráfico en el que se ofrecen los datos generados por los sensores. Para poder mostrar este gráfico es necesario elegir el número de muestras que se quieren consultar. En función de ese número se realizan las peticiones HTTP correspondientes a cada página para cada sensor o variable de la prueba que se quiera visualizar. A partir de esta selección se pueden visualizar los datos en la gráfica y desplazarse para ver los datos por completo.

Toda esta lógica se ejecuta en el controlador “results-id.js”. Este controlador se encarga de llevar a cabo todas las peticiones para cada variable y mostrar en la vista los datos seleccionados. Además, genera la gráfica que se desea visualizar mediante la librería “***n3-charts***”, que se trata de una adaptación de la librería de JavaScript D3JS [68] para AngularJS.

La monitorización en tiempo real también está contemplada gracias a una tarea que ejecuta un código en el cliente para comprobar el estado del test periódicamente. Si la prueba se está ejecutando, mediante la técnica “*polling*”, se hacen peticiones cada cierto tiempo al servicio que gestiona los datos almacenados por los sensores, y de esta manera, se proporciona la monitorización en tiempo real y se actualiza la vista con los datos correspondientes. El código que realiza esta tarea se muestra a continuación.

```
$scope.stop = $interval(function() {
      if(($scope.result.status === "running" || $scope.result.status === "ready") &&
!$scope.pauseTest) {
        if (!isNextOrPrev) {
          getTest({ignoreLoadingBar: true});
          $scope.numSensorRequest = $scope.numInitialSensorRequest;
          sendRequest($scope.urlArray);
          $scope.nextUrlArray = [];
          $scope.prevUrlArray = [];
          isNextOrPrev = true;
          isRealTime = true;
        }
      }
}, $scope.timeRequestGraph);
```

4.2.5. Integración de nuevos sensores en el sistema

Antes de integrar un nuevo sensor en el sistema se debe añadir un programa que se comunique con los sensores en el dispositivo interfaz. Los tres primeros pasos son comunes a todos los dispositivos interfaz ya que son los servicios ofrecidos en la nube y los debe ejecutar el usuario cuyo rol sea administrador. Estos pasos son:

1. Crear el modelo en el catálogo y obtener el identificador del modelo. Esto se realiza mediante la interfaz creada para la gestión del catálogo. El identificador es necesario para poder crear la llamada HTTP al recurso de la API REST del dispositivo que lanza el programa destinado a la captura de datos del sensor.

2. Añadir los parámetros correspondientes al modelo en el catálogo. Igual que en el paso 1, se realiza mediante la misma interfaz. Nuevamente, es necesario obtener los identificadores de los parámetros ya que son usados en el siguiente paso.

3. Creación del objeto JSON a enviar desde el servicio encargado de la gestión del catálogo al dispositivo interfaz, encargado de recibir la configuración del modelo.

Para ello es necesario disponer de los identificadores del modelo y de los parámetros y crear el objeto en el archivo “create_info.py” (Anexo D).

Se puede consultar un ejemplo de estos pasos detallados en la creación de un modelo para las pruebas realizadas en el Apartado 5.1.

Además, es necesario añadir la lógica en el dispositivo interfaz destinado a recibir los parámetros y lanzar el programa de captura de datos. El dispositivo empleado en las pruebas es una Raspberry Pi, por lo que la API REST desarrollada en este proyecto está orientada a Java. Por tanto, el **requisito** mínimo del dispositivo para poder ejecutar esta API REST es poder ejecutar código **Java**. Para implementar un nuevo modelo en la API REST seguir los pasos detallados en el Apartado 4.2.1.1, creando el controlador y el comando a ejecutar en el servicio.

En cuanto al desarrollo del programa encargado de la captura de las muestras tomadas por el sensor, si el dispositivo a utilizar es una Raspberry Pi se pueden seguir las directrices del Apartado 4.2.1.2.1 para comunicarse con un sensor. Si es otro tipo de dispositivo habrá que adaptar las lecturas a las librerías propias del dispositivo.

Las funciones reflejadas en el apartado 4.2.1.2.2 para el envío de datos están escritas en Python, por lo que si se desean usar estas funciones el dispositivo empleado debe ser capaz de ejecutar **Python** y tener instaladas las dependencias descritas en ese apartado.

Por último, la integración de nuevos sensores y dispositivos se realiza mediante la misma interfaz gráfica utilizada en la creación del modelo. En el Apartado 5.1.1 se muestran ejemplos reales de cómo añadir estos dispositivos al catálogo del sistema.

5. Pruebas y resultados

A lo largo de este capítulo se presentan las pruebas realizadas para comprobar el correcto funcionamiento del sistema así como mostrar la funcionalidad completa de la herramienta desarrollada.

El dispositivo encargado de comunicarse con la plataforma software, y por tanto con los servicios alojados en ella, utilizado en las pruebas es una Raspberry Pi que dispone de conexión Ethernet y permite utilizar adaptadores con conexión WiFi. Además, la utilización de este ordenador de placa reducida permite alimentar sensores mediante los pines GPIO, pudiendo modular la corriente de alimentación de éstos, y dispone de un bus SPI, permitiendo comunicarse con un conversor analógico-digital destinado a la captura digital de los valores de estos sensores. También se pueden realizar medidas de sensores resistivos con adaptación de circuitos RC permitiendo la lectura de los sensores mediante los pines GPIO. De esta manera, se aporta un dispositivo que ofrece una solución universal para la conectividad y la comunicación con diversos tipos de sensores, convirtiéndolos en sensores inteligentes, ya que disponen de un ordenador capaz de procesar sus datos *in situ*.

Las pruebas realizadas en este apartado quedan reflejadas en la Tabla 5.1 y detalladas en los apartados de este capítulo.

Tabla 5.1: Resumen de las pruebas realizadas.

Sensor	Tipo de prueba	Descripción
Potenciómetro	Circuito RC	Demostración de la adaptación de un circuito RC para la lectura de un sensor.
Fotorresistencia	Circuito RC	Utilización de un sensor real en una adaptación RC.
Potenciómetro	ADC	Demostración de la adaptación de un ADC para la lectura de un sensor.
Sensor de temperatura LM335	ADC	Utilización de un sensor real con un ADC. Además, se muestra la monitorización de variables propias del programa.
Fotorresistencia	ADC	En esta prueba además del sensor, se puede monitorizar si un LED está encendido o apagado.
Sensor de odorantes TGS2600 y sensor de temperatura LM35	ADC	Utilización de una nariz electrónica con un ADC. Esta nariz está modulada mediante regresión lineal, por ello se monitorizan las variables de la regresión. Además se monitoriza el valor de la nariz y de un sensor de temperatura.
Virtual	Carga	Envío simultáneo de múltiples peticiones para comprobar el comportamiento del servicio destinado a la gestión de resultados de los sensores a la hora de almacenar la información.
Virtual	Carga	Envío simultáneo de múltiples peticiones para comprobar el comportamiento del servicio destinado a la gestión del catálogo del sistema frente a la actualización del estado de una prueba.

5.1. Captura de datos mediante circuitos RC

La lectura de sensores resistivos se puede llevar a cabo gracias a la utilización de circuitos RC. Un circuito RC (Figura 5.1) es un circuito compuesto de resistencias, en este caso sensores, y condensadores alimentados por una fuente eléctrica. En el circuito se coloca una resistencia en serie con un condensador. Cuando se aplica un voltaje a través de estos componentes la tensión en el condensador incrementa. El tiempo de carga del circuito es directamente proporcional a las magnitudes de resistencia eléctrica (R) y capacidad (C) del condensador. El producto de la resistencia por la capacidad se llama constante de tiempo del circuito.

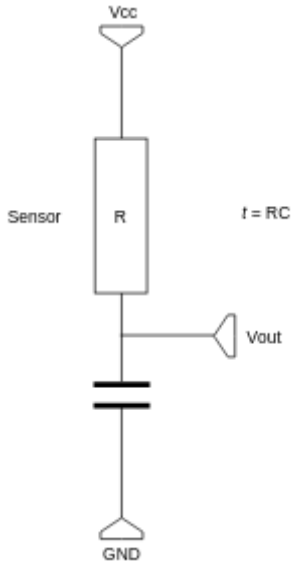

Figura 5.1: Circuito RC

El modelo creado para estas pruebas consiste en un programa que consulta el tiempo que tarda en cargarse el condensador y envía el dato periódicamente a la nube de la siguiente manera:

```
output_value = rc_time(sensor_pin_output)  # Measure timing using GPIO
# Crea la información que se envía a la plataforma
info = {
   "testId": test_id,
   "boardId": board_id,
   "sensorId": sensor_id,
   "inputValue": 3.3,
   "outputValue": output_value,
   "testTime": current_milli_time()
}
Thread(target=send_value, args=(info, )).start()  # Envía la información
time.sleep(sleep_time)  # Espera hasta la siguiente lectura
```

La lectura del tiempo de carga del condensador se realiza mediante la siguiente rutina:

```
def rc_time(pin):
    measurement = 0
    #  Descarga del condensador
    GPIO.setup(pin, GPIO.OUT)
    GPIO.output(pin, GPIO.LOW)
    time.sleep(0.1)

    GPIO.setup(pin, GPIO.IN)
    # Cuenta el número de ejecuciones hasta que el pin recibe una señal lógica “alta”
    while GPIO.input(pin) == GPIO.LOW:
        measurement += 1

    return measurement
```

En la sección de código anterior se puede observar en primera instancia cómo se descarga el condensador, para después contar el número de veces que se ejecuta el bucle hasta que el pin digital recibe una señal lógica “alta”, indicando que el condensador se ha cargado. El tiempo de carga es el número de lecturas realizado. El código completo se puede consultar en el Anexo F.

La creación del modelo en el sistema se realiza mediante el uso de la interfaz gráfica destinada a la gestión del inventario del sistema (dirección web del ejemplo: http://dev-api-catalog.pfc.com/docs). Para ello se han de seguir los siguientes pasos:

- Creación del modelo: nombre y descripción (Figura 5.2). Este modelo es genérico para cualquier sensor resistivo que se inserte en un circuito RC y del cual se pretenda capturar únicamente el valor que devuelve cada cierto tiempo.

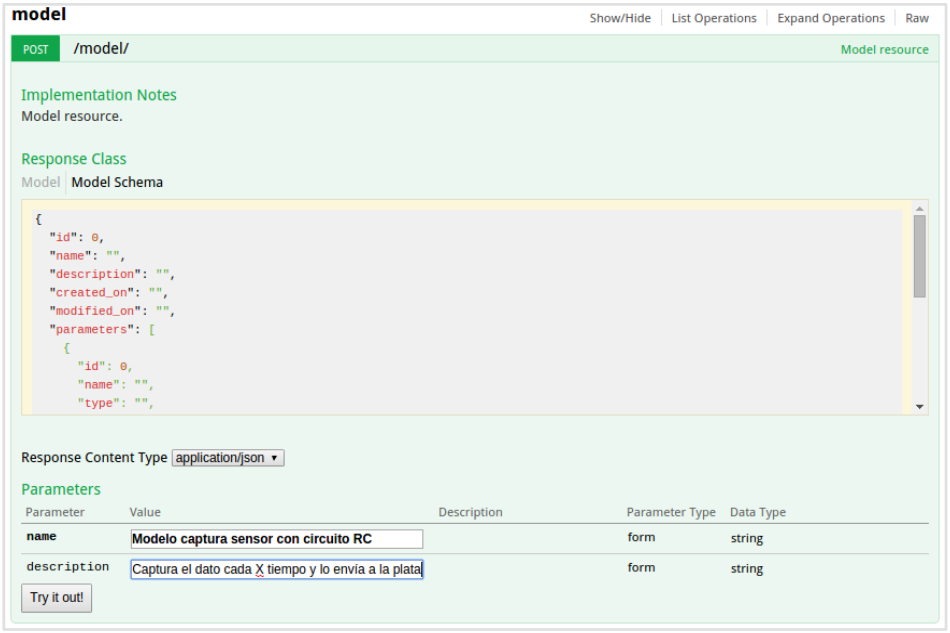

Figura 5.2: Creación de un modelo en la interfaz de usuario destinada a la gestión del catálogo.

- Definición de los parámetros (Figura 5.3). Este modelo consta de dos parámetros:
    - El sensor que se va a utilizar. En este caso al ser de tipo sensor sólo hay que indicar el nombre, el modelo al que pertenece el parámetro y el tipo.

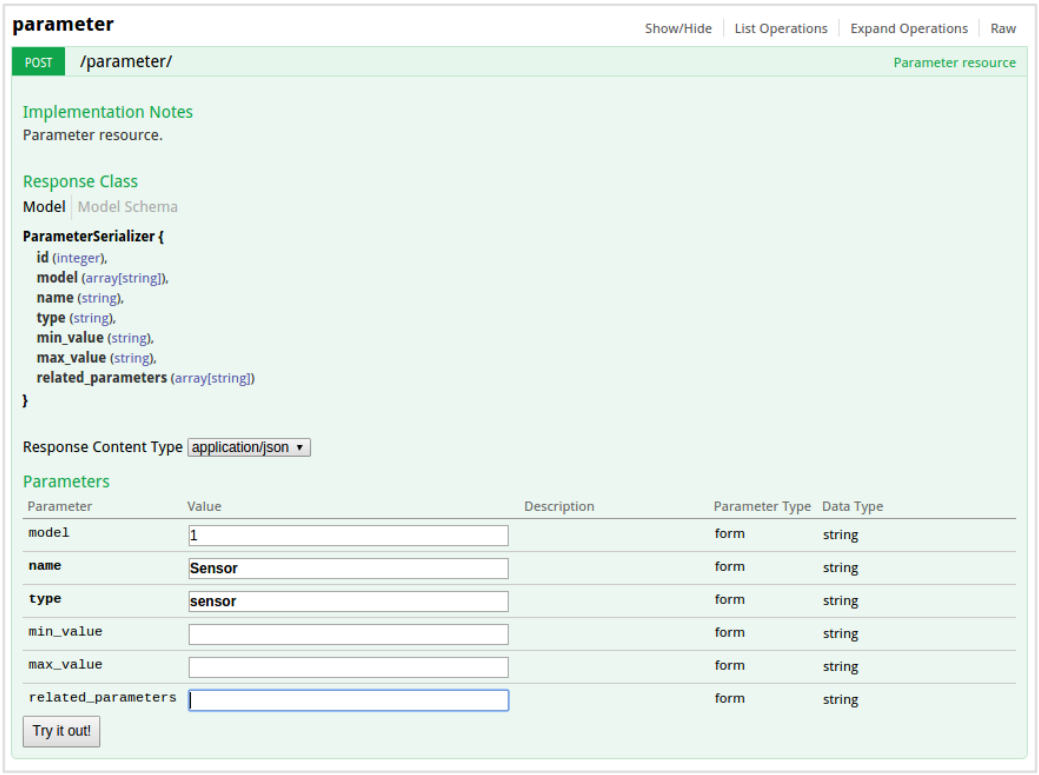

Figura 5.3: Creación del parámetro "sensor" en la UI destinada a la gestión del catálogo

- El tiempo de envío de datos en segundos. Este parámetro tiene que ser positivo y aceptar decimales.

Figura 5.4: Creación del parámetro "tiempo de envío" en la UI destinada a la gestión del catálogo.

Una vez creado el modelo en la base de datos que gestiona el catálogo hay que seguir los pasos descritos en el apartado 4.1.1.2. Para ello es necesario añadir la información de este modelo en el archivo “create_info.py” de la API de gestión de catálogo que crea el objeto JSON que se envía a la API embebida dentro del dispositivo. Esto se realiza de la siguiente manera:

```
if model == 1:
        sensor_id = 0
        sensor_pin_input = 0
        sensor_pin_output = 0
        sampling_time = 0

        for param in test_parameters:
            if param['model_parameter'] == 1:  # sensor id
                sensor = Sensor.objects.get(pk=param['value'])
                sensor_id = sensor.id
                sensor_pin_input = sensor.pin_input
                sensor_pin_output = sensor.pin_output

            if param['model_parameter'] == 2: # sampling Time Id
                sampling_time = param['value']

        sensor1 = {
            "sensorId": sensor_id,
            "pinInput": sensor_pin_input,
            "pinOutput": sensor_pin_output,
        }
        info = {
            "boardId": board_id,
            "testId": test.id,
            "samplingTime": sampling_time,
            "sensor1": sensor1
        }
```

En el código anterior se aprecia cómo se recorre cada parámetro recibido de la prueba que se desea ejecutar y se genera la información en formato JSON. Es necesario conocer los identificadores de cada parámetro del modelo y extraer la información, por eso es imprescindible anotarlos durante la creación a través de la interfaz.

Siguiendo los pasos descritos en el apartado 4.1.1.2. se crea en la API REST embebida dentro del dispositivo la entidad correspondiente al modelo , así como el servicio encargado de lanzar el programa y el endpoint que recibe los parámetros. Estos pasos están descritos en el apartado 4.2.1.1.

La demostración de la adaptación para la captura de los datos se realiza mediante dos pruebas: un potenciómetro para comprobar la respuesta del condensador y un fotorresistor con el que se demuestra el funcionamiento con un sensor.

5.1.1. Potenciómetro

El uso de un potenciómetro permite estudiar la respuesta y la adaptación de los circuitos RC en una Raspberry Pi. Para ello, se emplea un potenciómetro de 10 kΩ común y un condensador de 1 µF formando el circuito mostrado en la Figura 5.5.

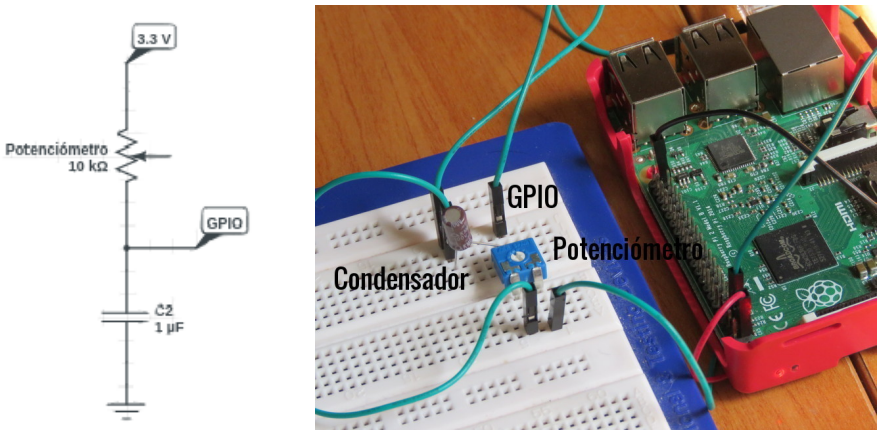

Figura 5.5: Diseño e implementación de un circuito RC con un potenciómetro.

La creación del sensor y de la Raspberry Pi en el servicio encargado de gestionar el inventario del sistema se realiza con la misma interfaz utilizada en la creación del modelo. Para ello hay que realizar las siguientes tareas:

- Creación del dispositivo Raspberry Pi (Figura 5.6). Se indica nombre, descripción, dirección del dispositivo y el número de pines del que dispone la placa, utilizados para evitar errores a la hora de ejecutar pruebas y proteger el dispositivo.

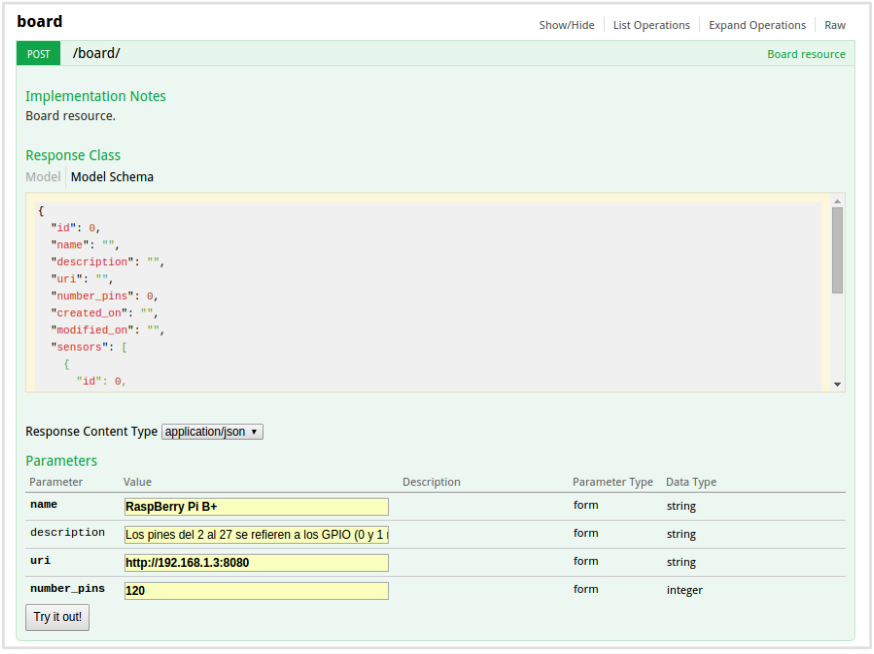

Figura 5.6: Creación del dispositivo "Raspberry Pi" en la UI destinada a la gestión del catálogo.

- Creación del sensor, en este caso el potenciómetro (Figura 5.7). Se indica nombre, descripción, el pin GPIO por el cual se lee la señal, el pin de entrada para alimentar el sensor (en este caso, es ignorado debido a la configuración del circuito, por lo que se indica un pin inexistente en la placa) y el identificador del dispositivo en el que se encuentra el sensor.

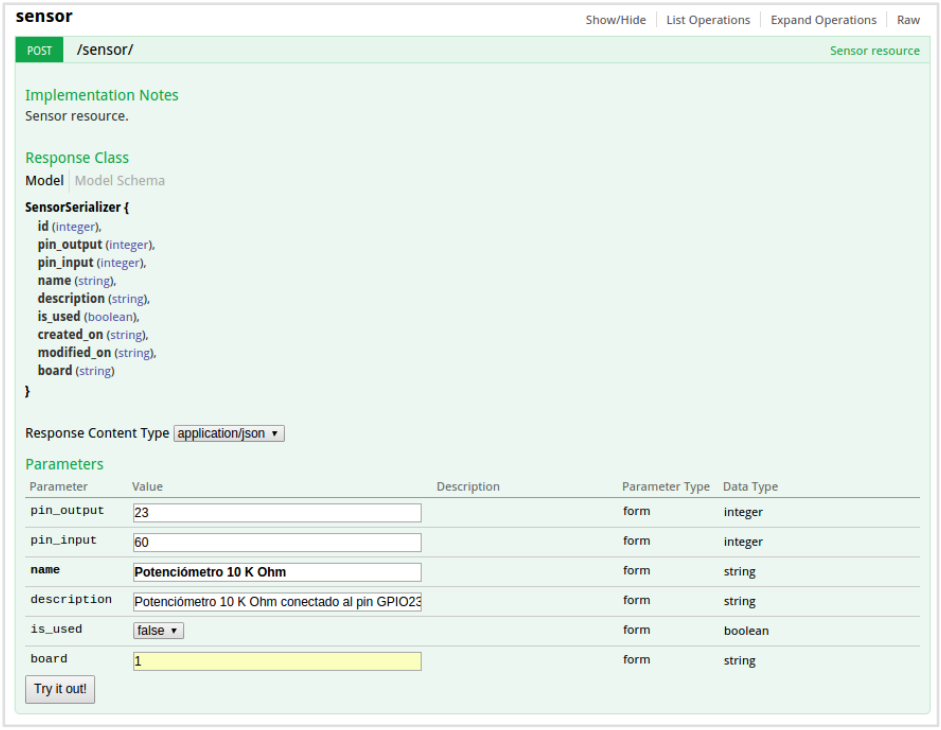

Figura 5.7: Creación del parámetro "tiempo de envío" en la UI destinada a la gestión de catálogo.

La creación de la prueba para ejecutar el modelo así como la visualización de los resultados se realiza mediante la interfaz destinada para este fin (dirección web del ejemplo: http://dev-api-front.pfc.com/#/create-test/). Se deben llevar a cabo los siguientes pasos:

1. Seleccionar el dispositivo en el que se va a ejecutar la prueba (Figura 5.8).

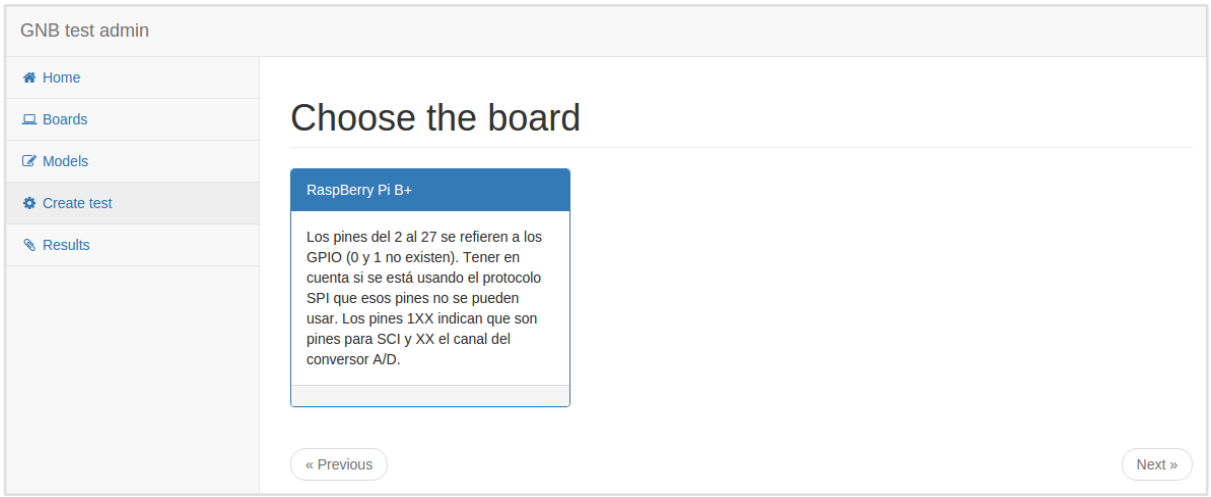

Figura 5.8: Elección del dispositivo para ejecutar una prueba en la UI para gestión de las pruebas.

2. Seleccionar el modelo de la prueba (Figura 5.9).

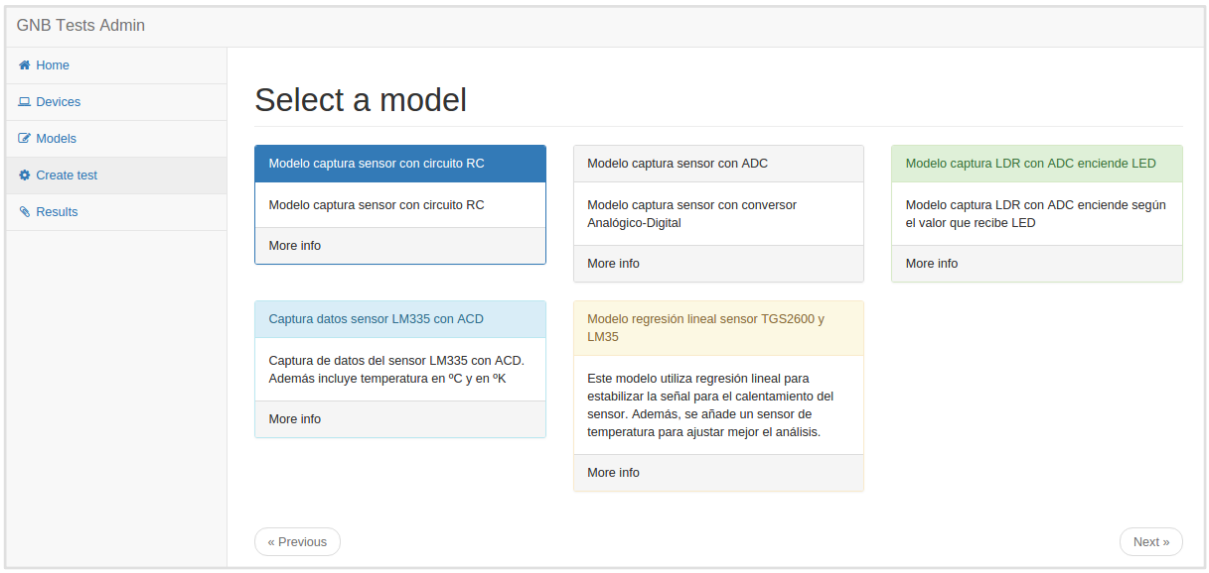

Figura 5.9: Elección del modelo a ejecutar en una prueba mediante la UI destinada a la gestión de las pruebas.

3. Rellenar el formulario con los parámetros del modelo deseados (Figura 5.10).

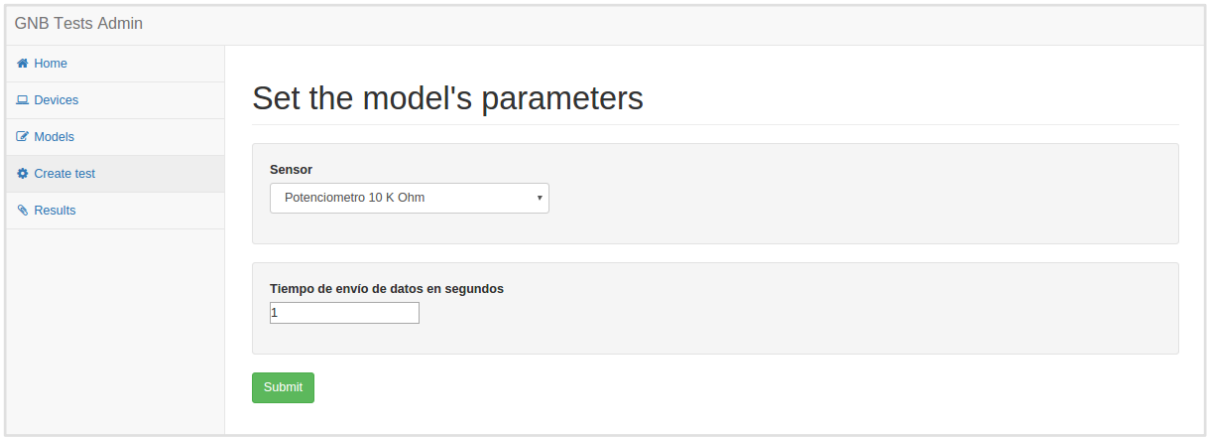

Figura 5.10: Elección de los parámetros del modelo a ejecutar en una prueba mediante la UI destinada a la gestión de las pruebas.

Una vez comienza la prueba, la interfaz redirige automáticamente a la visualización de los resultados.

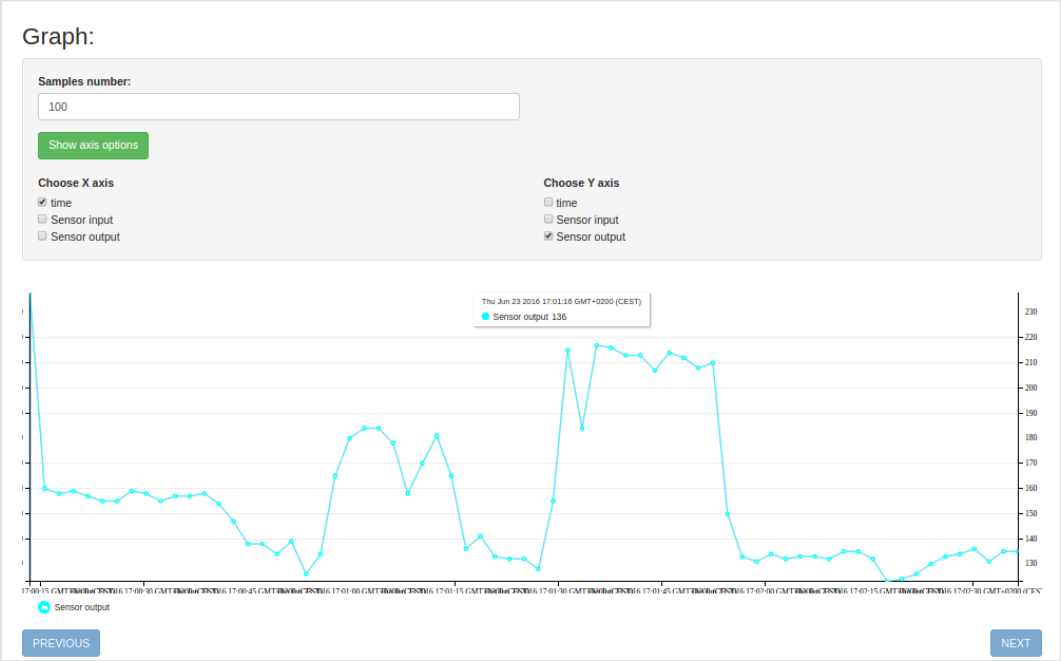

Figura 5.11: Visualización de los resultados de una prueba de un potenciómetro en un circuito RC.

Como se puede observar en la Figura 5.11, el programa funciona correctamente, ya que responde a la modificación de la resistencia del potenciómetro demostrando la posibilidad del uso de sensores resistivos mediante la adaptación de circuitos RC.

5.1.2. Fotorresistencia

En esta prueba se utiliza un sensor tipo LDR para comprobar el funcionamiento del programa anterior con un sensor fotorresistivo. En este caso, sólo es necesario crear el sensor en el catálogo, ya que el modelo y el dispositivo han sido creados en la prueba anterior. Por tanto, se siguen los mismos pasos durante la creación del sensor que en el ejemplo previo . En este caso, el pin de la Raspberry Pi correspondiente a la captura de los datos del sensor es el pin GPIO 12, y de la misma manera, el pin de entrada de alimentación del sensor es ignorado. En la Figura 5.12 se muestra el circuito implementado.

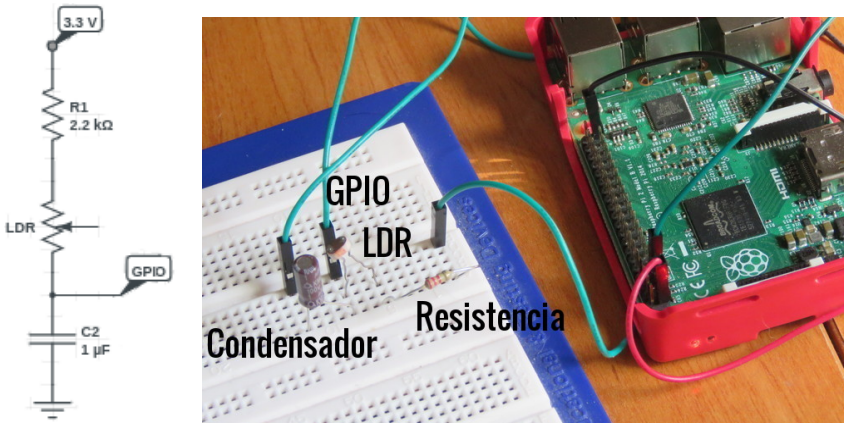

Figura 5.12: Diseño e implementación de un circuito RC con un sensor LDR.

Los resultados de una prueba ejecutada están reflejados en la Figura 5.13. Como se puede apreciar en la gráfica, el sensor comienza iluminado, ya que el valor es máximo, y después se tapa, cayendo el valor de la captura. Por último, se observa cómo se ha variado la incidencia de luz para estudiar la respuesta del sensor y el circuito ante las variaciones de luminosidad.

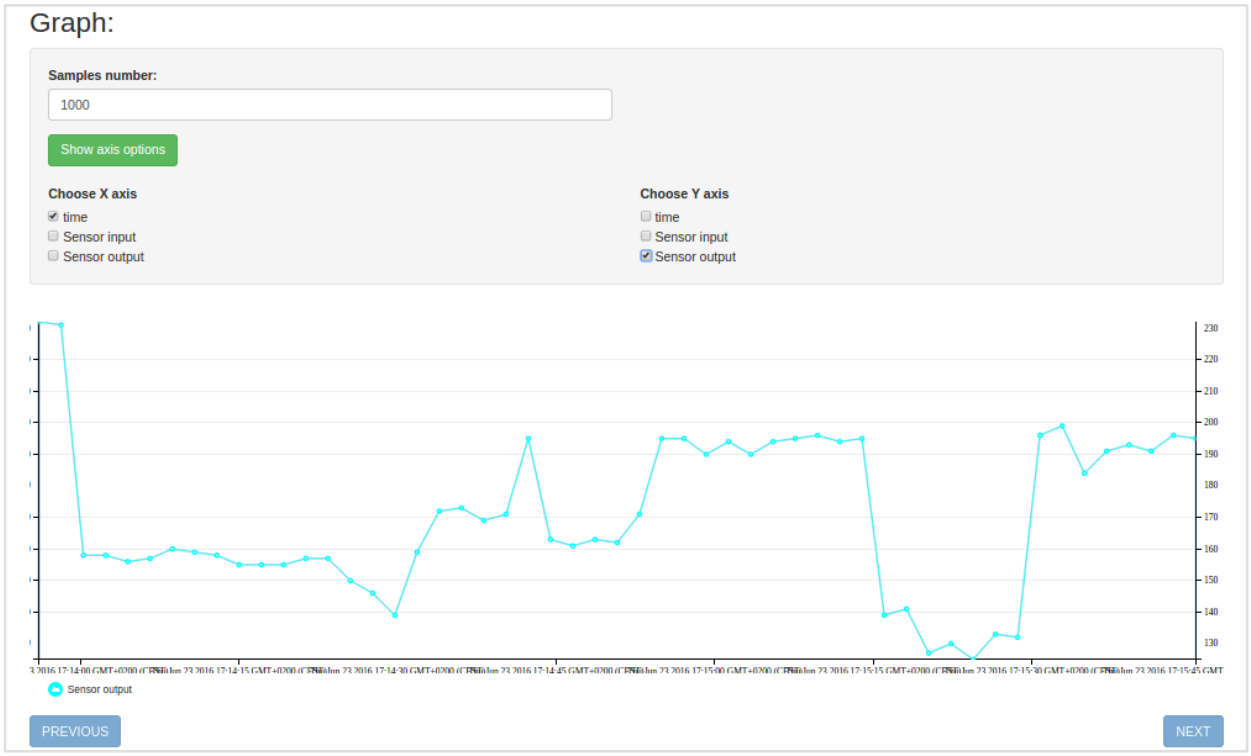

Figura 5.13: Visualización de los resultados de una prueba de un sensor LDR en un circuito RC.

5.2. Captura de datos mediante conversor analógico-digital

Los pines de la Raspberry Pi son digitales, por lo que se hace necesario buscar una alternativa que permita la lectura de sensores analógicos. La captura de datos procedentes de señales analógicas se puede realizar mediante conversores analógico-digitales (*Analogue Digital Converter*, ADC). En estas pruebas se utiliza un ADC modelo MCP3008 [36]. Se trata de un conversor con 8 canales de 10 bits de resolución e interfaz SPI. Esta conexión permite comunicarse con la Raspberry Pi mediante los pines GPIO destinados a este bus de datos. Las conexiones necesarias se detallan en la Tabla 5.1.

Tabla 5.2: Conexiones del ADC MCP3008 con los pines GPIO de la Raspberry Pi B+.

MCP3008	Raspberry Pi
Pin 16 - V_{DD}	5 V
Pin 15- V_{REF}	5 V
Pin 14 - ADNG	GND
Pin 13 - CLK	Pin 23 - SPI_SCLK
Pin 12 - D_{out}	Pin 21 - SPI_MISO
Pin 11 - D_{in}	Pin 19 - SPI_MOSI
Pin 10 - CS	Pin 24 - SPI_CEO
Pin 9 - DGND	GND

Todos los sensores que se conecten a este ACD deberán tener asignados como pin de salida un número comprendido entre el 100 y el 107, para indicar así el canal de lectura.

La comunicación a través del bus de datos SPI se realiza gracias a la librería **spidev**, siendo necesario habilitar el bus y leer los datos de la siguiente manera:

```
# SPI bus
spi = spidev.SpiDev()
spi.open(0, 0)

def read_channel(channel):
    _adc = spi.xfer2([1, (8 + channel) << 4, 0])
    data = ((_adc[1] & 3) << 8) + _adc[2]
    return data
```

Como se puede comprobar en el código, hay que indicar que el ADC tiene 8 canales y así poder leer el canal deseado. Una vez capturado el dato se puede proceder a su conversión a voltios:

```
def convert_volts(data, places=2):
    volts = (data * 5.0) / float(1023)
    volts = round(volts, places)
    return volts
```

Es necesario indicar el voltaje de referencia del ACD (5 V), así como la resolución (en este caso, 2^{10} - 1).

A continuación, se detallan distintas pruebas con un conversor analógico-digital y diversos tipos de sensores demostrando la **capacidad y universalidad del sistema**.

5.2.1. Potenciómetro

La utilización de un potenciómetro permite comprobar la comunicación entre el ADC y el dispositivo. El circuito utilizado consta de un potenciómetro de 10 kΩ alimentado con 5 V, como se muestra a continuación. El circuito se muestra en la Figura 5.14.

Figura 5.14: Diseño e implementación de un potenciómetro con un ADC modelo MCP3008.

La creación del potenciómetro en el catálogo se lleva a cabo como en las pruebas anteriores, definiendo como pin de salida el 100, indicando de esta manera que el canal de lectura del ACD es el 0.

Los parámetros del modelo utilizado para esta prueba son idénticos a los del modelo del apartado 5.1.1. Por este motivo, solamente se crea el modelo en el catálogo y se procede a actualizar la información de todos los parámetros y asignarlos a ambos modelos, evitando así la redundancia de datos. La actualización de los parámetros se muestra en la imagen 5.15, en la cual se puede observar cómo en el campo *body* se asignan ambos modelos.

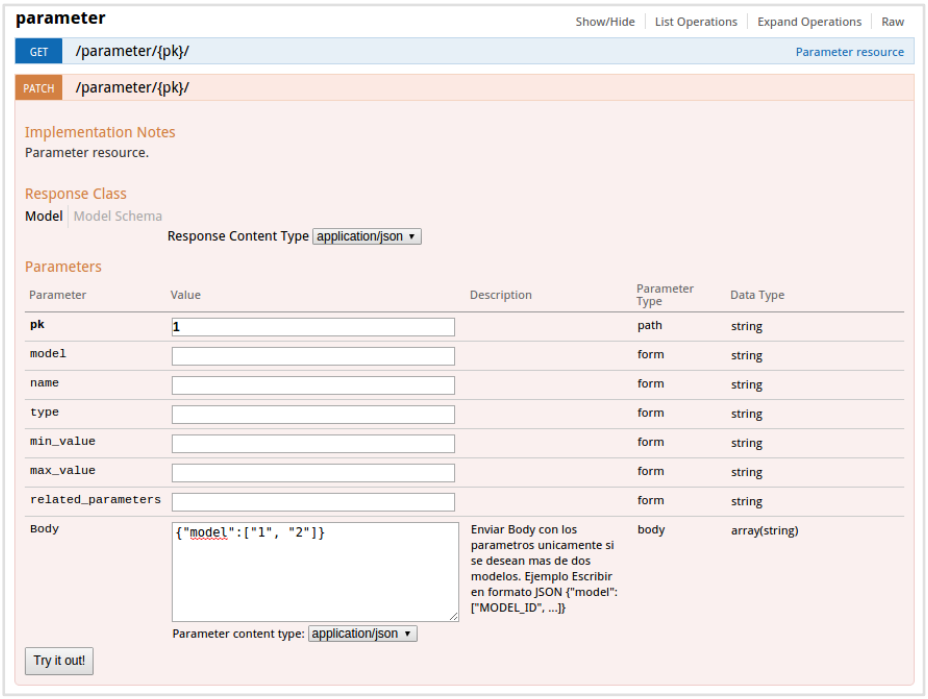

Figura 5.15: Asignación de los parámetros a distintos modelos mediante la UI de gestión del catálogo.

En la Figura 5.16 se muestran los resultados de la monitorización del potenciómetro cuando se ejecuta una prueba que captura el voltaje cada dos segundos. En el gráfico se puede observar cómo el voltaje que deja pasar el potenciómetro varía en función de la modificación de la resistencia variable del potenciómetro.

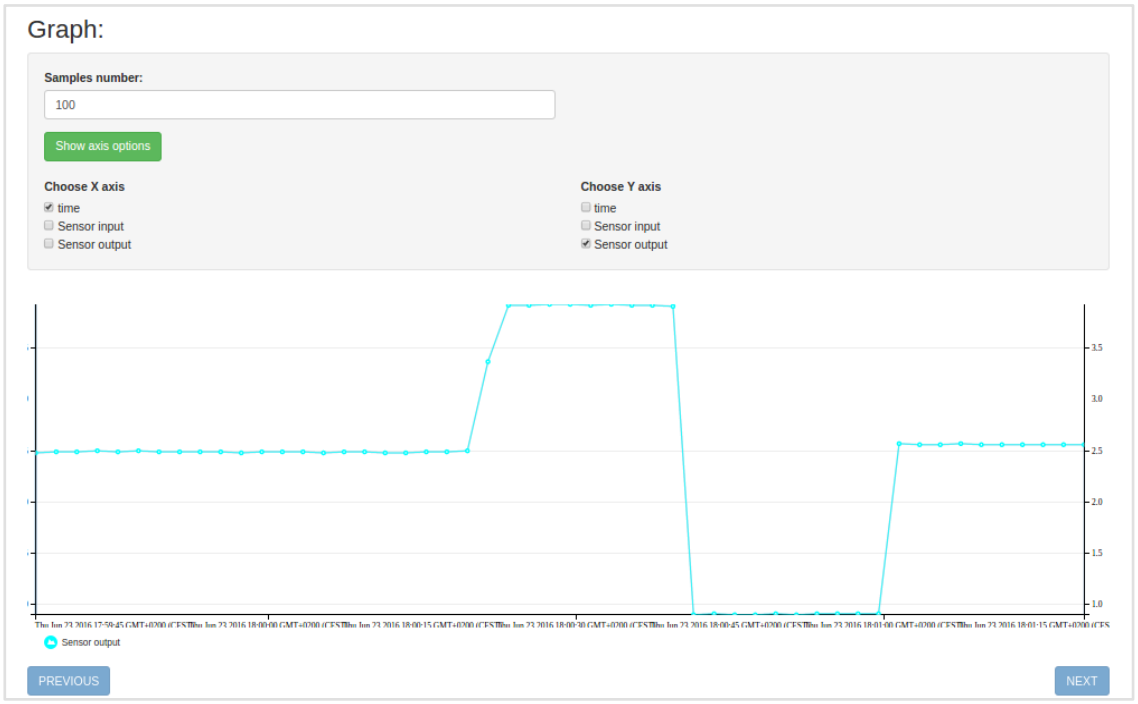

Figura 5.16: Visualización de los resultados de una prueba de un potenciómetro con un ADC en la UI de gestión de las pruebas.

5.2.2. Sensor de temperatura

El objetivo de esta prueba es capturar los datos de un sensor de temperatura mediante un ADC y el valor correspondiente de su temperatura en grados centígrados y Kelvin, enviando la información a la nube donde se almacenará la información. Para ello, se utiliza un sensor LM335 [32] cuya salida está calibrada entre -40º y 100º C y tiene una tensión de ruptura directamente proporcional a la temperatura absoluta con una relación de 10 mV/°K.

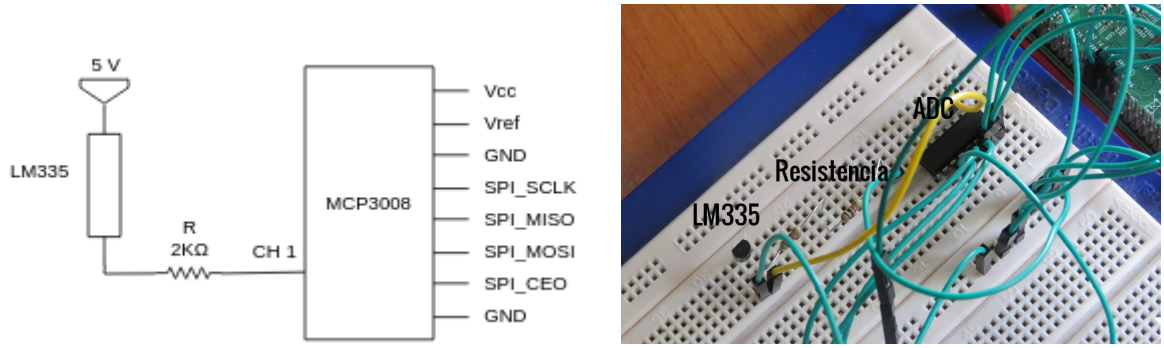

Figura 5.17: Diseño e implementación de un sensor LM335 con un ADC modelo MCP3008.

El circuito implementado se muestra en la Figura 5.17. En este modelo se desea almacenar el valor devuelto por el sensor y el valor correspondiente a la temperatura. Por lo tanto, es necesario crear un sensor que represente el valor de la temperatura (Figura 5.18) y poder persistir la información en el servicio que gestiona los resultados, además de monitorizar la información y mantenerla organizada. Los grados centígrados se almacenan en el campo *inputValue* y los grados Kelvin en el campo *outputValue.* De esta manera, la temperatura se convierte en un sensor “virtualizado”, lo que implica que el modelo requiere también un parámetro de tipo sensor que represente la variable.

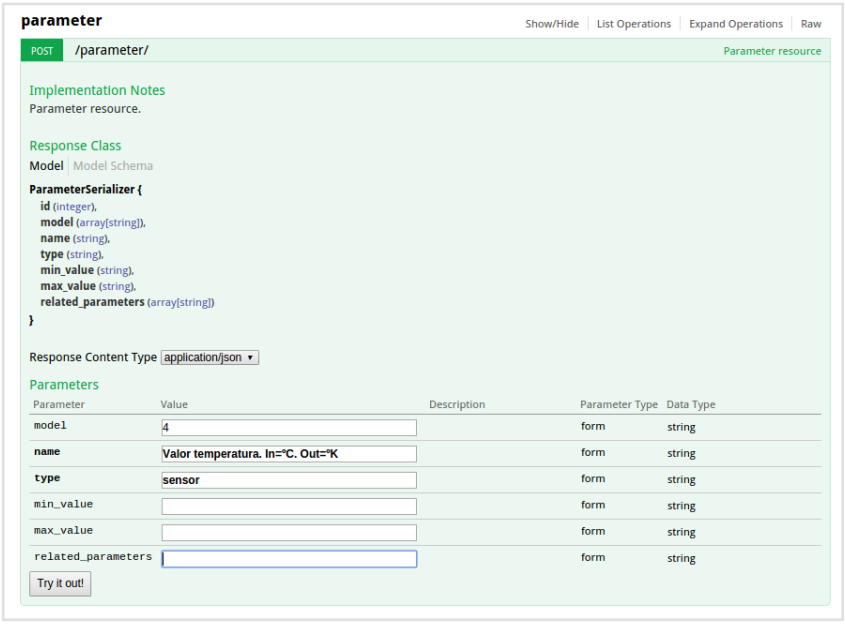

Figura 5.18: Creación mediante la UI de un parámetro de tipo sensor para representar el valor de una variable.

La Figura 5.19 muestra el resultado de una ejecución de este modelo. Se puede observar cómo varía el valor devuelto por el ACD correspondiente al sensor según se acerca o aleja una fuente de calor. En la Figura 5.20 se muestra el valor en grados centígrados.

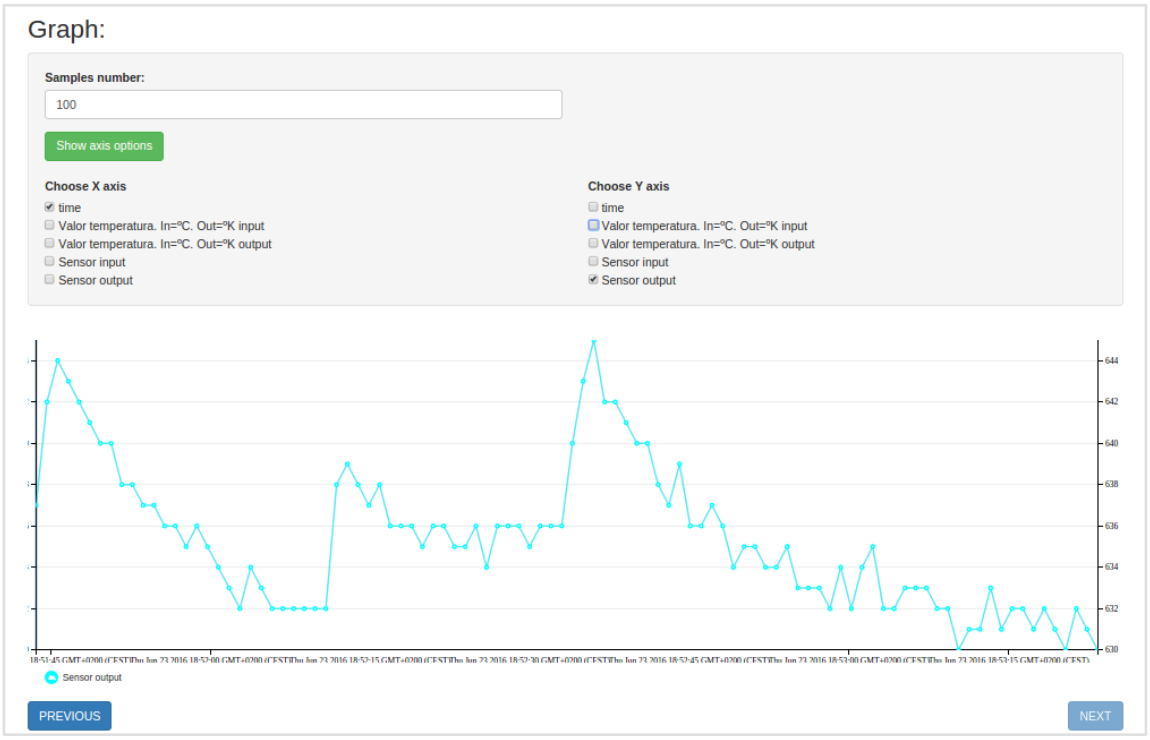

Figura 5.19: Visualización de los resultados de una prueba de un sensor LM335 con un ADC en la UI destinada a la gestión de las pruebas.

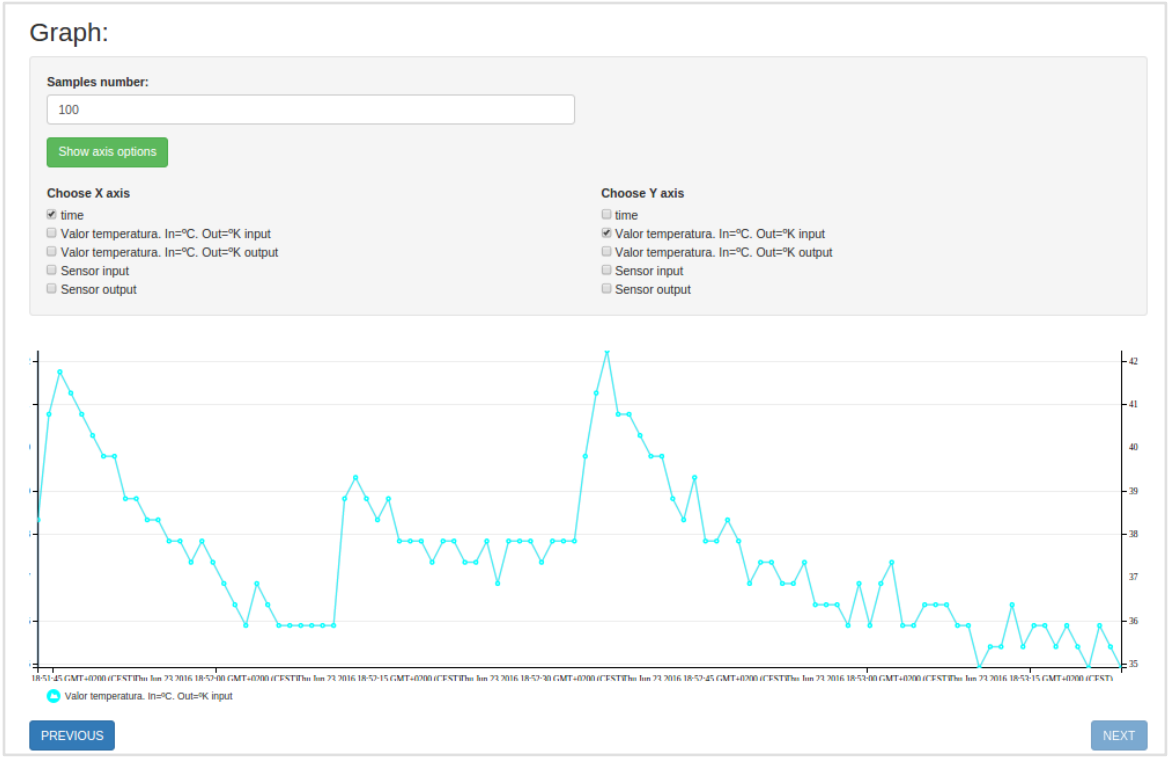

Figura 5.20: Visualización de la variable que representa el valor de un sensor LM335 en grados centígrados.

5.2.3. Fotorresistencia

Esta prueba consiste en capturar los datos de un sensor tipo LDR mediante un conversor analógico-digital. Además de la captura de los datos, se dota de inteligencia al programa para que se encienda un LED si el valor digital de la conversión del sensor se encuentra por debajo de un parámetro recibido en la configuración. El circuito se muestra en la Figura 5.21.

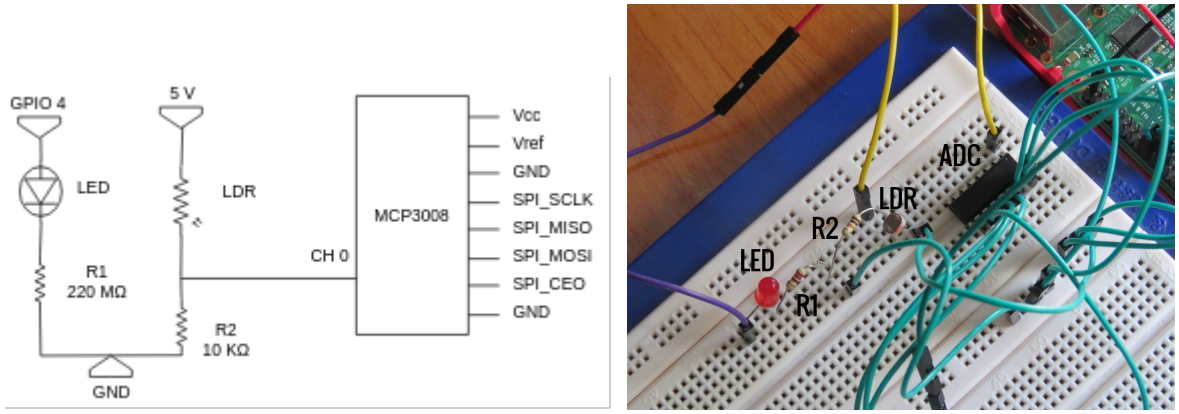

Figura 5.21: Diseño e implementación de un sensor LDR con un ADC modelo MCP3008 y un LED conectado a un pin GPIO.

Nuevamente, se crea el sensor indicando el canal del ADC por el cual se va a leer el valor y se asignan los parámetros de los anteriores modelos a este nuevo. Además, puesto que se quiere monitorizar cuándo el LED está encendido o apagado, es necesario crear un nuevo parámetro. Este parámetro será de tipo sensor, ya que de esta manera, se puede almacenar el valor de la variable en la plataforma, usando el parámetro *oututValue o inputValue*, como si de un sensor se tratara, y así recuperar y monitorizar el valor de la variable en cada momento. Por tanto, será necesario crear un sensor en la plataforma que represente a la variable del modelo que se desea monitorizar.

El valor que indica cuándo se enciende el LED también es un parámetro que está relacionado con este sensor, por lo que hay que establecer un nuevo parámetro y relacionarlo con la variable de tipo sensor creada.

Al disponer de una variable “sensorizada” el programa que ejecuta el modelo es capaz de utilizar el identificador de la variable para enviar a la plataforma si el LED está encendido o no de la siguiente manera:

```
# Comprueba si se tiene que encender el LED
on = 0
GPIO.output(led_pin_input, GPIO.LOW)
if adc < led_on_value:
    GPIO.output(led_pin_input, GPIO.HIGH)
    # Se facilita la visualización asignando el valor recibido para encender el LED
    on = led_on_value
# Envía el valor de la variable “on”.
```

```
info = {
   "testId": test_id,
   "boardId": board_id,
   "sensorId": led_id,
   "inputValue": on,
   "outputValue": 3.3,
   "testTime": current_milli_time()
}
Thread(target=send_value, args=(info, )).start()
```

En la Figura 5.22 se pueden observar los resultados de una prueba en la que la iluminación del sensor cae, y por tanto, se enciende el LED. La línea de color negro representa la salida del sensor. En azul se muestra la señal que indica si el LED está encendido o no.

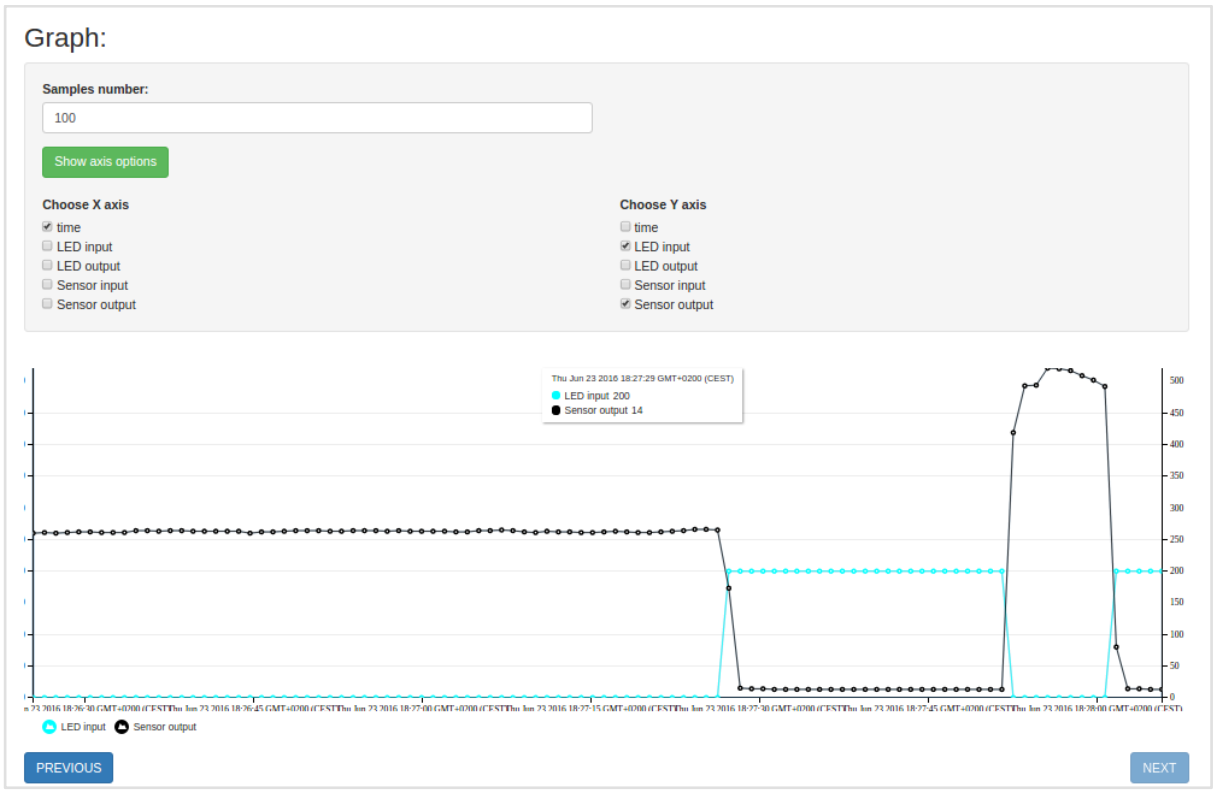

Figura 5.22: Visualización de la salida de un sensor LDR y de la variable que representa si un LED está encendido.

5.2.4. Sensor de odorantes

La utilización de sensores destinados a la detección de odorantes permite utilizar modelos de análisis complejos dentro del dispositivo encargado de enviar los datos, convirtiendo al conjunto sensor-dispositivo en una **nariz electrónica**.

El análisis empleado en esta prueba es mediante regresión lineal. La regresión lineal es un modelo matemático usado para aproximar la relación de dependencia entre una variable

dependiente, las variables independientes y un término aleatorio. Por lo tanto, permite hallar el valor esperado de una variable aleatoria “a” cuando “b” toma un valor específico.

En esta prueba se utiliza un sensor TGS2600 [57]. Estos sensores se caracterizan por tener una temperatura de calentamiento que hace variar la captura de la muestra. Existen estudios que indican que modulando esta temperatura se puede analizar el odorante existente en el ambiente reduciendo los tiempos [74]. Además, la temperatura ambiente y la humedad también afectan a la respuesta del sensor.

El circuito empleado para la nariz electrónica (Figura 5.23) se basa en el estudio realizado por Alejandro Pequeño Zurro en su Proyecto Fin de Carrera “Uso de una nariz electrónica ultra portátil en robots para la detección de fuentes de odorantes” [42]. La implementación de este circuito se muestra en la Figura 5.24.

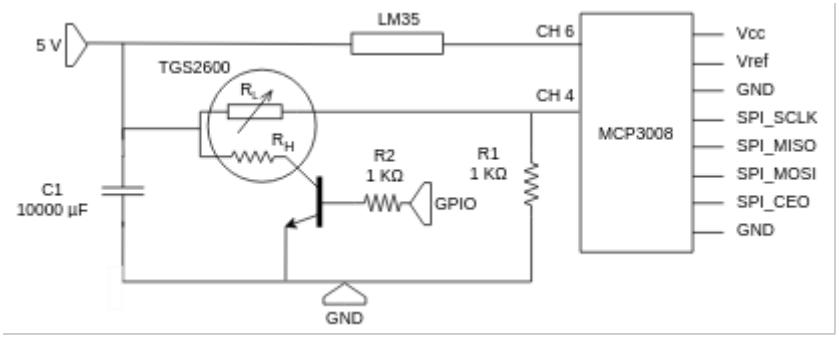

Figura 5.23: Circuito adaptado para la conversión de la señal de salida de un sensor TGS2600 mediante un ADC.

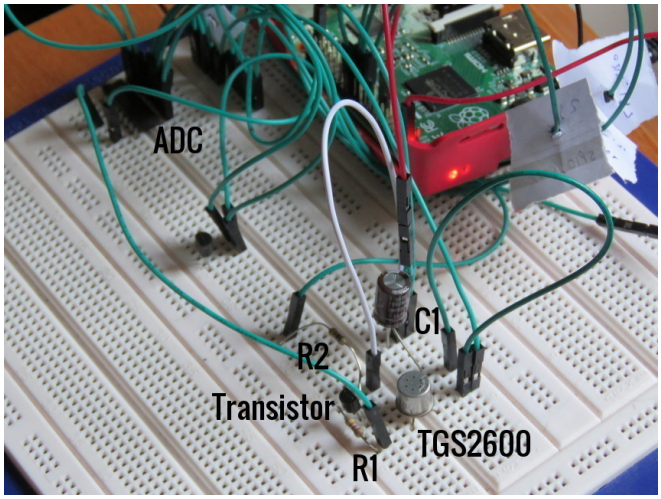

Figura 5.24: Implementación del circuito diseñado en la Figura 4.23.

Los sensores de odorantes necesitan un tiempo de estabilización de la señal, por lo que es necesario que el modelo disponga de un parámetro con el número de muestras que se desean capturar antes de comenzar el análisis. Además, el modelo consta de los siguientes parámetros:

- NM: número de lecturas del ADC.
- T: tiempo en segundos para la captura de datos.
- Sleep: periodo de estabilización cuando se configura una nueva temperatura de calentamiento del sensor.
- SamplesInicio: número de muestras que se capturan en la estabilización del sensor.
- Tendencia: tendencia de la modulación de la temperatura.
- AverageTemperature: temperatura promedio para el calentamiento del sensor.
- Duration: duración del experimento.

Para completar el modelo, a estos parámetros también hay que sumarle el propio sensor de odorantes, un sensor de temperatura para obtener el valor de la temperatura ambiente y los sensores “virtuales” que representan las variables de la regresión lineal. De esta manera, se puede monitorizar el análisis por regresión.

A continuación se muestran en detalle las capturas de las variables (Figura 5.25), así como la respuesta del sensor (Figura 5.26) a 31 grados centígrados cuando se expone a una fuente de odorante.

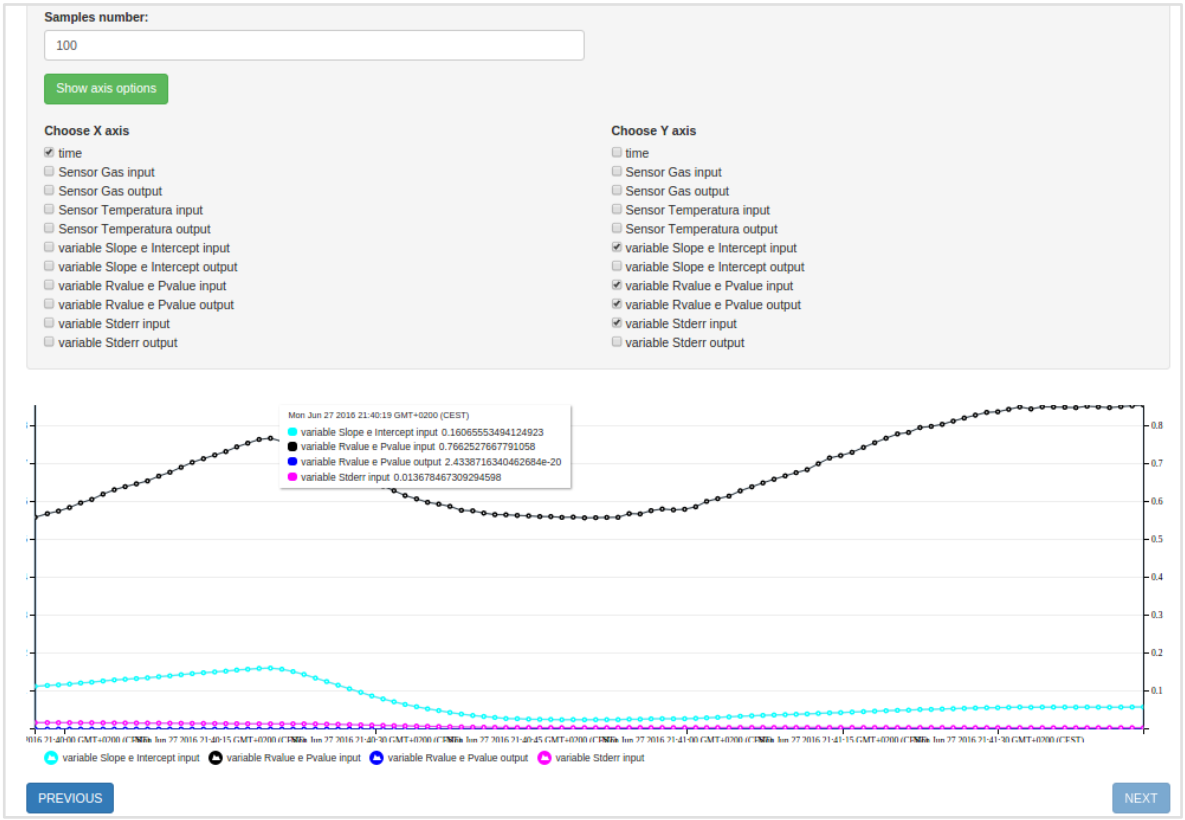

Figura 5.25: Visualización de las variables *slope, intercept, Rvalue, Pvalue* y *stderr* en una prueba con un sensor TGS2600 a una temperatura ambiente de 31ºC.

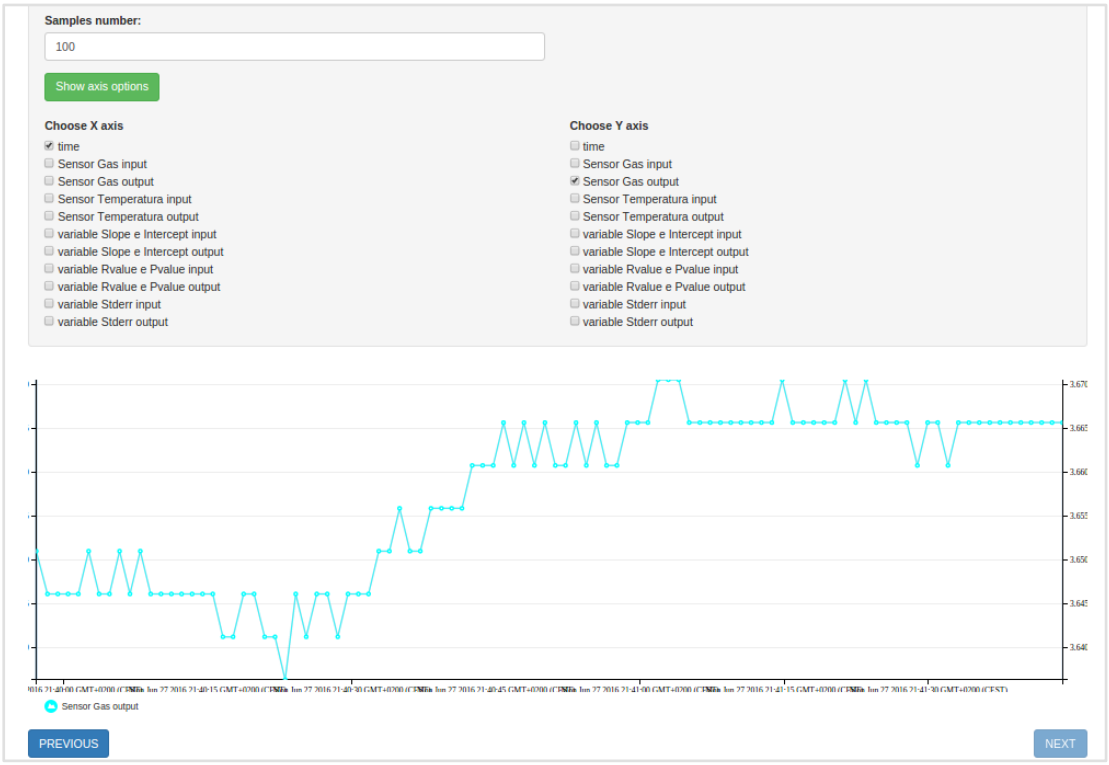

Figura 5.26: Visualización de la respuesta de un sensor TGS2600 a una temperatura ambiente de 31ºC.

En la Figura 5.27 se puede apreciar la modulación de la tensión aplicada al sensor para calentar su resistencia interna y la variación de la modulación cuando se expone una fuente de odorantes.

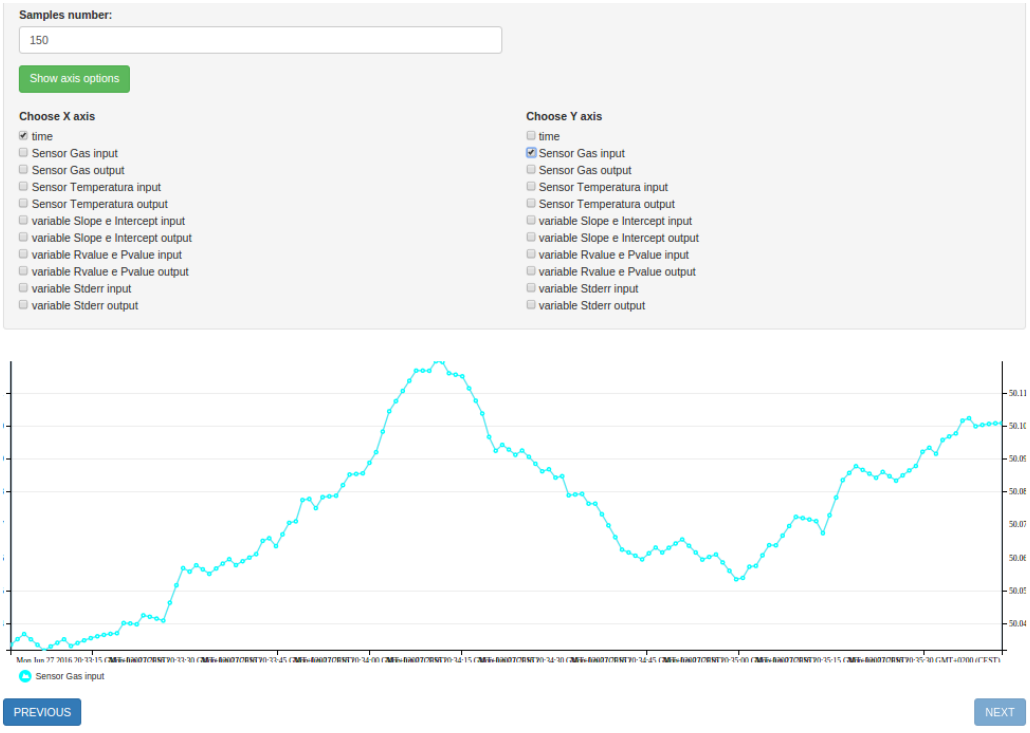

Figura 5.27: Visualización de la temperatura aplicada a un sensor TGS2600 ajustada por regresión lineal.

5.3. Prueba de carga

En la ingeniería del software, las pruebas de rendimiento son las pruebas que se realizan para determinar lo rápido que realiza una tarea un sistema en condiciones particulares de trabajo. Estas pruebas también pueden servir para validar y verificar otros atributos de calidad del sistema tales como la escalabilidad, la fiabilidad y el uso de los recursos.

Las pruebas de rendimiento sirven, entre otros propósitos, para detectar qué partes del sistema o de carga de trabajo provocan que el conjunto rinda de manera poco eficiente. Las herramientas utilizadas en el diagnóstico pueden consistir en monitorizaciones que midan qué partes de un dispositivo o software contribuyen de manera negativa al rendimiento o para establecer niveles del mismo que mantenga un tiempo de respuesta aceptable.

La prueba de carga, por tanto, es un tipo de prueba de rendimiento del software. Una prueba de carga se realiza generalmente para observar el comportamiento de una aplicación bajo una cantidad de peticiones esperada y muestra los tiempos de respuesta de las transacciones de la aplicación.

En este apartado se muestra la capacidad del servicio destinado a la gestión de los resultados de las pruebas soportando la recepción y el almacenamiento de los valores de los sensores. También se comprueba la respuesta del servicio que gestiona el catálogo con la intención de conocer qué carga es capaz de aguantar en la actualización de los valores de las pruebas, tales como el porcentaje de ejecución o el estado en el que se encuentra la prueba.

En las pruebas se ha utilizado como servidor un ordenador portátil que aloja todos los servicios y cuyas características son:

- Procesador Intel Core 2 Duo P8600 a 2,4 GHz
- 4 GB de memoria RAM
- Sistema operativo Linux, distribución Ubuntu 14.04
- Servidor web Nginx, utilizado como balanceador de carga de los servicios
- Base de datos Cassandra versión 2.2

Las pruebas de carga han sido realizadas con la herramienta **JMeter,** un proyecto Apache diseñado para realizar pruebas de este tipo y que ofrece una interfaz gráfica para el diseño y la ejecución de las pruebas. Tras la ejecución, ofrece una serie de datos con los tiempos de respuesta del servidor y los códigos de respuesta, pudiendo analizar los fallos del sistema y comprobar la carga aceptada por el servicio.

En la Figura 5.28 se muestra la configuración de JMeter indicando que se desean ejecutar 600 hilos con un bucle que se repita dos veces, con lo que se consiguen 1200 peticiones simultáneas.

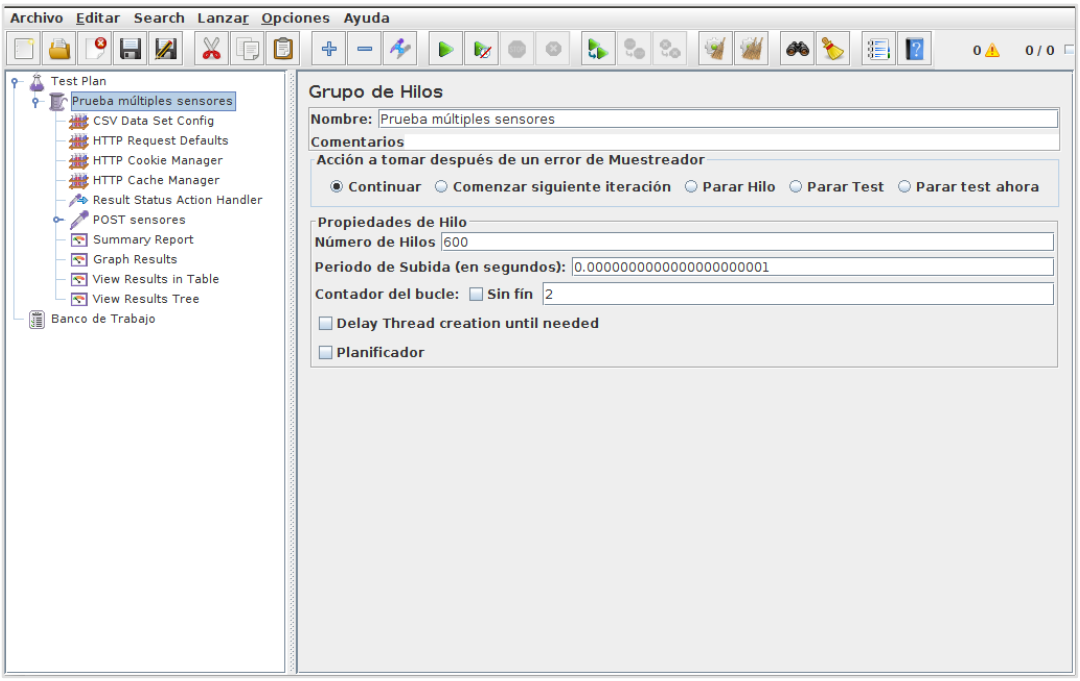

Figura 5.28: Interfaz gráfica de JMeter. Configuración del número de peticiones a realizar..

La Figura 5.29 muestra el cuerpo de la petición POST enviada por cada hilo, así como la ruta de destino. Las variables ${SENSORID} y ${TESTTIME} representan el identificador del sensor y el tiempo de captura de la muestra, respectivamente. Estos valores deben ser únicos en cada petición para no sobrescribir la información en la base de datos. Las variables se extraen de un fichero CSV en el que se especifican los valores.

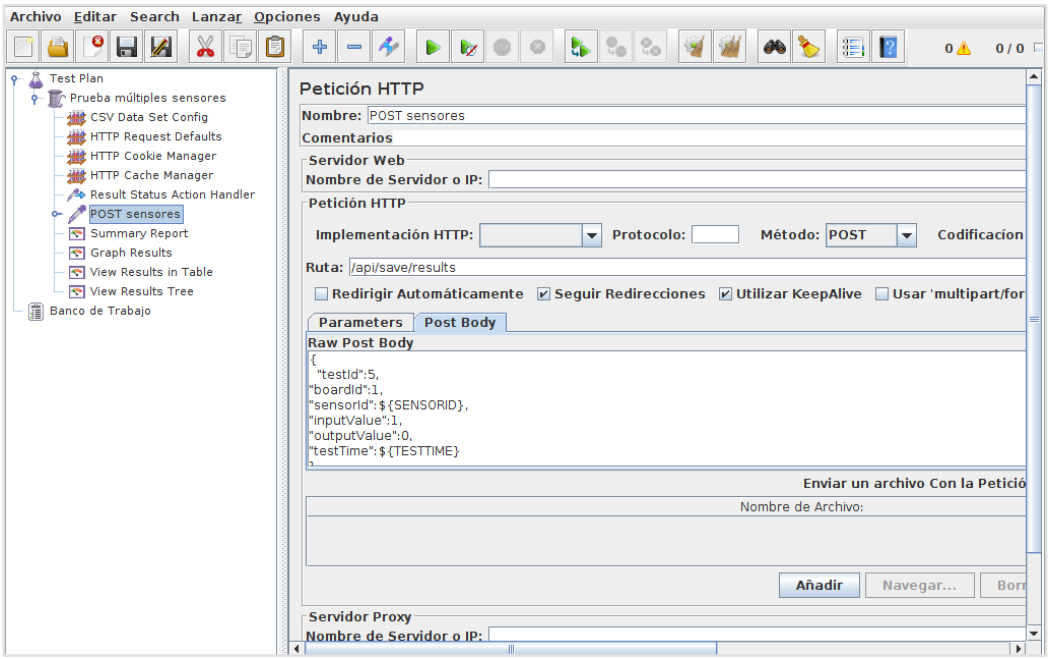

Figura 5.29: Petición POST a la ruta "/api/save/results" desde la interfaz gráfica de Jmeter.

5.3.1. Servicio de gestión de los resultados

Esta prueba consiste en el envío simultáneo de la información capturada por multitud de sensores, mandando una petición con el valor capturado por cada sensor. Para ello, se desarrolla una batería de pruebas en la que se incrementa progresivamente el número de sensores que van enviando los datos, comenzando con cien sensores y llegando a mil, momento en el cual el servidor empieza a desechar un número considerable de paquetes. Se simula un envío de dos muestras por sensor creando así un entorno de pruebas más real.

En la figura 5.30 se puede observar cómo el tiempo de respuesta medio por parte del servicio se mantiene prácticamente constante y por debajo de un segundo hasta 700 sensores. A partir de ese punto el tiempo comienza a aumentar exponencialmente, provocando la pérdida de paquetes cuando se simulan 900 sensores, llegando con 1000 sensores a una pérdida del 9% de las muestras enviadas.

El tiempo reflejado en la gráfica muestra el intervalo que tarda en llegar el paquete desde que es enviado, por lo que hay que tener en cuenta que puede variar según las condiciones de la red. Estos datos demuestran la velocidad de procesado por parte de la API REST desarrollada y la velocidad en la transacción con la base de datos Cassandra. Aunque los resultados se sitúan por encima de la carga esperada, podrían mejorarse cambiando la configuración de la base de datos instalada, puesto que se trata de una única instancia.

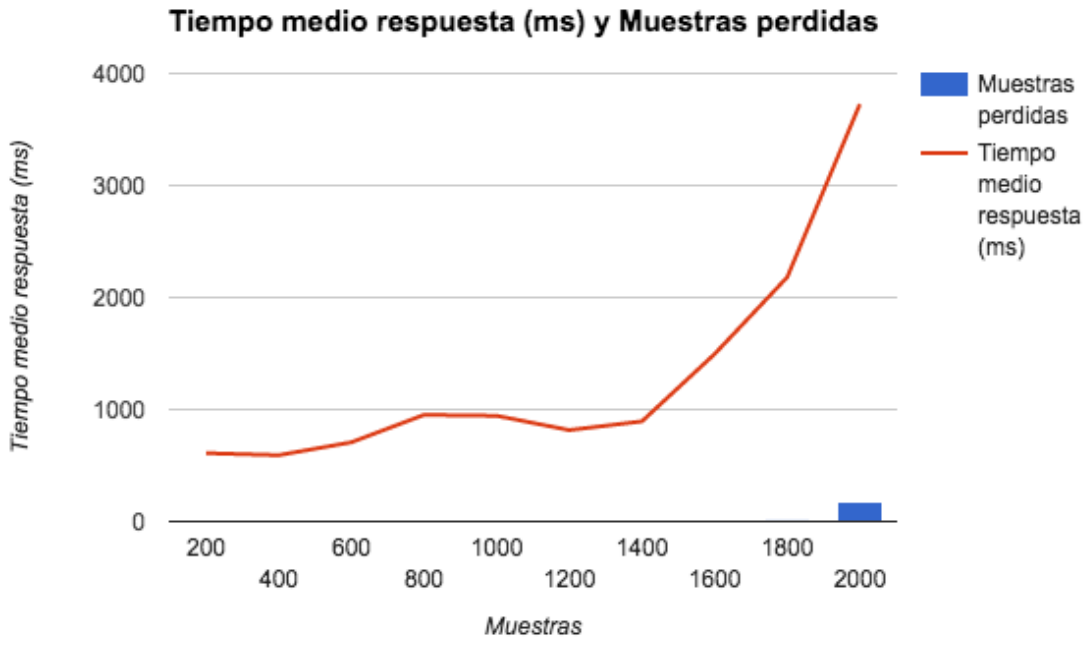

Figura 5.30: Prueba de carga del servicio de gestión de los resultados de los sensores. En la gráfica se muestra el tiempo medio de respuesta por parte del servicio asó como los paquetes perdidos en función del número de muestras.

5.3.2. Servicio de gestión del catálogo

Esta prueba consiste en que el mismo número de sensores utilizados en la prueba anterior actualicen el estado de la prueba en el servicio de gestión del catálogo del sistema, ya sea el porcentaje ejecutado o el estado de la prueba.

En la Figura 5.31 se muestra el resultado de la prueba. En ella se puede observar cómo el tiempo de respuesta por parte del servidor es exponencial. La pérdida de muestras comienza cuando se simulan 500 sensores, hasta alcanzar una tasa de error del 36% en el momento en que existen 1000 sensores intentando actualizar el estado de la prueba, situándose el tiempo medio de respuesta en torno a los 35 segundos. Esta respuesta demuestra el cuello de botella que supone utilizar SQLite como gestor de base de datos si se desea actualizar la información de la prueba con cada dato capturado. Sin embargo, se aprecia lo eficiente que es este gestor cuando existen pocas peticiones, ya que está pensado para un tráfico medio-bajo. Si se desease poder procesar un número mayor de peticiones habría que utilizar otro tipo de base de datos como pueda ser MySQL o PostgreSQL destinada a soportar un tráfico mayor de transacciones.

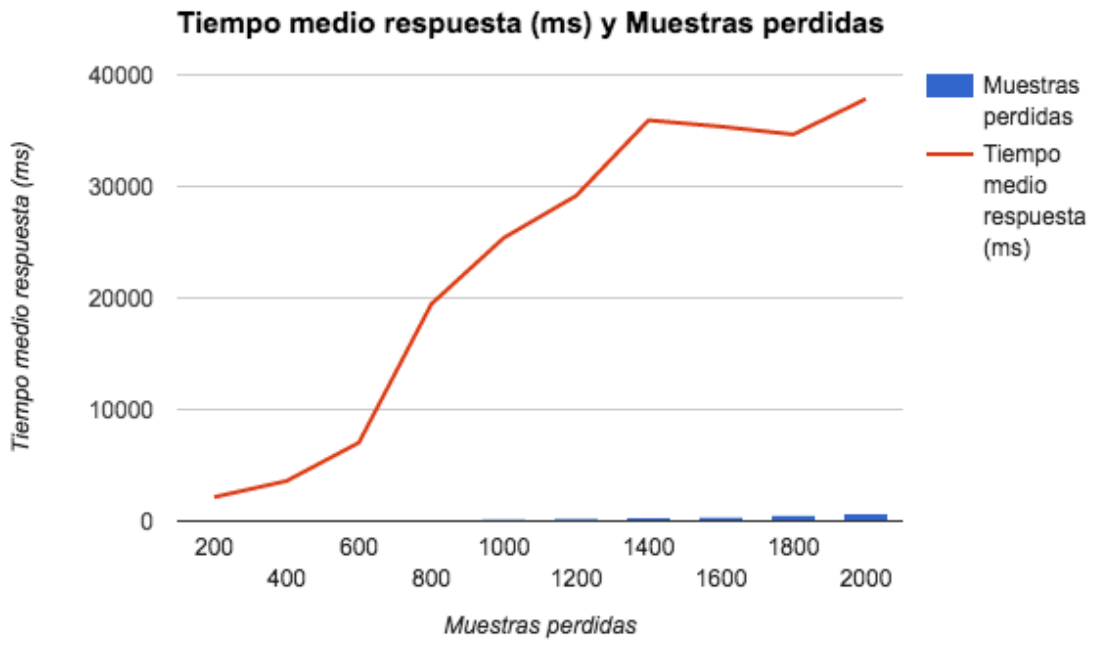

Figura 5.31: Prueba de carga del servicio de gestión del catálogo del sistema. En la gráfica se muestra el tiempo medio de respuesta por parte del servicio y los paquetes perdidos en función del número de muestras.

6. Conclusiones y trabajo futuro

6.1. Conclusiones

En este proyecto se ha llevado a cabo el diseño, el desarrollo y la implementación de un protocolo que permite la comunicación entre un ordenador o un *smartphone* con diversos tipos de sensores. Las pruebas se han desarrollado con potenciómetros para comprobar el funcionamiento del sistema y posteriormente se han aplicado distintos tipos de sensores: temperatura, fotorresistencias y sensores de odorantes. Además, se han realizado distintas pruebas de carga para demostrar y analizar la escalabilidad del sistema.

El protocolo implementado en la comunicación está basado en HTTP, lo que permite que el acceso a la información y el envío de la configuración de las pruebas se pueda realizar en remoto, descentralizando la ubicación del sensor o conjunto de sensores y la del usuario que interactúa con ellos. Por esta razón, resulta imprescindible la utilización de un dispositivo que permita la comunicación con el sensor y el envío de datos a través de la red. El dispositivo empleado en el proyecto ha sido un ordenador de placa reducida denominado Raspberry Pi. La utilización de este dispositivo, además de facilitar el envío y la recepción de los datos por protocolo HTTP, ejerce de interfaz de comunicación con el sensor mediante algún protocolo universal, como por ejemplo I^2C o SPI, o por comunicación directa utilizando los pines de entrada y salida de propósito general. Además, la Raspberry Pi proporciona un procesador *in situ* que permite el análisis de los datos en el propio dispositivo.

La recepción de las configuraciones se realiza mediante una API REST, lo que implica que cualquier dispositivo capaz de ejecutar esta API puede recibir los parámetros de configuración de las pruebas. De esta manera, se ofrece una solución completa y universal que dota de inteligencia a diversos tipos de sensores convirtiéndolos en sensores inteligentes, ya que los datos pueden ser analizados en el acto y se pueden tomar decisiones sobre ellos.

El envío de los datos de los sensores se realiza hacia una plataforma software en la “nube” cuya finalidad es almacenar la información enviada por los sensores facilitando su acceso en remoto. De esta manera, los datos de los sensores están persistidos en una máquina distinta a la empleada en la captura de los datos, favoreciendo el acceso a la información y protegiéndola frente a pérdidas por daños en el dispositivo utilizado en los sensores. La información recogida por los sensores es accesible para ser examinada, pudiendo ser utilizada en sistemas con técnicas de aprendizaje automático [24][55] para diferentes tipos de sensores y análisis empleados. Además, este sistema permite capturar los datos de múltiples sensores, centralizando la información y facilitando el acceso a los datos de todos ellos desde un único punto de entrada.

El sistema desarrollado permite también el envío de la configuración de las pruebas a ejecutar dentro del dispositivo. La configuración de estas pruebas en la “nube” se realiza gracias a un servicio dedicado a administrar y gestionar el catálogo de las pruebas que se pueden realizar, así como de los elementos implicados en ellas, como los sensores y los dispositivos, creando de esta manera un inventario para las pruebas.

El sistema se completa con distintas interfaces gráficas de usuario que facilitan el acceso a la información. Por un lado, se ofrece una interfaz que permite la creación del catálogo de

pruebas, sensores y dispositivos disponibles. El diseño de la interfaz es intuitivo permitiendo la gestión y visualización de la información de manera sencilla, así como la comunicación con el servicio destinado a almacenar dicha información mediante llamadas HTTP. Por otro lado, se ha desarrollado una interfaz que posibilita la creación de las pruebas así como la visualización y monitorización en tiempo real de los resultados de cada prueba ejecutada. Esta interfaz es una aplicación de página simple consiguiendo que la experiencia de usuario sea óptima, facilitando la monitorización en tiempo real. Además, esta última interfaz genera automáticamente toda la información que se puede monitorizar dependiendo de la prueba ejecutada y permite la descarga de un fichero con la información enviada por cada sensor implicado en la prueba.

Por último, el sistema permite que se puedan añadir nuevos sensores y programas de análisis de manera sencilla, de tal manera que los datos generados por los nuevos sensores también quedan registrados y se puedan visualizar y monitorizar con las herramientas aportadas. Por tanto, se han cumplido todos los objetivos planteados.

La arquitectura de microservicios empleada en la “nube” facilita que los servicios instalados en ella sean reutilizables según las necesidades de cada usuario, permitiendo además la creación de nuevos servicios que añadan nuevas funcionalidades sin afectar a los servicios activos. Además, los programas utilizados en la Raspberry Pi, así como la API REST desarrollada, pueden ser empleados en otros dispositivos que acepten estos lenguajes, aplicando las modificaciones necesarias. De esta manera, este proyecto ofrece toda la infraestructura necesaria para reutilizar las soluciones expuestas dependiendo del problema planteado, convirtiendo el sistema completo en una herramienta útil y versátil en el ámbito de la investigación.

6.2. Trabajo futuro

Este proyecto proporciona un sistema eficiente para el almacenamiento de datos de múltiples sensores, así como para la gestión y estructuración de la información que se genera en la ejecución de las pruebas. Además, provee una serie de buenas prácticas para mantener la información organizada e inventariada. Este sistema está pensado para su uso en el ámbito de la investigación, pero observando el desarrollo, no se descarta su implantación para la monitorización por parte de un usuario no científico.

El sistema refleja la intención de separar dos roles en la gestión de la información: el administrador del sistema y el usuario encargado de ejecutar las pruebas. Esto abre la posibilidad como trabajo futuro de la implantación de los servicios desarrollados en una nube pública, como pueda ser *Amazon AWS* o *Azure de Microsoft*, o algún servidor público, para que puedan ser utilizados desde cualquier punto del planeta, siempre que se disponga de conexión a Internet.

La puesta en producción en entornos públicos hace necesario securizar los servicios para que no puedan ser usados por agentes externos a la plataforma. Para ello, se exponen dos medidas básicas en la seguridad de la plataforma, las cuales deberán ser adoptadas y desarrolladas por la persona encargada de administrar el sistema:

- Implantación de certificados digitales en los dispositivos que dotan de conectividad a los sensores. Estos certificados deben ser conocidos únicamente por el servidor o servidores que alojan los servicios. De esta manera, la comunicación es segura y sólo es conocida por el servidor y el dispositivo. Además, como medida de seguridad adicional, se puede proceder a un filtrado por IP y que sólo los dispositivos puedan guardar y modificar los datos del servicio que gestiona los resultados de los sensores.

- Creación de usuarios con privilegios para ejecutar pruebas. Estos usuarios se deben identificar en la plataforma para poder tener acceso a los datos devueltos por los sensores. La autenticación por parte de los usuarios puede emplear el protocolo OAuth, ya que proporciona a los usuarios acceso a su información al mismo tiempo que protege los credenciales de su cuenta.

Para completar la universalidad del sistema desarrollado se propone el desarrollo de una API REST disponible para la plataforma Arduino, ya que se trata de una plataforma de bajo coste que permite utilizar multitud de sensores. También se propone la utilización de nuevos protocolos de comunicación para la toda de muestra de los sensores, como los protocolos empleados por distintos sensores digitales como pueda ser I^2C.

La creación de una librería con los códigos necesarios para distintos dispositivos en distintos lenguajes de programación facilita la labor del usuario que quiera utilizar el sistema, por lo que se convierte en un trabajo futuro interesante.

En el laboratorio, Sergio de la Cruz, en su Proyecto Fin de Carrera [12], ha diseñado una plataforma hardware destinada a capturar odorantes para su discriminación. Un trabajo futuro sería integrar ambos proyectos y así disponer en el laboratorio de una herramienta completa que permita el estudio y discriminación de los odorantes de manera remota.

Además, es importante poder etiquetar los datos almacenados en la nube con los eventos ocurridos en la adquisición de las muestras. Por ello, se propone modificar el servicio destinado a la gestión de los resultados de los sensores para poder etiquetar las muestras y que esta información quede almacenada, de manera que pueda ser visualizada mediante la herramienta gráfica destinada a la monitorización de los datos registrados por los sensores.

La arquitectura de microservicios planteada abre la posibilidad de generar nuevos servicios que completen la plataforma. Algunos ejemplos son: módulos encargados del análisis de los datos en el servidor, pudiendo aplicar técnicas de aprendizaje automático, o servicios que detecten la variación de algún valor tomado, como fugas de gases, y procedan a enviar un mensaje de alerta al usuario. La implantación de todos estos servicios no afectará al sistema actual debido al diseño de microservicios desarrollado.

Glosario

ACID: atomicidad, consistencia, aislamiento y durabilidad.
ADC: Analog-Digital Converter, Conversor analógico-digital.
API: Application Programming Interface, Interfaz de Programación de Aplicaciones.
API REST: API que cumple las condiciones de REST.
BLE: Bluetooth Low Energy.
Cloud: Nube, paradigma que permite ofrecer servicios de computación a través de una red.
CRUD: Create, Read, Update, Delete. Crear, leer, actualizar, borrar.
CSS: Cascading Style Sheets.
CSV: Comma-Separated Values.
DBMS: Database Management System. Sistema gestor de bases de datos.
DTO: Data Transfer Object. Objeto de transferencia de datos.
Endpoint: Punto de entrada a un servicio o un proceso.
E/S: Entrada/Salida.
Exponer una API REST: Ofrecer las rutas de acceso de la funcionalidad del servicio.
Framework: Estructura conceptual y tecnológica de soporte definido, con módulos de software, que sirven de base para la organización y el desarrollo de software.
GPIO: General Purpose Input/Output. Entrada/Salida de Propósito General.
GUI: Graphical User Interface. Interfaz gráfica de usuario.
HTML: HyperText Markup Language.
ID: Identificador.
IDE: Entorno de desarrollo integrado.
I^2C: Inter-Integrated Circuit.
IoT: Internet of Things. Internet de las Cosas.
JSON: JavaScript Object Notation. Formato de texto ligero para el intercambio de datos.
M2M: Machine to Machine. Máquina a máquina.
NFC: Near Field Communication.
Open source: Código abierto de libre distribución.
PWM: Pulse Wide Modulation
SaaS: Service as a Software.
SBC: Single Board Computer.
SDK: Software Development Kit.
SO: Sistema Operativo.
SPA: Single Page Application.
SPI: Serial Peripheral Interface.
SQL: Structured Query Language.
UART: Universal Asynchronous Receiver-Transmitter.
URL: Uniform Resource Locator.
USB: Universal Serial Bus
Wireframe: Guía visual que representa el esqueleto o estructura visual de un sitio web.
XML: eXtensible Markup Language. Lenguaje utilizado para almacenar datos en forma legible.
W3C: World Wide Web Consortium.

Referencias

[1] AMQP. (n.d.). https://www.amqp.org/, Consultada Julio 11, 2016

[2] AngularJS (n.d.). https://angularjs.org/ Consultada Julio 11, 2016

[3] Banana Pi - A Highend Single-Board Computer. (n.d.). http://www.bananapi.org/. Consultada Julio 11, 2016

[4] Bhaskar, D., & Mallick, B. (2015, 06). Performance Evaluation of MAC Protocol for IEEE 802. 11, 802. 11Ext. WLAN and IEEE 802. 15. 4 WPAN using NS-2. International Journal of Computer Applications IJCA, 119(16), 25-30. doi:10.5120/21153-4151

[5] Bootstrap · The world's most popular mobile-first and responsive front-end framework. (n.d.). http://getbootstrap.com/ Consultada Julio 11, 2016

[6] Bower. (n.d.). https://bower.io/ Consultada Julio 11, 2016

[7] Byrne, R., Benito-Lopez, F., & Diamond, D. (2010, 07). Materials science and the sensor revolution. Materials Today, 13(7-8), 16-23. doi:10.1016/s1369-7021(10)70124-3

[8] CMU-CREATE-Lab/esdr. (n.d.). https://github.com/CMU-CREATE-Lab/esdr Consultada Julio 11, 2016

[9] CSS Tutorial. (n.d.). http://www.w3schools.com/css/default.asp Consultada Julio 11, 2016

[10] Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications: IEEE standards for local area networks. (1985). New York, NY, USA: Institute of Electrical and Electronics Engineers.

[11] Cassandra. (n.d.). http://cassandra.apache.org/ Consultada Julio 11, 2016

[12] De la Cruz Gutiérrez, Sergio, (2016). Desarrollo de una plataforma para discriminación de odorantes mediante técnicas de modulación dinámica, Master's thesis, Escuela Politécnica Superior, Universidad Autónoma de Madrid.

[13] Django. (n.d.). https://www.djangoproject.com/ Consultada Julio 11, 2016

[14] Django REST Framework. (n.d.). http://www.django-rest-framework.org/ Consultada Julio 11, 2016

[15] Documentation | MQTT. (n.d.). http://mqtt.org/documentation Consultada Julio 11, 2016

[16] Dodson, S. (2003, October 09). The internet of things. https://www.theguardian.com/technology/2003/oct/09/shopping.newmedia Consultada Julio 11, 2016

[17] Estrin, D., Culler, D., Pister, K., & Sukhatme, G. (2002, 01). Connecting the physical world with pervasive networks. IEEE Pervasive Comput. IEEE Pervasive Computing, 1(1), 59-69. doi:10.1109/mprv.2002.993145

[18] Exploring the Foundations of Bluetooth. (2002). Bluetooth Application Developer's Guide, 69-102. doi:10.1016/b978-192899442-8/50005-7

[19] Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures.

[20] Frenzel, L. E. (2016). Inter-Integrated Circuit (I2C) Bus. Handbook of Serial Communications Interfaces, 65-68. doi:10.1016/b978-0-12-800629-0.00013-9

[21] Frenzel, L. E. (2016). Serial Peripheral Interface (SPI). Handbook of Serial Communications Interfaces, 143-145. doi:10.1016/b978-0-12-800629-0.00035-8

[22] Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020. (n.d.). http://www.gartner.com/newsroom/id/2636073 Consultada Julio 11, 2016
[23] Grunt: The JavaScript Task Runner. (n.d.). http://gruntjs.com/ Consultada Julio 11, 2016
[24] Gutierrez-Osuna, R. (2002, 06). Pattern analysis for machine olfaction: A review. IEEE Sensors J. IEEE Sensors Journal, 2(3), 189-202. doi:10.1109/jsen.2002.800688
[25] HTML(5) Tutorial. (n.d.). http://www.w3schools.com/html/ Consultada Julio 11, 2016
[26] Home - ARTIK. (n.d.). https://www.artik.io/ Consultada Julio 11, 2016
[27] Home Page SmartEverything - My CMS. (n.d.). http://www.smarteverything.it/ Consultada Julio 11, 2016
[28] IBM Watson Internet of Things. (n.d.). http://www.ibm.com/internet-of-things/ Consultada Julio 11, 2016
[29] IEEE 802.3 ETHERNET. (n.d.). http://www.ieee802.org/3/ Consultada Julio 11, 2016
[30] IoT platform - thethings.iO. (n.d.). Consultada Julio 11, 2016, from https://thethings.io/
[31] JavaScript Tutorial. (n.d.). http://www.w3schools.com/js/default.asp Consultada Julio 11, 2016
[32] LM335 (ACTIVE). (n.d.). http://www.ti.com/product/LM335 Consultada Julio 11, 2016
[33] Latest SOAP versions. (n.d.). https://www.w3.org/TR/soap/ Consultada Julio 11, 2016
[34] Laurant, S. S., Johnston, J., & Dumbill, E. (2001). Programming web services with XML-RPC. Beijing: O'Reilly.
[35] Libelium (n.d.). http://www.libelium.com/ Consultada Julio 11, 2016
[36] MCP3008. (n.d.). http://www.microchip.com/wwwproducts/en/MCP3008 Consultada Julio 11, 2016
[37] Meet the Beagles: Open Source Computing. (n.d.). https://beagleboard.org/ Consultada Julio 11, 2016
[38] MongoDB. (n.d.). https://www.mongodb.com/es Consultada Julio 11, 2016
[39] More Than 30 Billion Devices Will Wirelessly Connect to the Internet of Everything in 2020. (n.d.). https://www.abiresearch.com/press/more-than-30-billion-devices-will-wirelessly-conne/ Consultada Julio 11, 2016
[40] MySQL. (n.d.). https://www.mysql.com/ Consultada Julio 11, 2016
[41] Netduino :: Home. (n.d.). http://www.netduino.com/ Consultada Julio 11, 2016
[42] Pequeño Zurro, Alejandro (2015). Uso de una nariz electrónica ultra portátil en robots para la detección de fuentes odorantes, Master's thesis, Escuela Politécnica Superior, Universidad Autónoma de Madrid.
[43] Polastre, J., Szewczyk, R., Mainwaring, A., Culler, D., & Anderson, J. (n.d.). Analysis of Wireless Sensor Networks for Habitat Monitoring. Wireless Sensor Networks, 399-423. doi:10.1007/1-4020-7884-6_18
[44] PostgreSQL. (n.d.). http://www.postgresql.org.es/documentacion Consultada Julio 11, 2016
[45] Principles and Concepts of Cloud Computing. (2016, 02). Trustworthy Cloud Computing Safonov/Trustworthy Cloud Computing, 1-32. doi:10.1002/9781119114215.ch1
[46] RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1. (n.d.). https://tools.ietf.org/html/rfc2616 Consultada Julio 11, 2016

[47] RFC 6455 - The WebSocket protocol. (n.d.). https://tools.ietf.org/html/rfc6455 Consultada Julio 11, 2016
[48] RFC 7252 - The Constrained Application Protocol (CoAP). (n.d.). https://tools.ietf.org/html/rfc7252 Consultada Julio 11, 2016
[49] Raspberry Pi - Teach, Learn, and Make with Raspberry Pi. (n.d.). https://www.raspberrypi.org/ Consultada Julio 11, 2016
[50] Redis. (n.d.). http://redis.io/ Consultada Julio 11, 2016
[51] Richard Viel. (n.d.). https://www.lora-alliance.org/What-Is-LoRa/Technology Consultada Julio 11, 2016
[52] SB Admin 2 - Free Bootstrap Admin Theme. (n.d.). http://startbootstrap.com/template-overviews/sb-admin-2/ Consultada Julio 11, 2016
[53] SIGFOX - The Global Communications Service Provider for the Internet of Things (IoT). (n.d.). https://www.sigfox.com/ Consultada Julio 11, 2016
[54] SQLite Home Page. (n.d.). https://www.sqlite.org/ Consultada Julio 11, 2016
[55] Scott, S. M., James, D., & Ali, Z. (2006, 09). Data analysis for electronic nose systems. Microchimica Acta Microchim Acta, 156(3-4), 183-207. doi:10.1007/s00604-006-0623-9
[56] Somov, A., Baranov, A., Savkin, A., Spirjakin, D., Spirjakin, A., & Passerone, R. (2011, 11). Development of wireless sensor network for combustible gas monitoring. Sensors and Actuators A: Physical, 171(2), 398-405. doi:10.1016/j.sna.2011.07.016
[57] TGS2600. (n.d.). http://www.figaro.co.jp/en/product/entry/tgs2600.html Consultada Julio 11, 2016
[58] That 'Internet of Things' Thing. (n.d.). http://www.rfidjournal.com/articles/view?4986 Consultada Julio 11, 2016
[59] ThingWorx | An Enterprise IoT Solutions Company. (n.d.). https://www.thingworx.com/ Consultada Julio 11, 2016
[60] Universal Synchronous Asynchronous Receiver Transmitter (USART). (1988). Technical Aspects of Data Communication, 307-315. doi:10.1016/b978-1-4831-8400-5.50036-x
[61] Visible Things. (n.d.). https://silica.avnet.com/wps/portal/silica/marketstechnologies/internet-of-things/visible-things/ Consultada Julio 11, 2016
[62] Waldner, Jean-Baptiste (2007). Inventer l'Ordinateur du XXIeme Siècle. London: Hermes Science. pp. p254. ISBN 2746215160.
[63] WHAT IS ARDUINO? (n.d.). https://www.arduino.cc/ Consultada Julio 11, 2016
[64] Weiser, M. (1995). The Computer for the 21st Century. Readings in Human–Computer Interaction, 933-940. doi:10.1016/b978-0-08-051574-8.50097-2
[65] Welcome to Python.org. (n.d.). https://www.python.org/ Consultada Julio 11, 2016
[66] Wiring. (n.d.). http://wiring.org.co/ Consultada Julio 11, 2016
[67] Xively. (n.d.). https://xively.com/ Consultada Julio 11, 2016
[68] D3/d3. (n.d.). https://github.com/d3/d3/wiki Consultada Julio 11, 2016
[69] Nginx news. (n.d.). https://nginx.org/ Consultada Julio 11, 2016
[70] Spring.io. (n.d.). https://spring.io/ Consultada Julio 11, 2016
[71] The web's scaffolding tool for modern webapps | Yeoman. (n.d.). http://yeoman.io/ Consultada Julio 11, 2016
[72] ¿Qué es Carriots? (n.d.). https://www.carriots.com/que-es-carriots Consultada Julio 11, 2016

[73] ¿Qué es Java? (n.d.). https://www.java.com/es/about/whatis_java.jsp Consultada Julio 11, 2016
[74] Herrero-Carrón,Fernando, David J. Yáñez, Francisco de Borja Rodríguez, Pablo Varona, “An active, inverse temperature modulation strategy for single sensor odorant classification”, Sensors and Actuators B: Chemical
[75] Conceptos sobre APIs REST. (n.d.). http://asiermarques.com/2013/conceptos-sobre-apis-rest/ Consultada Julio 11, 2016
[76] Webber, J., Parastatidis, S., & Robinson, I. (2010). REST in Practice:. Beijing: O'Reilly.

Anexo A. Despliegue de los servicios del sistema

En este anexo se especifican los pasos que hay que seguir para instalar cada servicio y la configuración de los servidores necesaria para desplegar cada servicio en el sistema operativo Ubuntu.

Servicio gestión del catálogo del sistema

1. Instalar python versión 2.7.
2. Instalar y actualizar pip:
    a. sudo apt-get install python-pip
    b. sudo pip install --upgrade pip
3. Instalar virtualenvwrapper
    a. pip install virtualenvwrapper
    b. Configurar entorno:
        i. En el archivo bashrc añadir las siguientes líneas:
            - export WORKON_HOME=$HOME/.entornos
            - source /usr/local/bin/virtualenvwrapper.sh
4. Crear entorno
    a. mkvirtualenv api-catalog --no-site-packages
5. Activar entorno
    a. workon api-catalog
6. Ubicarse en el directorio del proyecto.
7. Instalar dependencias
    a. pip install -r -requirements.txt

Servicio gestión de los resultados de los sensores

1. Instalar java versión 8.
2. Instalar Maven.
    a. sudo apt-get install maven
3. mvn clean package.
4. Instalar cassandra versión 2.2.5.

Interfaz de administración de las pruebas

1. Instalar node y npm versión 4.3.
2. Instalar bower 1.7.7.
    a. sudo npm install bower -g
3. Instalar grunt 1.13.
    a. sudo npm install grunt-cli -g
4. Ubicarse en el directorio del proyecto.
5. sudo npm install --save
6. sudo bower install --save --allow-root
7. sudo grunt build

Despliegue de los servicios

En el despliegue de los servicios son necesarias varias herramientas: Gunicorn, Supervisor y Nginx.

- Gunicorn: Es un servidor WSGI HTTP para proyectos de Python.
    - Instalación: pip install gunicorn

- Supervisor: Permite el control a los usuarios de los procesos que se ejecutan en el sistema.
    - Instalación: sudo apt-get install supervisor
    - Configuración: Es necesario crear un archivo para cada API REST para que se ejecuten en el sistema:
        - sudo nano /etc/supervisor/conf.d/api_catalog.conf
          [program:api-catalog]
          command = /home/pfc/.entornos/api_catalog/bin/python
          /home/pfc/.entornos/api_catalog/bin/gunicorn api_catalog.w
          sgi:application
          directory = /home/pfc/api-catalog
          user = root

        - sudo nano /etc/supervisor/conf.d/api_results.conf
          [program:api-results]
          command = java -jar api-results-0.0.1-SNAPSHOT.jar
          directory = /home/pfc/api-results/target
          user = root
          stderr_logfile=/var/log/supervisor/api_results_err.log
          stdout_logfile=/var/log/supervisor/api_results_out.log
          autorestart=true
          stopsignal=KILL
    - sudo supervisorctl reread
    - sudo supervisorctl update
    - sudo supervisorctl start <program>

- Nginx: Es un servidor web ligero.
    - Instalación: sudo apt-get install nginx
    - Creación de los dominios de los servicios en el directorio sites-enabled:
        - API REST gestión del catálogo del sistema:

```
server {
        listen 80;
        server_name dev-api-catalog.pfc.com;

        access_log off;

        location /static {
            alias /home/pfc/api-catalog/static;
        }
```

```
        location / {
            proxy_pass http://127.0.0.1:8000;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

        }
    }
```

- API REST gestión de los resultados de los sensores:

```
    server {
        listen 80;
        server_name dev-api-results.pfc.com;

        access_log off;

        location / {
            proxy_pass http://127.0.0.1:8080;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

            if ($request_method = 'GET') {
                add_header 'Access-Control-Allow-Origin' '*';
                add_header 'Access-Control-Allow-Credentials' 'true';
                add_header 'Access-Control-Allow-Methods' 'GET, POST,
                        OPTIONS';
                add_header 'Access-Control-Allow-Headers' 'DNT,X-
    CustomHeader,Keep-Alive,User-Agent,X-Requested-With,If-Modified-
    Since,Cache-Control,Content-Type';
            }
        }
    }
```

- Interfaz de administración de las pruebas:

```
    server {
        listen 80;
        server_name dev-front.pfc.com;

        access_log off;
        root /home/pfc/front-pfc/dist;


        location / {
            try_files $uri $uri/ /index.html =404;

            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }
    }
```

- Crear enlace simbólico de los proyectos:
    - cd /etc/nginx/sites-enabled
    - sudo ln -s ../sites-available/<proyecto>

- Reiniciar el servicio:
    - sudo service nginx restart

Anexo B. Código API REST recepción de la configuración

Pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<groupId>com.gnb.api.board</groupId>
	<artifactId>api-board</artifactId>
	<version>1.0</version>
	<name>API board</name>
	<description>Board Api Rest to launch the analysis process</description>

	<properties>
		<!-- Enable Java 8 -->
		<java.version>1.8</java.version>
		<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
		<!-- Configure the main class of our Spring Boot application -->
		<start-class>api.Application</start-class>
	</properties>

	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>1.2.5.RELEASE</version>
	</parent>

	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.data</groupId>
			<artifactId>spring-data-rest-webmvc</artifactId>
		</dependency>

		<dependency>
			<groupId>org.apache.commons</groupId>
			<artifactId>commons-io</artifactId>
			<version>1.3.2</version>
		</dependency>

	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>

</project>
```

Application.java

```
package api;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Import;
import org.springframework.data.rest.webmvc.config.RepositoryRestMvcConfiguration;

@SpringBootApplication
@Import(RepositoryRestMvcConfiguration.class)
public class Application {

	private static Properties GLOBAL_PROPERTIES;

	public static void main(String[] args) throws IOException {
		loadGlobalProperties();
		//Run the application
		SpringApplication.run(Application.class, args);
	}

	/**
	 * Read domain.properties and store the values
	 * @throws IOException
	 */
	private static void loadGlobalProperties() throws IOException {
		Properties prop = new Properties();
		String propFileName = "global.properties";

		InputStream inputStream = Application.class.getClassLoader()
				.getResourceAsStream(propFileName);

		if (inputStream != null) {
			prop.load(inputStream);
		} else {
			throw new FileNotFoundException("property file '" + propFileName
					+ "' not found in the classpath");
		}

		GLOBAL_PROPERTIES = prop;
	}

	public static String getPythonProcessPath(){
		return GLOBAL_PROPERTIES.getProperty("global.python.process.path");
	}
}
```

ModelService.java

```
package api.service.interfaces;

public interface ModelService {

	int launchModelLdrRc(String boardId, String testId, String sensorId, Integer
		pinInput, Integer pinOutput, Double samplingTime);

	int launchModelSensorADC(String boardId, String testId, String sensorId, Integer
		pinInput, Integer pinOutput, Double samplingTime);
```

```
        int launchModelLdrADCLed(String boardId, String testId, String sensorId, Integer
                pinInput, Integer pinOutput, Double samplingTime, String ledId, Integer
                ledPinInput, Integer ledPinOutput, Double ledOnValue);

        int launchModelLm335ADC(String boardId, String testId, String sensorId, Integer
                pinInput, Integer pinOutput, Double samplingTime, String varId, Integer
                varPinInput, Integer varPinOutput);

        int launchModelGasSensorLm35ADC(String boardId, String testId, String sensorId,
                Integer pinInput, Integer pinOutput, String sensorId2, Integer pinInput2,
                Integer pinOutput2, int nm, double t, double sleep, int samplesInicio,
                double tendency, double averageTemperature, double duration, String
                sensorId3, Integer pinInput3, Integer pinOutput3, String sensorId4, Integer
                pinInput4, Integer pinOutput4, String sensorId5, Integer pinInput5, Integer
                pinOutput5);

        void killProcess(String pid);

}
```

ApiService.java

```
package api.service.implementation;

import java.io.IOException;
import java.lang.reflect.Field;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;
import api.exceptions.CommandException;
import api.service.interfaces.ModelService;
import static api.Application.getPythonProcessPath;

@Service
public class ApiService implements ModelService {

        private static final Logger LOGGER = LoggerFactory.getLogger(ApiService.class);

        @Override
        public void killProcess(String pid) {
                // Kill with keyboard interrupt
                String command = "kill -s INT " + pid;
                executeCommand(command);
        }

        @Override
        public int launchModelLdrRc(String boardId, String testId, String sensorId,
                        Integer pinInput, Integer pinOutput, Double samplingTime) {
                String pathToProcess = getPythonProcessPath();
                String command = pathToProcess + "model-ldr-circuit-rc.py " + boardId + " "
                                + testId + " " + sensorId + " " + pinInput + " " + pinOutput
                                + " " + samplingTime + " >> " + pathToProcess +
                                 "model-ldr-circuit-rc-" + System.currentTimeMillis() +".log";

                LOGGER.info("Command to launch: " + command);
                return executeCommand(command);
        }

        @Override
        public int launchModelSensorADC(String boardId, String testId, String sensorId,
                        Integer pinInput, Integer pinOutput, Double samplingTime) {
                String pathToProcess = getPythonProcessPath();
                String command = pathToProcess + "model-sensor-adc.py " + boardId + " "
```

```
                        + testId + " " + sensorId + " " + pinInput + " " + pinOutput
                        + " " + samplingTime + " >> " + pathToProcess +
                        "model-sensor-adc-" + System.currentTimeMillis() +".log";

        LOGGER.info("Command to launch: " + command);
        return executeCommand(command);
    }

    @Override
    public int launchModelLdrADCLed(String boardId, String testId, String sensorId,
                    Integer pinInput, Integer pinOutput, Double samplingTime, String
                    ledId, Integer ledPinInput, Integer ledPinOutput, Double ledOnValue){

        String pathToProcess = getPythonProcessPath();
        String command = pathToProcess + "model-ldr-adc-led.py " + boardId + " "
                        + testId + " " + sensorId + " " + pinInput + " " + pinOutput
                        + " " + samplingTime + " " + ledId + " " + ledPinInput
                        + " " + ledPinOutput + " " + ledOnValue + " >> "
                        + pathToProcess + "model-ldr-adc-led-" +
                        System.currentTimeMillis() +".log";

        LOGGER.info("Command to launch: " + command);
        return executeCommand(command);
    }

    @Override
    public int launchModelLm335ADC(String boardId, String testId, String sensorId,
                    Integer pinInput, Integer pinOutput, Double samplingTime, String
                    varId, Integer varPinInput, Integer varPinOutput) {

        String pathToProcess = getPythonProcessPath();
        String command = pathToProcess + "model-lm335-adc.py " + boardId + " "
                        + testId + " " + sensorId + " " + pinInput + " " + pinOutput
                        + " " + samplingTime + " " + varId + " " + varPinInput
                        + " " + varPinOutput  + " >> " + pathToProcess +
                        "model-lm335-adc-" + System.currentTimeMillis() +".log";

        LOGGER.info("Command to launch: " + command);
        return executeCommand(command);
    }

    @Override
    public int launchModelGasSensorLm35ADC(String boardId, String testId, String
                    sensorId, Integer pinInput, Integer pinOutput, String sensorId2,
                    Integer pinInput2, Integer pinOutput2, int nm, double t, double
                    sleep, int samplesInicio, double tendency, double averageTemperature,
                    double duration, String sensorId3, Integer pinInput3, Integer
                    pinOutput3, String sensorId4, Integer pinInput4, Integer pinOutput4,
                    String sensorId5, Integer pinInput5, Integer pinOutput5) {

        String pathToProcess = getPythonProcessPath();
        String command = pathToProcess + "model-gas-sensor-regression-lm35.py " +
                boardId + " "  + testId + " " + sensorId + " " + pinInput + " " +
                pinOutput + " " + sensorId2 + " " + pinInput2 + " " + pinOutput2+ " "
                + nm + " " + t + " " + sleep + " " + samplesInicio + " " + tendency +
                " " + averageTemperature + " " + duration + " " + sensorId3 + " " +
                pinInput3 + " " + pinOutput3 + " " + sensorId4 + " " + pinInput4 +" "
                + pinOutput4 + " " + sensorId5 + " " + pinInput5 + " " + pinOutput5 +
                " >> " + pathToProcess + "model-gas-sensor-regression-lm35-" +
                System.currentTimeMillis() +".log";

                LOGGER.info("Command to launch: " + command);
                return executeCommand(command);
    }
```

```
	/**
	 * Execute the command and return the PID
	 * @param command
	 * @return PID
	 */
	private int executeCommand(String command) {
		Process process;
		int pid = 0;
		try {
			process = Runtime.getRuntime().exec(command);
			if(process.getClass().getName().equals("java.lang.UNIXProcess")) {
				// get the PID on unix/linux systems
				Field f = process.getClass().getDeclaredField("pid");
				f.setAccessible(true);
				pid = f.getInt(process);
			}
		} catch (NoSuchFieldException | SecurityException | IllegalArgumentException
				| IllegalAccessException | IOException e) {
			LOGGER.error(e.getMessage());
			throw new CommandException(command);
		}
		return pid;
	}
}
```

ModelLdrRcDTO.java

```
package api.entity.model.dto;

import javax.validation.Valid;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;

public class ModelLdrRcDTO {

	@NotNull
	private String boardId;
	@NotNull
	private String testId;

	@NotNull
	@Valid
	private ModelLdrRcSensorDTO sensor1;

	@NotNull
	@Min(0)
	private Double samplingTime;

	//Getters and setters
}
```

ModelLdrRcSensorDTO.java

```
package api.entity.model.dto;

import javax.validation.constraints.NotNull;

public class ModelLdrRcSensorDTO {

	@NotNull
```

```
        private String sensorId;

        @NotNull
        private Integer pinInput;

        @NotNull
        private Integer pinOutput;

        //Getters and setters
}
```

ModelSensorAdcDTO.java

```
package api.entity.model.dto;

import javax.validation.Valid;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;

public class ModelSensorAdcDTO {

        @NotNull
        private String boardId;
        @NotNull
        private String testId;

        @NotNull
        @Valid
        private ModelSensorAdcSensorDTO sensor1;

        @NotNull
        @Min(0)
        private Double samplingTime;

        //Getters and setters
}
```

ModelSensorAdcSensorDTO.java

```
package api.entity.model.dto;

import javax.validation.constraints.NotNull;

public class ModelSensorAdcSensorDTO {

        @NotNull
        private String sensorId;

        @NotNull
        private Integer pinInput;

        @NotNull
        private Integer pinOutput;

        //Getters and setters
}
```

ModelLdrAdcDTO.java

```
package api.entity.model.dto;

import javax.validation.Valid;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
```

```
public class ModelLdrAdcDTO {

	@NotNull
	private String boardId;
	@NotNull
	private String testId;

	@NotNull
	@Valid
	private ModelSensorAdcSensorDTO sensorLDR;

	@NotNull
	@Min(0)
	private Double samplingTime;

	@NotNull
	@Valid
	private ModelLdrLedAdcSensorDTO led;

	//Getters and setters
}
```

ModelLdrLedAdcSensorDTO.java

```
package api.entity.model.dto;

import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;

public class ModelLdrLedAdcSensorDTO {

	@NotNull
	private String sensorId;

	@NotNull
	private Integer pinInput;

	@NotNull
	private Integer pinOutput;

	@NotNull
	private Double ledOnValue;

	//Getters and setters
}
```

ModelLm335AdcDTO.java

```
package api.entity.model.dto;

import javax.validation.Valid;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;

public class ModelLm335AdcDTO {

	@NotNull
	private String boardId;
	@NotNull
	private String testId;

	@NotNull
```

```
        @Valid
        private ModelSensorAdcSensorDTO sensorLM335;

        @NotNull
        @Valid
        private ModelSensorAdcSensorDTO varTemperature;

        @NotNull
        @Min(0)
        private Double samplingTime;

        //Getters and setters
}
```

ModelGasSensorLm35AdcDTO.java

```
package api.entity.model.dto;

import javax.validation.Valid;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;

public class ModelGasSensorLm35AdcDTO {

        @NotNull
        private String boardId;
        @NotNull
        private String testId;

        @NotNull
        @Valid
        private ModelSensorAdcSensorDTO sensorLM35;

        @NotNull
        @Valid
        private ModelSensorAdcSensorDTO sensorGas;

        @NotNull
        @Valid
        private ModelSensorAdcSensorDTO varSlopeIntercept;

        @NotNull
        @Valid
        private ModelSensorAdcSensorDTO varRvaluePvalue;

        @NotNull
        @Valid
        private ModelSensorAdcSensorDTO varStdErr1;

        private int nm;

        private double t;

        private double sleep;

        private int samplesInicio;

        private double tendency;

        @NotNull
        @Min(0)
        @Max(100)
        private double averageTemperature;
```

```
        @NotNull
        @Min(0)
        private double duration;

        //Constructor
        public ModelGasSensorLm35AdcDTO() {}

        //Getters and setters
}
```

ApiModelController.java

```
package api.controller;

import javax.validation.Valid;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import api.entity.model.dto.ModelGasSensorLm35AdcDTO;
import api.entity.model.dto.ModelLdrAdcDTO;
import api.entity.model.dto.ModelLdrRcDTO;
import api.entity.model.dto.ModelLm335AdcDTO;
import api.entity.model.dto.ModelSensorAdcDTO;
import api.exceptions.CommandException;
import api.service.implementation.ApiService;

/**
 * This controller provides the private API that is used to launch the model.
 * @author Luis Pulido
 */
@RestController
@RequestMapping("/api/model")
public class ApiModelController {

        private static final Logger LOGGER =
                LoggerFactory.getLogger(ApiModelController.class);

        private final ApiService service;

    @Autowired
```

```
ApiModelController(ApiService service) {
    this.service = service;
}

@RequestMapping(value="/1", method = RequestMethod.POST, consumes =
   {"application/json"}, produces = "application/json")
@ResponseStatus(HttpStatus.ACCEPTED)
public @ResponseBody int create(@RequestBody @Valid ModelLdrRcDTO modelEntry) {
   LOGGER.info("Processing a new model entry with information: {}", modelEntry);
   int pid = service.launchModelLdrRc(modelEntry.getBoardId(), modelEntry.getTestId(),
          modelEntry.getSensor1().getSensorId(),modelEntry.getSensor1().getPinInput(),
          modelEntry.getSensor1().getPinOutput(), modelEntry.getSamplingTime());
          LOGGER.info("Process launched with pid {}", pid);
          return pid;
   }

@RequestMapping(value="/2", method = RequestMethod.POST, consumes =
   {"application/json"}, produces = "application/json")
@ResponseStatus(HttpStatus.ACCEPTED)
public @ResponseBody int create(@RequestBody @Valid ModelSensorAdcDTO modelEntry) {
   LOGGER.info("Processing a new model entry with information: {}", modelEntry);
   int pid = service.launchModelSensorADC(modelEntry.getBoardId(),
          modelEntry.getTestId(), modelEntry.getSensor1().getSensorId(),
          modelEntry.getSensor1().getPinInput(),modelEntry.getSensor1().getPinOutput()
          , modelEntry.getSamplingTime());
   LOGGER.info("Process launched with pid {}", pid);
   return pid;
}

@RequestMapping(value="/3", method = RequestMethod.POST, consumes =
   {"application/json"}, produces = "application/json")
@ResponseStatus(HttpStatus.ACCEPTED)
public @ResponseBody int create(@RequestBody @Valid ModelLdrAdcDTO modelEntry) {
   LOGGER.info("Processing a new model entry with information: {}", modelEntry);
   int pid = service.launchModelLdrADCLed(modelEntry.getBoardId(),
          modelEntry.getTestId(), modelEntry.getSensorLDR().getSensorId(),
          modelEntry.getSensorLDR().getPinInput(),
          modelEntry.getSensorLDR().getPinOutput(), modelEntry.getSamplingTime(),
          modelEntry.getLed().getSensorId(), modelEntry.getLed().getPinInput(),
          modelEntry.getLed().getPinOutput(), modelEntry.getLed().getLedOnValue());
   LOGGER.info("Process launched with pid {}", pid);
   return pid;
}


@RequestMapping(value="/4", method = RequestMethod.POST, consumes =
```

```
    {"application/json"}, produces = "application/json")
@ResponseStatus(HttpStatus.ACCEPTED)
public @ResponseBody int create(@RequestBody @Valid ModelLm335AdcDTO modelEntry) {
    LOGGER.info("Processing a new model entry with information: {}", modelEntry);
    int pid = service.launchModelLm335ADC(modelEntry.getBoardId(),
            modelEntry.getTestId(), modelEntry.getSensorLM335().getSensorId(),
            modelEntry.getSensorLM335().getPinInput(),
            modelEntry.getSensorLM335().getPinOutput(), modelEntry.getSamplingTime(),
            modelEntry.getVarTemperature().getSensorId(),
            modelEntry.getVarTemperature().getPinInput(),
            modelEntry.getVarTemperature().getPinOutput());
    LOGGER.info("Process launched with pid {}", pid);
    return pid;
}

@RequestMapping(value="/5", method = RequestMethod.POST, consumes =
    {"application/json"}, produces = "application/json")
@ResponseStatus(HttpStatus.ACCEPTED)
    public @ResponseBody int create(@RequestBody @Valid ModelGasSensorLm35AdcDTO
            modelEntry) {
            LOGGER.info("Processing a new model entry with information: {}",modelEntry);
            int pid = service.launchModelGasSensorLm35ADC(modelEntry.getBoardId(),
                    modelEntry.getTestId(), modelEntry.getSensorGas().getSensorId(),
                    modelEntry.getSensorGas().getPinInput(),
                    modelEntry.getSensorGas().getPinOutput(),
                     modelEntry.getSensorLM35().getSensorId(),
                     modelEntry.getSensorLM35().getPinInput(),
                     modelEntry.getSensorLM35().getPinOutput(), modelEntry.getNm(),
                     modelEntry.getT(), modelEntry.getSleep(),
                     modelEntry.getSamplesInicio(), modelEntry.getTendency(),
                     modelEntry.getAverageTemperature(), modelEntry.getDuration(),
                     modelEntry.getVarSlopeIntercept().getSensorId(),
                     modelEntry.getVarSlopeIntercept().getPinInput(),
                     modelEntry.getVarSlopeIntercept().getPinOutput(),
                     modelEntry.getVarRvaluePvalue().getSensorId(),
                    modelEntry.getVarRvaluePvalue().getPinInput(),
                    modelEntry.getVarRvaluePvalue().getPinOutput(),
                    modelEntry.getVarStdErr1().getSensorId(),
                    modelEntry.getVarStdErr1().getPinInput(),
                    modelEntry.getVarStdErr1().getPinOutput());

                    LOGGER.info("Process launched with pid {}", pid);
                    return pid;
            }

                @ExceptionHandler
                @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
```

```
                public void handleCommandException(CommandException ex) {
                }
}
```

ApiKillProcessController.java

```
package api.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import api.service.implementation.ApiService;

/**
 * This controller provides the private API that is used to launch the model.
 * @author Luis Pulido
 */
@RestController
@RequestMapping("/api/kill")
public class ApiKillProcessController {

    private static final Logger LOGGER =
        LoggerFactory.getLogger(ApiKillProcessController.class);

    private final ApiService service;

    @Autowired
    ApiKillProcessController(ApiService service) {
        this.service = service;
    }

       @RequestMapping(value = "{pid}", method = RequestMethod.GET)
       @ResponseStatus(HttpStatus.OK)
       public @ResponseBody void killProcess(@PathVariable("pid") String pid) {
              LOGGER.info("Processing a new process with pid: {}", pid);
              service.killProcess(pid);
              LOGGER.info("Process killed");
       }

}
```

Anexo C. Código API REST gestión de los resultados de los sensores

Pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<groupId>com.gnb.pmne</groupId>
	<artifactId>api-results</artifactId>
	<version>1.0</version>
	<name>api-results</name>
	<description>API to stores results data through private ajax POST request and to get
the results data through ajax GET request (OAuth security)</description>
	<properties>
		<java.version>1.8</java.version>
		<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
		<start-class>api.Application</start-class>
	</properties>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>1.3.0.RELEASE</version>
	</parent>
	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.data</groupId>
			<artifactId>spring-data-cassandra</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.data</groupId>
			<artifactId>spring-data-rest-webmvc</artifactId>
		</dependency>
		<dependency>
			<groupId>org.apache.commons</groupId>
			<artifactId>commons-io</artifactId>
			<version>1.3.2</version>
		</dependency>
		<dependency>
		    <groupId>net.sf.supercsv</groupId>
		    <artifactId>super-csv</artifactId>
		    <version>2.4.0</version>
		</dependency>
	</dependencies>
	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>
</project>
```

Cassandra.properties

```
cassandra.contactpoints=127.0.0.1
cassandra.port=9042
cassandra.keyspace=apiresults
```

SCHEMA.CQL

```
CREATE TABLE IF NOT EXISTS results (
    test_id text,
    test_time bigint,
    board_id text,
    sensor_id text,
    input_value double,
    output_value double,
    event_time bigint,
    PRIMARY KEY (test_id, sensor_id, test_time)
)
WITH comment='' AND read_repair_chance = 2.0 AND CLUSTERING ORDER BY (sensor_id ASC,
test_time DESC);
```

Application.java

```
package api;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;

import org.apache.commons.io.IOUtils;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Import;
import org.springframework.data.cassandra.config.java.AbstractCassandraConfiguration;
import org.springframework.data.cassandra.convert.MappingCassandraConverter;
import org.springframework.data.cassandra.core.CassandraOperations;
import org.springframework.data.cassandra.core.CassandraTemplate;
import org.springframework.data.cassandra.mapping.BasicCassandraMappingContext;
import org.springframework.data.cassandra.repository.config.EnableCassandraRepositories;
import org.springframework.data.rest.webmvc.config.RepositoryRestMvcConfiguration;

import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.Session;


@SpringBootApplication
@EnableCassandraRepositories
@Import(RepositoryRestMvcConfiguration.class)
public class Application  extends AbstractCassandraConfiguration {

        private static String KEYSPACE_NAME = "apiresults";
        private static String CONTACT_POINTS = "127.0.0.1";
        private static int PORT = 9042;

        public static String DOMAIN = "http://localhost:8080";

        public static void main(String[] args) throws IOException {
                loadDomainProperties();
```

```
        getSession();
        //Run the application
        SpringApplication.run(Application.class, args);
}

/**
 * Read domain.properties and store the values
 * @throws IOException
 */
private static void loadDomainProperties() throws IOException {
        Properties prop = new Properties();
        String propFileName = "domain.properties";

        InputStream inputStream = Application.class.getClassLoader()
                        .getResourceAsStream(propFileName);

        if (inputStream != null) {
                prop.load(inputStream);
        } else {
                throw new FileNotFoundException("property file '" + propFileName
                                + "' not found in the classpath");
        }

        DOMAIN = prop.getProperty("domain.dns");
}

@Override
protected String getKeyspaceName() {
        return KEYSPACE_NAME;
}

@Override
protected String getContactPoints() {
        return CONTACT_POINTS;
}

@Override
protected int getPort() {
        return PORT;
}

@Bean(name = "casops")
public CassandraOperations operations() throws Exception {
        return new CassandraTemplate(session().getObject(),
                new MappingCassandraConverter(new BasicCassandraMappingContext()));
}

/**
 * Get the Cassandra session and create the keyspace and the schema.
 * @return
 * @throws IOException
 */
@Bean(name="cassandrasession")
public static Session getSession() throws IOException{
        loadCassandraProperties();

        //Create keyspace and schema
        Cluster cluster = Cluster.builder().addContactPoint(CONTACT_POINTS).build();
        Session session = cluster.connect();
        session.execute("CREATE KEYSPACE IF NOT EXISTS "
                        + KEYSPACE_NAME
                        + " WITH replication "
                        + "= {'class':'SimpleStrategy', 'replication_factor':3};");
        session.execute("USE " + KEYSPACE_NAME);
        checkKeySpace(session);
```

```
                return session;
        }

        /**
         * Read cassandra.properties and store the values
         * @throws IOException
         */
        private static void loadCassandraProperties() throws IOException {
                Properties prop = new Properties();
                String propFileName = "cassandra.properties";

                InputStream inputStream =
                        Application.class.getClassLoader().getResourceAsStream(propFileName);

                if (inputStream != null) {
                        prop.load(inputStream);
                } else {
                        throw new FileNotFoundException("property file '" + propFileName
                                        + "' not found in the classpath");
                }

                CONTACT_POINTS = prop.getProperty("cassandra.contactpoints");
                PORT = Integer.parseInt(prop.getProperty("cassandra.port"));
                KEYSPACE_NAME = prop.getProperty("cassandra.keyspace");
        }

        /**
         * Create the Schema. Read the SCHEMA.cql file and execute the sentences.
         * @param session Cassandra session
         * @throws IOException
         */
        private static void checkKeySpace(Session session) throws IOException {
                InputStream dataScheme =
                        Application.class.getResourceAsStream("/SCHEMA.cql");
                String scheme = IOUtils.toString(dataScheme);
                String[] cfs = scheme.split(";");
                for (String cf : cfs) {
                        session.execute(cf);
                }
        }
}
```

Utils.java

```
package api.util;

import java.util.Collections;
import java.util.List;

import com.google.common.collect.Lists;

import api.entity.dto.ResultsDTO;
import api.entity.table.Results;
import api.exception.ResultsNotFoundException;
import api.repository.ResultsRepository;


public class Utils {

        public static List<Results> findResultsByTestIdAndSensorId(String testId, String
                sensorId, int size, ResultsRepository repository) {
            List<Results> results=repository.findByTestIdAndSensorId(testId,sensorId,size);
            if(results.size() == 0 )
                throw new ResultsNotFoundException(testId, sensorId);
```

```
            return results;
        }

        public static List<Results> findNextResultsByTestIdAndSensorId(String testId,
                String sensorId, long eventTime, int size, ResultsRepository repository,
                boolean isChecked) {

                List<Results> results = repository.findByTestIdAndSensorIdNextPage(testId,
                sensorId, eventTime, size);

                if(results.size() == 0 && !isChecked)
                        throw new ResultsNotFoundException(testId, sensorId);
                Collections.reverse(results);
                return results;
        }

        public static List<Results> findPreviousResultsByTestIdAndSensorId(String testId,
                String sensorId, long eventTime, int size, ResultsRepository repository,
                boolean isChecked) {

                List<Results> results =
                        repository.findByTestIdAndSensorIdPreviousPage(testId, sensorId,
                                eventTime, size);
                if(results.size() == 0 && !isChecked)
                        throw new ResultsNotFoundException(testId, sensorId);
                return results;
        }

        public static List<ResultsDTO> convertListToListResultsDTO(Iterable<Results>
                listResults) {

                List<ResultsDTO> listResultsDTO = Lists.newArrayList();
                for(Results result : listResults)
                        listResultsDTO.add(convertToDTO(result));
                return listResultsDTO;
        }

        public static ResultsDTO convertToDTO(Results model) {
                ResultsDTO dto = new ResultsDTO();
                dto.setTestId(model.getPrimaryKey().getTestId());
                dto.setTestTime(model.getPrimaryKey().getTestTime());
                dto.setBoardId(model.getBoardId());
                dto.setSensorId(model.getPrimaryKey().getSensorId());
                dto.setInputValue(model.getInputValue());
                dto.setOutputValue(model.getOutputValue());
                dto.setEventTime(model.getEventTime());
                return dto;
        }
}
```

Results.java

```
package api.entity.table;

import org.springframework.data.cassandra.mapping.Column;
import org.springframework.data.cassandra.mapping.PrimaryKey;
import org.springframework.data.cassandra.mapping.Table;

/**
 * Representation of Cassandra table results.
 *
 * @author Luis Pulido
 *
 */
```

```
@Table(value = "results")
public class Results {

	@PrimaryKey
	private ResultsKey primaryKey;

	@Column(value = "board_id")
	private String boardId;

	@Column(value = "input_value")
	private double inputValue;

	@Column(value = "output_value")
	private double outputValue;

	@Column(value = "event_time")
	private long eventTime;

	//Constructors
	public Results() {}

	private Results(Builder builder) {
	 this.primaryKey = ResultsKey.getBuilder()
	                                    .testId(builder.testId)
	                                    .testTime(builder.testTime)
	                                    .sensorId(builder.sensorId)
	                                    .build();
	 this.boardId = builder.boardId;
	 this.inputValue = builder.inputValue;
	 this.outputValue = builder.outputValue;
	 this.eventTime = builder.eventTime;
	}

	//Getters and setters

	public static Builder getBuilder() {
		return new Builder();
	}

	/**
	* Builder pattern to create a Results
	* @author Luis Pulido
	*
	*/
	public static class Builder {
		private String testId;
		private long testTime;
		private String boardId;
		private String sensorId;
		private double inputValue;
		private double outputValue;
		private long eventTime;

		private Builder() {}

		public Builder testId(String testId){
			this.testId = testId;
			return this;
		}
		public Builder testTime(long testTime){
			this.testTime = testTime;
			return this;
		}
		public Builder boardId(String boardId){
			this.boardId = boardId;
			return this;
```

```
		}
		public Builder sensorId(String sensorId){
			this.sensorId = sensorId;
			return this;
		}
		public Builder inputValue(double inputValue){
			this.inputValue = inputValue;
			return this;
		}
		public Builder outputValue(double outputValue){
			this.outputValue = outputValue;
			return this;
		}
		public Builder eventTime(long eventTime){
			this.eventTime = eventTime;
			return this;
		}

		public Results build(){
			Results build = new Results(this);
			return build;
		}
	}
}
```

ResultsKey.java

```
package api.entity.table;

import java.io.Serializable;

import org.springframework.cassandra.core.Ordering;
import org.springframework.cassandra.core.PrimaryKeyType;
import org.springframework.data.cassandra.mapping.PrimaryKeyClass;
import org.springframework.data.cassandra.mapping.PrimaryKeyColumn;

/**
 * Composite primary key for {@link Results}
 * @author Luis Pulido
 *
 */
@PrimaryKeyClass
public class ResultsKey implements Serializable{

	private static final long serialVersionUID = 40723597991640197321;

	@PrimaryKeyColumn(name = "test_id", ordinal = 0, type = PrimaryKeyType.PARTITIONED)
	private String testId;

	@PrimaryKeyColumn(name = "sensor_id", ordinal = 1, type = PrimaryKeyType.CLUSTERED,
			ordering = Ordering.ASCENDING)
	private String sensorId;

	@PrimaryKeyColumn(name = "test_time", ordinal = 2, type = PrimaryKeyType.CLUSTERED,
			ordering = Ordering.DESCENDING)
	private long testTime;

	//Constructors
	public ResultsKey() {}

	private ResultsKey(Builder builder) {
		this.testId = builder.testId;
		this.testTime = builder.testTime;
		this.sensorId = builder.sensorId;
	}
```

```
	//Getters and setters

	//As implements Serializable class is necessary override this methods
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;

		result = (int) (prime * result + testTime);
		result = prime * result + ((testId == null) ? 0 : testId.hashCode());
		result = prime * result + ((sensorId == null) ? 0 : sensorId.hashCode());

		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		ResultsKey other = (ResultsKey) obj;
		if (testTime != other.testTime)
			return false;
		if (testId == null) {
			if (other.testId != null)
				return false;
		} else if (!testId.equals(other.testId))
			return false;
		if (sensorId == null) {
			if (other.sensorId != null)
				return false;
		} else if (!sensorId.equals(other.sensorId))
			return false;

		return true;
	}


	public static Builder getBuilder() {
		return new Builder();
	}

	/**
	 * Builder pattern to create the composite primary key for Results
	 *
	 * @author Luis
	 *
	 */
	public static class Builder {
		private String testId;
		private String sensorId;
		private long testTime;

		private Builder() {}

		public Builder testId(String testId){
			this.testId = testId;
			return this;
		}
		public Builder sensorId(String sensorId){
			this.sensorId = sensorId;
			return this;
```

```
		}
		public Builder testTime(long testTime){
			this.testTime = testTime;
			return this;
		}

		public ResultsKey build(){
			ResultsKey build = new ResultsKey(this);
			return build;
		}
	}
}
```

ResulstsDTO.java

```
package api.entity.dto;

import javax.annotation.Nonnegative;
import javax.validation.constraints.NotNull;

import org.hibernate.validator.constraints.NotEmpty;

/**
 * This data transfer object contains the information of a single {@link Results}
 * entry and specifies validation rules that are used to ensure that only
 * valid information can be saved to the used database.
 * @author Luis Pulido
 *
 */
public class ResultsDTO {

	@NotEmpty
	@NotNull
	private String testId;

	private long eventTime;
	@NotEmpty
	@NotNull
	private String boardId;
	@NotEmpty
	@NotNull
	private String sensorId;
	@NotNull
	private double inputValue;
	@NotNull
	private double outputValue;
	@NotNull
	@Nonnegative
	private long testTime;

	//Getters and setters
}
```

ListResultsDTOPaginated.java

```
package api.entity.dto;

import java.util.List;

public class ListResultsDTOPaginated {

	private String nextPage;
```

```
        private String previousPage;
        private List<ResultsDTO> Results;

        //Getters and setters
}
```

ResultsRepository.java

```
package api.repository;

import java.util.List;

import org.springframework.data.cassandra.repository.CassandraRepository;
import org.springframework.data.cassandra.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

import api.entity.table.Results;

@RepositoryRestResource(collectionResourceRel = "results", path = "results")
public interface ResultsRepository extends CassandraRepository<Results> {

        @Query("SELECT * FROM results WHERE test_id=?0 AND sensor_id=?1 LIMIT ?2 ")
        public List<Results> findByTestIdAndSensorId(@Param("test_id") String testId,
                        @Param("sensor_id") String sensorId,
                        @Param("size") int size);

        @Query("SELECT * FROM results WHERE test_id=?0 AND sensor_id=?1 AND test_time > ?2
                ORDER BY sensor_id DESC LIMIT ?3")
        public List<Results> findByTestIdAndSensorIdNextPage(@Param("test_id") String
                testId, @Param("sensor_id") String sensorId, @Param("event_time") long
                eventTime, @Param("size") int size);

        @Query("SELECT * FROM results WHERE test_id=?0 AND sensor_id=?1 AND test_time < ?2
                LIMIT ?3")
        public List<Results> findByTestIdAndSensorIdPreviousPage(@Param("test_id") String
                testId, @Param("sensor_id") String sensorId, @Param("event_time") long
                eventTime, @Param("size") int size);

        @Query("SELECT * FROM results WHERE test_id=?0")
        public List<Results> findByTestId(@Param("test_id") String testId);
}
```

ApiResultsController.java

```
package api.controller;

import java.text.ParseException;
import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import com.google.common.collect.Lists;
```

```
import api.entity.dto.ListResultsDTOPaginated;
import api.entity.dto.ResultsDTO;
import api.entity.table.Results;
import api.exception.SizeParameterException;
import api.exception.UriResultsException;
import api.repository.ResultsRepository;
import static api.Application.DOMAIN;
import static api.util.Utils.*;
/**
 * This controller provides the private API with OAuth that is used to retrieve the
information
 * of {@link Results} entries.
 * @author Luis Pulido
 */
@RestController
@RequestMapping("/api/results")
public class ApiResultsController {

	private static final Logger LOGGER =
LoggerFactory.getLogger(ApiResultsController.class);

	private final ResultsRepository repository;

	@Autowired
	ApiResultsController(ResultsRepository repository){
		this.repository = repository;
	}

	@RequestMapping(method = RequestMethod.GET)
	public List<ResultsDTO> findAll() {
		LOGGER.info("Finding all results entries");
		return convertListToListResultsDTO(repository.findAll());
	}

	@RequestMapping(value = "{test-id}/sensor/{sensor-id}", method = RequestMethod.GET)
	@ResponseStatus(HttpStatus.OK)
	public ListResultsDTOPaginated findByTestIdAndSensorId(
			@PathVariable("test-id") String testId,
			@PathVariable("sensor-id") String sensorId,
			@RequestParam(value="size", required=false, defaultValue="10") int
			 size,
			@RequestParam(value="date", required=false, defaultValue="0") long
			 testTime,
			 @RequestParam(value="page", required=false, defaultValue="") String
			 page) throws ParseException {

		LOGGER.info("Finding results entry with testId: {} | sensorId: {} | size {}
			 | date {} | page {} ", testId, sensorId, size, testTime, page);

		//Check correct parameters.
		checkParameters(page, testTime, size);

		List<Results> results = Lists.newArrayList();

		if(page.equals("next"))
			results = findNextResultsByTestIdAndSensorId(testId, sensorId,
				testTime, size, repository, false);
		else if(page.equals("previous"))
			results = findPreviousResultsByTestIdAndSensorId(testId, sensorId,
				testTime, size, repository, false);
		else
			results = findResultsByTestIdAndSensorId(testId, sensorId, size,
				repository);
```

```
        String nextPage = getNextPage(results, page, testTime, testId, sensorId,
                size);
        String previousPage = getPreviousPage(results, page, testId, sensorId,
                size);

        return getResultsPaginated(nextPage, previousPage, results);
}

private String getPreviousPage(List<Results> results, String page, String testId,
        String sensorId, int size) {
        String previousPage = null;

        long previousTestTime =
                results.get(results.size()-1).getPrimaryKey().getTestTime();
        if(findPreviousResultsByTestIdAndSensorId(testId, sensorId,previousTestTime,
                size, repository, true).size() != 0)
                previousPage = DOMAIN + "/api/results/" + testId + "/sensor/" +
                        sensorId + "?size=" + size  + "&date=" + previousTestTime +
                "&page=previous";

        return previousPage;
}

private String getNextPage(List<Results> results, String page, long testTime,
        String testId, String sensorId, int size) {
        String nextPage = null;
        long nextTestTime = results.get(0).getPrimaryKey().getTestTime();
        if((page.equals("next") || page.equals("previous")) && testTime != 0) {
                if(findNextResultsByTestIdAndSensorId(testId, sensorId, nextTestTime,
                        size, repository, true).size() != 0)
                        nextPage = DOMAIN + "/api/results/" + testId + "/sensor/" +
                                sensorId + "?size=" + size  + "&date=" +
                                results.get(0).getPrimaryKey().getTestTime() +
                                "&page=next";
        }
        return nextPage;
}

private ListResultsDTOPaginated getResultsPaginated(String nextPage, String
        previousPage, List<Results> results) {

        ListResultsDTOPaginated resultsPaginated = new ListResultsDTOPaginated();
        resultsPaginated.setNextPage(nextPage);
        resultsPaginated.setPreviousPage(previousPage);
        resultsPaginated.setResults(convertListToListResultsDTO(results));
        return resultsPaginated;
}

/**
 * Check the request parameters
 *
 * @param page
 * @param testTime
 * @param size
 */
private void checkParameters(String page, long testTime, int size) {
        if((page.equals("next") || page.equals("previous")) && testTime == 0)
                throw new UriResultsException("date");
        if(!(page.equals("next") || page.equals("previous")) && testTime != 0)
                throw new UriResultsException("page");
        if(size <= 0)
                throw new SizeParameterException();
        }
}
```

ApiSaveResultsController.java

```
package api.controller;

import javax.validation.Valid;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import api.entity.dto.ResultsDTO;
import api.entity.table.Results;
import api.repository.ResultsRepository;
import static api.util.Utils.convertToDTO;
import java.util.Date;

/**
 * This controller provides the private API with SSL that is used to store the information
 * of {@link Results} entries.
 * @author Luis Pulido
 */
@RestController
@RequestMapping("/api/save/results")
public class ApiSaveResultsController {

	private static final Logger LOGGER =
		LoggerFactory.getLogger(ApiSaveResultsController.class);

	private final ResultsRepository repository;

	@Autowired
	public ApiSaveResultsController(ResultsRepository repository){
		this.repository = repository;
	}

	@RequestMapping(method = RequestMethod.POST, consumes = {"application/json"})
	@ResponseStatus(HttpStatus.CREATED)
	public ResultsDTO create(@RequestBody @Valid ResultsDTO resultsEntry) {
		LOGGER.info("Creating a new results entry with information: {}",
			resultsEntry);

		LOGGER.info("Creating a new results entry with information: {}", new
			Date());

		Results created = Results.getBuilder()
					.testId(resultsEntry.getTestId())
					.testTime(resultsEntry.getTestTime())
					.boardId(resultsEntry.getBoardId())
					.sensorId(resultsEntry.getSensorId())
					.eventTime(System.currentTimeMillis())
					.inputValue(resultsEntry.getInputValue())
					.outputValue(resultsEntry.getOutputValue())
					.build();

		created = repository.save(created);

		LOGGER.info("Created a new todo entry with information: {}", created);

		return convertToDTO(created);
	}
}
```

CSVFileDownloadController.java

```
package api.controller;

import static api.util.Utils.convertListToListResultsDTO;
import java.io.IOException;
import java.util.List;
import javax.servlet.http.HttpServletResponse;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import org.supercsv.io.CsvBeanWriter;
import org.supercsv.io.ICsvBeanWriter;
import org.supercsv.prefs.CsvPreference;
import api.entity.dto.ResultsDTO;
import api.repository.ResultsRepository;

@RestController
@RequestMapping("/api/downloadCSV")
public class CSVFileDownloadController {

	private static final Logger LOGGER =
		LoggerFactory.getLogger(CSVFileDownloadController.class);

	private final ResultsRepository repository;

	@Autowired
	CSVFileDownloadController(ResultsRepository repository){
		this.repository = repository;
	}

	@RequestMapping(value = "{test-id}", method = RequestMethod.GET)
	@ResponseStatus(HttpStatus.OK)
	public void downloadCSV(HttpServletResponse response,
			@PathVariable("test-id") String testId) throws IOException {
		LOGGER.info("download " + testId);
		String csvFileName = "test_" + testId + "_" + System.currentTimeMillis() +
		".csv";
		 response.setContentType("text/csv");
		 // retrieve the data
		 String headerKey = "Content-Disposition";
		 String headerValue = String.format("attachment; filename=\"%s\"",
			csvFileName);
		 response.setHeader(headerKey, headerValue);
		 List<ResultsDTO> listResults =
			convertListToListResultsDTO(repository.findByTestId(testId));
		 // uses the Super CSV API to generate CSV data from the model data
		 ICsvBeanWriter csvWriter = new CsvBeanWriter(response.getWriter(),
			CsvPreference.STANDARD_PREFERENCE);
		 String[] header = { "testId", "sensorId", "testTime", "boardId",
			"inputValue", "outputValue", "eventTime" };
		 csvWriter.writeHeader(header);

		 for (ResultsDTO result : listResults) {
		     csvWriter.write(result, header);
		 }
		 csvWriter.close();
	}
}
```

Anexo D. Código API REST gestión del catálogo del sistema

Requirements.txt

```
alabaster==0.7.6
Babel==2.1.1
Django==1.8.4
django-cors-headers==1.1.0
django-rest-swagger==0.3.4
djangorestframework==3.2.4
docutils==0.12
gunicorn==19.3.0
Jinja2==2.8
MarkupSafe==0.23
Pygments==2.0.2
pytz==2015.6
PyYAML==3.11
requests==2.8.1
six==1.10.0
snowballstemmer==1.2.0
Sphinx==1.3.1
sphinx-rtd-theme==0.1.9
wheel==0.24.0
```

Settings.py

```
import os

BASE_DIR = os.path.dirname(os.path.dirname(__file__))
SECRET_KEY = ')h7a=-1r)8_33!*m=l#n0@5&#xgg6i!=f12^(uur=cc)&=me3g'
DEBUG = True
ALLOWED_HOSTS = []

# Application definition

INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'catalog',
    'rest_framework_swagger',
    'corsheaders',
)
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'django.middleware.security.SecurityMiddleware',
```

```
)
ROOT_URLCONF = 'api_catalog.urls'

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]

WSGI_APPLICATION = 'api_catalog.wsgi.application'

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}

LANGUAGE_CODE = 'en-us'
TIME_ZONE = 'UTC'
USE_I18N = True
USE_L10N = True
USE_TZ = True

STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'static')

################################
## django-restframework
################################
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.LimitOffsetPagination',
    'PAGE_SIZE': 10
}

################################
## django-rest-swagger
################################
SWAGGER_SETTINGS = {
    "exclude_namespaces": ["internal_apis"],    #  List URL namespaces to ignore

    'info': {
        'description': 'This is a simple UI designed to manage the information about the '
                       'system that is required to launch the sensor\'s tests. '
                       'To create a test and visualize the results go to '
                       '<a href="http://dev-front.pfc.com">'
                       'http://dev-front.pfc.com</a> ',
        'title': 'GNB API Catalog UI',
    },
}
################################
## django-cors-headers
################################
CORS_ORIGIN_ALLOW_ALL = True
```

Urls.py

```
from rest_framework.routers import SimpleRouter
from django.conf.urls import include, url, patterns
from django.contrib import admin
from catalog.views import *

router = SimpleRouter()
router.register(r'board', BoardViewSet)
router.register(r'sensor', SensorViewSet)
router.register(r'model', ModelViewSet)
router.register(r'parameter', ParameterViewSet)

router_test = SimpleRouter()
router_test.register(r'test', TestViewSet)
# router_test.register(r'test-parameter', TestParameterViewSet)

internal_apis = patterns('', url(r'^', include(router_test.urls)), )

urlpatterns = [
    url(r'^docs/', include('rest_framework_swagger.urls')),
    url(r'^admin/', include(admin.site.urls)),
    url(r'^', include(router.urls))
]

urlpatterns = urlpatterns + patterns('',
       url(r'^', include(internal_apis, namespace="internal_apis")),)
```

Utils.py

```
from models import Sensor, Parameter, Board
from rest_framework.exceptions import ValidationError


def check_sensor_request(data, pk):
        board_pk = data.get('board')
        # If board parameter is not present then put the 'pin' parameter to null and
'is_used' parameter to false.
        if board_pk:
            board = Board.objects.get(pk=board_pk)
            board_number_pins = board.number_pins
            if not data.get('pin_input'):
                raise ValidationError({"pin_input": "This field may not be blank."})
            if not data.get('pin_output'):
                raise ValidationError({"pin_output": "This field may not be blank."})

            data_pin_input = int(data.get('pin_input'))
            data_pin_output = int(data.get('pin_output'))

            # Check if the pin_input provided is valid. The pins starts in 0
            if data_pin_input > (board_number_pins-1):
                raise ValidationError({"pin_input": "The pin parameter must be less than "
                      + str(board_number_pins) + "."})

            # Check if the pin_output provided is valid. The pins starts in 0
            if data_pin_output > (board_number_pins-1):
                raise ValidationError({"pin_output": "The pin parameter must be less than "
                      + str(board_number_pins) + "."})

            if data_pin_input == data_pin_output:
                raise ValidationError({"pin_input": "Can not be the same that pin_output.",
                                       "pin_output":"Can not be the same that pin_input."})

            for sensor in Sensor.objects.filter(board=board):
```

```
                # Check if are more than 2 sensors with the same assigned pin board
                if (sensor.pin_input == data_pin_input or sensor.pin_output ==
data_pin_input) \ and (pk is None or (pk is not None and sensor.id != pk)):
                    raise ValidationError({"pin_input": "Invalid pk \"" +
                      str(data_pin_input) + "\" - object is occupied."})
                if (sensor.pin_input == data_pin_output or sensor.pin_output ==
data_pin_output)\ and (pk is None or (pk is not None and sensor.id != pk)):
                    raise ValidationError({"pin_output": "Invalid pk \"" +
str(data_pin_output) + "\" - object is occupied."})


def check_valid_test_serializer(data):
        model_parameters = Parameter.objects.filter(model=data.get('model'))
        test_parameters = data.get('parameters')

        error_message = {}

        # Check if all request parameters are valid
        error_parameters = []
        error_parameters, test_parameters_objects =
check_request_test_parameters(test_parameters, model_parameters,

error_parameters)
        for error in error_parameters:
            if error:
                error_message["parameters"] = error_parameters

        # Check if all model parameters are present in the test request
        error_model_parameters = []
        error_model_parameters = check_request_model_parameters(test_parameters_objects,
model_parameters, error_model_parameters)
        for error in error_model_parameters:
            if error:
                error_message["parameters_required"] = error_model_parameters

        if error_message:
            raise ValidationError(error_message)


def check_request_test_parameters(test_parameters, model_parameters, error_parameters):
    """
    Check if the parameters for the test in the request are valid.

    :param test_parameters: request parameters
    :param model_parameters: model parameters selected in the request
    :param error_parameters: errors are added into this dictionary
    :return: the dictionary completed and the request parameters object
    """
    test_parameters_objects = []
    for test_parameter in test_parameters:
        parameter_object = Parameter.objects.get(pk=test_parameter['model_parameter'])

        error_parameter = {}
        # If parameter not in the model. Break the loop and check the next.
        if parameter_object not in model_parameters:
            error_parameter["model_parameter"] = "Invalid pk \"" + str(parameter_object.id)
+ \ "\" - object is not present in the model."
            error_parameters.append(error_parameter)
            break

        error_parameter = check_parameter_type(parameter_object,
parameter_object.type.lower(), test_parameter['value'], error_parameter)

        model_related_parameters = []
        for param in parameter_object.related_parameters.all():
            model_related_parameters.append(Parameter.objects.get(pk=param.id))
```

```
        error_related_parameters = []
        test_related_parameters_objects = []
        for related_parameter in test_parameter['related_parameters']:
            related_parameter_object =
Parameter.objects.get(pk=related_parameter['model_parameter'])

            error_related_parameter = {}
            if related_parameter_object not in model_related_parameters:
                error_related_parameter["model_parameter"] = "Invalid pk \"" +
str(related_parameter_object.id) + "\" - object is not present in the model."
                test_related_parameters_objects.append(related_parameter_object)
                error_related_parameters.append(error_related_parameter)
                break

            if related_parameter_object in test_related_parameters_objects:
                error_related_parameter["model_parameter"] = "Invalid pk \"" +
str(related_parameter_object.id) + "\" - object is repeated."
                test_related_parameters_objects.append(related_parameter_object)
                error_related_parameters.append(error_related_parameter)
                break

            error_related_parameter = check_parameter_type(related_parameter_object,
                  related_parameter_object.type.lower(), related_parameter['value'],
error_related_parameter)

            test_related_parameters_objects.append(related_parameter_object)
            error_related_parameters.append(error_related_parameter)

        for param in test_parameters_objects:
            if parameter_object == param[0]:
                error_parameter["model_parameter"] = "Invalid pk \"" +
str(parameter_object.id) + "\" - object is repeated."

        test_parameters_objects.append((parameter_object, test_related_parameters_objects))

        for error in error_related_parameters:
            if error:
                error_parameter["related_parameters"] = error_related_parameters

        error_parameters.append(error_parameter)

    return error_parameters, test_parameters_objects


def check_request_model_parameters(test_parameters_objects, model_parameters,
error_model_parameters):
    """
    Check if the parameters for the model test in the request ara valid.

    :param test_parameters_objects: request parameters object
    :param model_parameters: model parameters selected in the request
    :param error_model_parameters: errors are added into this dictionary
    :return: the dictionary completed
    """
    for model_parameter in model_parameters:
        model_related_parameters = []
        for param in model_parameter.related_parameters.all():
            model_related_parameters.append(Parameter.objects.get(pk=param.id))

        model_present = False
        error_model_parameter = {}
        for test_param in test_parameters_objects:
            if model_parameter == test_param[0]:
                model_present = True
                error_related = False
```

```
                error_model_related_parameters = []
                for related in model_related_parameters:
                    error_model_related_parameter = {}
                    if related not in test_param[1]:
                        error_model_related_parameter["model_parameter"] = "pk \"" +
str(related.id) + "\" - object is required."
                        error_related = True
                    error_model_related_parameters.append(error_model_related_parameter)

                if error_related:
                    error_model_parameter["related_parameter"] =
error_model_related_parameters

        if not model_present:
            error_model_parameter["model_parameter"] = "pk \"" + str(model_parameter.id) +
"\" - object is required."

        error_model_parameters.append(error_model_parameter)
    return error_model_parameters


def check_parameter_type(parameter_object, parameter_type, parameter_value, error):
    """
    Check if the type and the value of the parameter are valid.

    :param parameter_object:
    :param parameter_type:
    :param parameter_value:
    :param error: errors are added into this dictionary
    :return: param error completed.
    """
    if parameter_type == 'sensor':
        try:
            if Sensor.objects.get(pk=parameter_value).is_used:
                error["value"] = "Invalid pk \"" + str(parameter_value) + "\" - sensor is
being used."
        except Exception as e:
            error["value"] = e.message

    if parameter_type == 'string':
        if parameter_object.min_value:
            if len(parameter_value) < parameter_object.min_value:
                error["value"] = "Must be greater than " + str(parameter_object.min_value)
        if parameter_object.max_value:
            if len(parameter_value) > parameter_object.max_value:
                error["value"] = "Must be lower than " + str(parameter_object.max_value)

    if parameter_object.type == 'integer':
        try:
            int(parameter_value)
        except Exception as e:
            print(e.message)
            error["value"] = "Must be an integer."

        if parameter_object.min_value:
            if int(parameter_value) < int(parameter_object.min_value):
                error["value"] = "Must be greater than " + str(parameter_object.min_value)
        if parameter_object.max_value:
            if int(parameter_value) > int(parameter_object.max_value):
                error["value"] = "Must be lower than " + str(parameter_object.max_value)

    if parameter_object.type == 'float':
        try:
            float(parameter_value)
        except Exception as e:
            print(e.message)
```

```
            error["value"] = "Must be a float."

        if parameter_object.min_value:
            if float(parameter_value) < int(parameter_object.min_value):
                error["value"] = "Must be greater than " + str(parameter_object.min_value)
        if parameter_object.max_value:
            if float(parameter_value) > int(parameter_object.max_value):
                error["value"] = "Must be lower than " + str(parameter_object.max_value)

    return error
```

Validators.py

```
from django.core.exceptions import ValidationError


def validate_positive(value):
    if value < 0:
        raise ValidationError('The parameter must be a positive value')


def validate_greater_than_0(value):
    if value <= 0:
        raise ValidationError('The parameter must be a greater than 0')
```

Models.py

```
from django.db import models
from .validators import validate_positive, validate_greater_than_0


class Board(models.Model):
    name = models.TextField()
    description = models.TextField(null=True, blank=True)
    uri = models.URLField()
    number_pins = models.PositiveSmallIntegerField(validators=[validate_greater_than_0])
    created_on = models.DateTimeField(auto_now_add=True)
    modified_on = models.DateTimeField(auto_now=True)


class Sensor(models.Model):
    board = models.ForeignKey(Board, related_name='sensors', null=True)
    pin_output = models.IntegerField(validators=[validate_positive], null=True)
    pin_input = models.IntegerField(validators=[validate_positive], null=True)
    name = models.TextField()
    description = models.TextField(null=True, blank=True)
    is_used = models.BooleanField(default=False)
    created_on = models.DateTimeField(auto_now_add=True)
    modified_on = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ['pin_output']


class Model(models.Model):
    name = models.TextField()
    description = models.TextField(null=True, blank=True)
    created_on = models.DateTimeField(auto_now_add=True)
    modified_on = models.DateTimeField(auto_now=True)
```

```
class Parameter(models.Model):
    model = models.ManyToManyField(Model, related_name='parameters', blank=True)
    name = models.TextField()
    type = models.TextField()
    min_value = models.TextField(blank=True, null=True)
    max_value = models.TextField(blank=True, null=True)
    related_parameters = models.ManyToManyField('self', blank=True)

    def __eq__(self, other):
        return self.id == other.id and self.name == other.name and self.type == other.type
\ and self.min_value == other.min_value and self.max_value == other.max_value


class Test(models.Model):
    created_on = models.DateTimeField(auto_now_add=True)
    finished_on = models.DateTimeField(blank=True, null=True)
    status = models.TextField(default="waiting",blank=True, null=True)
    progress = models.FloatField(validators=[validate_positive], null=True, default=0)
    pid = models.IntegerField(validators=[validate_positive], null=True)
    model = models.ForeignKey(Model)
    board = models.ForeignKey(Board)


class TestParameter(models.Model):
    test = models.ForeignKey(Test, related_name='parameters', null=True)
    # name = models.TextField()
    model_parameter = models.ForeignKey(Parameter, related_name='model_parameter')
    value = models.TextField()
    related_parameters = models.ManyToManyField('self', blank=True)
```

Serializers.py

```
from models import *
from rest_framework.serializers import ModelSerializer


class SensorSerializer(ModelSerializer):
    class Meta:
        model = Sensor


class BoardSerializer(ModelSerializer):
    sensors = SensorSerializer(many=True, read_only=True)

    class Meta:
        model = Board
        fields = ('id', 'name', 'description', 'uri', 'number_pins', 'created_on',
'modified_on', 'sensors')


class ParameterSerializer(ModelSerializer):
    class Meta:
        model = Parameter
        fields = ('id', 'model', 'name', 'type', 'min_value', 'max_value',
'related_parameters')


class ParameterValueSerializer(ModelSerializer):
    class Meta:
        model = Parameter
        fields = ('id', 'name', 'type', 'min_value', 'max_value')


class ParameterModelSerializer(ModelSerializer):
```

```
    related_parameters = ParameterValueSerializer(many=True, read_only=True)

    class Meta:
        model = Parameter
        fields = ('id', 'name', 'type', 'min_value', 'max_value', 'related_parameters')


class ModelSerializer(ModelSerializer):
    parameters = ParameterModelSerializer(many=True, read_only=True)

    class Meta:
        model = Model
        fields = ('id', 'name', 'description', 'created_on', 'modified_on', 'parameters')


class TestParameterValueSerializer(ModelSerializer):
    class Meta:
        model = TestParameter
        fields = ('model_parameter', 'value')


class TestParameterSerializer(ModelSerializer):
    related_parameters = TestParameterValueSerializer(many=True)

    class Meta:
        model = TestParameter
        fields = ('model_parameter', 'value', 'related_parameters')


class TestSerializer(ModelSerializer):
    parameters = TestParameterSerializer(many=True)

    class Meta:
        model = Test
        fields = ('id', 'created_on', 'finished_on', 'model', 'board', 'status',
'progress', 'pid', 'parameters')

    def create(self, validated_data):
        parameters = validated_data.pop('parameters')
        test = Test.objects.create(**validated_data)
        for parameter in parameters:
            related_parameters = parameter.pop('related_parameters')
            parameter = TestParameter.objects.create(test=test, **parameter)
            for related_parameter in related_parameters:
                related_parameter = TestParameter.objects.create(**related_parameter)
                parameter.related_parameters.add(related_parameter)
        return test


class TestParameterValueRetrieveSerializer(ModelSerializer):
    model_parameter = ParameterValueSerializer(read_only=True)

    class Meta:
        model = TestParameter
        fields = ('model_parameter', 'value')


class TestParameterRetrieveSerializer(ModelSerializer):
    related_parameters = TestParameterValueRetrieveSerializer(many=True)
    model_parameter = ParameterValueSerializer(read_only=True)

    class Meta:
        model = TestParameter
        fields = ('model_parameter', 'value', 'related_parameters')
```

```
class TestRetrieveSerializer(ModelSerializer):
    parameters = TestParameterRetrieveSerializer(many=True)
    model = ModelSerializer(read_only=True)

    class Meta:
        model = Test
        fields = ('id', 'created_on', 'finished_on', 'model', 'board', 'status',
'progress', 'pid', 'parameters')
```

Views.py

```
from serializers import *
from rest_framework.viewsets import ModelViewSet
from rest_framework.response import Response
from rest_framework import status
from utils import check_valid_test_serializer, check_sensor_request
import requests
from create_info import create_info
from rest_framework.decorators import detail_route


class BoardViewSet(ModelViewSet):
    """ Board resource. """

    queryset = Board.objects.all()
    serializer_class = BoardSerializer


class SensorViewSet(ModelViewSet):
    """ Sensor resource. """

    queryset = Sensor.objects.all()
    serializer_class = SensorSerializer

    def create(self, request, *args, **kwargs):
        # If the sensor is new cannot be still used
        request.data["is_used"] = False
        # If the sensor has not "board" field then cannot have an assigned pin
        if not request.data.get('board'):
            request.data["pin_output"] = None
            request.data["pin_input"] = None
        return super(SensorViewSet, self).create(request, args, kwargs)

    def perform_create(self, serializer):
        check_sensor_request(self.request.data, None)
        serializer.save()

    def update(self, request, *args, **kwargs):
        # If the sensor has not "board" field then cannot have an assigned pin
        if not request.data.get('board'):
            request.data["pin_output"] = None
            request.data["pin_input"] = None
            request.data["is_used"] = False
        return super(SensorViewSet, self).update(request, args, kwargs)

    def perform_update(self, serializer):
        check_sensor_request(self.request.data, self.get_object().id)
        serializer.save()


class ModelViewSet(ModelViewSet):
    """ Model resource. """

    queryset = Model.objects.all()
```

```
    serializer_class = ModelSerializer


class ParameterViewSet(ModelViewSet):
    """ Parameter resource.    """

    queryset = Parameter.objects.all()
    serializer_class = ParameterSerializer

    def partial_update(self, request, *args, **kwargs):
        """
        ---
        response_serializer: ParameterSerializer
        parameters:
            - name: Body
              description: Enviar Body con los parametros unicamente si se desean mas de
               dos modelos. Ejemplo Escribir en formato JSON {"model":["MODEL_ID", ...]}

              required: true
              type: array(string)
              paramType: body

        """
        return super(ParameterViewSet, self).partial_update(request, *args, **kwargs)


class TestViewSet(ModelViewSet):
    """ Test resource. """

    queryset = Test.objects.all()
    #serializer_class = TestSerializer

    def get_serializer_class(self):
        if self.action == 'list' or self.action == 'retrieve':
            return TestRetrieveSerializer
        return TestSerializer

    @detail_route(methods=['get'])
    def stop(self, request, pk=None):
        instance = self.get_object()

        if instance.status != "running" and instance.status != "ready":
            return Response("Test is " + instance.status + ".",
status=status.HTTP_400_BAD_REQUEST)

        # Make the request to stop the test
        board_uri = instance.board.uri + "/api/kill/" + str(instance.pid)

        try:
            r = requests.get(board_uri)
        except requests.ConnectionError:
            return Response("Board connection error.",
status=status.HTTP_503_SERVICE_UNAVAILABLE)

        if r.status_code != status.HTTP_200_OK:
            return Response(r.content, status=r.status_code)

        # Update the test
        instance.status = "stopped"
        instance.pid = None
        instance.save()

        serializer = self.get_serializer(instance)
        return Response(serializer.data)

    def create(self, request, *args, **kwargs):
```

```
        serializer = self.get_serializer(data=request.data)
        serializer.is_valid(raise_exception=True)

        data = self.request.data

        # Check the request serializer
        check_valid_test_serializer(data)

        print(data)

        # Save the info to obtain the test id
        test = serializer.save()

        # Create info to send to the board.
        info = create_info(data.get('model'), data['board'], data.get('parameters'), test)
        print("info")
        print (info)

        # Make the request to launch the test
        board = Board.objects.get(pk=data['board'])
        board_uri = board.uri + "/api/model/" + str(data.get('model'))
        try:
            r = requests.post(board_uri, json=info)
        except requests.ConnectionError:
            test.delete()
            return Response("Board connection error.",
status=status.HTTP_503_SERVICE_UNAVAILABLE, headers={})

        if r.status_code != status.HTTP_202_ACCEPTED:
            test.delete()
            return Response(r.content, status=r.status_code)

        # Update the test
        test.pid = r.content
        test.save()

        headers = self.get_success_headers(serializer.data)
        return Response(serializer.data, status=status.HTTP_201_CREATED, headers=headers)

    def perform_update(self, serializer):
        data = self.request.data
        # Check the request serializer
        # check_valid_test_serializer(serializer, data)
        # Save the info to obtain the test id
        serializer.save()
```

Createinfo.py

```
from .models import Sensor


def create_info(model, board_id, test_parameters, test):
    if model == 1:
        sensor_id = 0
        sensor_pin_input = 0
        sensor_pin_output = 0
        sampling_time = 0

        for param in test_parameters:
            if param['model_parameter'] == 1:  # sensor id
                sensor = Sensor.objects.get(pk=param['value'])
                sensor_id = sensor.id
                sensor_pin_input = sensor.pin_input
                sensor_pin_output = sensor.pin_output
```

```
            if param['model_parameter'] == 2:  # sampling Time Id
                sampling_time = param['value']

        sensor1 = {
            "sensorId": sensor_id,
            "pinInput": sensor_pin_input,
            "pinOutput": sensor_pin_output,
        }

        info = {
            "boardId": board_id,
            "testId": test.id,
            "samplingTime": sampling_time,
            "sensor1": sensor1
        }

    elif model == 2:
        sensor_id = 0
        sensor_pin_input = 0
        sensor_pin_output = 0
        sampling_time = 0

        for param in test_parameters:
            if param['model_parameter'] == 1:  # sensor id
                sensor = Sensor.objects.get(pk=param['value'])
                sensor_id = sensor.id
                sensor_pin_input = sensor.pin_input
                sensor_pin_output = sensor.pin_output

            if param['model_parameter'] == 2:  # sampling Time Id
                sampling_time = param['value']

        sensor1 = {

            "sensorId": sensor_id,
            "pinInput": sensor_pin_input,
            "pinOutput": sensor_pin_output,
        }

        info = {
            "boardId": board_id,
            "testId": test.id,
            "samplingTime": sampling_time,
            "sensor1": sensor1
        }

    elif model == 3:
        sensor_id = 0
        sensor_pin_input = 0
        sensor_pin_output = 0
        led_id = 0
        led_pin_input = 0
        led_pin_output = 0
        led_on_value = 0
        sampling_time = 0

        for param in test_parameters:
            if param['model_parameter'] == 1:  # sensor id
                sensor = Sensor.objects.get(pk=param['value'])
                sensor_id = sensor.id
                sensor_pin_input = sensor.pin_input
                sensor_pin_output = sensor.pin_output

            if param['model_parameter'] == 2:  # sampling Time Id
                sampling_time = param['value']
```

```
            if param['model_parameter'] == 3:  # variable LED ID
                led = Sensor.objects.get(pk=param['value'])
                led_id = led.id
                led_pin_input = led.pin_input
                led_pin_output = led.pin_output

                for related_parameter in param['related_parameters']:
                    if related_parameter['model_parameter'] == 4:
                        led_on_value = related_parameter['value']

        led = {
            "sensorId": led_id,
            "pinInput": led_pin_input,
            "pinOutput": led_pin_output,
            "ledOnValue": led_on_value,
        }

        sensor_ldr = {
            "sensorId": sensor_id,
            "pinInput": sensor_pin_input,
            "pinOutput": sensor_pin_output,
        }

        info = {
            "boardId": board_id,
            "testId": test.id,
            "samplingTime": sampling_time,
            "sensorLDR": sensor_ldr,
            "led": led
        }

    elif model == 4:
        sensor_id = 0
        sensor_pin_input = 0
        sensor_pin_output = 0
        temperature_id = 0
        temperature_pin_input = 0
        temperature_pin_output = 0
        sampling_time = 0

        for param in test_parameters:
            if param['model_parameter'] == 1:  # sensor id
                sensor = Sensor.objects.get(pk=param['value'])
                sensor_id = sensor.id
                sensor_pin_input = sensor.pin_input
                sensor_pin_output = sensor.pin_output

            if param['model_parameter'] == 2:  # sampling Time Id
                sampling_time = param['value']

            if param['model_parameter'] == 5:  # variable Temperature ID
                temperature = Sensor.objects.get(pk=param['value'])
                temperature_id = temperature.id
                temperature_pin_input = temperature.pin_input
                temperature_pin_output = temperature.pin_output

        var_temperature = {
            "sensorId": temperature_id,
            "pinInput": temperature_pin_input,
            "pinOutput": temperature_pin_output,
        }

        sensor_lm335 = {
            "sensorId": sensor_id,
            "pinInput": sensor_pin_input,
```

```
            "pinOutput": sensor_pin_output,
        }

        info = {
            "boardId": board_id,
            "testId": test.id,
            "samplingTime": sampling_time,
            "sensorLM335": sensor_lm335,
            "varTemperature": var_temperature
        }

    elif model == 5:

        sensor_gas = sensor_temp = var_slope = var_value = var_std = {}
        nm = t = sleep = samples_inicio = tendency = average_temperature = duration = 0

        for param in test_parameters:
            if param['model_parameter'] == 6:  # sensor gas id
                sensor_gas = Sensor.objects.get(pk=param['value'])

            if param['model_parameter'] == 7:  # sensor temp Id
                sensor_temp = Sensor.objects.get(pk=param['value'])

            if param['model_parameter'] == 8:  # var slope intercept Id
                var_slope = Sensor.objects.get(pk=param['value'])

            if param['model_parameter'] == 9:  # var value Id
                var_value = Sensor.objects.get(pk=param['value'])

            if param['model_parameter'] == 10:  # var std Id
                var_std = Sensor.objects.get(pk=param['value'])

            if param['model_parameter'] == 11:  # nm
                nm = param['value']

            if param['model_parameter'] == 12:  # nm
                t = param['value']

            if param['model_parameter'] == 13:  # nm
                sleep = param['value']

            if param['model_parameter'] == 14:  # nm
                samples_inicio = param['value']

            if param['model_parameter'] == 15:  # nm
                tendency = param['value']

            if param['model_parameter'] == 16:  # nm
                average_temperature = param['value']

            if param['model_parameter'] == 17:  # nm
                duration = param['value']

        sensor_gas_info = {
            "sensorId": sensor_gas.id,
            "pinInput": sensor_gas.pin_input,
            "pinOutput": sensor_gas.pin_output
        }

        sensor_temp_info = {
            "sensorId": sensor_temp.id,
            "pinInput": sensor_temp.pin_input,
            "pinOutput": sensor_temp.pin_output
        }

        var_slope_intercept_info = {
```

```
            "sensorId": var_slope.id,
            "pinInput": var_slope.pin_input,
            "pinOutput": var_slope.pin_output
        }

        var_rvalue_pvalue = {
            "sensorId": var_value.id,
            "pinInput": var_value.pin_input,
            "pinOutput": var_value.pin_output
        }

        var_std_err1 = {
            "sensorId": var_std.id,
            "pinInput": var_std.pin_input,
            "pinOutput": var_std.pin_output
        }

        info = {
            "boardId": board_id,
            "testId": test.id,
            "sensorGas": sensor_gas_info,
            "sensorLM35": sensor_temp_info,
            "varSlopeIntercept": var_slope_intercept_info,
            "varRvaluePvalue": var_rvalue_pvalue,
            "varStdErr1": var_std_err1,
            "nm": nm,
            "t": t,
            "sleep": sleep,
            "samplesInicio": samples_inicio,
            "tendency": tendency,
            "averageTemperature": average_temperature,
            "duration": duration
        }

    else:
        info = {}

    return info
```

Anexo E. Código interfaz de administración de las pruebas.

Bower.json

```
{
  "name": "app",
  "version": "0.0.0",
  "dependencies": {
    "angular": "^1.3.0",
    "angular-animate": "~1.4.7",
    "angular-bootstrap": "~0.14.2",
    "angular-cookies": "^1.3.0",
    "angular-loading-bar": "~0.9.0",
    "angular-messages": "~1.4.8",
    "angular-options-disabled": "*",
    "angular-resource": "^1.3.0",
    "angular-route": "^1.3.0",
    "angular-sanitize": "^1.3.0",
    "angular-touch": "^1.3.0",
    "bootstrap": "^3.2.0",
    "checklist-model": "~0.9.0",
    "components-font-awesome": "~4.4.0",
    "metisMenu": "~2.2.0",
    "restangular": "~1.5.1"
  },
  "devDependencies": {
    "angular-mocks": "^1.3.0"
  },
  "appPath": "app",
  "moduleName": "appApp",
  "overrides": {
    "bootstrap": {
      "main": [
        "less/bootstrap.less",
        "dist/css/bootstrap.css",
        "dist/js/bootstrap.js"
      ]
    }
  }
}
```

Package.json

```
{
  "name": "app",
  "private": true,
  "devDependencies": {
    "grunt": "^0.4.5",
    "grunt-angular-templates": "^0.5.7",
    "grunt-autoprefixer": "^2.0.0",
    "grunt-concurrent": "^1.0.0",
    "grunt-contrib-clean": "^0.6.0",
    "grunt-contrib-concat": "^0.5.0",
    "grunt-contrib-connect": "^0.9.0",
    "grunt-contrib-copy": "^0.7.0",
    "grunt-contrib-cssmin": "^0.12.0",
    "grunt-contrib-htmlmin": "^0.4.0",
    "grunt-contrib-imagemin": "^1.0.0",
    "grunt-contrib-jshint": "^0.11.0",
```

```
    "grunt-contrib-uglify": "^0.7.0",
    "grunt-contrib-watch": "^0.6.1",
    "grunt-filerev": "^2.1.2",
    "grunt-google-cdn": "^0.4.3",
    "grunt-karma": "*",
    "grunt-newer": "^1.1.0",
    "grunt-ng-annotate": "^0.9.2",
    "grunt-svgmin": "^2.0.0",
    "grunt-usemin": "^3.0.0",
    "grunt-wiredep": "^2.0.0",
    "jit-grunt": "^0.9.1",
    "jshint-stylish": "^1.0.0",
    "karma-jasmine": "*",
    "karma-phantomjs-launcher": "*",
    "time-grunt": "^1.0.0"
  },
  "engines": {
    "node": ">=0.10.0"
  },
  "scripts": {
    "test": "grunt test"
  },
  "dependencies": {
    "n3-charts": "~2.0.16"
  }
}
```

Index.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title></title>
  <meta name="description" content="">
  <meta name="viewport" content="width=device-width">
  <!-- Place favicon.ico and apple-touch-icon.png in the root directory -->
  <!-- build:css(.) styles/vendor.css -->
  <!-- bower:css -->
  <link rel="stylesheet" href="bower_components/angular-loading-bar/build/loading-bar.css"
/>
  <link rel="stylesheet" href="bower_components/bootstrap/dist/css/bootstrap.css" />
  <link rel="stylesheet" href="bower_components/components-font-awesome/css/font-
awesome.css" />
  <link rel="stylesheet" href="bower_components/metisMenu/dist/metisMenu.css" />
  <!-- endbower -->

  <!-- endbuild -->
  <!-- build:css(.tmp) styles/main.css -->
  <link rel="stylesheet" href="styles/timeline.css" />
  <link rel="stylesheet" href="styles/sb-admin-2.css" />
  <link rel="stylesheet" href="styles/main.css" />
  <!-- endbuild -->

</head>
<body ng-app="appApp">
<!--[if lte IE 8]>
<p class="browsehappy">You are using an <strong>outdated</strong> browser. Please <a
href="http://browsehappy.com/">upgrade your browser</a> to improve your experience.</p>
<![endif]-->

<div id="wrapper">

  <!-- include navbar-->
  <div ng-include="'views/navbar.html'"  role="navigation"> </div>
```

```
  <!-- Views-->
  <div id="page-wrapper">
      <div ng-view=""></div>
  </div>
</div>
<!-- /#page-wrapper -->


<div class="footer">

</div>


</div>
<!-- /#wrapper -->


<!-- build:js(.) scripts/vendor.js -->
<!-- bower:js -->
<script src="bower_components/jquery/dist/jquery.js"></script>
<script src="bower_components/angular/angular.js"></script>
<script src="bower_components/angular-animate/angular-animate.js"></script>
<script src="bower_components/angular-bootstrap/ui-bootstrap-tpls.js"></script>
<script src="bower_components/angular-cookies/angular-cookies.js"></script>
<script src="bower_components/angular-loading-bar/build/loading-bar.js"></script>
<script src="bower_components/angular-messages/angular-messages.js"></script>
<script src="bower_components/angular-options-disabled/src/angular-options-
disabled.js"></script>
<script src="bower_components/angular-resource/angular-resource.js"></script>
<script src="bower_components/angular-route/angular-route.js"></script>
<script src="bower_components/angular-sanitize/angular-sanitize.js"></script>
<script src="bower_components/angular-touch/angular-touch.js"></script>
<script src="bower_components/bootstrap/dist/js/bootstrap.js"></script>
<script src="bower_components/checklist-model/checklist-model.js"></script>
<script src="bower_components/metisMenu/dist/metisMenu.js"></script>
<script src="bower_components/lodash/lodash.js"></script>
<script src="bower_components/restangular/dist/restangular.js"></script>
<!-- endbower -->
<!-- endbuild -->

<!-- build:js({.tmp,app}) scripts/scripts.js -->
<script src="scripts/app.js"></script>
<script src="scripts/controllers/main.js"></script>
<script src="scripts/controllers/boards.js"></script>
<script src="scripts/controllers/boards-modal.js"></script>
<script src="scripts/controllers/create-test-board.js"></script>
<script src="scripts/controllers/create-test-model.js"></script>
<script src="scripts/controllers/create-test-parameters.js"></script>
<script src="scripts/controllers/models.js"></script>
<script src="scripts/controllers/models-modal.js"></script>
<script src="scripts/controllers/results.js"></script>
<script src="scripts/controllers/results-id.js"></script>
<script src="scripts/sb-admin-2.js"></script>
<script src="scripts/utils.js"></script>
<script src="scripts/n3-charts/build/LineChart.js"></script>
<!-- endbuild -->

<!-- build:js({.tmp,app}) scripts/d3.min.js -->
<script src="scripts/d3/d3.v3.js"></script>
<!-- endbuild -->

</body>
</html>
```

Navbar.html

```
<!-- Navigation -->
<nav class="navbar navbar-default navbar-static-top" role="navigation" style="margin-
bottom: 0">
  <div class="navbar-header">
    <button type="button" class="navbar-toggle" data-toggle="collapse" data-
target=".navbar-collapse">
      <span class="sr-only">Toggle navigation</span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
    </button>
    <a class="navbar-brand" href="index.html">GNB Tests Admin</a>
  </div>
  <!-- /.navbar-header -->

  <div class="navbar-default sidebar" role="navigation">
    <div class="sidebar-nav navbar-collapse">
      <ul class="nav" id="side-menu">
        <li>
          <a href="#/dashboard" ng-class="{active: isActive('/')}"><i class="fa fa-home fa-
fw"></i> Home</a>
        </li>
        <li>
          <a href="#/boards" ng-class="{active: isActive('/boards')}"><i class="fa fa-
laptop fa-fw"></i> Devices</a>
        </li>
        <li>
          <a href="#/models" ng-class="{active: isActive('/models')}"><i class="fa fa-edit
fa-fw"></i> Models</a>
        </li>
        <li>
          <a href="#/create-test/board" ng-class="{active: isActive('/create-test')}">
            <i class="fa fa-cog fa-fw"></i> Create test</a>
        </li>
        <li>
          <a href="#/results" ng-class="{ active: isActive('/results')}">
            <i class="fa fa-paperclip fa-fw"></i> Results</a>
        </li>
      </ul>
    </div>
    <!-- /.sidebar-collapse -->
  </div>
  <!-- /.navbar-static-side -->
</nav>
```

App.js

```
'use strict';

/**
 * @ngdoc overview
 * @name appApp
 * @description
 * # appApp
 *
 * Main module of the application.
 */
angular
  .module('appApp', [
    'ngAnimate',
    'ngCookies',
    'ngResource',
    'ngRoute',
    'ngSanitize',
```

```
    'ngTouch',
    'ui.bootstrap',
    'ngMessages',
    'checklist-model',
    'ngOptionsDisabled',
    'n3-line-chart',
    'angular-loading-bar'
  ])
  .config(['cfpLoadingBarProvider', function(cfpLoadingBarProvider) {
    cfpLoadingBarProvider.spinnerTemplate = '<div> <div
class="loading">Loading&#8230;</div></div>';
  }])
  .config(function($routeProvider) {
    $routeProvider
      .when('/', {
        templateUrl: 'views/main.html',
        controller: 'MainCtrl',
        controllerAs: 'main'
      })
      .when('/boards', {
        templateUrl: 'views/boards.html',
        controller: 'BoardsCtrl',
        controllerAs: 'boards'
      })
      .when('/models', {
        templateUrl: 'views/models.html',
        controller: 'ModelsCtrl',
        controllerAs: 'models'
      })
      .when('/create-test/board', {
        templateUrl: 'views/create-test-board.html',
        controller: 'CreateTestBoardCtrl',
        controllerAs: 'createTestBoard'
      })
      .when('/create-test/model', {
        templateUrl: 'views/create-test-model.html',
        controller: 'CreateTestModelCtrl',
        controllerAs: 'createTestModel'
      })
      .when('/create-test/parameters', {
        templateUrl: 'views/create-test-parameters.html',
        controller: 'CreateTestParametersCtrl',
        controllerAs: 'createTestParameters'
      })
      .when('/results', {
        templateUrl: 'views/results.html',
        controller: 'ResultsCtrl',
        controllerAs: 'results'
      })
      .when('/results/:resultId', {
        templateUrl: 'views/results-id.html',
        controller: 'ResultsIdCtrl',
        controllerAs: 'resultsId'
      })
      .otherwise({
        redirectTo: '/'
      });
  })
  .factory('boardStored', function() {
    var savedData = {};
    function set(data) {
      savedData = data;
    }
    function get() {
      return savedData;
    }
```

```
    return {
      set: set,
      get: get
    }
  })
  .factory('modelStored', function() {
    var savedData = {};

    function set(data) {
      savedData = data;
    }

    function get() {
      return savedData;
    }

    return {
      set: set,
      get: get
    }
  })
  .run(function($rootScope, $log, $location, boardStored, modelStored) {

    $rootScope.PANEL_STYLE = ['panel-primary', 'panel-default', 'panel-success', 'panel-
info', 'panel-warning',
      'panel-danger', 'panel-green', 'panel-yellow', 'panel-red'];
    $rootScope.BOARD_URL = 'http://dev-api-catalog.pfc.com/board';
    $rootScope.MODEL_URL = 'http://dev-api-catalog.pfc.com/model';

    $rootScope.testModelStep = function(object, path) {
      boardStored.set(object);
      $location.path(path);
    };

    $rootScope.testParametersStep = function(object, path) {
      modelStored.set(object);
      $location.path(path);
    };

    /*
     Compare the path to put the class active
     */
    $rootScope.isActive = function (viewLocation) {
      //This comparison is for main root '/'
      if(viewLocation.length > 1)
        return $location.path().substr(0, viewLocation.length) === viewLocation;
      else
        return $location.path() === viewLocation;
    };

  })
  .directive("serializer", function(){
    return {
      restrict: "A",
      scope: {
        onSubmit: "&serializer"
      },
      link: function(scope, element){
        // assuming for brevity that directive is defined on <form>

        var form = element;

        form.submit(function(event){
          event.preventDefault();
          var serializedData = form.serialize();
```

```
          scope.onSubmit({data: serializedData});
        });

      }
    };
  })
  .directive('numberConverter', function() {
    return {
      priority: 1,
      restrict: 'A',
      require: 'ngModel',
      link: function(scope, element, attr, ngModel) {
        function toModel(value) {
          return "" + value; // convert to string
        }

        function toView(value) {
          return parseInt(value); // convert to number
        }

        ngModel.$formatters.push(toView);
        ngModel.$parsers.push(toModel);
      }
    };
  });
```

Utils.js

```
function chunk(arr, size) {
  var newArr = [];
  for (var i=0; i<arr.length; i+=size) {
    newArr.push(arr.slice(i, i+size));
  }
  return newArr;
}

function getResultsRequest(url, $http, $log, $scope) {
  $log.info('Request to: ' + url);
  $scope.actualPage = url;
  $http.get(url)
    .success(function (response, status, headers, config) {
      //process success scenario.
      $scope.results = response.results;
      $scope.nextPage = response.next;
      $scope.previousPage = response.previous;
      $log.info('Response received');
    }).error(function (err, status, headers, config) {
      //process error scenario.
      alert('err');
      alert(err);
      alert(status);
    });
}

// Speed up calls to hasOwnProperty
var hasOwnProperty = Object.prototype.hasOwnProperty;

function isEmpty(obj) {

  // null and undefined are "empty"
  if (obj == null) return true;

  // Assume if it has a length property with a non-zero value
```

```
  // that that property is correct.
  if (obj.length > 0)    return false;
  if (obj.length === 0)  return true;

  // Otherwise, does it have any properties of its own?
  // Note that this doesn't handle
  // toString and valueOf enumeration bugs in IE < 9
  for (var key in obj) {
    if (hasOwnProperty.call(obj, key)) return false;
  }

  return true;
}
```

Sb-admin-2.js

```
$(function() {
    $('#side-menu').metisMenu();
});

//Loads the correct sidebar on window load,
//collapses the sidebar on window resize.
// Sets the min-height of #page-wrapper to window size
$(function() {
    $(window).bind("load resize", function() {
        var topOffset = 50;
      var width = (this.window.innerWidth > 0) ? this.window.innerWidth :
this.screen.width;
        if (width < 768) {
            $('div.navbar-collapse').addClass('collapse');
            topOffset = 100; // 2-row-menu
        } else {
            $('div.navbar-collapse').removeClass('collapse');
        }

        var height = ((this.window.innerHeight > 0) ? this.window.innerHeight :
this.screen.height) - 1;
        height = height - topOffset;
        if (height < 1) {
          height = 1;
        }
        if (height > topOffset) {
            $("#page-wrapper").css("min-height", (height) + "px");
        }
    });

    var url = window.location;
    var element = $('ul.nav a').filter(function() {
        return this.href === url || url.href.indexOf(this.href) === 0;
    }).addClass('active').parent().parent().addClass('in').parent();
    if (element.is('li')) {
        element.addClass('active');
    }
});
```

Boards.html

```
<div class="row">
  <div class="col-lg-12">
    <h1 class="page-header">Boards</h1>
  </div>
  <!-- /.col-lg-12 -->
</div>
```

```
<!-- /.row -->

<div class="col-sm-4 col-xs-12" ng-repeat="board in results">
  <div class="panel {{PANEL_STYLE[$index]}}" ng-click="open(board)">
    <div class="panel-heading">
      {{board.name}}
    </div>
    <div class="panel-body">
      <p>{{board.description}}</p>
    </div>
    <div class="panel-footer">
      Show more info
    </div>
  </div>
</div>
<!-- /.col-sm-4 col-xs-12-->

<div class="row">
  <div class="col-xs-12">
    <ul total-items="totalItems" ng-model="currentPage" class="pager ng-isolate-scope ng-
valid">
      <li class="previous" ng-class="{disabled: previousPage == null, previous: align}">
        <a class="ng-binding" href="" ng-click=previousPageRequest()>
          « Previous
        </a>
      </li>
      <li class="next" ng-class="{disabled: nextPage == null, next: align}">
        <a class="ng-binding" href="" ng-click="nextPageRequest()">
          Next »
        </a>
      </li>
    </ul>
  </div>
</div>

<script type="text/ng-template" id="boardModalContent.html">
  <div class="modal-header">
    <h3 class="modal-title">Name: {{board.name}}</h3>
  </div>
  <div class="modal-body">
    <p><b>ID: </b> {{board.id}}</p>
    <p><b>Created on: </b> {{board.created_on | date:'medium'}}</p>
    <p><b>Modified on: </b> {{board.modified_on | date:'medium'}}</p>
    <p><b>Description: </b>{{board.description}}</p>
    <p><b>Number of pins: </b>{{board.number_pins}}</p>
    <p><b>URI: </b>{{board.uri}}</p>
    <p><b>Sensors:</b></p>
    <ul>
      <li ng-repeat="sensor in board.sensors">
        <p><b>Name: </b>{{sensor.name}}</p>
        <p><b>Description: </b>{{sensor.description}}</p>
        <p><b>Is used: </b>{{sensor.is_used}}</p>
        <p><b>Pin: </b>{{sensor.pin}}</p>
      </li>
    </ul>
  </div>
  <div class="modal-footer">
    <button class="btn btn-primary" type="button" ng-click="close()">Close</button>
  </div>
</script>
```

Boards.js

```
'use strict';

angular.module('appApp')
  //.controller('BoardsCtrl', function ($scope, Restangular)  {
  .controller('BoardsCtrl', function ($http, $scope, $uibModal, $log) {

    $scope.actualPage = 'http://dev-api-catalog.pfc.com/board?limit=9';
    $scope.nextPage = null;
    $scope.previousPage = null;

    getResultsRequest($scope.actualPage, $http, $log, $scope);

    $scope.nextPageRequest = function() {
      if($scope.nextPage!=null)
        getResultsRequest($scope.nextPage, $http, $log, $scope);
    }

    $scope.previousPageRequest = function() {
      if($scope.previousPage!=null)
        getResultsRequest($scope.previousPage, $http, $log, $scope);
    }

    $scope.open = function (boardObject) {
      var modalInstance = $uibModal.open({
        animation: true,
        templateUrl: 'boardModalContent.html',
        controller: 'BoardModalInstanceCtrl',
        size: 'lg',
        resolve: {
          board: function () {
            return boardObject;
          }
        }
      });

      $log.info('Modal open at: ' + new Date());

      modalInstance.result.then(function () {
      }, function () {
        $log.info('Modal dismissed at: ' + new Date());
      });
    };

  });
```

Models.html

```
<div class="row">
  <div class="col-lg-12">
    <h1 class="page-header">Models</h1>
  </div>
  <!-- /.col-lg-12 -->
</div>
<!-- /.row -->

<div class="col-sm-4 col-xs-12" ng-repeat="model in results">
  <div class="panel {{PANEL_STYLE[$index]}}" ng-click="open(model)">
    <div class="panel-heading">
      {{model.name}}
    </div>
    <div class="panel-body">
      <p>{{model.description}}</p>
    </div>
    <div class="panel-footer">
```

```
      Show more info
    </div>
  </div>
</div>
<!-- /.col-sm-4 col-xs-12-->

<div class="row">
  <div class="col-xs-12">
    <ul total-items="totalItems" ng-model="currentPage" class="pager ng-isolate-scope ng-
valid">
      <li class="previous" ng-class="{disabled: previousPage == null, previous: align}">
        <a class="ng-binding" href="" ng-click=previousPageRequest()>
          « Previous
        </a>
      </li>
      <li class="next" ng-class="{disabled: nextPage == null, next: align}">
        <a class="ng-binding" href="" ng-click="nextPageRequest()">
          Next »
        </a>
      </li>
    </ul>
  </div>
</div>

<script type="text/ng-template" id="modelModalContent.html">
  <div class="modal-header">
    <h3 class="modal-title">{{model.name}}</h3>
  </div>
  <div class="modal-body">
    <p><b>ID:</b> {{model.id}}</p>
    <p><b>Created on:</b> {{model.created_on | date:'medium'}}</p>
    <p><b>Modified on:</b> {{model.modified_on | date:'medium'}}</p>
    <p><b>Description:</b>{{model.description}}</p>
    <p><b>Parameters:</b></p>
    <ul>
      <li ng-repeat="parameter in model.parameters">
        {{parameter.name}}
        <ul>
          <li ng-repeat="related_parameter in parameter.related_parameters">
            {{related_parameter.name}}
        </ul>
      </li>
    </ul>

  </div>
  <div class="modal-footer">
    <button class="btn btn-primary" type="button" ng-click="close()">Close</button>
  </div>
</script>
```

Models.js

```
'use strict';

angular.module('appApp')
  .controller('ModelsCtrl', function ($http, $scope, $uibModal, $log) {
    $scope.panelStyle = ['panel-primary', 'panel-default', 'panel-success', 'panel-info',
'panel-warning',
      'panel-danger', 'panel-green', 'panel-yellow', 'panel-red'];

    $scope.actualPage = 'http://dev-api-catalog.pfc.com/model?limit=9';
    $scope.nextPage = null;
    $scope.previousPage = null;
```

```
    getResultsRequest($scope.actualPage, $http, $log, $scope);

    $scope.nextPageRequest = function() {
      if($scope.nextPage!=null)
        getResultsRequest($scope.nextPage, $http, $log, $scope);
    }

    $scope.previousPageRequest = function() {
      if($scope.previousPage!=null)
        getResultsRequest($scope.previousPage, $http, $log, $scope);
    }

    $scope.open = function (modelObject) {
      var modalInstance = $uibModal.open({
        animation: true,
        templateUrl: 'modelModalContent.html',
        controller: 'ModelModalInstanceCtrl',
        size: 'lg',
        resolve: {
          model: function () {
            return modelObject;
          }
        }
      });

      $log.info('Modal open at: ' + new Date());

      modalInstance.result.then(function () {
      }, function () {
        $log.info('Modal dismissed at: ' + new Date());
      });
    };

  });
```

Create-test-board.html

```
<div class="row">
  <div class="col-lg-12">
    <h1 class="page-header">Choose a device</h1>
  </div>
  <!-- /.col-lg-12 -->
</div>
<!-- /.row -->

<div class="row">
  <div class="col-sm-4 col-xs-12 gallery" ng-repeat="board in results">
    <div class="panel {{PANEL_STYLE[$index]}}" ng-click="testModelStep(board, '/create-
test/model')">
      <div class="panel-heading">
        {{board.name}}
      </div>
      <div class="panel-body">
        {{board.description}}
      </div>
      <div class="panel-footer">
        More info
      </div>
    </div>
  </div>
  <!-- /.col-sm-4 col-xs-12-->
</div>
<!-- /.row -->
```

```
<div class="row">
  <div class="col-xs-12">
    <ul total-items="totalItems" ng-model="currentPage" class="pager ng-isolate-scope ng-
valid">
      <li class="previous" ng-class="{disabled: previousPage == null, previous: align}">
        <a class="ng-binding" href="" ng-click=newPageRequest(previousPage)>
          « Previous
        </a>
      </li>
      <li class="next" ng-class="{disabled: nextPage == null, next: align}">
        <a class="ng-binding" href="" ng-click=newPageRequest(nextPage)>
          Next »
        </a>
      </li>
    </ul>
  </div>
</div>
```

Create-test-board.js

```
'use strict';

angular.module('appApp')
  .controller('CreateTestBoardCtrl', function ($http, $scope, $uibModal, $log, $rootScope)
{

    $scope.actualPage = $rootScope.BOARD_URL+'?limit=9';
    $scope.nextPage = null;
    $scope.previousPage = null;

    console.log('peticion');
    getResultsRequest($scope.actualPage, $http, $log, $scope);

    $scope.newPageRequest = function(page) {
      console.log('new');
      if(page!=null)
        getResultsRequest(page, $http, $log, $scope);
    }
  });
```

Create-test-model.html

```
<div class="row">
  <div class="col-lg-12">
    <h1 class="page-header">Select a model</h1>
  </div>
  <!-- /.col-lg-12 -->
</div>
<!-- /.row -->

<div class="row">
  <div class="col-sm-4 col-xs-12 gallery" ng-repeat="model in results">
    <div class="panel {{PANEL_STYLE[$index]}}" ng-click="testParametersStep(model,
'/create-test/parameters')">
      <div class="panel-heading">
        {{model.name}}
      </div>
      <div class="panel-body">
        {{model.description}}
      </div>
      <div class="panel-footer">
        More info
      </div>
```

```
    </div>
  </div>
  <!-- /.col-sm-4 col-xs-12-->
</div>
<!-- /.row -->

<div class="row">
  <div class="col-xs-12">
    <ul total-items="totalItems" ng-model="currentPage" class="pager ng-isolate-scope ng-
valid">
      <li class="previous" ng-class="{disabled: previousPage == null, previous: align}">
        <a class="ng-binding" href="" ng-click=newPageRequest(previousPage)>
          « Previous
        </a>
      </li>
      <li class="next" ng-class="{disabled: nextPage == null, next: align}">
        <a class="ng-binding" href="" ng-click=newPageRequest(nextPage)>
          Next »
        </a>
      </li>
    </ul>
  </div>
</div>
```

Create-test-model.js

```
'use strict';

angular.module('appApp')
  .controller('CreateTestModelCtrl', function ($http, $scope, $uibModal, $log, $rootScope,
$location, boardStored) {

    var board = boardStored.get();
    $log.info('Board input');$log.info(board);

    if (isEmpty(board)) {
      //FIXME maybe show alert
      $log.info('Board empty. Redirect to /create-test/board');
      $location.path('/create-test/board');
    }

    //TODO make request with the board id to show only valid models in this board
    $scope.actualPage = $rootScope.MODEL_URL+'?limit=9';
    $scope.nextPage = null;
    $scope.previousPage = null;

    getResultsRequest($scope.actualPage, $http, $log, $scope);

    $scope.newPageRequest = function(page) {
      if(page!=null)
        getResultsRequest(page, $http, $log, $scope);
    }
  });
```

Create-test-parameters.html

```
<div class="row">
  <div class="col-lg-12">
    <h1 class="page-header">Set the model's parameters</h1>
  </div>
  <!-- /.col-lg-12 -->
</div>
<!-- /.row -->
<form ng-submit="submitTable()" name="myForm">
```

```
  <input type="hidden"
         ng-init="formData['model']=model.id"
         ng-model="formData['model']">
  <input type="hidden"
         ng-init="formData['board']=board.id"
         ng-model="formData['board']">
  <div class="row">
    <input type="hidden"
           ng-init="formData['parameters']=[]"
           ng-model="formData['parameters']">
    <div class="col-xs-12" ng-repeat="parameter in model.parameters">
      <div class="well">
        <div class="row">
          <div class="col-sm-4">
            <div class="form-group">
              <label>{{parameter.name}}</label>
              <input type="hidden"
                     ng-
init="formData['parameters'][$index]['model_parameter']=parameter.id"
                     ng-model="formData['parameters'][$index]['model_parameter']">
              <div ng-if="parameter.type=='sensor'">
                <select class="form-control" ng-
model="formData['parameters'][$index]['value']" name={{parameter.name}} required>
                  <option ng-repeat="sensor in board.sensors"
value="{{sensor.id}}">{{sensor.name}}</option>
                </select>
              </div>
              <div ng-if="parameter.type=='integer' || parameter.type=='float'">
                <input name={{parameter.name}} type="number" class="input-small"
                       value={{parameter.min_value}}  step="0.000001"
                       ng-
init="formData['parameters'][$index]['value']=parameter.min_value"
                       number-converter
                       ng-min={{parameter.min_value}} ng-max={{parameter.max_value}}
                       ng-model="formData['parameters'][$index]['value']" required>
              </div>
              <div ng-if="parameter.type=='string'">
                <input type="text" class="input-small"
                       name={{parameter.name}}
                       value={{parameter.min_value}}
                       ng-
init="formData['parameters'][$index]['value']=parameter.min_value"
                       min={{parameter.min_value}} max={{parameter.max_value}}
                       ng-model="formData['parameters'][$index]['value']" required>
              </div>
              <div ng-messages="myForm[parameter.name].$error">
                <div ng-message="required"><p class="text-danger">This field is
required</p></div>
                <div ng-message="min"><p class="text-danger">The minimum value is
{{parameter.min_value}}</p></div>
                <div ng-message="max"><p class="text-danger">The maximum value is
{{parameter.min_value}}</p></div>
                <div ng-message="minlength"><p class="text-danger">{{parameter.name}} must
be over {{parameter.min_value}} characters</p></div>
                <div ng-message="maxlength"><p class="text-danger">{{parameter.name}} must
not exceed {{parameter.max_value}} characters</p></div>
              </div>
            </div>
          </div>
          <div class="col-sm-8">
            <input type="hidden"
                   ng-init="formData['parameters'][$index]['related_parameters']=[]"
                   ng-model="formData['parameters'][$index]['related_parameters']">
            <div ng-repeat="related_parameter in parameter.related_parameters">
              <input type="hidden"
                     ng-
```

```
init="formData['parameters'][$parent.$index]['related_parameters'][$index]['model_parameter
']
                          =related_parameter.id"
                    ng-
model="formData['parameters'][$parent.$index]['related_parameters'][$index]['model_paramete
r']">
              <div class="controls form-inline">
                <label>{{related_parameter.name}}</label>
                <div ng-if="related_parameter.type=='integer' ||
related_parameter.type=='float'">
                  <input name={{related_parameter.name}} type="number" class="input-small"
step="0.000001"
                         value={{related_parameter.min_value}}
                         ng-min="{{related_parameter.min_value}}" ng-
max="{{related_parameter.max_value}}"
                         number-converter
                         ng-
init="formData['parameters'][$parent.$parent.$index]['related_parameters'][$parent.$index][
'value']=related_parameter.min_value"
                         ng-
model="formData['parameters'][$parent.$parent.$index]['related_parameters'][$parent.$index]
['value']"
                         required>
                </div>

                <div ng-if="related_parameter.type=='string'">
                  <input name={{related_parameter.name}} type="text" class="input-small"
value={{related_parameter.min_value}}
                         ng-minlength={{related_parameter.min_value}} ng-
maxlength={{related_parameter.max_value}}
                         ng-
init="formData['parameters'][$parent.$parent.$index]['related_parameters'][$parent.$index][
'value']=related_parameter.min_value"
                         ng-
model="formData['parameters'][$parent.$parent.$index]['related_parameters'][$parent.$index]
['value']"
                         required>
                </div>
                <div ng-messages="myForm[related_parameter.name].$error">
                  <div ng-message="required"><p class="text-danger">This field is
required</p></div>
                  <div ng-message="minlength"><p class="text-
danger">{{related_parameter.name}} must be over {{related_parameter.min_value}}
characters</p></div>
                  <div ng-message="maxlength"><p class="text-
danger">{{related_parameter.name}} must not exceed 1000 characters</p></div>
                  <div ng-message="min"><p class="text-danger">The minimum value is
{{related_parameter.min_value}}</p></div>
                  <div ng-message="max"><p class="text-danger">The maximum value is
{{related_parameter.max_value}}</p></div>
                </div>
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
    <!-- /.col-sm-4 col-xs-12-->
  </div>
  <!-- /.row -->
  <div class="row">
    <div class="col-xs-12">
      <button ng-disabled="!myForm.$valid" class="btn btn-success"
type='submit'>Submit</button>
    </div>
```

```
  </div>
</form>
```

Create-test-parameters.js

```
'use strict';

angular.module('appApp')
  .controller('CreateTestParametersCtrl', function ($http, $scope, $uibModal, $log,
$rootScope, $location, $window,
                                                    boardStored, modelStored) {

    $scope.board = boardStored.get();
    $log.info('Board input');$log.info($scope.board);

    $scope.model = modelStored.get();
    $log.info('Model input');$log.info($scope.model);

    if (isEmpty($scope.board) || isEmpty($scope.model)) {
      //FIXME maybe show alert
      $log.info('Board empty. Redirect to /create-test/board');
      $location.path('/create-test/board');
    }

    $scope.formData = {};

    $scope.submitTable = function () {
      if($scope.formData.$valid){
        $window.alert("Invalid form");
        return;
      }
      $log.info(JSON.stringify($scope.formData));
      $http.post("http://dev-api-catalog.pfc.com/test/", $scope.formData)
        .success(function (response, status, headers, config) {
          $log.info(response);
          $log.info('Response received');
          var path = '/results/' + (response.id).toString();
          $location.path(path);
        }).error(function (err, status, headers, config) {
          $window.alert(JSON.stringify(err));
      });
    };

  });
```

Results.html

```
<div class="row">
  <div class="col-lg-12">
    <h1 class="page-header">Results</h1>
  </div>
</div>

<div class="row" id="table">
  <div class="col-xs-12 table-responsive">
    <div class="">
      <table class="table table-striped table-hover">
        <thead>
        <tr>
          <th>ID</th>
          <th>Created on</th>
          <th>Board Id</th>
          <th>Model name</th>
          <th>PID</th>
```

```
          <th>Progress</th>
          <th>Status</th>
          <th>Finished on</th>
          <th>Show results</th>
        </tr>
        </thead>
        <tbody>
        <tr ng-repeat-start="row in results" ng-click="row.expanded = !row.expanded"
class="clickable">
          <td>{{row.id}}</td>
          <td>{{row.created_on | date:'yyyy-MM-dd HH:mm:ss Z'}}</td>
          <td style="text-align: center;">{{row.board}}</td>
          <td>{{row.model.name}}</td>
          <td>{{row.pid}}</td>
          <td>{{row.progress | number:1}}</td>
          <td>{{row.status}}</td>
          <td>{{row.finished_on | date:'yyyy-MM-dd HH:mm:ss Z'}}</td>
          <td>
            <a class="btn" style="color: #5CB85C;" ng-click="setSelected(row)">
              <i class="fa fa-plus-circle fa-2x"></i>
            </a>
          </td>
        </tr>

        <tr ng-if="row.expanded" ng-repeat-end="">
          <td colspan="12">
            <div class="row">
              <label>Parameters</label>
              <div class="col-xs-12" ng-repeat="parameter in row.parameters">
                <div class="well">
                  <div class="row">
                    <div class="col-xs-4">
                      <p><b>{{parameter.model_parameter.name}}</b>: {{parameter.value}}</p>
                    </div>
                    <div class="col-xs-8">
                      <div ng-repeat="related_parameter in parameter.related_parameters">
                        <p><b>{{related_parameter.model_parameter.name}}</b>:
{{related_parameter.value}}</p>
                      </div>
                    </div>
                  </div>
                </div>
              </div>
            </div>
          </td>
        </tr>

        </tbody>
      </table>
    </div>
  </div>
</div>

<div class="row">
  <div class="col-xs-12">
    <ul total-items="totalItems" ng-model="currentPage" class="pager ng-isolate-scope ng-
valid">
      <li class="previous" ng-class="{disabled: previousPage == null, previous: align}">
        <a class="ng-binding" href="" ng-click=previousPageRequest()>
          « Previous
        </a>
      </li>
      <li class="next" ng-class="{disabled: nextPage == null, next: align}">
        <a class="ng-binding" href="" ng-click="nextPageRequest()">
          Next »
        </a>
```

```
      </li>
    </ul>
  </div>
</div>
```

Results.js

```
'use strict';

angular.module('appApp')
  .controller('ResultsCtrl', function ($http, $scope, $log, $location, $anchorScroll) {

    $scope.actualPage = 'http://dev-api-catalog.pfc.com/test?limit=25';
    $scope.nextPage = null;
    $scope.previousPage = null;

    getResultsRequest($scope.actualPage, $http, $log, $scope);

    $scope.nextPageRequest = function() {
      if($scope.nextPage!=null) {
        getResultsRequest($scope.nextPage, $http, $log, $scope);
        goToTopTable();
      }
    };

    $scope.previousPageRequest = function() {
      if($scope.previousPage!=null) {
        getResultsRequest($scope.previousPage, $http, $log, $scope);
        goToTopTable();
      }
    };

    var goToTopTable = function(){
      $location.hash('table');
      $anchorScroll();
    }

    $scope.setSelected = function(obj) {
      $location.path('/results/'+obj.id);
    };
  });
```

Results-id.html

```
<div class="row">
  <div class="col-xs-12">
    <div class="row page-header">
      <div class="col-xs-9">
        <h1>Test: {{result.id}}</h1>
      </div>
      <div class="col-xs-3">
        <a target="_self" ng-href="{{csvLink}}">
          <button class="btn btn-primary">DOWNLOAD Results</button>
        </a>
      </div>
    </div>
  </div>
</div>
<!-- /.row -->

<div class="row">
  <div class="col-xs-12">
    <div class="row">
      <div class="col-xs-10">
```

```
        <h2>Status: {{result.status}}</h2>
      </div>
      <div class="col-xs-2">
        <button class="btn btn-danger" ng-disabled="result.status !== 'ready' &&
result.status !== 'running' "
                ng-click="stopTest();">STOP test</button>
      </div>
    </div>
    <div class="progress progress-striped active">
      <div class="progress-bar progress-bar-success" role="progressbar" aria-
valuenow="{{result.progress}}"
           aria-valuemin="0" aria-valuemax="100" style="width: {{result.progress}}%">
        <span class="sr-only">40% Complete (success)</span>
      </div>
    </div>
  </div>
</div>

<div class="row">
  <div class="col-xs-12">
    <h2>Model:</h2>
  </div>
</div>

<div class="row">
  <div class="col-xs-12">
    <div class="well">
      <h5>{{result.model.name}}</h5>
      <h6>{{result.model.description}}</h6>
    </div>
  </div>
</div>

<div class="row">
  <div class="col-xs-12">
    <h2>Parameters:</h2>
  </div>
</div>

<div class="row">
  <div class="col-xs-12" ng-repeat="parameter in result.parameters">
    <div class="well">
      <div class="row">
        <div class="col-xs-4">
          <p><b>{{parameter.model_parameter.name}}</b>: {{parameter.value}}</p>
        </div>
        <div class="col-xs-8">
          <div ng-repeat="related_parameter in parameter.related_parameters">
            <p><b>{{related_parameter.model_parameter.name}}</b>:
{{related_parameter.value}}</p>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>

<div class="row">
  <div class="col-xs-12">
    <h2>Graph:</h2>
  </div>
</div>

<div class="row">
  <div class="col-xs-12">
    <div class="well">
```

```
      <div class="row">
        <div class="col-xs-6">
          <form ng-submit="requestSensorValues()" name="samplesForm">
            <div class="form-group">
              <label for="samplesNumber">Samples number:</label>
              <input type="number" min="1" ng-model="samplesNumber" class="form-control"
id="samplesNumber" name="samplesNumber" required>
              <div ng-messages="samplesForm.samplesNumber.$error" role="alert">
                <div ng-message="required" style="color: #D82424;">This field is
required</div>
                <div ng-message="min" style="color: #D82424;">This value is not
correct</div>
              </div>
            </div>
            <button ng-disabled="!samplesForm.$valid" class="btn btn-success"
type='submit'>Show axis options</button>
          </form>
        </div>
      </div>
      <br>
      <div ng-show="showGraphParameters" class="row">
        <div class="col-xs-6">
          <label>Choose X axis</label>
          <span ng-repeat="option in xAxis.availableOptions" class="block">
            <input type="checkbox" ng-model="option.checked" ng-
click="updateSelection($index, xAxis.availableOptions)" />
            {{option.name}}
          </span>
        </div>
        <div class="col-xs-6">
          <label>Choose Y axis</label>
          <span ng-repeat="option in yAxis.availableOptions" class="block">
            <input type="checkbox" checklist-model="yAxis.selectedOption" checklist-
value="option"> {{option.name}}
          </span>
        </div>
      </div>

    </div>
  </div>
</div>

<div class="row" >
  <div class="col-xs-12">
    <div class="chart">
      <linechart data="data" options="options"></linechart>
    </div>
    <br>
    <br>
  </div>
</div>

<div class="row">
  <div class="col-xs-6">
    <button class="btn btn-primary" ng-click="prevPagesRequest()" ng-
disabled="prevDisabled">PREVIOUS</button>
  </div>
  <div class="col-xs-6">
    <button class="btn btn-primary pull-right" ng-click="nextPagesRequest()" ng-
disabled="nextDisabled">NEXT</button>
  </div>
  <br>
  <br>
</div>
```

Results-id.js

```
'use strict';

angular.module('appApp')
  .controller('ResultsIdCtrl', function ($http, $scope, $uibModal, $log, $routeParams,
$window, $location, $anchorScroll, $interval, $rootScope) {

    var invalidTest = function(){
      $window.alert('Invalid testId');
      $location.path('/results');
    };

    if(!(Number($routeParams.resultId) % 1 === 0)){
      invalidTest();
      return;
    }

    $scope.actualPage = 'http://dev-api-catalog.pfc.com/test/'+$routeParams.resultId;
    $scope.sensorResponse = new Map();
    $scope.samplesNumber = 10; // default samples number
    getTest();
    $scope.showGraphParameters = false;
    $scope.urlArray = [];
    $scope.nextUrlArray = [];
    $scope.prevUrlArray = [];

    /**
     * Function to send the stop signal
     */
    $scope.stopTest = function () {
      $http.get('http://dev-api-catalog.pfc.com/test/'+$routeParams.resultId + '/stop/')
        .success(function (response, status, headers, config) {
          getTest();
          $window.alert('Test stopped');
        })
        .error(function (err, status, headers, config) {
          $window.alert('Error stopping the test: ' + err);
        });
    };

    /**
     * Function to get the CSV file
     */
    $scope.csvLink = 'http://dev-api-
results.pfc.com/api/downloadCSV/'+$routeParams.resultId;

    /**
     * reset axis function
     */
    function resetAxis() {
      $scope.xAxis = {
        availableOptions: [],
        selectedOption: {} //This sets the default value of the select in the ui
      };
      $scope.yAxis = {
        availableOptions: [],
        selectedOption: [] //This sets the default value of the select in the ui
      };
    }

    /**
     * Get the test by id
     */
    function getTest(header) {
```

```
      $http.get($scope.actualPage, header)
        .success(function (response, status, headers, config) {
          $scope.result = response;
          // if header is not present is the first request
          if(header === undefined) {
            $scope.numSensorRequest = 0;
          }
          response.parameters.forEach(function(element, index, array){
            if(element.model_parameter.type === "sensor") {
              if(header === undefined) {
                $scope.numSensorRequest += 1;
                $scope.numInitialSensorRequest = $scope.numSensorRequest;
              }
            }
          });
        })
        .error(function (err, status, headers, config) {
          invalidTest();
        });
    }

    /**
     * Request all the sensor values
     */
    $scope.requestSensorValues = function () {
      $scope.numSensorRequest = $scope.numInitialSensorRequest;
      resetAxis();

      $scope.xAxis.availableOptions.push({id: 'time', sensorId: null, name: "time",
selectable: false});
      $scope.yAxis.availableOptions.push({id: 'time', sensorId: null, name: "time",
selectable: false});

      var response = $scope.result;
      $scope.urlArray = [];
      $scope.nextUrlArray = [];
      $scope.prevUrlArray = [];

      response.parameters.forEach(function(element, index, array){
        if(element.model_parameter.type === "sensor") {
          var input = element.model_parameter.name + " input";
          var output = element.model_parameter.name + " output";
          $scope.xAxis.availableOptions.push({id: 'input', sensorId: element.value, name:
input, selectable: false});
          $scope.yAxis.availableOptions.push({id: 'input', sensorId: element.value, name:
input, selectable: false});

          $scope.xAxis.availableOptions.push({id: 'output', sensorId: element.value, name:
output, selectable: false});
          $scope.yAxis.availableOptions.push({id: 'output', sensorId: element.value, name:
output, selectable: false});

          var url = "http://dev-api-results.pfc.com/api/results/" + response.id +
"/sensor/" + element.value +
            '?size=' + $scope.samplesNumber;

          $scope.urlArray.push(url);
        }
      });

      sendRequest($scope.urlArray);
    };

    $scope.nextDisabled = true;
    $scope.prevDisabled = true;
```

```
    function sendRequest(urlArray) {
      urlArray.forEach(function(element, index, array) {
        $http.get(element)
          .then(function successCallback(response) {
              // console.log(response);
              $scope.sensorResponse.set(response.data.results[0].sensorId,
response.data.results);
              $scope.nextUrlArray.push(response.data.nextPage);
              $scope.prevUrlArray.push(response.data.previousPage);
              if(response.data.previousPage === null) {
                $scope.prevDisabled = true;
              } else {
                $scope.prevDisabled = false;
              }
              if(response.data.nextPage === null) {
                $scope.nextDisabled = true;
              } else {
                $scope.nextDisabled = false;
              }

              $scope.numSensorRequest -= 1;
            },
            function errorCallback(response) {

            });
      });
      $scope.urlArray = urlArray;
    }

    var isNextOrPrev = false;
    $scope.nextPagesRequest = function() {
      $scope.numSensorRequest = $scope.numInitialSensorRequest;
      sendRequest($scope.nextUrlArray);
      $scope.nextUrlArray = [];
      $scope.prevUrlArray = [];
      isNextOrPrev = true;
    };

    $scope.prevPagesRequest = function() {
      $scope.numSensorRequest = $scope.numInitialSensorRequest;
      sendRequest($scope.prevUrlArray);
      $scope.nextUrlArray = [];
      $scope.prevUrlArray = [];
      isNextOrPrev = true;
    };

    /**
     * Show info when all requests have been completed
     */
    $scope.$watch(
      'numSensorRequest',
      function (newValue, oldValue) {
        if(newValue === 0) {
          $scope.showGraphParameters = true;
          if(isNextOrPrev){
            createGraphDataSet();
            isNextOrPrev = false;
          }
        } else {
          if(!isRealTime)
            $scope.showGraphParameters = false;
        }
      });

    $scope.updateSelection = function(position, entities) {
      angular.forEach(entities, function(subscription, index) {
```

```
    if (position != index)
      subscription.checked = false;
    else
      $scope.xAxis.selectedOption = entities[position];
  });
};

$scope.$watchCollection('xAxis.selectedOption', function(newValue, oldValue) {
  if(newValue !== undefined) {
    createGraphDataSet();
  } else {
    resetGraph();
  }
});

$scope.$watchCollection('yAxis.selectedOption', function(newValue, oldValue) {
  if(newValue !== undefined && newValue.length > 0) {
    createGraphDataSet();
  } else {
    resetGraph();
  }
});

var isRealTime = false;
$scope.pauseTest = false;
$scope.timeRequestGraph = 4000;

 $scope.stop = $interval(function() {
  if(($scope.result.status === "running" || $scope.result.status === "ready" ||
    $scope.result.status === "waiting" ) && !$scope.pauseTest) {
    if (!isNextOrPrev) {
      getTest({ignoreLoadingBar: true});

      $scope.numSensorRequest = $scope.numInitialSensorRequest;
      sendRequest($scope.urlArray);
      $scope.nextUrlArray = [];
      $scope.prevUrlArray = [];
      isNextOrPrev = true;
      isRealTime = true;
    }
  }
}, $scope.timeRequestGraph);

var dereg = $rootScope.$on('$locationChangeSuccess', function() {
  $interval.cancel($scope.stop);
  dereg();
});

/**
 * Reset the graph
 */
function resetGraph() {
  $scope.data = {
    dataset0: []
  };

  $scope.options = {
    margin: {
      top: 20,
      right: 30,
      bottom: 20,
      left: 10
    },
    series: [],
    axes: {x: {key: "x"}}
  };
```

```
    }

    var colors= ['aqua', 'black', 'blue', 'fuchsia', 'gray', 'green',
      'lime', 'maroon', 'navy', 'olive', 'orange', 'purple', 'red',
      'silver', 'teal', 'yellow'];
    /**
     * Create graph
     */
    function createGraphDataSet() {
      if(Object.keys($scope.xAxis.selectedOption).length === 0 ||
$scope.yAxis.selectedOption.length === 0) {
        return;
      }
      var xType = 'linear';
      var dataset = [];
      var x = [];
      var xAxisSensorId = $scope.xAxis.selectedOption.sensorId;
      var xAxisSelectedType = $scope.xAxis.selectedOption.id;

      if(xAxisSensorId === null && xAxisSelectedType === 'time') {
        var key = '';
        $scope.sensorResponse.forEach(function(value, key2) {
          key = key2;
        }, $scope.sensorResponse);
        xType = 'date';
        $scope.sensorResponse.get(key.toString()).forEach(function(element, index, array) {
          x.push(new Date(element.testTime));
        });
      } else if (xAxisSensorId !== null) {
        if(xAxisSelectedType === 'input') {
          var data = $scope.sensorResponse.get(xAxisSensorId.toString());
          data.forEach(function(element, index, array) {
            x.push(element.inputValue);
          });
        } else if(xAxisSelectedType === 'output'){
          var data = $scope.sensorResponse.get(xAxisSensorId.toString());
          data.forEach(function(element, index, array) {
            x.push(element.outputValue);
          });
        }
      }

      var y = [];
      var series = [];
      $scope.yAxis.selectedOption.forEach(function(element, index, array) {
        console.log(element)
        var yAxisSensorId = element.sensorId;
        var yAxisSelectedType = element.id;
        var yArray = {};

        if(yAxisSensorId === null && yAxisSelectedType === 'time') {
          var key = '';
          $scope.sensorResponse.forEach(function(value, key2) {
            key = key2;
          }, $scope.sensorResponse);
          $scope.sensorResponse.get(key.toString()).forEach(function(element, index, array)
{
            yArray[index] = element.testTime;
          });
        } else if (yAxisSensorId !== null) {
          console.log("yAxisSensorId " + yAxisSensorId)

          if(yAxisSelectedType === 'input') {
            var data = $scope.sensorResponse.get(yAxisSensorId.toString());
            data.forEach(function(element, index, array) {
              yArray[index] = element.inputValue;
```

```
          });
        } else if(yAxisSelectedType === 'output'){
          var data = $scope.sensorResponse.get(yAxisSensorId.toString());
          data.forEach(function(element, index, array) {
            yArray[index] = element.outputValue;
          });
        }
      }

      var serieKey = 'val_' + index.toString();
      var serieId = 'serie_' + index.toString();
      series.push({
        axis: "y",
        dataset: "dataset0",
        key: serieKey,
        label: element.name,
        color: colors[index%colors.length],
        type: ['line', 'dot', 'area'],
        id: serieId
      });
      y.push(yArray);
    });
    for(var i = 0; i < x.length; i++) {
      var object = {};

      object['x'] = x[i];

      for(var j = 0; j < y.length; j++) {
        var key = 'val_'+ j.toString();
        var value = y[j][i];

        object[key] = value;
      }

      dataset.push(object);
    }
    $scope.data = {
      dataset0: dataset
    };
    $scope.options = {
      margin: {
        top: 20,
        right: 30,
        bottom: 20,
        left: 10
      },
      zoom: {
        x:true,
        y:true
      },
      doubleClickEnabled: true,
      series: series,
      axes: {
        x: {
          type: xType,
          key: "x",
          includeZero: true,
          tickFormat: function(value, index) {
              return value;
            }
        }
      }
    };
  }
});
```

Anexo F. Códigos de las pruebas

Model-gas-sensor-regression-lm35.py

```
#!/usr/bin/python

import spidev
import sys
import os
import time
import random
import requests
import json
from threading import Thread
import RPi.GPIO as GPIO
from scipy import stats


def current_milli_time():
    return int(round(time.time() * 1000))


def send_value(_info):
    """
    Function to send POST info to create a new entry in the api-results service
    :param _info:
    """
    api_results_uri = "http://dev-api-results.pfc.com/api/save/results"
    try:
        r = requests.post(api_results_uri, json=_info)

        if r.status_code != 201:
            sys.stderr.write("send_value: Fail to create result entry: \n" +
json.dumps(_info, sort_keys=True,

indent=4) + "\n")
            error_msg_1 = "ERROR: send_value: Fail to create result entry: \n" +
json.dumps(_info, sort_keys=True,

indent=4)
            Thread(target=update_test_info, args=(test_id, {"status":
error_msg_1},)).start()
            GPIO.cleanup()
            os._exit(os.EX_DATAERR)

    except requests.exceptions.RequestException as e:
        sys.stderr.write("send_value: " + "\n")
        error_msg_2 = "ERROR: send_value: " + str(e)
        Thread(target=update_test_info, args=(test_id, {"status": error_msg_2},)).start()
        GPIO.cleanup()
        os._exit(os.EX_UNAVAILABLE)


def update_test_info(_test_id, _info):
    """
    Function to send PATCH info to update the Test in the api-catalog service
    :param _test_id:
    :param _info:
    """
    api_catalog_uri = "http://dev-api-catalog.pfc.com/test/"+_test_id+'/'
    headers = {'Content-Type': 'application/json'}

    try:
```

```
        r = requests.patch(api_catalog_uri, json=_info, headers=headers)

        if r.status_code != 200:
            sys.stderr.write("update_test_info: Fail to update test entry: \n" +
json.dumps(_info, sort_keys=True,

indent=4) + "\n")
            error_msg_1 = "ERROR: update_test_info: Fail to update test entry: \n" +
json.dumps(_info, sort_keys=True,

indent=4)
            Thread(target=update_test_info, args=(test_id, {"status":
error_msg_1},)).start()
            GPIO.cleanup()
            os._exit(os.EX_DATAERR)

    except requests.exceptions.RequestException as e:
        sys.stderr.write("update_test_info: " + "\n")
        error_msg_2 = "ERROR: update_test_info: " + str(e)
        Thread(target=update_test_info, args=(test_id, {"status": error_msg_2},)).start()
        GPIO.cleanup()
        os._exit(os.EX_UNAVAILABLE)


def read_channel(channel):
    """
    Function to read SPI data from MCP3008 chip
    :param channel: must be an integer 0-7
    :return: chanel value
    """

    _adc = spi.xfer2([1, (8 + channel) << 4, 0])
    data = ((_adc[1] & 3) << 8) + _adc[2]
    return data


def convert_volts(data, places=2):
    """
    Function to convert data to voltage level, rounded to specified number of decimal
places.
    :param data: channel data
    :param places: places to round
    :return: volts
    """

    volts = (data * 5.0) / float(1023)
    volts = round(volts, places)
    return volts


def convert_temp(data, places):
    """
    Function to calculate temperature from
    LM335 data, rounded to specified
    number of decimal places.
    The sensor increase 10 mV/C
    :param data:
    :param places:
    :return: temperature in Kelvin
    """

    temperature = ((data * 500) / float(1023))
    temperature = round(temperature, places)
    return temperature

def read_value_tgs2620(channel):
```

```
    """
    Read the sensor NM times and calculate the mean
    :param channel: MCP3008 channel corresponding to the sensor
    :return:
    """

    value = 0
    residue = 0
    read_channel(channel)
    for i in range(nm):
        value += read_channel(channel)
        r = random.uniform(0, tsub)
        time.sleep(r)
        residue += (tsub-r)

    time.sleep(residue)
    return value/nm


def ini_regression_tgs2620(sample_count, temperature_tgs2620, channel):
    """
    Function to stabilize the sensor
    :param sample_count: Number of sample
    :param temperature_tgs2620: sensor temperature to apply
    :param channel: Sensor pin output
    :return:
    """

    random.seed()  # Initialize the basic random number generator
    nose_sensor.start(temperature_tgs2620)  # Start the PMV duty cycle

    value_tgs2620 = read_value_tgs2620(channel)

    # Store the data to use in the regression
    x.append(sample_count)
    concent_tgs2620.append(value_tgs2620)
    temp_tgs2620.append(temperature_tgs2620)

    # Calculating the internal resistance
    rs_tgs_2620 = ((Vc*Rl_2620) / (value_tgs2620/1000.)) - Rl_2620

    print 'Muestra Regresion_ini['+str(sample_count) + ']\n>> Valor: ' +
str(convert_volts(value_tgs2620, 2)) + \
          'V   >> Rs: ' + str(rs_tgs_2620) + '     >> Temperatura: ' +
str(temperature_tgs2620) + '\n'

    return convert_volts(value_tgs2620,5), temperature_tgs2620


def regression_tgs2620(sample_count, temperature_tgs2620, channel):

    value_tgs2620 = read_value_tgs2620(channel)

    # Heating temperature sensor adaption using linear regression
    slope, intercept, r_value, p_value, std_err1 = \
        stats.linregress(x[(sample_count - samples_init):(sample_count - 1)],
                         concent_tgs2620[(sample_count - samples_init):(sample_count - 1)])

    print("slope: " + str(slope) + " | intercept: " + str(intercept) + " |r_value: " +
str(r_value) + " |p_value: " +
          str(p_value) + " |std_err1: " + str(std_err1))

    new_temperature_tgs2620 = temperature_tgs2620 - (slope * tendency)

    if new_temperature_tgs2620 < 10.0:
        new_temperature_tgs2620 = 10.0
```

```
    elif new_temperature_tgs2620 > 90.0:
        new_temperature_tgs2620 = 90.0

    # Reset PWM value
    nose_sensor.ChangeDutyCycle(new_temperature_tgs2620)

    x.append(sample_count)
    concent_tgs2620.append(value_tgs2620)
    temp_tgs2620.append(new_temperature_tgs2620)

    # Calculating the internal resistance
    rs_tgs_2620 = ((Vc*Rl_2620) / (value_tgs2620/1000.)) - Rl_2620

    print 'Muestra Regresion_ini['+str(sample_count) + ']\n>> Valor: ' +
str(convert_volts(value_tgs2620, 2)) + \
          'V   >> Rs: ' + str(rs_tgs_2620) + '     >> Temperatura: ' +
str(new_temperature_tgs2620) + '\n'


    return convert_volts(value_tgs2620,5), new_temperature_tgs2620, slope, intercept,
r_value, p_value, std_err1


x = []  # Array to store number of sample
concent_tgs2620 = []  # Array to store concentration samples
temp_tgs2620 = []  # Array to stor the heating sensor

# Sensor TGS2620.
Rl_2620 = 1000  # Omh
Vc = 5  # V


if __name__ == '__main__':
    GPIO.setmode(GPIO.BCM)

    # Parameters from input
    board_id = str(sys.argv[1])
    test_id = str(sys.argv[2])
    # TGS info
    gas_sensor_id = int(sys.argv[3])
    gas_sensor_pin_input = int(sys.argv[4])
    gas_sensor_pin_output = int(sys.argv[5])
    if gas_sensor_pin_output >= 100:  # Define sensor channels. Greater than 100 indicates
that is a adc channel
        gas_sensor_pin_output -= 100
    # LM35 info
    temp_sensor_id = int(sys.argv[6])
    temp_sensor_pin_input = int(sys.argv[7])
    temp_sensor_pin_output = int(sys.argv[8])
    if temp_sensor_pin_output >= 100:  # Define sensor channels. Greater than 100 indicates
that is a adc channel
        temp_sensor_pin_output -= 100
    # global variables
    nm = int(sys.argv[9])
    t = float(sys.argv[10])
    tsub = t/nm
    sleep = float(sys.argv[11])
    samples_init = int(sys.argv[12])
    tendency = float(sys.argv[13])
    average_temperature = float(sys.argv[14])
    duration = float(sys.argv[15])
    # Ids to store the regression info
    regression_slope_intercept_id = int(sys.argv[16])
    regression_slope_intercept_pin_input = int(sys.argv[17])  # Param not used
    regression_slope_intercept_pin_output = int(sys.argv[18])  # Param not used
    regression_rvalue_pvalue_id = int(sys.argv[19])
```

```
regression_rvalue_pvalue_pin_input = int(sys.argv[20])  # Param not used
regression_rvalue_pvalue_pin_output = int(sys.argv[21])  # Param not used
regression_stderr1_id = int(sys.argv[22])
regression_stderr1_pin_input = int(sys.argv[23])  # Param not used
regression_stderr1_pin_output = int(sys.argv[24])  # Param not used

print ("****************************")
print ("START MODEL LDR CIRCUIT RC. PARAMETERS:")
print ("\t boardId: %s" % str(board_id))
print ("\t testId: %s" % str(test_id))
print ("\t gasSensorId: %s" % str(gas_sensor_id))
print ("\t gasSensorPinInput: %s" % str(gas_sensor_pin_input))
print ("\t gasSensorPinOutput: %s" % str(gas_sensor_pin_output))
print ("\t----------------------------")
print ("\t tempSensorId: %s" % str(temp_sensor_id))
print ("\t tempSensorPinInput: %s" % str(temp_sensor_pin_input))
print ("\t tempSensorPinOutput: %s" % str(temp_sensor_pin_output))
print ("\t----------------------------")
print ("\t NM: %s" % str(nm))
print ("\t T: %s" % str(t))
print ("\t Tsub: %s" % str(tsub))
print ("\t sleep: %s" % str(sleep))
print ("\t samples_init: %s" % str(samples_init))
print ("\t tendency: %s" % str(tendency))
print ("\t average_temperature: %s" % str(average_temperature))
print ("\t duration: %s" % str(duration))
print ("\t----------------------------")
print ("\t slopeInterceptId: %s" % str(regression_slope_intercept_id))
print ("\t rValue_pValueId: %s" % str(regression_rvalue_pvalue_id))
print ("\t stderr1Id: %s" % str(regression_stderr1_id))
print ("****************************")

# GPIO to heat the sensor
GPIO.setup(gas_sensor_pin_input, GPIO.OUT)
nose_sensor = GPIO.PWM(gas_sensor_pin_input, 100)

# SPI bus
spi = spidev.SpiDev()
spi.open(0, 0)

# Update the test status to running
Thread(target=update_test_info, args=(test_id, {"status": "running"},)).start()

try:
    # Heating the sensor. Take samples_init before starting the experiment
    count = 1
    while count <= samples_init:
        tick_reg = time.time()
        value_tgs2620, new_temperature_tgs2620 = \
            ini_regression_tgs2620(count, average_temperature, gas_sensor_pin_output)
        count += 1
        tack_reg = time.time()
        time.sleep(sleep - round((tack_reg - tick_reg),1))

        tsend = current_milli_time()
        # Create the temperature sensor info to send
        info = {
            "testId": test_id,
            "boardId": board_id,
            "sensorId": temp_sensor_id,
            "inputValue": 0,
            "outputValue": convert_temp(read_channel(temp_sensor_pin_output), 2),
            "testTime": tsend
        }
        print(info)
        Thread(target=send_value, args=(info, )).start()
```

```
            # Create the gas sensor info to send
            info = {
                "testId": test_id,
                "boardId": board_id,
                "sensorId": gas_sensor_id,
                "inputValue": new_temperature_tgs2620,
                "outputValue": value_tgs2620,
                "testTime": tsend
            }
            Thread(target=send_value, args=(info, )).start()

            # Create the slope intercept info to send
            info = {
                "testId": test_id,
                "boardId": board_id,
                "sensorId": regression_slope_intercept_id,
                "inputValue": -1,
                "outputValue": -1,
                "testTime": tsend
            }
            Thread(target=send_value, args=(info, )).start()

            # Create the r_value p_value info to send
            info = {
                "testId": test_id,
                "boardId": board_id,
                "sensorId": regression_rvalue_pvalue_id,
                "inputValue": -1,
                "outputValue": -1,
                "testTime": tsend
            }
            Thread(target=send_value, args=(info, )).start()

            # Create the std_err1 info to send
            info = {
                "testId": test_id,
                "boardId": board_id,
                "sensorId": regression_stderr1_id,
                "inputValue": -1,
                "outputValue": 0,
                "testTime": current_milli_time()
            }
            Thread(target=send_value, args=(info, )).start()


        print '\n\nSe van a capturar: ' + str(duration * 60) + ' muestras\n\n'

        # Start the experiment
        while count <= ((duration * 60) + samples_init):
            tick_regression = time.time()
            value_tgs2620, new_temperature_tgs2620, slope, intercept, r_value, p_value,
std_err1 = \
                regression_tgs2620(count, average_temperature, gas_sensor_pin_output)
            count += 1
            tack_regression = time.time()
            time.sleep(sleep - round((tack_regression - tick_regression),1))

            tsend = current_milli_time()

            # Create the gas sensor info to send
            info = {
                "testId": test_id,
                "boardId": board_id,
                "sensorId": temp_sensor_id,
                "inputValue": 0,
```

```
                "outputValue": convert_temp(read_channel(temp_sensor_pin_output), 2),
                "testTime": tsend
            }
            Thread(target=send_value, args=(info, )).start()


             # Create the gas sensor info to send
            info = {
                "testId": test_id,
                "boardId": board_id,
                "sensorId": gas_sensor_id,
                "inputValue": new_temperature_tgs2620,
                "outputValue": value_tgs2620,
                "testTime": tsend
            }
            Thread(target=send_value, args=(info, )).start()

             # Create the slope intercept info to send
            info = {
                "testId": test_id,
                "boardId": board_id,
                "sensorId": regression_slope_intercept_id,
                "inputValue": slope,
                "outputValue": intercept,
                "testTime": tsend
            }
            Thread(target=send_value, args=(info, )).start()

             # Create the r_value p_value info to send
            info = {
                "testId": test_id,
                "boardId": board_id,
                "sensorId": regression_rvalue_pvalue_id,
                "inputValue": r_value,
                "outputValue": p_value,
                "testTime": tsend
            }
            Thread(target=send_value, args=(info, )).start()

             # Create the std_err1 info to send
            info = {
                "testId": test_id,
                "boardId": board_id,
                "sensorId": regression_stderr1_id,
                "inputValue": std_err1,
                "outputValue": 0,
                "testTime": current_milli_time()
            }
            Thread(target=send_value, args=(info, )).start()

            # Update the test progress
            progress = 100*(count/((duration * 60) + samples_init))
            Thread(target=update_test_info, args=(test_id, {"progress":
progress},)).start()

        # Update the test status to finished
        Thread(target=update_test_info, args=(test_id, {"status": "finished", "progress":
100},)).start()
        print('\nEXPERIMENTO FINALIZADO CON EXITO')

    except KeyboardInterrupt:
        # exits when press CTRL+C or command kill -s INT [PID]
        print "\n keyboard interrupt"  # print value of counter

    except:
        # this catches ALL other exceptions including errors.
```

```
        print ("Other error or exception occurred! \n ", sys.exc_info()[0])
        error_msg = "Other error or exception occurred! \n " + str(sys.exc_info()[0])
        Thread(target=update_test_info, args=(test_id, {"status": error_msg},)).start()

    finally:
        GPIO.cleanup()
        end_time = time.time()
        print 'Experimento terminado:      ' + str(end_time)
```

Model-ldr-adc-led.py

[...]

```
if __name__ == "__main__":
    # Parameters from input
    board_id = str(sys.argv[1])
    test_id = str(sys.argv[2])
    sensor_id = int(sys.argv[3])
    sensor_pin_input = int(sys.argv[4])  # Param not used
    sensor_pin_output = int(sys.argv[5])
    if sensor_pin_output >= 100:  # Define sensor channels. Greater than 100 indicates that
is a adc channel
        sensor_pin_output -= 100
    sleep_time = float(sys.argv[6])
    led_id = int(sys.argv[7])
    led_pin_input = int(sys.argv[8])  # Pin to turn on de LED
    led_pin_output = int(sys.argv[9])  # Param not used
    led_on_value = float(sys.argv[10])  # Define the LDR value to turn on the LED

    print ("****************************")
    print ("START MODEL LDR ADC WITH LED. PARAMETERS:")
    print ("\t boardId: %s" % str(board_id))
    print ("\t testId: %s" % str(test_id))
    print ("\t sensorId: %s" % str(sensor_id))
    print ("\t sensorPinInput: %s" % str(sensor_pin_input))
    print ("\t sensorPinOutput: %s" % str(sensor_pin_output))
    print ("\t samplingTime: %s" % str(sleep_time))
    print ("\t ledId: %s" % str(led_id))
    print ("\t ledPinInput: %s" % str(led_pin_input))
    print ("\t ledPinOutput: %s" % str(led_pin_output))
    print ("****************************")

    # SPI bus
    spi = spidev.SpiDev()
    spi.open(0, 0)

    # GPIO bus
    GPIO.setmode(GPIO.BCM)
    GPIO.setup(led_pin_input, GPIO.OUT)

    # Update the test status to running
    Thread(target=update_test_info, args=(test_id, {"status": "running"},)).start()

    try:
        while True:

            # Read the light sensor data
            adc = read_channel(sensor_pin_output)
            pot_volts = convert_volts(adc, 2)

            # Print out results
            print ("--------------------------------------------")
            print ("Lectura ADC: ", adc)
            print("Voltaje: {}V".format(pot_volts))
```

```
            # Create the info to send
            info = {
                "testId": test_id,
                "boardId": board_id,
                "sensorId": sensor_id,
                "inputValue": 5,
                "outputValue": adc,
                "testTime": current_milli_time()
            }
            Thread(target=send_value, args=(info, )).start()

            # Check if LED have to be turned on/off
            on = 0
            GPIO.output(led_pin_input, GPIO.LOW)
            if adc < led_on_value:
                GPIO.output(led_pin_input, GPIO.HIGH)
                on = led_on_value

            info = {
                "testId": test_id,
                "boardId": board_id,
                "sensorId": led_id,
                "inputValue": on,
                "outputValue": 3.3,
                "testTime": current_milli_time()
            }
            Thread(target=send_value, args=(info, )).start()

            # Wait before repeating loop
            time.sleep(sleep_time)

    except KeyboardInterrupt:
        # exits when press CTRL+C or command kill -s INT [PID]
        print "\n keyboard interrupt"  # print value of counter

    except:
        # this catches ALL other exceptions including errors.
        print ("Other error or exception occurred! \n ", sys.exc_info()[0])
        error_msg = "Other error or exception occurred! \n " + str(sys.exc_info()[0])
        Thread(target=update_test_info, args=(test_id, {"status": error_msg},)).start()

    finally:
        GPIO.cleanup()  # this ensures a clean exit
        print "Exit program"
```

Model-ldr-circuit-rc.py

[...]

```
if __name__ == "__main__":
    # Parameters from input
    board_id = str(sys.argv[1])
    test_id = str(sys.argv[2])
    sensor_id = int(sys.argv[3])
    sensor_pin_input = int(sys.argv[4])
    sensor_pin_output = int(sys.argv[5])
    sleep_time = float(sys.argv[6])

    print ("****************************")
    print ("START MODEL LDR CIRCUIT RC. PARAMETERS:")
    print ("\t boardId: %s" % str(board_id))
    print ("\t testId: %s" % str(test_id))
    print ("\t sensorId: %s" % str(sensor_id))
```

```
    print ("\t sensorPinInput: %s" % str(sensor_pin_input))
    print ("\t sensorPinOutput: %s" % str(sensor_pin_output))
    print ("\t samplingTime: %s" % str(sleep_time))
    print ("****************************")

    # GPIO bus
    GPIO.setmode(GPIO.BCM)

    # Update the test status to running
    Thread(target=update_test_info, args=(test_id, {"status": "running"},)).start()

    try:
        while True:
            output_value = rc_time(sensor_pin_output)  # Measure timing using GPIO

            # Create the info to send
            info = {
                "testId": test_id,
                "boardId": board_id,
                "sensorId": sensor_id,
                "inputValue": 5,
                "outputValue": output_value,
                "testTime": current_milli_time()
            }
            Thread(target=send_value, args=(info, )).start()
            print(info)
            time.sleep(sleep_time)

    except KeyboardInterrupt:
        # exits when press CTRL+C or command kill -s INT [PID]
        print "\n keyboard interrupt"  # print value of counter

    except:
        # this catches ALL other exceptions including errors.
        print ("Other error or exception occurred! \n ", sys.exc_info()[0])
        error_msg = "Other error or exception occurred! \n " + str(sys.exc_info()[0])
        Thread(target=update_test_info, args=(test_id, {"status": error_msg},)).start()

    finally:
        GPIO.cleanup()  # this ensures a clean exit
        print "Exit program"
```

Model-lm335-adc.py

[...]

```
if __name__ == "__main__":
    # Parameters from input
    board_id = str(sys.argv[1])
    test_id = str(sys.argv[2])
    sensor_id = int(sys.argv[3])
    sensor_pin_input = int(sys.argv[4])  # Param not used
    sensor_pin_output = int(sys.argv[5])
    if sensor_pin_output >= 100:  # Define sensor channels. Greater than 100 indicates that
is a adc channel
        sensor_pin_output -= 100
    sleep_time = float(sys.argv[6])
    temperature_id = int(sys.argv[7])  # Used to store the info in the sensor variable
    temperature_pin_input = int(sys.argv[8])  # Param not used
    temperature_pin_output = int(sys.argv[9])  # Param not used

    print ("****************************")
    print ("START MODEL LM335 ADC. PARAMETERS:")
    print ("\t boardId: %s" % str(board_id))
```

```
print ("\t testId: %s" % str(test_id))
print ("\t sensorId: %s" % str(sensor_id))
print ("\t sensorPinInput: %s" % str(sensor_pin_input))
print ("\t sensorPinOutput: %s" % str(sensor_pin_output))
print ("\t samplingTime: %s" % str(sleep_time))
print ("\t temperatureId: %s" % str(temperature_id))
print ("****************************")

# SPI bus
spi = spidev.SpiDev()
spi.open(0, 0)

# Update the test status to running
Thread(target=update_test_info, args=(test_id, {"status": "running"},)).start()

try:
    while True:

        # Read the temperature sensor data
        adc = read_channel(sensor_pin_output)
        pot_volts = convert_volts(adc, 2)
        temp = convert_temp(adc, 2)

        # Print out results
        print ("-----------------------------------------")
        print ("Lectura ADC: ", adc)
        print("Voltaje: {}V".format(pot_volts))
        print("TEMP: {} Celsius".format(temp-273))
        print("TEMP: {} Kelvin".format(temp))

        # Create the sensor info to send
        info = {
            "testId": test_id,
            "boardId": board_id,
            "sensorId": sensor_id,
            "inputValue": 5,
            "outputValue": adc,
            "testTime": current_milli_time()
        }
        Thread(target=send_value, args=(info, )).start()

        # Create the temperature info to send
        celsius = temp - 273
        info = {
            "testId": test_id,
            "boardId": board_id,
            "sensorId": temperature_id,
            "inputValue": celsius,
            "outputValue": temp,
            "testTime": current_milli_time()
        }
        Thread(target=send_value, args=(info, )).start()

        # Wait before repeating loop
        time.sleep(sleep_time)

except KeyboardInterrupt:
    # exits when press CTRL+C or command kill -s INT [PID]
    print "\n keyboard interrupt"  # print value of counter

except:
    # this catches ALL other exceptions including errors.
    print ("Other error or exception occurred! \n ", sys.exc_info()[0])
    error_msg = "Other error or exception occurred! \n " + str(sys.exc_info()[0])
    Thread(target=update_test_info, args=(test_id, {"status": error_msg},)).start()
```

```
    finally:
        print "Exit program"
```

Model-sensor-adc.py

[...]

```
if __name__ == "__main__":
    # Parameters from input
    board_id = str(sys.argv[1])
    test_id = str(sys.argv[2])
    sensor_id = int(sys.argv[3])
    sensor_pin_input = int(sys.argv[4])  # Param not used
    sensor_pin_output = int(sys.argv[5])
    if sensor_pin_output >= 100:  # Define sensor channels. Greater than 100 indicates that
is a adc channel
        sensor_pin_output -= 100
    sleep_time = float(sys.argv[6])

    print ("****************************")
    print ("START MODEL LDR CIRCUIT RC. PARAMETERS:")
    print ("\t boardId: %s" % str(board_id))
    print ("\t testId: %s" % str(test_id))
    print ("\t sensorId: %s" % str(sensor_id))
    print ("\t sensorPinInput: %s" % str(sensor_pin_input))
    print ("\t sensorPinOutput: %s" % str(sensor_pin_output))
    print ("\t samplingTime: %s" % str(sleep_time))
    print ("****************************")

    # SPI bus
    spi = spidev.SpiDev()
    spi.open(0, 0)

    # Update the test status to running
    Thread(target=update_test_info, args=(test_id, {"status": "running"},)).start()

    try:
        while True:

            # Read the analg sensor data
            adc = read_channel(sensor_pin_output)
            pot_volts = convert_volts(adc, 2)

            # Print out results
            print ("-----------------------------------------")
            print ("Lectura ADC: ", adc)
            print("Voltaje: {}V".format(pot_volts))

            # Create the info to send
            info = {
                "testId": test_id,
                "boardId": board_id,
                "sensorId": sensor_id,
                "inputValue": 5,
                "outputValue": pot_volts,
                "testTime": current_milli_time()
            }
            Thread(target=send_value, args=(info, )).start()

            # Wait before repeating loop
            time.sleep(sleep_time)

    except KeyboardInterrupt:
        # exits when press CTRL+C or command kill -s INT [PID]
```

```
    print "\n keyboard interrupt"  # print value of counter

except:
    # this catches ALL other exceptions including errors.
    print ("Other error or exception occurred! \n ", sys.exc_info()[0])
    error_msg = "Other error or exception occurred! \n " + str(sys.exc_info()[0])
    Thread(target=update_test_info, args=(test_id, {"status": error_msg},)).start()

finally:
    print "Exit program"
```

Anexo G. Presupuesto

1. Ejecución Material

- Compra de ordenador personal (Software incluido) ……..…….. 2.000 €
- Material de oficina ……..……..... 50 €
- Componentes electrónicos ……..……..... 100 €
- Total de ejecución material ……..…….. 2.150 €

2. Gastos generales

- 16 % sobre Ejecución Material ……..…... 344€

3. Beneficio Industrial

- 6 % sobre Ejecución Material …….. 129 €

4. Honorarios Proyecto

- 1800 horas a 15 € / hora ……..…….. 27.000 €

5. Material fungible

- Gastos de impresión ……. 150 €
- Encuadernación …… 50 €

6. Subtotal del presupuesto

- Subtotal Presupuesto ……..……..… 29.823 €

7. I.V.A. aplicable

- 21% Subtotal Presupuesto ……..…… 6.262,83 €

8. Total presupuesto

- Total Presupuesto ……..…… 36.085,83 €

Madrid, julio de 2016
El Ingeniero Jefe de Proyecto

Fdo.: Luis Pulido García-Duarte
Ingeniero de Telecomunicación

Anexo H. Pliego de condiciones

Este documento contiene las condiciones legales que guiarán la realización, en este proyecto, del diseño, desarrollo e implementación de un protocolo de comunicaciones entre sensores en red y un computador. En lo que sigue, se supondrá que el proyecto ha sido encargado por una empresa cliente a una empresa consultora con la finalidad de realizar dicho sistema. Dicha empresa ha debido desarrollar una línea de investigación con objeto de elaborar el proyecto. Esta línea de investigación, junto con el posterior desarrollo de los programas está amparada por las condiciones particulares del siguiente pliego.

Supuesto que la utilización industrial de los métodos recogidos en el presente proyecto ha sido decidida por parte de la empresa cliente o de otras, la obra a realizar se regulará por las siguientes:

Condiciones generales

1. La modalidad de contratación será el concurso. La adjudicación se hará, por tanto, a la proposición más favorable sin atender exclusivamente al valor económico, dependiendo de las mayores garantías ofrecidas. La empresa que somete el proyecto a concurso se reserva el derecho a declararlo desierto.

2. El montaje y mecanización completa de los equipos que intervengan será realizado totalmente por la empresa licitadora.

3. En la oferta, se hará constar el precio total por el que se compromete a realizar la obra y el tanto por ciento de baja que supone este precio en relación con un importe límite si este se hubiera fijado.

4. La obra se realizará bajo la dirección técnica de un Ingeniero Superior de Telecomunicación, auxiliado por el número de Ingenieros Técnicos y Programadores que se estime preciso para el desarrollo de la misma.

5. Aparte del Ingeniero Director, el contratista tendrá derecho a contratar al resto del personal, pudiendo ceder esta prerrogativa a favor del Ingeniero Director, quien no estará obligado a aceptarla.

6. El contratista tiene derecho a sacar copias a su costa de los planos, pliego de condiciones y presupuestos. El Ingeniero autor del proyecto autorizará con su firma las copias solicitadas por el contratista después de confrontarlas.

7. Se abonará al contratista la obra que realmente ejecute con sujeción al proyecto que sirvió de base para la contratación, a las modificaciones autorizadas por la superioridad o a las órdenes que con arreglo a sus facultades le hayan comunicado por escrito al Ingeniero Director de obras siempre que dicha obra se haya ajustado a los preceptos de los pliegos de condiciones, con arreglo a los cuales, se harán las modificaciones y la valoración de las diversas unidades sin que el importe total pueda exceder de los presupuestos aprobados.

Por consiguiente, el número de unidades que se consignan en el proyecto o en el presupuesto, no podrá servirle de fundamento para entablar reclamaciones de ninguna clase, salvo en los casos de rescisión.

8. Tanto en las certificaciones de obras como en la liquidación final, se abonarán los trabajos realizados por el contratista a los precios de ejecución material que figuran en el presupuesto para cada unidad de la obra.

9. Si excepcionalmente se hubiera ejecutado algún trabajo que no se ajustase a las condiciones de la contrata pero que sin embargo es admisible a juicio del Ingeniero Director de obras, se dará conocimiento a la Dirección, proponiendo a la vez la rebaja de precios que el Ingeniero estime justa y si la Dirección resolviera aceptar la obra, quedará el contratista obligado a conformarse con la rebaja acordada.

10. Cuando se juzgue necesario emplear materiales o ejecutar obras que no figuren en el presupuesto de la contrata, se evaluará su importe a los precios asignados a otras obras o materiales análogos si los hubiere y cuando no, se discutirán entre el Ingeniero Director y el contratista, sometiéndolos a la aprobación de la Dirección. Los nuevos precios convenidos por uno u otro procedimiento, se sujetarán siempre al establecido en el punto anterior.

11. Cuando el contratista, con autorización del Ingeniero Director de obras, emplee materiales de calidad más elevada o de mayores dimensiones de lo estipulado en el proyecto, o sustituya una clase de fabricación por otra que tenga asignado mayor precio o ejecute con mayores dimensiones cualquier otra parte de las obras, o en general, introduzca en ellas cualquier modificación que sea beneficiosa a juicio del Ingeniero Director de obras, no tendrá derecho sin embargo, sino a lo que le correspondería si hubiera realizado la obra con estricta sujeción a lo proyectado y contratado.

12. Las cantidades calculadas para obras accesorias, aunque figuren por partida alzada en el presupuesto final (general), no serán abonadas sino a los precios de la contrata, según las condiciones de la misma y los proyectos particulares que para ellas se formen, o en su defecto, por lo que resulte de su medición final.

13. El contratista queda obligado a abonar al Ingeniero autor del proyecto y director de obras así como a los Ingenieros Técnicos, el importe de sus respectivos honorarios facultativos por formación del proyecto, dirección técnica y administración en su caso, con arreglo a las tarifas y honorarios vigentes.

14. Concluida la ejecución de la obra, será reconocida por el Ingeniero Director que a tal efecto designe la empresa.

15. La garantía definitiva será del 4% del presupuesto y la provisional del 2%.

16. La forma de pago será por certificaciones mensuales de la obra ejecutada, de acuerdo con los precios del presupuesto, deducida la baja si la hubiera.

17. La fecha de comienzo de las obras será a partir de los 15 días naturales del replanteo oficial de las mismas y la definitiva, al año de haber ejecutado la provisional, procediéndose si no existe reclamación alguna, a la reclamación de la fianza.

18. Si el contratista al efectuar el replanteo, observase algún error en el proyecto, deberá comunicarlo en el plazo de quince días al Ingeniero Director de obras, pues transcurrido ese plazo será responsable de la exactitud del proyecto.

19. El contratista está obligado a designar una persona responsable que se entenderá con el Ingeniero Director de obras, o con el delegado que éste designe, para todo relacionado con ella. Al ser el Ingeniero Director de obras el que interpreta el proyecto, el contratista deberá consultarle cualquier duda que surja en su realización.

20. Durante la realización de la obra, se girarán visitas de inspección por personal facultativo de la empresa cliente, para hacer las comprobaciones que se crean oportunas. Es obligación del contratista, la conservación de la obra ya ejecutada hasta la recepción de la misma, por lo que el deterioro parcial o total de ella, aunque sea por agentes atmosféricos u otras causas, deberá ser reparado o reconstruido por su cuenta.

21. El contratista, deberá realizar la obra en el plazo mencionado a partir de la fecha del contrato, incurriendo en multa, por retraso de la ejecución siempre que éste no sea debido a causas de fuerza mayor. A la terminación de la obra, se hará una recepción provisional previo reconocimiento y examen por la dirección técnica, el depositario de efectos, el interventor y el jefe de servicio o un representante, estampando su conformidad el contratista.

22. Hecha la recepción provisional, se certificará al contratista el resto de la obra, reservándose la administración el importe de los gastos de conservación de la misma hasta su recepción definitiva y la fianza durante el tiempo señalado como plazo de garantía. La recepción definitiva se hará en las mismas condiciones que la provisional, extendiéndose el acta correspondiente. El Director Técnico propondrá a la Junta Económica la devolución de la fianza al contratista de acuerdo con las condiciones económicas legales establecidas.

23. Las tarifas para la determinación de honorarios, reguladas por orden de la Presidencia del Gobierno el 19 de Octubre de 1961, se aplicarán sobre el denominado en la actualidad “Presupuesto de Ejecución de Contrata” y anteriormente llamado ”Presupuesto de Ejecución Material” que hoy designa otro concepto.

Condiciones particulares

La empresa consultora, que ha desarrollado el presente proyecto, lo entregará a la empresa cliente bajo las condiciones generales ya formuladas, debiendo añadirse las siguientes condiciones particulares:

1. La propiedad intelectual de los procesos descritos y analizados en el presente trabajo, pertenece por entero a la empresa consultora representada por el Ingeniero Director del Proyecto.

2. La empresa consultora se reserva el derecho a la utilización total o parcial de los resultados de la investigación realizada para desarrollar el siguiente proyecto, bien para su publicación o bien para su uso en trabajos o proyectos posteriores, para la misma empresa cliente o para otra.

3. Cualquier tipo de reproducción aparte de las reseñadas en las condiciones generales, bien sea para uso particular de la empresa cliente, o para cualquier otra aplicación, contará con autorización expresa y por escrito del Ingeniero Director del Proyecto, que actuará en representación de la empresa consultora.

4. En la autorización se ha de hacer constar la aplicación a que se destinan sus reproducciones así como su cantidad.

5. En todas las reproducciones se indicará su procedencia, explicitando el nombre del proyecto, nombre del Ingeniero Director y de la empresa consultora.

6. Si el proyecto pasa la etapa de desarrollo, cualquier modificación que se realice sobre él, deberá ser notificada al Ingeniero Director del Proyecto y a criterio de éste, la empresa consultora decidirá aceptar o no la modificación propuesta.

7. Si la modificación se acepta, la empresa consultora se hará responsable al mismo nivel que el proyecto inicial del que resulta el añadirla.

8. Si la modificación no es aceptada, por el contrario, la empresa consultora declinará toda responsabilidad que se derive de la aplicación o influencia de la misma.

9. Si la empresa cliente decide desarrollar industrialmente uno o varios productos en los que resulte parcial o totalmente aplicable el estudio de este proyecto, deberá comunicarlo a la empresa consultora.

10. La empresa consultora no se responsabiliza de los efectos laterales que se puedan producir en el momento en que se utilice la herramienta objeto del presente proyecto para la realización de otras aplicaciones.

11. La empresa consultora tendrá prioridad respecto a otras en la elaboración de los proyectos auxiliares que fuese necesario desarrollar para dicha aplicación industrial, siempre que no haga explícita renuncia a este hecho. En este caso, deberá autorizar expresamente los proyectos presentados por otros.

12. El Ingeniero Director del presente proyecto, será el responsable de la dirección de la aplicación industrial siempre que la empresa consultora lo estime oportuno. En caso contrario, la persona designada deberá contar con la autorización del mismo, quien delegará en él las responsabilidades que ostente.