Grado en Ingeniería Informática

Trabajo Final de Grado

Desarrollo de un servicio de auditoría para una plataforma de *Internet of Things*

Autor:
Pedro Domingo Mengual

Supervisor:
Javier Muñoz ferrara
Tutor académico:
Lledó Museros Cabedo

Fecha de lectura: 22 de Junio de 2018
Curso académico 2017/2018

Resumen

Este documento contiene la memoria del trabajo final de grado en el que se ha desarrollado un servicio de auditorías para la plataforma de *Internet of Things* de IoTsens.

El proyecto se ha realizado en esta misma empresa, que constituye la rama de Internet de las cosas del grupo empresarial Grupo Gimeno.

Concretamente, el proyecto ha consistido, en primer lugar, en desarrollar un sistema que permita la auditoría de datos procedentes de los *microservicios* que forman parte de la arquitectura de la empresa, comunicando a éstos últimos con el nuevo sistema y, en segundo lugar, en implementar una vista *Web* que permita filtrar y visualizar dichos datos.

Para el desarrollo del proyecto se ha realizado un análisis de requisitos, donde se ha definido la funcionalidad deseada conjuntamente con el cliente, estudiando todas las tecnologías y herramientas que forman parte del ecosistema de la empresa, para posteriormente implantar el resultado del desarrollo.

Así mismo, también se han llevado a cabo una serie de pruebas para comprobar el correcto funcionamiento del servicio y la calidad del mismo.

Palabras clave

Internet de las cosas, auditorías, aplicación Web, Java, microservicio, ciudad inteligente.

Abstract

This document contains the memory of the end-of-degree project in which an audit service has been developed for the IoTSens Internet of Things platform.

The project has been developed in this same company, which constitutes the Internet of Things branch of the Grupo Gimeno corporate group.

Specifically, the project has consisted, firstly, in developing a system that allows the audit of data from the different microservices that are part of the company's architecture, and secondly to develop a Web view that allows the data to be filtered and visualized.

For the development of the project, a requirements analysis has been performed, where the desired functionality has been defined in conjunction with the client, studying all the technologies and tools that take part in the company's ecosystem, to subsequently implant the result of the development.

Moreover, a series of tests have been performed in order to ensure the correct behaviour of

the service and the quality of the same.

Keywords

Internet of things, audits, Web aplication, java, microservice, smart city.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

El contexto en el que se encuentra la empresa es el del *Internet de las cosas*, un concepto que describe la idea de objetos del día a día conectados a Internet, normalmente para ser gestionados por otros equipos.

IoTSens proporciona una vertical de soluciones de *Internet de las cosas* escalable e interoperable, que recolecta e intercambia datos del mundo físico con el mundo digital, lo que resulta en una mejora en eficiencia, exactitud y beneficio económico [4] .

En este capítulo se describe de forma más amplia el contexto en el que se desarrolla el proyecto y los principales objetivos del mismo.

1.1. Descripción de la empresa

El Grupo Gimeno [5] es un grupo empresarial que se fundó hace más de 140 años en Castellón con la constitución de FACSA, con el objetivo de dotar a la capital de una moderna red de distribución de agua potable. Actualmente, cuenta con más de 30 empresas operando en territorio nacional que operan gracias a los 4.100 profesionales que forman parte de su equipo humano.

Con el paso del tiempo, Grupo Gimeno ha ido evolucionando hasta consolidar su presencia en sectores económicos tan diversos como la construcción, el turismo, el ocio, o las nuevas tecnologías, surgiendo tres divisiones a partir de dicha evolución: Gimeno Servicios, Gimeno Construcción y Gimeno Turismo y Ocio.

Dentro de la división de servicios del grupo se encuentra ADC Infraestructuras y Sistemas, S.L., que ofrece cobertura tecnológica el resto de empresas que forman parte de Grupo Gimeno, y se divide en tres departamentos: *Software*, *Datacenter* y *Comunicaciones y Microinformática*.

El proyecto se ha desarrollado en IoTSens, que nace en 2013 como departamento propio de ADC Infraestructuras y Sistemas, con el fin de vender, tanto a empresas públicas como privadas,

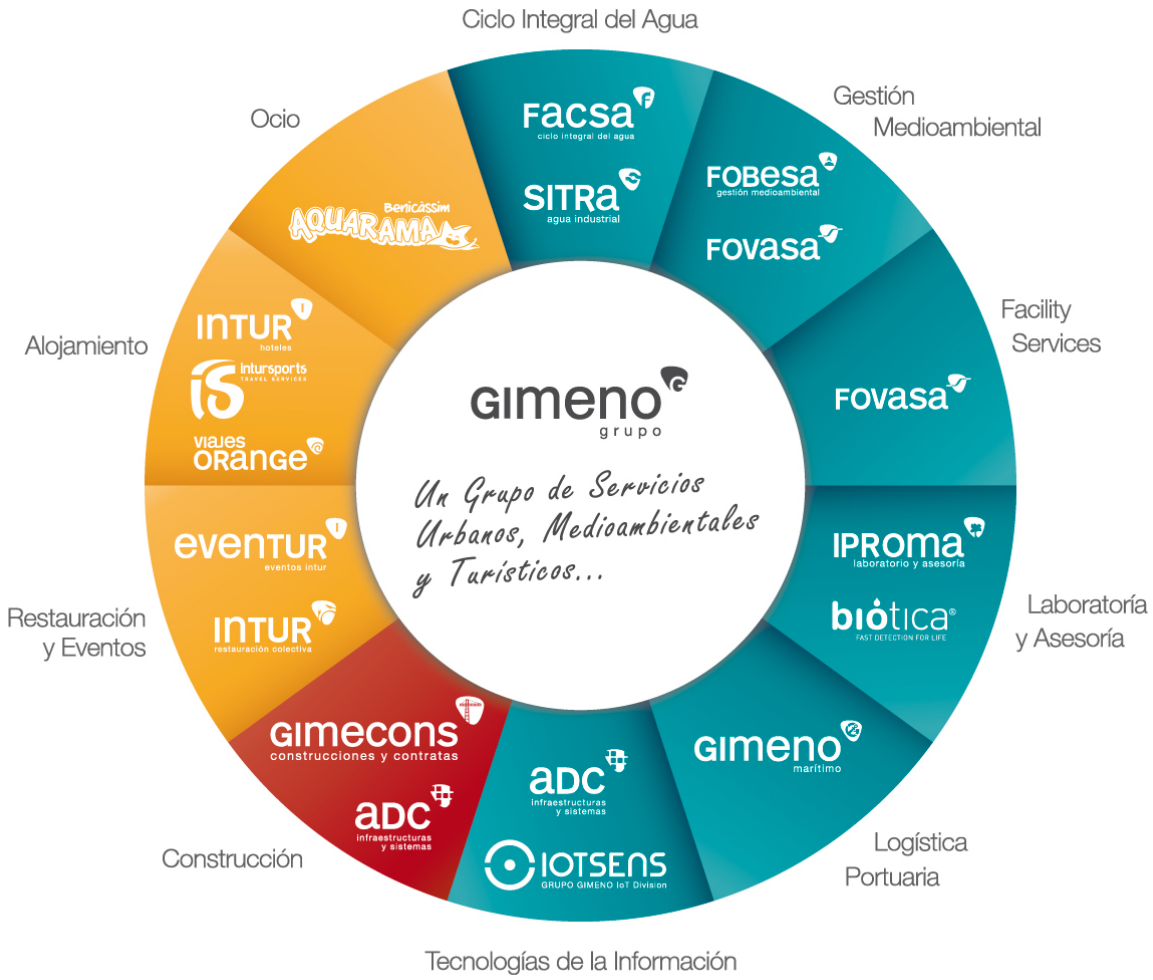


Figura 1.1: Organigrama de las divisiones del Grupo Gimeno.

soluciones del *Internet de las cosas* compuestas por verticales eficientes y colaborativas. Dichas verticales se encuentran enumeradas en la sección 1.3.

1.2. El Internet de las cosas

El *Internet de las cosas*, también conocido como *Internet of things* o IoT, tiene como objetivo hacer que, mediante ciertos dispositivos o tecnologías, estos objetos se comuniquen entre sí para, por tanto, conseguir que sean más independientes e "inteligentes"[14].

Se trata de un concepto en auge y sobre el que es cada vez más habitual oír hablar, especialmente debido a que proporciona soluciones a problemas cotidianos, como optimizar los sistemas de riego en agricultura o enviar alarmas dependiendo de las situaciones meteorológicas, aunque también es cada vez más común ver dispositivos IoT en los hogares, levantando

las persianas automáticamente al detectar el sol, o activando el sistema de aire acondicionado cuando el sensor detecta que la temperatura del hogar es superior a la deseada, por ejemplo.

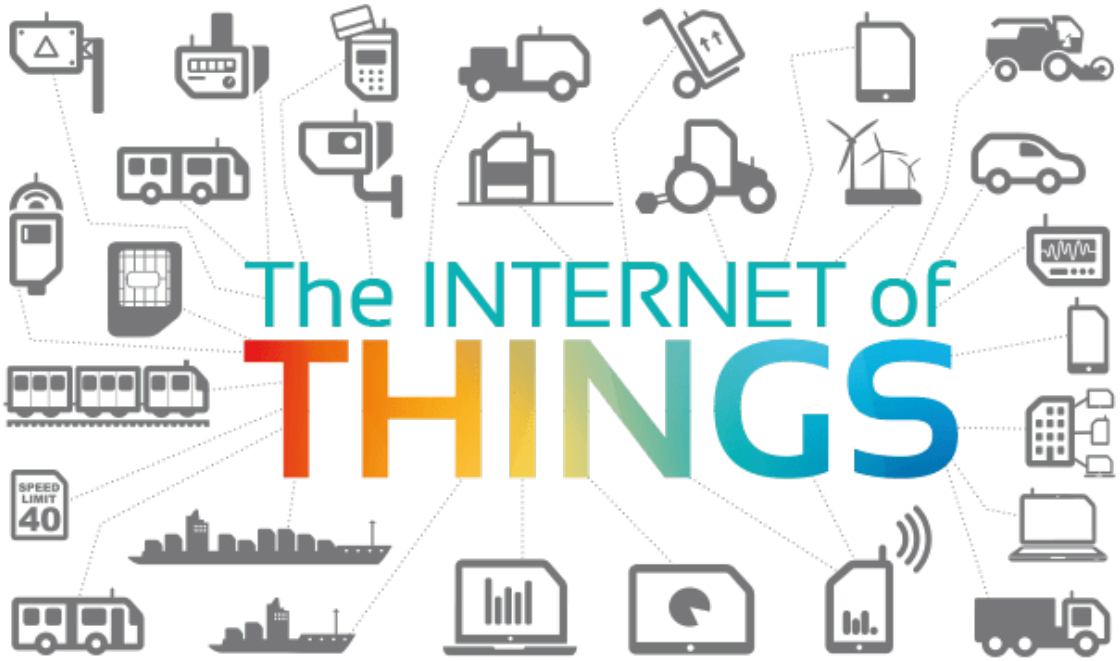


Figura 1.2: Concepto general de IoT

El concepto *Smart City*, por otro lado, se refiere al de un área urbana que usa diferentes tipos de sensores de recopilación de datos para proporcionar información que es usada para administrar los recursos eficientemente [10].

IoTsens se basa en estos conceptos para proporcionar los servicios que ofrecen a empresas en sus distintas verticales, especialmente en el ámbito de las ciudades inteligentes.

1.3. IoTsens

IoTsens es una de las empresas pioneras en el ámbito del *Internet of Things* en España, proporcionando una red de comunicaciones y una plataforma *software* de procesamiento de la información para *Smart Metering* y *Smart Cities*, basándose en estándares de los mismos. Para ello, su arquitectura permite integrar de forma dinámica todo tipo de agentes, como actuadores, sensores o sistemas de información externos, que trata y explota de forma homogénea para que estos sean explotados mediante alarmas, comparativas, tendencias, paneles de control, etc.

Por ello, mediante dicha integración dinámica se ofrece un sistema integral de *Smart City* con comunicación bidireccional, donde mediante una arquitectura basada en colas altamente escalables, se transmiten y consumen los distintos mensajes para su tratamiento y gestión. El servidor central es el encargado de procesar dichos mensajes o de generar otros nuevos mediante componentes especializados en cada una de las distintas tecnologías utilizadas, y a continuación los distribuye a través de los diferentes procesos especializados en su codificación

y decodificación, para posteriormente ser utilizados para generar informes, alarmas, estadísticas o cualquier otra forma de explotación de los mismos.

Figura 1.3: Logotipo de IoTSens

Mediante una arquitectura tan flexible, IoTsens desarrolla todo tipo de soluciones verticales. Actualmente consta de las siguientes: *SmartWaste*, encargada de la gestión de basuras, *SmartWater*, encargada de la monitorización del ciclo integral del agua, *SmartIndustrial*, que controla y gestión de plantas industriales, *SmartAgriculture*, encargada de monitorizar datos procedentes de tierras de cultivo, y *SmartCity*, que permite controlar y administrar dispositivos que se encuentran en cualquier parte del mundo.

1.4. Contexto y motivación del proyecto

Este proyecto surge de la necesidad por parte de IoTsens de auditar y monitorizar las acciones de los usuarios y las distintas aplicaciones y procesos que forman parte de su arquitectura. Esto permite recolectar datos sobre acciones específicas de los mismos, detectar problemas de autorización y control de acceso e investigar actividades sospechosas que se realicen en la plataforma. La aplicación a desarrollar formará parte de la arquitectura de IoTsens, ofreciendo un valor añadido dentro de su conjunto de soluciones.

La plataforma existente consta de una estructura formada por las siguientes cuatro capas:

1. **Capa de sensores y dispositivos**: se sitúan los dispositivos en los lugares requeridos y estos envían la información recogida al servidor central.
2. **Capa de conectividad**: formada por las distintas redes de comunicación, formadas por estándares como LoRa o UNE [3].
3. **Capa de almacenamiento de datos**: formada por servidores que se encargan de gestionar las colas de datos y los almacenan, para que estos sean accesibles por el resto de aplicaciones.
4. **Capa de aplicación**: representada por las distintas aplicaciones que ofrece la empresa.

El proyecto desarrollado se centra en la capa de aplicación, permitiendo almacenar todas las acciones que necesiten ser auditadas, y los cambios que éstas producen en los datos utilizados por las diferentes aplicaciones de la plataforma.

Figura 1.4: Representación de la arquitectura de IoTsens

Adicionalmente, las diferentes aplicaciones ofrecidas por IoTsens se encuentran implementadas mediante una arquitectura orientada a micro-servicios, construidas como un conjunto de pequeños servicios los cuales se ejecutan en su propio proceso y se comunican mediante el protocolo HTTP [11], lo que añade cierta complicación al proyecto.

Por tanto, el nuevo servicio debe permitir almacenar registros de auditorías de los diferentes micro-servicios que forman parte de la arquitectura, almacenando el usuario que realiza la llamada y la aplicación de origen, además de los datos que se han producido en la misma.

1.5. Objetivos del proyecto

El principal objetivo de este proyecto es el registro y visualización de auditorías de las diferentes funcionalidades que se ofrecen en la plataforma de IoTsens.

El objetivo principal se puede desglosar en los siguientes sub-objetivos:

- Sustituir las tablas de auditoría de cada micro-servicio por una funcionalidad separada e independiente.

- Adición de las acciones que se ejecuten en la plataforma al sistema de auditorías.
- Posibilidad de visualización, mediante una interfaz web, tanto del estado anterior como del posterior de los datos asociados.
- Consulta de los registros mediante diferentes filtros.
- Eliminación de registros que se consideren innecesarios.

1.6. Estructura de la memoria

Esta memoria se encuentra dividida en siete capítulos: *Introducción, Planificación del proyecto, Tecnologías utilizadas, Análisis y diseño del sistema, Desarrollo e implementación, Verificación y validación y Conclusiones*, en conjunción diferentes anexos que complementarán algunos de los capítulos anteriores.

En el capítulo de *Introducción* se pretende dar una visión general del contexto del proyecto y su relación con el *Internet de las cosas*, así como dar una visión general de la empresa en donde se realiza.

En segundo lugar, en el capítulo de *Planificación del proyecto*, se describen las distintas tareas que forman parte del proyecto, junto a un coste temporal asociado a cada una de ellas.

A continuación, en *Tecnologías utilizadas*, se explican las herramientas y estándares que se han utilizado en el desarrollo e implantación del proyecto, así como los diferentes estándares y convenios adoptados por la propia empresa. En el *Análisis de requisitos* se precisan los bloques que componen la aplicación, especificando los requisitos de datos y funcionales, además de los prototipos de interfaz de usuario.

En siguiente capítulo, *Análisis y diseño del sistema* se expone el proceso seguido durante la fase de análisis, así como los detalles de la arquitectura del proyecto.

En siguiente lugar, en *Desarrollo e implementación*, se muestran las funcionalidades de las que consta el proyecto, las técnicas y patrones empleados, y ciertas decisiones de implementación que se han tomado a lo largo del desarrollo del proyecto.

Seguidamente, en *Verificación y validación* se detallan las pruebas realizadas para asegurar un correcto funcionamiento del resultado del desarrollo.

Finalmente, en el capítulo de *Conclusiones y trabajo futuro*, se exponen los resultados, impresiones y reflexiones alcanzadas una vez finalizado el proyecto y la estancia en prácticas, detallando así mismo posibles mejoras del proyecto que puedan ser implementadas en un futuro.

Capítulo 2

Planificación del proyecto

Este capítulo pretende aportar una visión general de la etapa de planificación, describiendo primeramente la metodología para a continuación exponer la planificación temporal y el desglose de tareas. Por último, se realiza una estimación de los recursos necesarios para llevar a cabo el proyecto, junto a su seguimiento.

Esta etapa cuenta con gran importancia en el proyecto, pues la precisión con la que se desglosen las tareas y estimen los recursos influirá en el desarrollo del mismo.

2.1. Metodología de trabajo

En un primer momento y con el objetivo de conseguir una rápida adopción de los estándares y convenios utilizados por la empresa en su día a día, se realizó una primera etapa de formación donde se puso especial énfasis en los paradigmas de trabajo, en las tecnologías a utilizar, y en los estándares utilizados en las mismas

Una vez finalizada esta etapa inicial, y tras establecer con la empresa algunos detalles referentes al proyecto, se comenzó con la etapa de implementación.

En este caso, la empresa apuesta por el uso de metodologías ágiles, donde los requisitos y soluciones evolucionan con el tiempo según la necesidad del proyecto [12]. En concreto, la empresa decidió el uso de SCRUM. Dado que la empresa se suele encontrar en un entorno con requisitos cambiantes, la adopción de estas metodologías permite obtener flexibilidad, efectividad y calidad al producirse un cambio de requisitos.

Periódicamente, la empresa se reúne con el cliente para mostrar los avances producidos, permitiendo detectar los errores existentes de forma temprana, sin que supongan un cambio radical en el proyecto.

En la Figura 2.1 se muestran las diferentes partes del ciclo de vida de SCRUM, que se detallan a continuación:

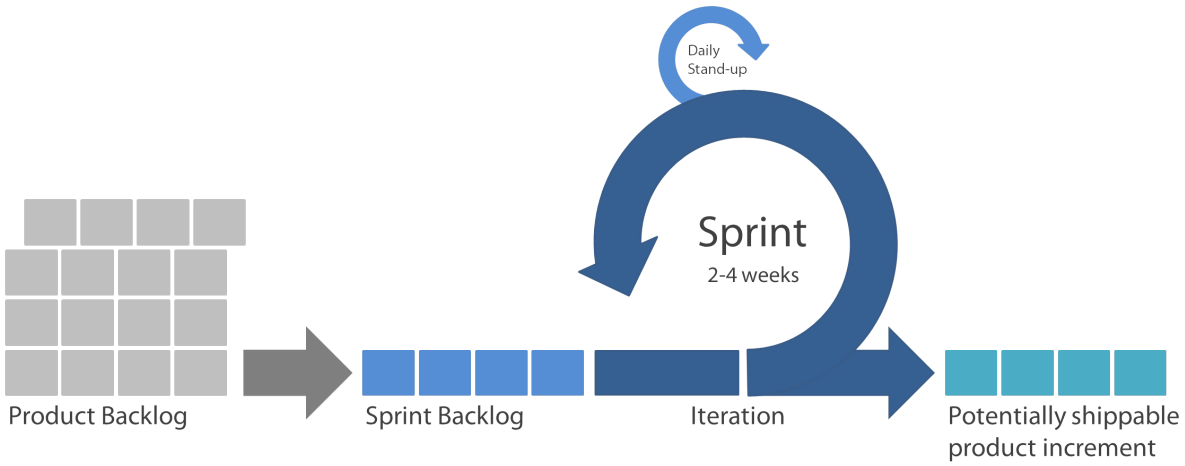


Figura 2.1: Representación del ciclo de vida Scrum

1. *Product Backlog*: Pila de requisitos que debe satisfacer el sistema. Puede evolucionar durante todo el ciclo de vida hasta el cierre del desarrollo.
2. *Sprint Backlog*: Pila de tareas que se desarrollarán en el Sprint actual. Pueden estar priorizadas según la necesidad del usuario y/o la complejidad de las mismas.
3. *Sprint*: Iteración que puede durar entre una y cuatro semanas en la que se desarrollan las tareas definidas en el *sprint backlog*, y tras la que el producto incrementa funcionalmente, añadiendo valor respecto a la iteración anterior. En este caso, la empresa opta por sprints de 3 semanas.

Adicionalmente, en la empresa se realizan una serie de reuniones: una reunión de revisión y retrospectiva del sprint en la que se presentan y discuten los resultados e impresiones del último sprint, una reunión de planificación, en la que se selecciona el subconjunto de requisitos de la pila del producto que formarán la pila del sprint, y reuniones diarias o *daily meetings* de quince minutos en las que se facilita la transferencia de información y comunicación poniendo de manifiesto puntos que pueden ayudar a otros compañeros.

Estas reuniones son llevadas a cabo por los diferentes roles que forman parte de la metodología SCRUM, detallados a continuación:

- *Scrum Manager*: Responsable del proyecto y encargado de coordinar al equipo, detectando además los problemas que puedan llegar a surgir.
- *Product Owner*: Cliente o usuario del sistema que decide la funcionalidad del producto.
- *Team*: Equipo que realiza les diferentes tareas de construcción del producto
- *Stakeholders*: Interesados en el proyecto, tanto internos como externos.

Cabe destacar que, tal y como se comenta en el siguiente capítulo, la empresa dispone de la herramienta Jira, que ofrece la funcionalidad necesaria para gestionar las iteraciones de los diferentes proyectos que se desarrollen.

2.2. Planificación temporal

Aunque el proyecto se ha desarrollado mediante una metodología ágil, por claridad este proyecto se documenta con un proceso de ingeniería de *software* clásico, de forma que se pueden observar cuatro partes totalmente diferenciadas:

- **Análisis de requisitos**: se realiza un estudio de las necesidades del cliente y de los actores que influyen en el futuro sistema, obteniendo como resultado un diagrama de casos de uso, una serie de requisitos de usuario y diversos prototipos que marcarán cómo será la interfaz del sistema.
- **Diseño e implementación**: diseño de la arquitectura del sistema a alto nivel, junto a un diseño a nivel de componentes, diseño de clases e identificación de patrones de diseño, entre otros. Se obtiene el diseño real de la interfaz de usuario a partir de los prototipos, llevando a cabo la construcción del sistema.
- **Validación y verificación**: comprobación del sistema en diferentes escenarios mediante una serie de pruebas unitarias, donde se comprueban algunos de los componentes del sistema de forma independiente, y pruebas de integración, donde se enlazan los distintos componentes para asegurar que éstos cooperan correctamente.
- **Despliegue y puesta en marcha**: lanzamiento del producto en un entorno real. Puede dividirse en dos etapas: Lanzamiento del producto a un grupo reducido de usuarios o *testers* que reportan errores encontrados, y lanzamiento final a todos los usuarios.

La planificación temporal fue definida conjuntamente con la empresa, estableciendo unos plazos y horario de trabajo en los que completar el proyecto, definiendo además las tareas a realizar.

2.2.1. Calendario de trabajo

Con el fin de estimar los costes temporales, se propuso que el alumno trabajara en la empresa de lunes a viernes de 8:30 a 13:30, realizando trescientas horas de trabajo, iniciándose la estancia el 14 de Febrero, y finalizando la misma el 21 de Mayo.

2.2.2. Planificación temporal

Aunque se han utilizado metodologías ágiles durante el desarrollo del proyecto, a continuación se muestra el EDT equivalente a la descomposición de las tareas en los diferentes *sprints*. Además, se incluyen tareas relacionadas con la redacción de la propuesta técnica y la formación en las tecnologías utilizadas en la empresa.

Las tareas realizadas se pueden ver en el esquema de descomposición de trabajo (EDT) de la Figura 2.2.

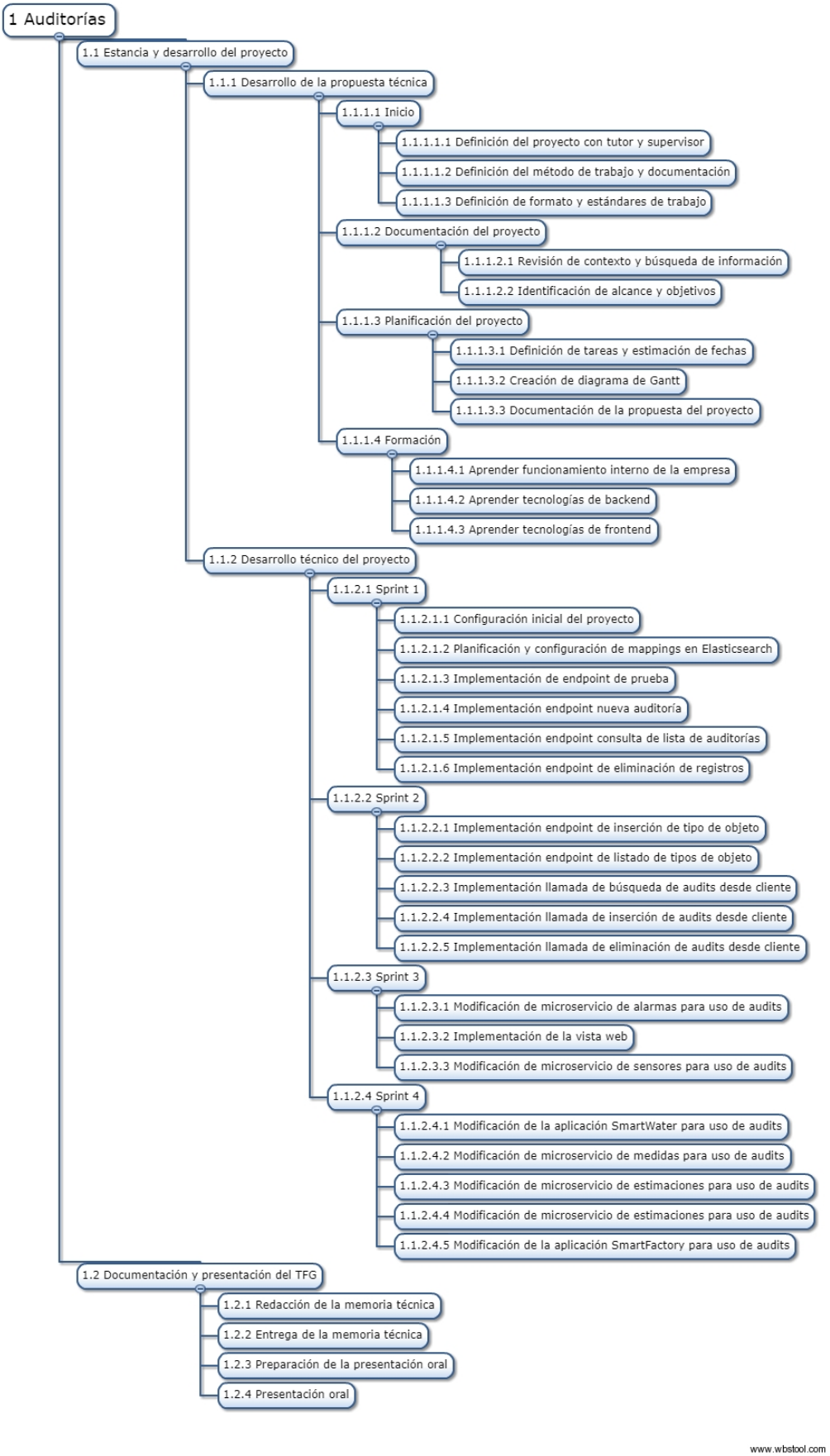


Figura 2.2: Diagrama EDT del proyecto.

En la siguiente tabla se muestra la estimación temporal de las tareas llevadas a cabo:

EDT	Nombre	Duración	Comienzo	Fin	Predecesoras
1	**Auditorías**	**513 horas**	**mié. 14/02/18**	**mar 22/05/18**	
1.1	**Estancia y desarrollo del proyecto**	**392 horas**	**mié. 14/02/18**	**vie. 27/04/18**	
1.1.1	**Desarrollo de la propuesta técnica**	**92 horas**	**mié. 14/02/18**	**jue. 01/03/18**	
1.1.1.1	**Inicio**	**11 horas**	**mié. 14/02/18**	**jue. 15/02/18**	
1.1.1.1.1	Definir proyecto con tutor y supervisor	1 hora	mié. 14/02/18	mié. 14/02/18	
1.1.1.1.2	Definir método de trabajo y documentación	5 horas	mié. 14/02/18	mié. 14/02/18	1.1.1.1.1
1.1.1.1.3	Definir formato y estándares de trabajo	5 horas	mié. 14/02/18	jue. 15/02/18	1.1.1.1.2
1.1.1.2	**Documentar el proyecto**	**24 horas**	**jue. 15/02/18**	**mar 20/02/18**	
1.1.1.2.1	Revisar contexto y buscar información	16 horas	jue. 15/02/18	lun. 19/02/18	1.1.1.1
1.1.1.2.2	Identificar alcance y objetivos	8 horas	lun. 19/02/18	mar 20/02/18	1.1.1.2.1
1.1.1.3	**Planificar el proyecto**	**32 horas**	**mar 20/02/18**	**lun. 26/02/18**	
1.1.1.3.1	Definir tareas y estimar fechas	8 horas	mar 20/02/18	mié. 21/02/18	1.1.1.2
1.1.1.3.2	Crear diagrama de Gantt	8 horas	mié. 21/02/18	jue. 22/02/18	1.1.1.3.1
1.1.1.3.3	Documentar la propuesta del proyecto	16 horas	jue. 22/02/18	lun. 26/02/18	1.1.1.3.2
1.1.1.3.4	Entregar la propuesta técnica	0 horas	lun. 26/02/18	lun. 26/02/18	1.1.1.3.3
1.1.1.4	**Formación**	**25 horas**	**lun. 26/02/18**	**jue. 01/03/18**	
1.1.1.4.1	Funcionamiento interno de la empresa	5 horas	lun. 26/02/18	lun. 26/02/18	1.1.1.3
1.1.1.4.2	Aprender las tecnologías de backend	10 horas	mar 27/02/18	mié. 28/02/18	1.1.1.4.1
1.1.1.4.3	Aprender las tecnologías de frontend	10 horas	mié. 28/02/18	jue. 01/03/18	1.1.1.4.2
1.1.2	**Desarrollo técnico del proyecto**	**300 horas**	**jue. 01/03/18**	**vie. 27/04/18**	
1.1.2.1	**Sprint 1**	**70 horas**	**jue. 01/03/18**	**jue. 15/03/18**	
1.1.2.1.1	Planificación del sprint	5 horas	jue. 01/03/18	vie. 02/03/18	1.1.1.4
1.1.2.1.2	Configuración inicial del proyecto	5 horas	vie. 02/03/18	vie. 02/03/18	1.1.2.1.1
1.1.2.1.3	Configuración inicial de mappings en elasticsearch	20 horas	vie. 02/03/18	jue. 08/03/18	1.1.2.1.2
1.1.2.1.4	Implementación del endpoint de nueva auditoría	20 horas	jue. 08/03/18	lun. 12/03/18	1.1.2.1.3
1.1.2.1.5	Implementación del endpoint de consulta de lista de auditorías	10 horas	lun. 12/03/18	mar 13/03/18	1.1.2.1.4
1.1.2.1.6	Implementación del endpoint de eliminación de registros	10 horas	mié. 14/03/18	jue. 15/03/18	1.1.2.1.5
1.1.2.2	**Sprint 2**	**75 horas**	**jue. 15/03/18**	**jue. 29/03/18**	
1.1.2.2.1	Implementación enpoint inserción de tipos de objeto	5 horas	jue. 15/03/18	jue. 15/03/18	1.1.2.1
1.1.2.2.2	Implementación endpoint inserción de tipo de objeto	3 horas	jue. 15/03/18	vie. 16/03/18	1.1.2.2.1
1.1.2.2.3	Implementación endpoint listado de tipos de objeto	2 horas	vie. 16/03/18	vie. 16/03/18	1.1.2.2.2
1.1.2.2.4	Implementación llamada de búsqueda desde cliente	25 horas	vie. 16/03/18	jue. 22/03/18	1.1.2.2.3
1.1.2.2.5	Implementación llamaba de inserción de audits desde cliente	20 horas	jue. 22/03/18	mar 27/03/18	1.1.2.2.4
1.1.2.2.6	Implementación llamada de eliminación de audits desde cliente	20 horas	mar 27/03/18	jue. 29/03/18	1.1.2.2.5
1.1.2.3	**Sprint 3**	**75 horas**	**jue. 29/03/18**	**vie. 13/04/18**	
1.1.2.3.1	Modificación de microservicio de alarmas para uso de audits	30 horas	jue. 29/03/18	vie. 06/04/18	1.1.2.2
1.1.2.3.2	Implementación de la vista web	25 horas	vie. 06/04/18	mié. 11/04/18	1.1.2.3.1
1.1.2.3.3	Modificación de microservicio de sensores para uso de audits	20 horas	mié. 11/04/18	vie. 13/04/18	1.1.2.3.2
1.1.2.4	**Sprint 4**	**80 horas**	**lun. 16/04/18**	**vie. 27/04/18**	
1.1.2.4.1	Modificación de la aplicación SmartWater para uso de audits	20 horas	lun. 16/04/18	mié. 18/04/18	1.1.2.3
1.1.2.4.2	Modificación de microservicio de medidas para uso de audits	10 horas	mié. 18/04/18	jue. 19/04/18	1.1.2.4.1
1.1.2.4.3	Modificación de microservicio de estimaciones para uso de audits	15 horas	jue. 19/04/18	lun. 23/04/18	1.1.2.4.2
1.1.2.4.4	Modificación de microservicio de estimaciones para uso de audits	15 horas	lun. 23/04/18	mié. 25/04/18	1.1.2.4.3
1.1.2.4.5	Modificación de la aplicación SmartFactory para uso de audits	20 horas	mié. 25/04/18	vie. 27/04/18	1.1.2.4.4
1.2	**Documentación y presentación del TFG**	**121 horas**	**lun. 30/04/18**	**mar 22/05/18**	
1.2.1	Redacción de la memoria técnica	100 horas	lun. 30/04/18	jue. 17/05/18	1.1
1.2.2	Entrega de la memoria técnica	0 horas	jue. 17/05/18	jue. 17/05/18	1.2.1
1.2.3	Preparación de la presentación oral	20 horas	jue. 17/05/18	lun. 21/05/18	1.2.2
1.2.4	Presentación oral	1 hora	mar 22/05/18	mar 22/05/18	1.2.3

Cuadro 2.1: Distribución de tareas del proyecto.

Finalmente, en la Figura 2.3 se muestra el diagrama de Gantt correspondiente al proyecto.

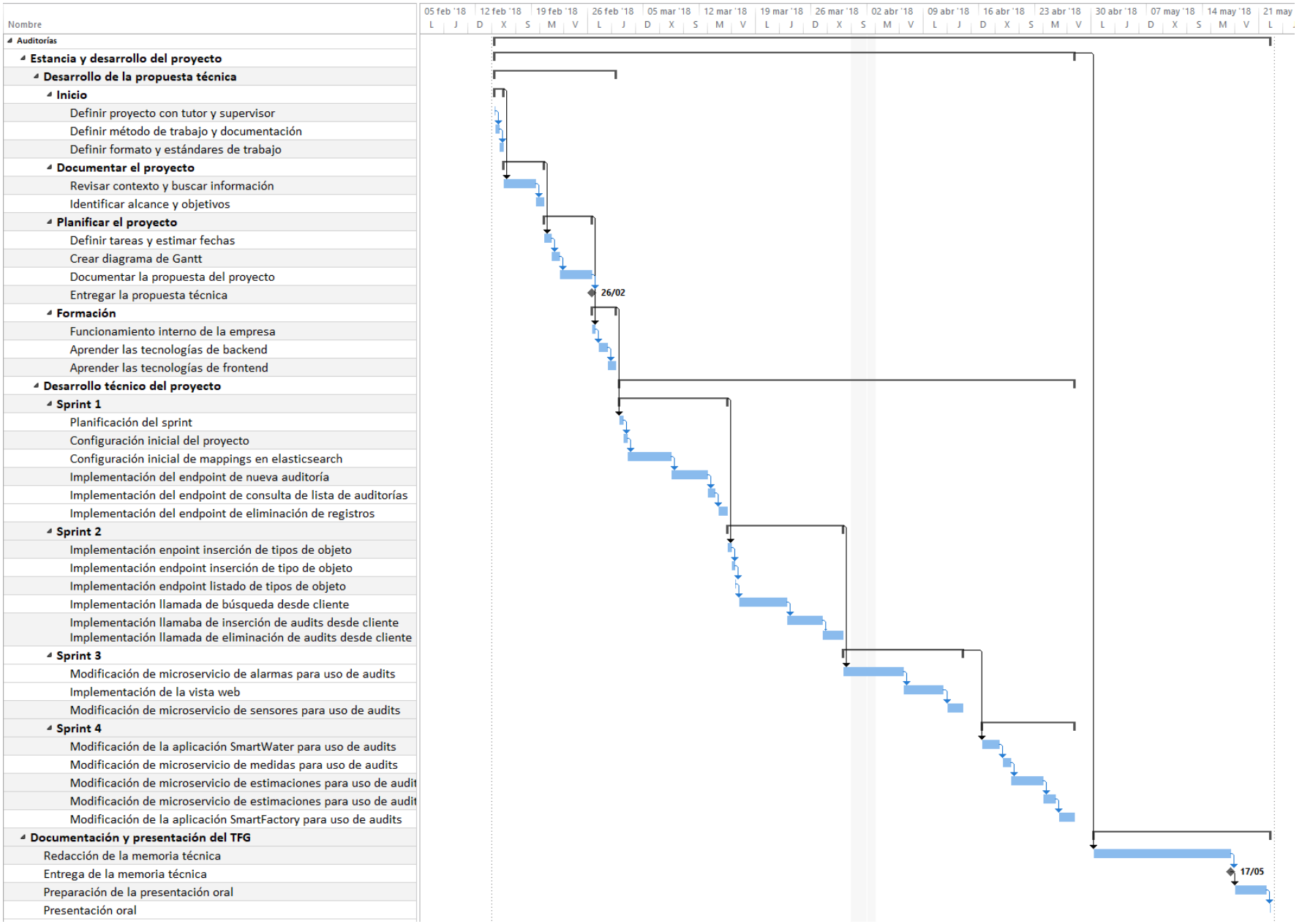


Figura 2.3: Diagrama de Gantt del proyecto.

2.3. Costes del proyecto

El coste total del proyecto se ha estimado suponiendo que el sueldo coincide con el sueldo medio de un trabajador Junior en España en 2018, es decir, de alrededor de unos 1200 mensuales para una jornada de 8 horas [6].

Aunque este salario no se corresponde con la remuneración de la empresa durante la estancia en prácticas, permite obtener una mejor visión de cuánto sería el coste del desarrollo en condiciones habituales. La estimación de los costes, incluyendo gastos adicionales, es la siguiente:

- Ordenador de sobremesa: 600€
- Licencia IntelliJ Ultimate (3 meses): 150€
- Programador: 7.2 €/hora ×300 horas = 2160€

Para calcular el coste total del proyecto se deben tener en cuenta las desviaciones y los costes indirectos, que supondremos de un 30 y un 20 por ciento, respectivamente:

Coste total:
(2160€×+ desviaciones (30 %) + costes indirectos (20 %))= 3240€

Conviene tener en cuenta que este precio no contiene los gastos de soporte y ampliaciones posteriores. En tal caso, la empresa cobra al cliente una cantidad a especificar por cada jornada de trabajo adicional de los programadores. Tampoco se incluye en el precio final el coste del ordenador y licencias de software, pues se usarán en otros proyectos y por tanto se amortizará su coste.

Capítulo 3

Tecnologías utilizadas

En este capítulo se presentan las diferentes tecnologías que se han utilizado durante el desarrollo de la aplicación, así como algunos estándares, convenios y paradigmas adoptados por la propia empresa.

La mayoría de tecnologías utilizadas durante el desarrollo del proyecto han sido impuestas por la empresa, ya que se utilizan en el resto de aplicaciones ofrecidas en IoTsens, lo que favorece la consistencia de la arquitectura y la reducción de la curva de aprendizaje tanto en la incorporación de nuevos miembros en el equipo y en el desarrollo de nuevas aplicaciones.

Cabe destacar que no se muestran únicamente tecnologías y herramientas utilizadas para el desarrollo del código, sino también las utilizadas para la gestión y documentación de los diferentes proyectos. No obstante, debido al gran número de tecnologías empleadas, sólo se enseñan las más significativas.

3.1. Planificación y documentación

Tal y como se ha mencionado en el capítulo dedicado a la metodología de trabajo, la empresa ha adoptado la metodología ágil SCRUM. Con el objetivo de facilitar la gestión y seguimiento de los diferentes proyectos, se cuenta con la herramienta Jira.

Además, con el fin de documentar el desarrollo, se dispone también de la herramienta Confluence, que permite a los programadores disponer de toda la información necesaria en páginas de formato *wiki* donde se detallan aspectos como las tecnologías utilizadas, información relevante de la arquitectura, o ejemplos de utilización de librerías, entre otros.

3.1.1. Jira

Jira es una plataforma web de la compañía Atlassian destinada a la gestión integral de metodologías ágiles. Esto permite que el *Scrum Manager* pueda gestionar por completo todo el

trabajo a realizar, y distribuir las distintas tareas entre los diferentes integrantes del equipo de desarrollo.

Adicionalmente, esta plataforma implementa un tablero *Kanban* de igual forma que un sistema convencional de tarjetas adhesivas, representando las tareas a realizar mediante la forma de una tarjeta en la que aparece de forma visual e intuitiva el tipo de tarea mediante iconos (historia de usuario, tarea, subtarea, incidencia, mejora,...). También contiene el identificador de la tarea, un título descriptivo, la foto de perfil del responsable de la tarea y los puntos de historia relacionados con su estimación.

Cabe destacar que, en caso de que una historia de usuario sea excesivamente amplia, ésta puede ser descompuesta en diferentes sub-tareas.

La herramienta permite distribuir las tarjetas en múltiples columnas personalizables que marcan el estado de cada tarea. En este caso la empresa ha convenido en utilizar las siguientes:

- Por hacer: Estado inicial de una tarea al iniciar un sprint. Una tarea permanece en este estado hasta que alguien decide comenzar a implementar la funcionalidad asociada a la misma.

- En progreso: Tareas empezadas por el usuario asignado y que se está implementando en ese instante de tiempo. Es habitual que no haya más de una tarea en este estado de un mismo usuario al mismo tiempo.

- En pruebas: Indica que la tarea ha sido acabada y en ese momento otro desarrollador está verificando que cumple la funcionalidad y calidad deseada.

- Hecho: Indica que la tarea ya ha sido verificada y que, por tanto, se puede dar por cerrada.

En su uso habitual, el *Scrum Manager* crea sprints definiendo su fecha de inicio y final, y se asignan tareas a cada uno de los distintos desarrolladores. Además, la herramienta permite obtener diferentes informes y gráficas que muestran cómo ha sido el desarrollo del sprint.

La Figura 3.1 muestra un ejemplo del *backlog* una vez iniciado el sprint, mientras que la Figura 3.2 enseña un ejemplo de cómo se vería un informe una vez iniciado el sprint, donde la línea roja marca los puntos de historia pendientes de completar, y la línea gris indica los puntos de historia ideales que faltarían por completar en una situación ideal.

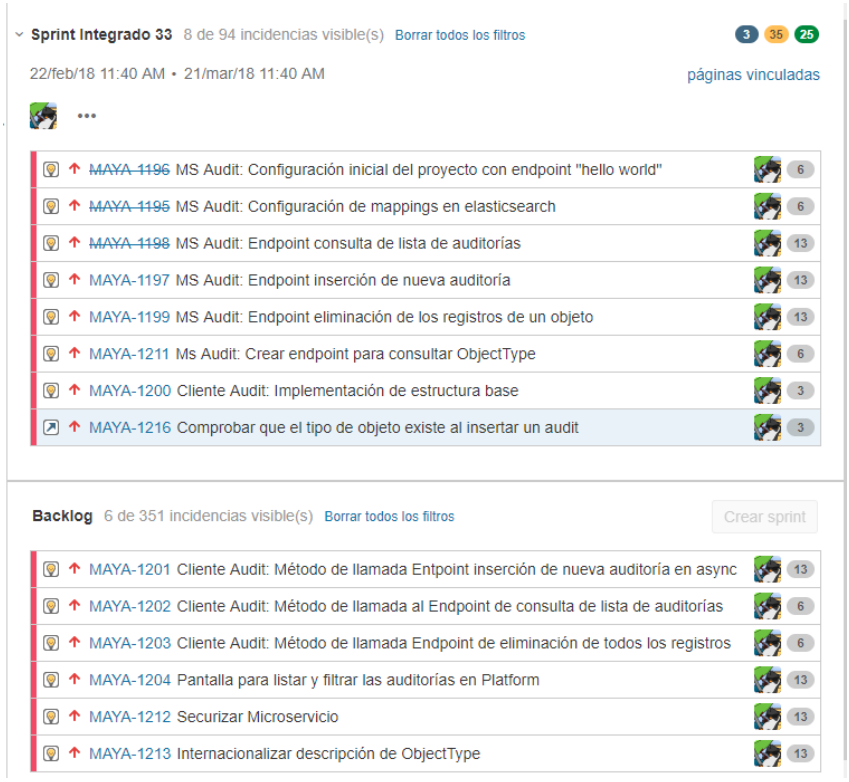


Figura 3.1: Visualización del *backlog* en Jira.

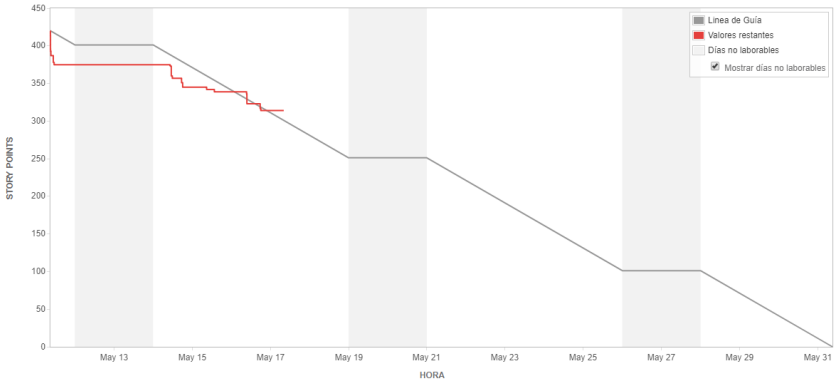


Figura 3.2: Ejemplo de informe de Jira una vez empezado el Sprint.

3.1.2. Confluence

Confluence es otra herramienta de la suite Atlassian que proporciona la funcionalidad de una una *wiki*, páginas web de carácter informativo, editables desde el navegador, donde son los propios desarrolladores los creadores de contenido.

Una de las principales ventajas de una *wiki* es que permite crear y mejorar las páginas de forma inmediata por medio de una interfaz realmente simple, dando una gran libertad al

usuario.

Esto hace que más gente participe en su modificación y comparta sus conocimientos, a diferencia de los sistemas tradicionales, donde resulta más difícil que los usuarios del sitio contribuyan a mejorarlo.

Además, la empresa cuenta con una cultura donde se valora altamente la documentación del código, y de toda aquella información que pueda resultar de interés al resto de usuarios. Por ejemplo, en la Figura 3.3 se observa un ejemplo de página en Confluence, en este caso dedicado al uso de la herramienta Git.

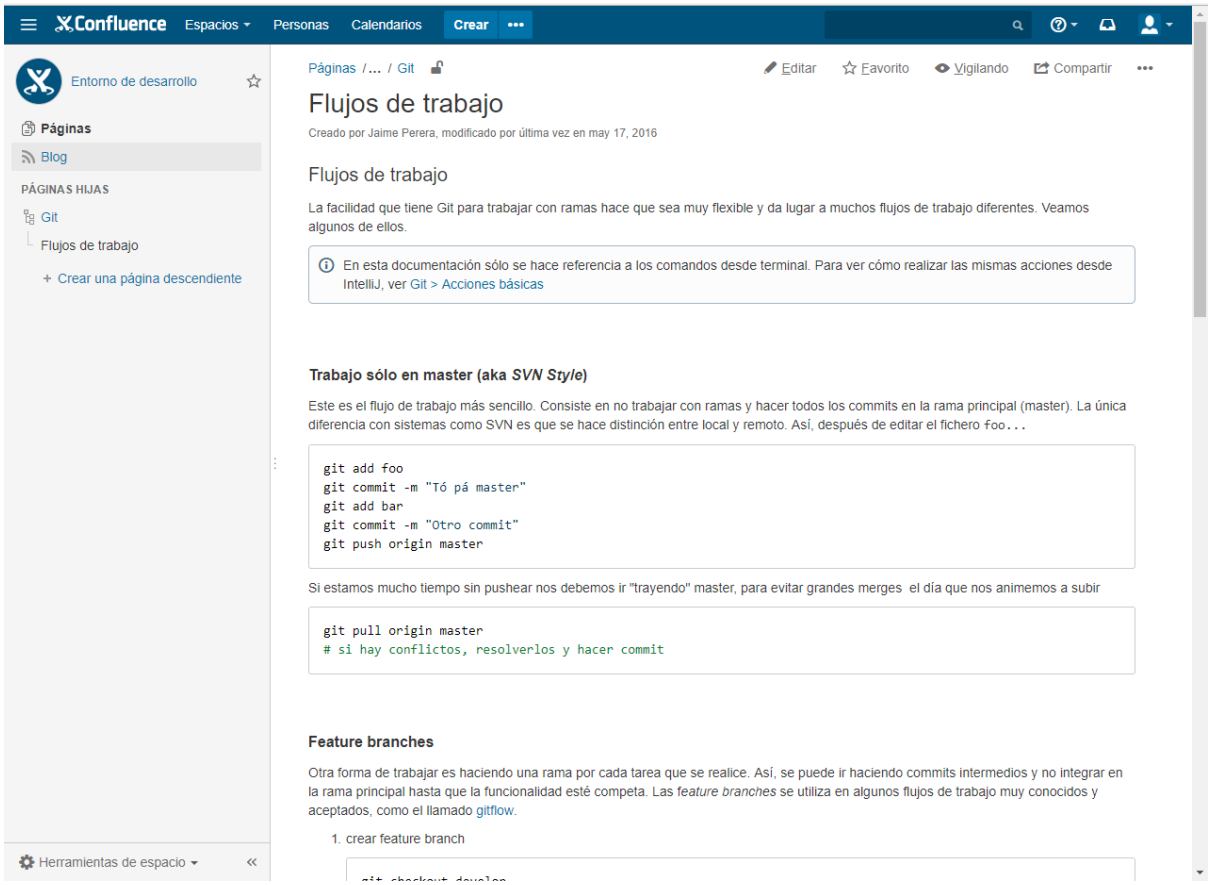


Figura 3.3: Página en Confluence sobre el flujo de trabajo con Git.

El uso de Confluence ha resultado gratamente satisfactorio, siendo realmente útil a la hora de encontrar todo tipo de información relativa a bibliotecas internas de la empresa, ahorrando mucho del tiempo que de otra forma se perdería a la hora de trabajar con muchos de los componentes internos.

Es de especial interés mantener dicha información para que nuevos empleados puedan aprender fácilmente muchos de los detalles internos de la empresa, reduciendo la curva de aprendizaje y mejorando notablemente la productividad de los trabajadores.

3.2. Estándares y paradigmas

A continuación se indican los estándares y paradigmas que han sido utilizados a lo largo largo del desarrollo del proyecto. Éstos han sido adoptados por la empresa y se encuentran orientados al sector de los servicios Web.

3.2.1. REST

Representational State Transfer (REST) es un estilo de arquitectura de *software* que define un conjunto de propiedades y restricciones basadas en HTTP [9]. Esto supone una arquitectura cliente-servidor en la que éste último no almacena ningún estado. El cliente, por su parte, realiza peticiones CRUD HTTP mediante los métodos GET, POST, PUT, DELETE, que el servidor interpreta devolviendo la llamada pertinente para cada caso.

Dicha llamada depende además de la URL en la que se realice la petición. Dependiendo de en qué URL se realice la llamada, el servidor realizará una operación u otra.

En la mayoría de aplicaciones de la empresa se utiliza esta tecnología para realizar tareas de comunicación entre la vista de una aplicación web y su *back end* o entre este último y los micro-servicios que le pueden proporcionar funcionalidades adicionales. En este caso, la empresa suele utilizar JSON (*JavaScript Object Notation*), un subconjunto de la notación literal de objetos de JavaScript, para formatear las respuestas y, en caso de que fuese necesario, el mensaje de la petición.

A continuación se muestra un ejemplo de petición REST al micro-servicio de auditorías que se ha desarrollado durante el proyecto. En este caso, se pretende obtener los detalles de un tipo de objeto a auditar. Este proceso se divide en las siguientes tres etapas:

1. El cliente realiza una petición GET a la URL correspondiente a la operación deseada en el micro-servicio de auditorías.

   `http://172.20.16.228:31107/v1/objectTypes/5`

2. El servidor recibe la petición, realiza las consultas correspondientes en las bases de datos, y devuelve la siguiente respuesta JSON:


```
{
    "success": true,
    "msg": null,
    "data": {
        "id": 5,
        "name": "EVENT_GENERATOR",
        "description": "Event Generator. It may be boolean, exact value, lower, upper or inactivity based."
    },
```


Figura 3.4: Resultado de la llamada REST para obtener los detalles de un tipo de objeto.

3. El cliente que ha realizado la llamada recibe la respuesta, continuando con su ejecución.

En este caso, el recurso en el micro-servicio se ha implementado utilizando las herramientas proporcionadas por el *framework* Spring Boot. Más adelante se muestra una parte del código implementado para proporcionar el recurso que se ha utilizado en este ejemplo.

3.2.2. AJAX

Tradicionalmente el cliente web solicitaba una página completa ya lista para mostrar. No obstante, actualmente el uso de AJAX, una serie de técnicas de desarrollo que permiten la creación de clientes asíncronos, permite que las aplicaciones envíen y reciban datos sin afectar a la visualización de la página existente, es decir, sin la obligación de recargar la página completa [2].

En este caso, las aplicaciones se ejecutan en el navegador de los usuarios, manteniendo una comunicación asíncrona con el servidor en segundo plano y realizando las peticiones oportunas para obtener los datos que se necesiten sin realizar molestos refrescos de página, tal y como se muestra en la Figura 3.5 [13].

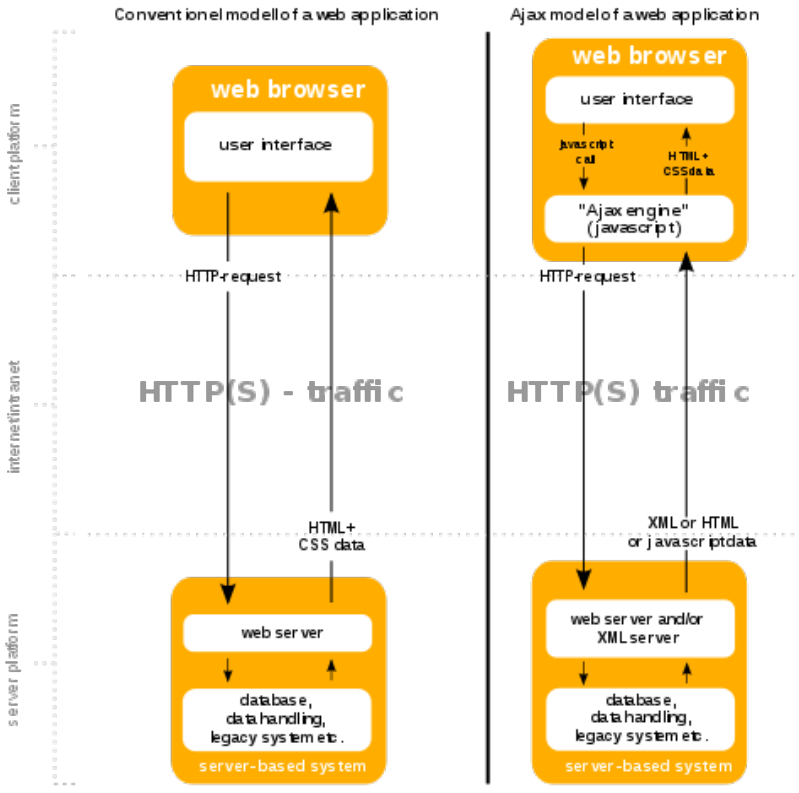


Figura 3.5: Representación del funcionamiento de AJAX

3.2.3. TDD

Durante la implementación del *back end* ha llegado a resultar útil el uso de TDD (*Test Driven Development*), es decir, el desarrollo guiado por pruebas.

TDD es una práctica de ingeniería del software que involucra crear pruebas de software antes incluso de desarrollar dicho software, con la finalidad de luego implementar la funcionalidad deseada, *refactorizando* en cada iteración.

Debido a que supone un gran esfuerzo temporal, no se ha utilizado esta metodología excepto en las ocasiones más sensibles y en las que se requería de una visión temprana de ciertos requisitos, como los métodos de acceso a la base de datos, con el fin de obtener una aplicación más robusta y de más calidad.

3.3. Herramientas de desarrollo

En este apartado se describen las distintas herramientas que han sido utilizadas durante el desarrollo de proyecto. Estas han sido adoptadas por la empresa y propuestas por la misma debido a las facilidades que aportan en el momento de desarrollo, integración y despliegue del proyecto.

3.3.1. IntelliJ

El IDE o entorno de desarrollo integrado que se ha elegido ha sido IntelliJ IDEA 2018. Este IDE se enfoca principalmente en analizar el código, buscando conexiones entre símbolos a lo largo de todos los ficheros y lenguajes del proyecto, usando esta información para proporcionar asistencia de código, navegación rápida, análisis de errores y distintas opciones de *refactorización*.

La versión *Ultimate*, además, permite integrar cómodamente lenguaje Java con Typescript, el lenguaje utilizado para el desarrollo de la vista, lo que ha resultado especialmente cómodo. Concretamente, se ha utilizado *Java Development Kit* 8 (JDK8) y Typescript 2.4.

3.3.2. Git y GitLab

Con el objetivo de controlar las distintas versiones del código de forma fácil se utiliza Git, un sistema distribuido de control de versiones de código libre que se encuentra diseñado para ser utilizado tanto en proyectos pequeños como grandes con una gran velocidad y eficiencia.

Al tratarse de un sistema distribuido, la empresa cuenta con un servidor GitLab en el que poder alojar el código, de tal forma que los distintos desarrolladores puedan tenerlo siempre disponible. Además, GitLab ofrece las herramientas necesarias para la implementación del

workflow Gitflow, que define un modelo de ramificaciones alrededor de las *release* del proyecto, lo que proporciona un *framework* robusto con el que gestionar grandes proyectos, tal y como se muestra en la Figura 3.6.

Según este *workflow* el trabajo se organiza en varias ramas. En la rama *master* debe encontrarse el código que se considera listo para subir a producción. En la rama *develop* se incluye el código que formará parte de la siguiente versión del proyecto.

Además, se crean diversas ramas auxiliares. Cada nueva funcionalidad o *feature* se desarrolla en una rama independiente que posteriormente, cuando se encuentre finalizada, se unirá a *develop*. Además, se pueden crear ramas específicas para solucionar errores en producción o ramas específicas para cada *release*.

Figura 3.6: Esquema de uso de Gitflow

3.3.3. Jenkins

Jenkins es un servidor de *continuous delivery* que permite automatizar las tareas no-humanas del proceso de desarrollo mediante integración continua.

En la empresa Jenkins se encuentra configurado de tal forma que se lanza el proceso de construcción cada vez que se produce un *merge* (fusión de dos ramas) contra la rama `develop` en el servidor de GitLab.

Esto resulta especialmente interesante al poder ser usado homogéneamente con Gitflow, lo que facilita el proceso a los desarrolladores. Además, permite desplegar las dependencias de los proyectos automáticamente, ejecutar análisis de código mediante diferentes *plug-in* y ejecutar otros proyectos automáticamente cada cierto tiempo, como test de interfaz, para asegurar un correcto funcionamiento de toda la plataforma.

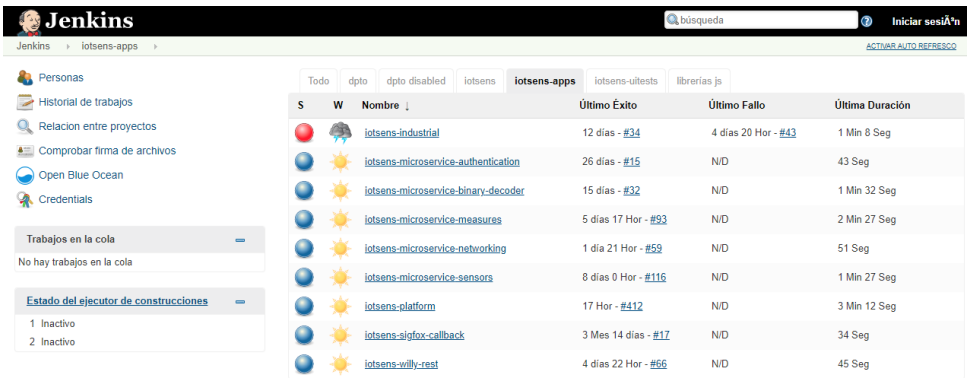


Figura 3.7: Pantalla principal del servidor Jenkins.

3.4. Tecnologías del servidor y acceso a datos

Las tecnologías aquí presentes se han utilizado en el *back end* del proyecto, incluyendo micro-servicio, cliente API del micro-servicio, servidor de la plataforma y sistemas de almacenamiento de datos.

3.4.1. Spring Boot

Spring Boot es un *framework* enfocado en el desarrollo de aplicaciones de forma rápida y fácil. Cuenta con diferentes módulos, que al ser importados permiten ampliar la funcionalidad de la aplicación con herramientas como un servidor Tomcat integrado, JPA, o JDBC, entre otros.

A modo de ejemplo, en la Figura 3.8 se enseña el código necesario para implementar el servicio REST encargado de recibir las llamadas desde la vista. En este caso, la llamada devuelve los detalles de un tipo de objeto a auditar (método `getObjectType`).

```java
@RestController
@RequestMapping("/objectTypes")
public class ObjectTypeResource {

    @Autowired
    ObjectTypeService objectTypeService;

    @GetMapping
    public RestResponse<List<ObjectType>> getObjectTypes() {
        List<ObjectType> objectTypes = objectTypeService.getObjectTypes();
        return new RestResponse<>(objectTypes, objectTypes.size());
    }

    @GetMapping("{id}")
    public RestResponse<ObjectType> getObjectType(@PathVariable Long id) {
        ObjectType objectType = objectTypeService.getObjectType(id);
        return new RestResponse<>(objectType);
    }
}
```

Figura 3.8: Ejemplo de *endpoint* REST en Spring Boot

3.4.2. Apache Maven

Maven es una herramienta dedicada a la gestión y construcción de proyectos Java. El uso de esta herramienta es especialmente interesante, ya que facilita procesos muy comunes como el nombrado de versiones, compilación y ejecución del código y, especialmente, la resolución de dependencias.

Las dependencias, que se encuentran declaradas en el fichero `pom.xml`, son buscadas en *Maven Central*, los repositorios por defecto, y en otros repositorios que se hayan especificado, y descargadas para ser utilizadas automáticamente en el proyecto.

3.4.3. MySQL

MySQL es un sistema de gestión de bases de datos relacionales propiedad de Oracle. Adicionalmente, en la empresa se cuenta con un cliente PHPMyAdmin, una interfaz visual que permite administrar las diferentes bases de datos de las aplicaciones y micro-servicios existentes, permitiendo ejecutar consultas fácilmente, o sentencias SQL si se necesitase.

Para realizar la conexión entre el proyecto y la base de datos se ha utilizado Hibernate y QueryDSL. Hibernate es una herramienta basada en el estándar JPA que permite un mapeado automático de las clases Java a un modelo relacional, cuyo *schema* puede ser configurado mediante anotaciones en el propio código o mediante ficheros XML. Por otra parte, QueryDSL es un *framework* de Java que permite la generación de consultas con seguridad respecto al tipo de objetos que cuenta con una estructura similar al lenguaje SQL, reduciendo la posibilidad de realizar operaciones sintácticamente inválidas y el tiempo dedicado a su definición.

No obstante, MySQL únicamente se utiliza cuando el volumen de datos es pequeño o mediano. En casos donde el volumen de datos es muy grande se utiliza Elasticearch.

3.4.4. Elasticsearch

Elasticsearch es un motor de búsqueda basado en Lucene [1] con un sistema distribuido mediante una interfaz HTTP y documentos JSON que, mediante su cliente Java, permite realizar búsquedas en un excepcional tiempo, lo que se consigue mediante su sistema de índices y *shards*.

Los índices (equivalente a tablas en SGBD relacionales) se dividen en *shards*, y cada uno de ellos puede contener ninguna o más de una réplicas. Cada nodo puede almacenar uno o más *shards* y actúa como coordinador para delegar operaciones a los *shards* correctos. Los datos relacionados se suelen almacenar en el mismo índice, que una vez creado no puede ser cambiado.

Debido a la gran cantidad de filtros de búsqueda que se deben permitir al usuario, el uso de un motor de búsqueda tan especializado y rápido como este ha supuesto una gran rapidez en el uso de la aplicación, con búsquedas de pocos milisegundos aún con decenas de miles de datos disponibles.

3.4.5. JUnit

JUnit es un *framework* para la construcción de test unitarios en lenguaje Java. Mediante su uso se definen varios métodos de testeo, que se marcan mediante la anotación `@Test`. En dichos métodos se pueden realizar varias comprobaciones de los valores esperados en diferentes partes del mismo. En caso de que en todas las comprobaciones el valor resultante sea el esperado la prueba será exitosa, pero en caso de que exista al menos una comprobación en la que el valor recibido dista del esperado, el test habrá fallado.

Esto resulta especialmente útil para controlar las pruebas de regresión, permitiendo a los desarrolladores estar seguros de que las modificaciones de partes del código no afectan negativamente a la funcionalidad previa.

Durante el proyecto se ha utilizado JUnit en conjunción con Mockito, una herramienta que permite simular el comportamiento de objetos de otras clases de forma muy sencilla, aislando a una clase de sus dependencias, lo que permite asegurar que los errores encontrados durante las pruebas realmente se encuentran en los métodos comprobados y no en alguna de las clases de las que dependen.

También ha sido útil el uso de MockWebServer, una herramienta dedicada a la simulación de llamadas REST. Esto ha permitido simular las llamadas desde el cliente al micro-servicio de auditorías para comprobar que no existe ningún error en la lógica de este proceso.

Finalmente, para la comprobación de las pruebas de interfaz de usuario se ha utilizado JUnit en conjunción con Selenium, una herramienta que mediante su API para el lenguaje Java y sencillos *scripts* permite automatizar las acciones que realizaría un usuario sobre un navegador Web.

3.5. Tecnologías del cliente Web

A continuación se presentan las herramientas y tecnologías utilizadas para la elaboración del cliente y su vista, es decir, el *front end* del proyecto.

3.5.1. HTML5

HTML5 (*HyperText Markup Language*) es el estándar web más utilizado y reconocido por el World Wide Web Consortium, la organización encarada de gestionar las tecnologías utilizadas en la red.

Se ha utilizado este lenguaje para implementar los diferentes componentes de la vista para que ésta sea accesible desde un navegador web. Esto ha sido posible además gracias a la utilización de Angular.

3.5.2. CSS3 y SASS

Las hojas de estilo en cascada o CSS, por sus siglas en inglés, permiten definir el aspecto y estilos de un documento HTML.

En este caso, y con el objetivo de obtener un mayor dinamismo y eficiencia en la programación del aspecto web, se ha utilizado SASS (*Syntactically Awesome StyleSheets*), una extensión de CSS que permite el uso de variables, reglas anidadas o importaciones en una línea, entre otros, para conseguir que los estilos se encuentren bien organizados y en documentos de escaso tamaño.

Complementariamente se ha utilizado Bootstrap, una biblioteca que contiene plantillas de diseño de componentes HTML y CSS que se encuentra especialmente enfocado en conseguir consistencia entre todas las partes de la vista web.

Además, se emplean *CSS Media Queries*, una técnica que permite incluir un bloque de propiedades CSS cuando se cumplen una serie de condiciones. En este caso su uso conjunto con Bootstrap ha permitido desarrollar una vista web que adapta su estructura dependiendo del tamaño de la ventana, lo que permite que se pueda visualizar cómodamente tanto desde un navegador de escritorio como desde el navegador de un teléfono móvil, sin importar el sistema operativo del dispositivo.

3.5.3. TypeScript

Typescript es un lenguaje de programación libre y de código abierto desarrollado por Microsoft. Es un superconjunto de JavaScript, un lenguaje de programación orientado a objetos, interpretado y dialecto del estándar ECMAScript, al que añade tipificación estático y objetos basados en clases.

El código en TypeScript posteriormente se compila a JavaScript, para poder ser ejecutado en cualquier motor que soporte ECMAScript 3 o superior, en Node.js o, en este caso, en cualquier navegador web.

3.5.4. Angular 2

Angular 2 o simplemente Angular es la evolución de AngularJS, un *framework* de Javascript de código abierto mantenido por Google, que se utiliza para crear y mantener aplicaciones Web de una sola página con el objetivo de aumentar las aplicaciones basadas en navegador con capacidad de Modelo-Vista-Controlador (MVC).

Mediante Angular se implementan los distintos módulos y componentes (algunos reutilizables) que formarán la plantilla HTML final. Los componentes se comunican mediante AJAX con el servidor, actualizan los datos y refrescan automáticamente la pantalla, sin necesidad de recargarla.

Cabe destacar que este *framework* obliga a la utilización de TypeScript.

Adicionalmente, se han utilizado componentes de la biblioteca PrimeNG, una colección de componentes reutilizables y ya implementados, que permiten disminuir el tiempo de desarrollo y facilitar la integración en pantallas complejas.

Capítulo 4

Análisis y diseño del sistema

Este capítulo presenta un análisis de los requisitos del sistema desarrollado. Los requisitos representan la funcionalidad que debe ser implementada y han sido especificados conjuntamente con la empresa antes del desarrollo de los mismos.

Cabe destacar que a lo largo del desarrollo la lista de requisitos ha sufrido variaciones. No obstante, al estar utilizando una metodología ágil, esto no ha supuesto un mayor problema y los cambios se han podido resolver sin tener que retrasar de forma significativa el desarrollo.

4.1. Análisis del sistema

A continuación se especifican los distintos requisitos que el sistema debe satisfacer, tanto los requisitos funcionales como los requisitos de datos.

4.1.1. Historias de usuario

Las historias de usuario permiten definir en alto nivel los requisitos que deben ser satisfechos, incluyendo los diferentes actores que forman parte del mismo. La Tabla 4.1 muestra las historias en las que se basa este proyecto:

ID	Historia de usuario
HU01	Como usuario necesito que el sistema audite cada vez que realizo una acción.
HU02	Como administrador necesito listar todos los registros de las auditorías realizadas.
HU03	Como administrador necesito filtrar los registros existentes.
HU04	Como administrador necesito limitar la cantidad de registros mostrados.
HU05	Como administrador necesito eliminar las auditorías de un objeto determinado.
HU06	Como administrador necesito añadir nuevos tipos de objeto.
HU07	Como usuario necesito listar los tipos de objeto existentes.

Cuadro 4.1: Historias de usuario.

4.1.2. Diagrama de casos de uso

Los casos de uso del proyecto pueden observarse en conjunción con los actores que las realizan en un diagrama de casos de uso, que muestra las relaciones entre los mismos:

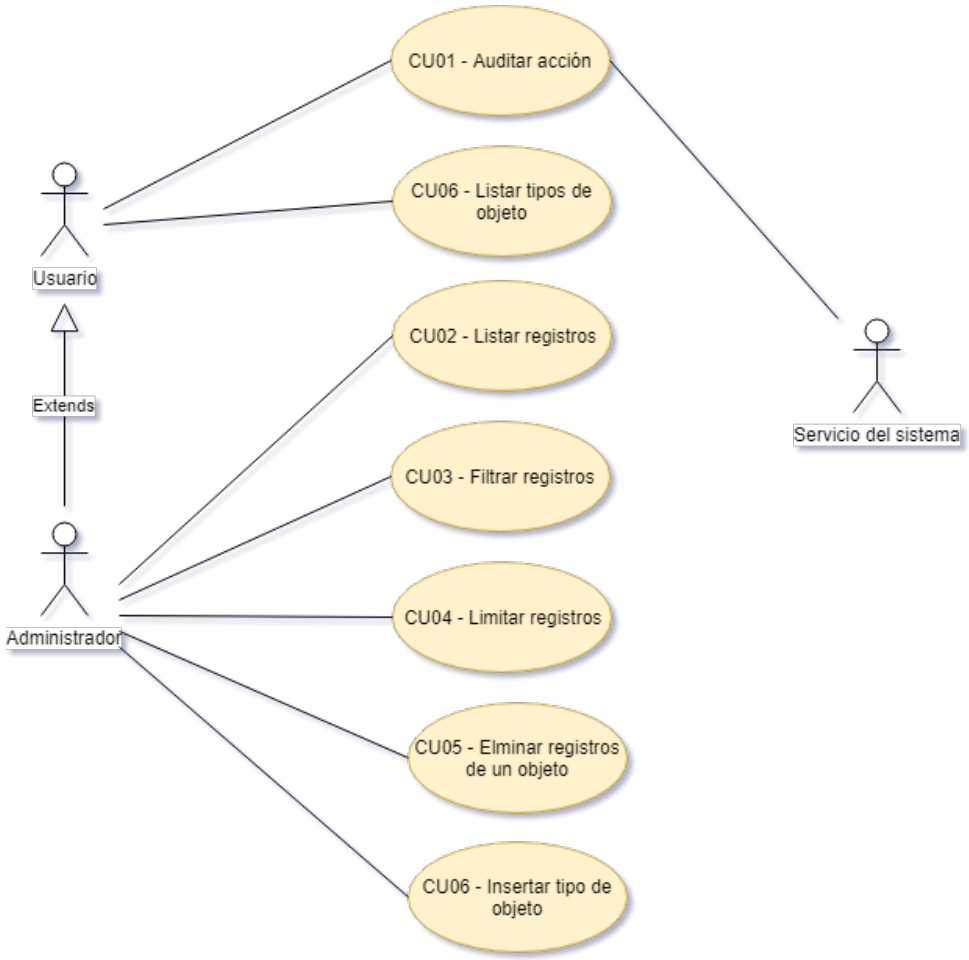


Figura 4.1: Diagrama de casos de uso

4.1.3. Requisitos funcionales

Los requisitos funcionales definen las características y alcance del sistema a desarrollar, como que se necesitará un filtrado de los registros mediante unos parámetros concretos, o cómo se gestionan estas llamadas cuando no se proporcionan los datos necesarios, por ejemplo.

Dichos requisitos, junto a otros datos relevantes, se encuentran adjuntos en el Anexo A.

4.1.4. Requisitos de datos

Los requisitos de datos se han definido a continuación con el objetivo de definir las entidades y atributos que se van a almacenar en el sistema. Éstos son obtenidos a partir de los casos de uso mencionados anteriormente.

En el proyecto desarrollado la cantidad de datos diferentes a almacenar no es tan amplia como en otro tipo de proyectos enfocados a la construcción de una aplicación web completa, pues al tratarse de una auditoría se intenta almacenar de forma genérica todos los datos, en lugar de almacenarlos de una forma concreta.

RD-1 Audit	
ID	RD-1
Nombre	Audit
Datos	Identificador, fecha, acción, nombre de usuario, id del objeto, nombre de la aplicación, tipo de objeto, datos nuevos y datos antiguos.
Comentarios	Cada *audit* se corresponde con una operación realizada en la plataforma de la empresa. El único dato necesario respecto al usuario, su nombre de usuario, se solicitará directamente a la API que la empresa proporciona para estos fines, lo que implica que no hará falta almacenar otra información sobre él. La acción podrá ser *CREATE, UPDATE* o *DELETE*. No se auditarán las consultas de los usuarios.

Cuadro 4.2: RD-1 Audit

RD-2 Tipo de objeto	
ID	RD-2
Nombre	Tipo de objeto
Datos	Identificador, nombre y descripción.
Comentarios	Los tipos de objeto indican el tipo de dato que ha sido cambiado en el sistema. Por ejemplo, al crear un nuevo sensor, el tipo de objeto será *SENSOR*.

Cuadro 4.3: RD-2 Tipo de objeto

4.1.5. Diagrama de clases

Un diagrama de clases ayuda a obtener una visión general de los requisitos de datos y las relaciones existentes entre ellos.

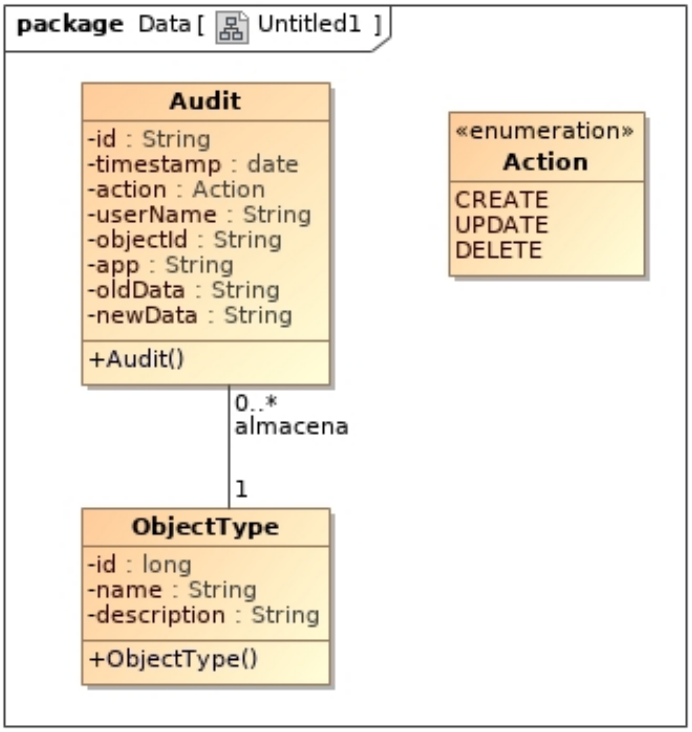


Figura 4.2: Diagrama de clases UML.

El diagrama de clases de la Figura 4.2 aporta la visión más sencilla de las relaciones entre los escasos datos que tienen un papel importante en el proyecto. Un registro de auditoría almacena toda la información necesaria para representar una acción y los datos que se han modificado en ella. El objeto del que se extraen dichos datos será de un tipo.

En el Anexo B se pueden encontrar los diagramas de clases que responden a la implementación final de las distintas partes del proyecto (micro-servicio y cliente del mismo) permitiendo obtener una visión más específica y detallada de la implementación. También se incluye otro diagrama de ejemplo que permita ilustrar cómo se integra el cliente en otras aplicaciones de la empresa, en concreto en el micro-servicio de *Event Definitions*.

4.1.6. Mockups

Los *mockups* se utilizan para obtener una aproximación del resultado visual que tendrá la aplicación final. No representan el aspecto definitivo, los elementos que aparecen en él pueden cambiar una vez empiece su implementación.

En este caso sólo se ha desarrollado un *mockup*, ya que desde un inicio se pensó que la parte visual de la aplicación debería concentrarse en una única pantalla, integrándose así en la plataforma central de la empresa.

La pantalla, mostrada en la Figura 4.3 permite, por un lado, obtener el listado de los registros

resultantes de las auditorías y, por otro lado, filtrar los registros mediante los criterios que se consideren oportunos.

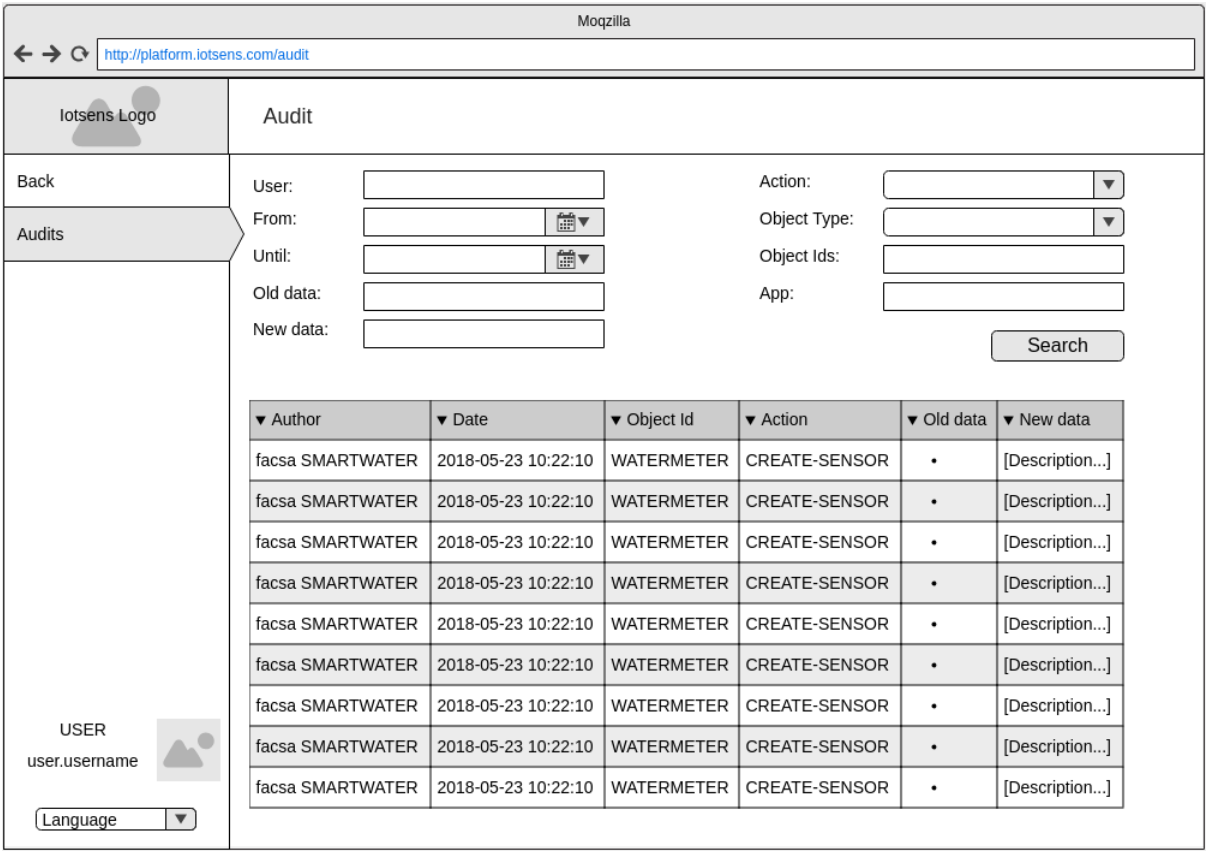


Figura 4.3: *Mockup* de la pantalla de auditorías.

Desde un primer momento, por criterio de la empresa, se decidió no ofrecer la posibilidad de insertar y/o eliminar registros desde la propia interfaz, permitiendo realizar estas operaciones únicamente mediante llamadas REST el hacia micro-servicio de auditorías.

Tras el diseño, para el que se ha utilizado la herramienta *Moqups* (`app.moqups.com`), quedó una idea más clara de cómo debería quedar visualmente la aplicación. El visto bueno de la empresa significó eliminar la posibilidad de empezar el desarrollo con la preocupación de tener que rediseñar la parte visual.

4.2. Arquitectura

La arquitectura define la estructura e interacción entre los distintos componentes que forman el sistema, centrándose en el aspecto global y no en los detalles de cada una de las partes que lo componen.

A continuación se explicará brevemente el estilo arquitectónico usado, la arquitectura en

alto nivel de la empresa y la arquitectura de las aplicaciones de la misma.

4.2.1. Arquitectura cliente/servidor y arquitectura de micro-servicios

Durante el desarrollo del proyecto se han implementado componentes en diversas arquitecturas: cliente/servidor y arquitectura de micro-servicios.

En la arquitectura cliente/servidor el cliente (navegador web) solicita y envía datos al servidor mediante peticiones HTTP. El servidor una vez recibe la petición realiza las operaciones correspondientes y devuelve una respuesta con el estado y datos que sean necesarios.

En este caso el servidor también puede considerarse como un cliente pues, para poder llevar a cabo toda su funcionalidad, envía peticiones a los distintos micro-servicios existentes en la empresa.

En una arquitectura de micro-servicios los componentes que en una arquitectura cliente/-servidor clásica formarían parte del mismo proyecto, ya que se ocupan de diferentes funcionalidades, se descomponen en pequeños servicios. Cada uno de los micro-servicios es una unidad independiente que gestiona su propia base de datos.

La Figura 4.5 muestra una abstracción de estos conceptos. El cliente web llama al servidor de alguna de las verticales de la empresa, que a su vez llama a los micro-servicios. Estos últimos pueden llegar a comunicarse entre ellos, aunque deben intentar conseguir la máxima independencia posible.

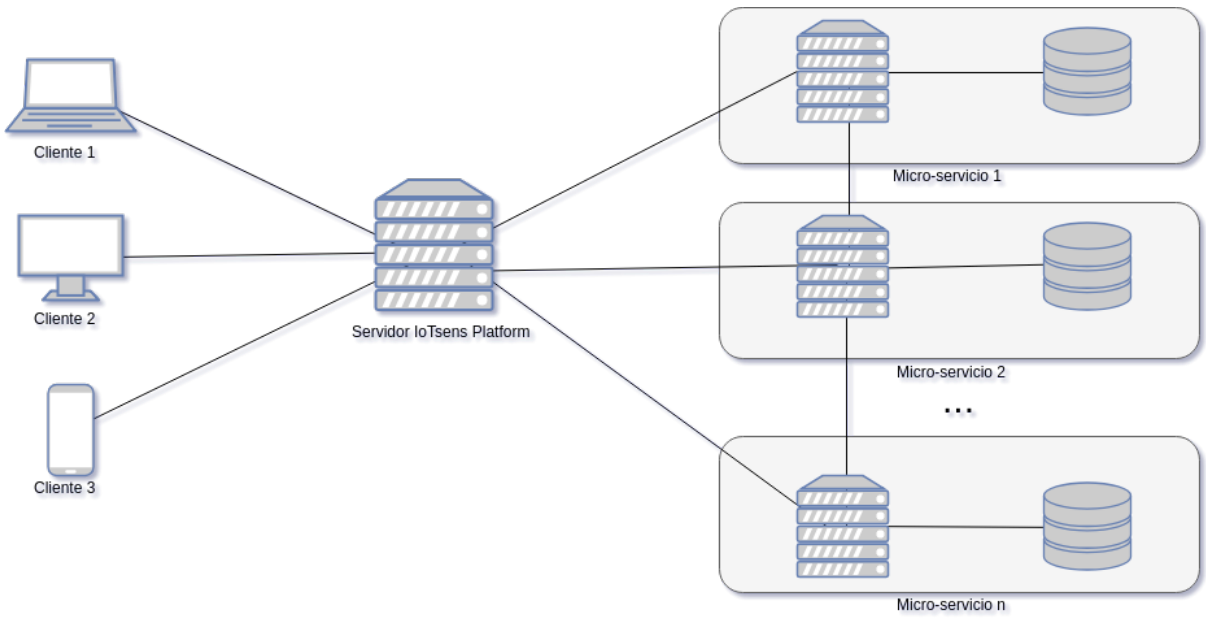


Figura 4.4: Representación de la arquitectura de la plataforma.

El objetivo del proyecto es que los micro-servicios existentes pasen a comunicarse con el

nuevo micro-servicio de auditorías, enviando una nueva petición cada vez que en ellos se produce alguna acción de creación, modificación o eliminación de datos.

4.2.2. Arquitectura IoTsens

La arquitectura mostrada anteriormente forma parte de la arquitectura global de la empresa, mostrada en la Figura 4.5.

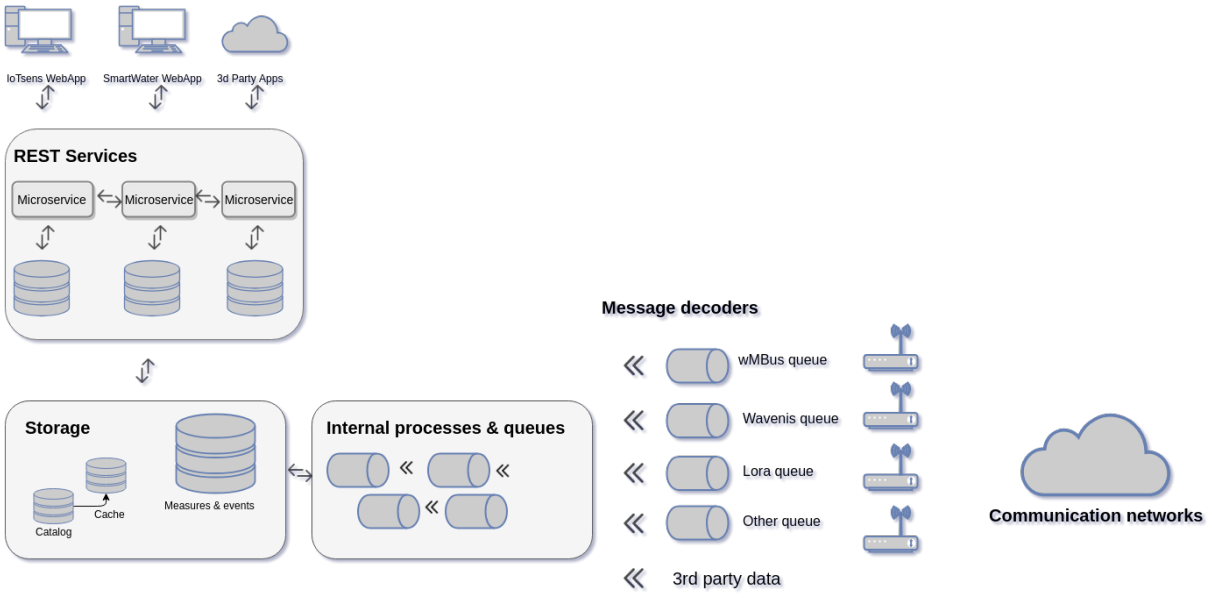


Figura 4.5: Representación de la arquitectura de IoTsens.

Los sensores de la empresa envían mensajes con las medidas obtenidas a una serie de colas que procesan la información recibida y la decodifican a un formato interno. A continuación los datos de las medidas se almacenan en base de datos y se comprueban sus valores para generar alarmas en caso de que fuese necesario.

El proyecto se ha centrado en la capa de micro-servicios y en el cliente y servidor de la plataforma *Smart City* de IoTsens.

4.2.3. Arquitectura de las aplicaciones

Las arquitecturas desarrolladas en IoTsens siguen una estructura común, basada en capas y servicios. Los usuarios utilizan una aplicación web que se comunica mediante peticiones REST, que a su vez llaman a los diferentes servicios de la aplicación.

Dichos servicios leen y modifican datos mediante la capa de persistencia, encargada de almacenar y cargar información desde la base de datos, y aplican la lógica necesaria, enviándolos finalmente en como respuesta de la petición.

En otras ocasiones los servicios llamarán a otros servicios, tanto propios de la aplicación como a micro-servicios, que proporcionan otra funcionalidad necesaria para su correcta ejecución.

La Figura 4.6 muestra las distintas partes de una aplicación en IoTsens. Cada una de las capas que las forman se describe a continuación:

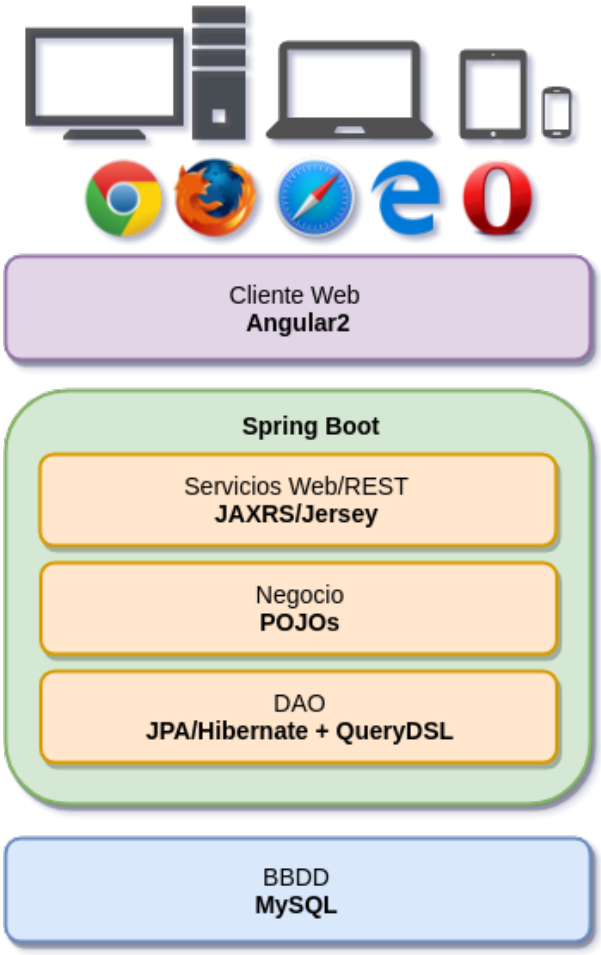


Figura 4.6: Representación de la arquitectura de las aplicaciones de IoTsens.

- Clientes Web: Se construyen, tal y como se ha mencionado en capítulos anteriores, mediante el *framework* Angular. Los clientes no contienen demasiada lógica, sino que únicamente realizan peticiones a los servicios REST de la aplicación y muestran los datos recibidos.
- Servicios Web/REST: Los servicios se asocian a un recurso (GET, PUT, POST, DELETE) en una ruta concreta, pudiendo recibir una serie de parámetros adicionales.

  Los servicios llaman a la capa de persistencia, aplican la lógica que sea necesaria y devuelven una respuesta en formato JSON. Dicha respuesta cuenta con una estructura común para todas las aplicaciones de la empresa.
- Negocio: Los objetos de negocio utilizados son objetos Java puros, también llamados POJOs (*Plain Old Java Objects*), que no dependen de ninguna biblioteca o *framework* externos.

En ellos se definen todos los campos necesarios para el funcionamiento de la aplicación y los métodos necesarios para ello.

- Persistencia/DAO: Se utiliza el patrón *Data Access Object*, proporcionando métodos que recuperan y almacenan los distintos objetos en uso.

  Dicho almacenamiento tiene lugar en bases de datos MySQL, que siguen el modelo entidad-relación. No obstante, debido al gran número de entradas de auditorías que habría que almacenar, en este proyecto se ha acordado con la empresa el uso de Elasticsearch, que proporciona un mejor rendimiento en las búsquedas con grandes volúmenes de datos.

  Cabe destacar que el uso de JPA/Hibernate hace que las operaciones contra las bases de datos relacionales sean de gran sencillez, pues no hace falta implementar las sentencias SQL, sino que se hace automáticamente.

Capítulo 5

Desarrollo e implementación

En este capítulo se detallan los detalles de la implementación del proyecto, tales como los patrones de diseño utilizados o el entorno en el que se ha producido la implementación.

5.1. Entorno de desarrollo

Se ha trabajado de forma remota sobre una máquina con Ubuntu 12 mediante conexión SSH. Mediante el sistema de ventanas X11, y configurando correctamente las pantallas, se visualizan las diferentes aplicaciones necesarias para la implementación, como el IDE o la consola bash.

Esto además permite que todas las máquinas dispongan de las mismas condiciones, facilitando el trabajo del departamento de Datacenter, encargado del mantenimiento y asistencia remota.

5.2. Principios de diseño

En la empresa se tiene mucho énfasis en la aplicación de los principios SOLID [7], que permiten que el sistema a implementar sea más fácilmente mantenible y ampliable. Estos principios son los siguientes:

- Principio de responsabilidad única (SRP): Una clase debe ser utilizada únicamente con un propósito, no puede tener más de un uso.

  Identificar y separar correctamente unas responsabilidades de las otras hará que el diseño del *software* sea menos rígido, permitiendo mayor flexibilidad y escalabilidad en el futuro.
- Principio de abierto/cerrado (OCP): Las clases deben estar abiertas a extensiones pero cerradas a modificaciones. Debido a que el software requiere cambios, y que unas entidades dependen de otras, las modificaciones en el código de una de ellas puede generar indeseables efectos colaterales en cascada.

Esto es fácilmente evitable mediante el uso de herencia, donde se amplía la funcionalidad de la clase padre en los métodos de la clase hija.

- Principio de sustitución de Liskov (LSP): Si una función espera recibir un parámetro de un tipo, pero en su lugar le pasamos un objeto de otro tipo que hereda del primero, la función debe funcionar correctamente.

  Este principio se encuentra muy relacionado con el anterior; si una función no cumple con el principio LSP entonces tampoco cumple el OCP, pues para trabajar correctamente con subtipos necesitaría saber demasiado de la clase padre, que debería ser modificada.

- Principio de segregación de interfaces (ISP): Este principio también se emplea de forma colateral con el principio de responsabilidad única. El principio de segregación de interfaces defiende que los clientes no deberían estar obligados a depender de clases o interfaces que no necesitan usar.

  Esto ocurre, por ejemplo, cuando una clase tiene más métodos de los que el cliente necesita, seguramente porque sirve a varios objetos cliente que llevan a cabo responsabilidades diferentes. Una posible solución sería que dicha clase implementase dos interfaces diferentes, cada una con los métodos necesarios, y que los objetos cliente decidan cuál de esas interfaces necesitan utilizar.

- Principio de inversión de dependencias (DIP): Según este principio, una clase no debería depender directamente de otra clase, sino de una abstracción (interfaz) de la misma.

Además de los principios SOLID, la empresa intenta promover el seguimiento de otros tres principios:

- Legibilidad de código: Se debe intentar que el código desarrollado cuente con nombres de variables descriptivos y que los métodos implementados sean relativamente cortos, facilitando la lectura, y su comprensión, a otros desarrolladores.

- No abusar de los comentarios: Se debe intentar que el código sea suficientemente autoexplicativo como para no necesitar la ayuda de comentarios en el propio código. Esto se consigue mediante un formateo claro, métodos cortos y concretos, y evitando *oneliners* de gran tamaño.

- Uso adecuado de excepciones: Es necesario conocer con cierta profundidad el propósito de las excepciones a lanzar, además de hacer un uso correcto de esta funcionalidad.

  Se recomienda el uso de excepciones en situaciones que de otra forma devolverían un valor nulo o códigos de error, por ejemplo. Las aplicaciones de IoTsens suelen implementar un *mapper* que es capaz de capturar las excepciones lanzadas en otras aplicaciones y/o micro-servicios para actuar en consecuencia.

  Además, es recomendable incluir mensajes informativos que ayuden al usuario a comprender el motivo por el que se ha producido un error.

5.3. Patrones de diseño

A lo largo de la implementación del proyecto se han utilizado diversos patrones de diseño, que han permitido ahorrar tiempo y esfuerzo manteniendo la validez y calidad del código.

Aunque se ha implementado un gran número de patrones, a continuación se muestran únicamente aquellos de mayor relevancia y que han tenido más impacto en el desarrollo.

5.3.1. Patrón Modelo-Vista-Controlador

El patrón Modelo-Vista-Controlador (MVC) se encuentra enfocado en desacoplar las clases en la programación de interfaces gráficas de usuario, agrupándolas en diferentes roles.

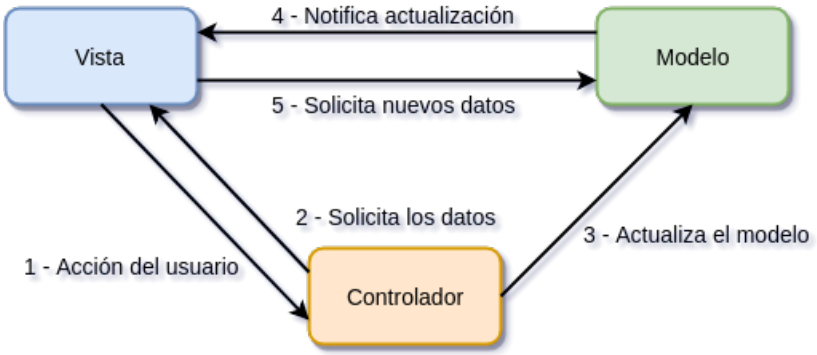


Figura 5.1: Representación del patrón MVC.

La Figura 5.1 representa las interacciones entre los roles. El modelo sólo conoce a la vista, pero no al controlador, mientras que tanto la vista como el controlador conocen al resto de roles. A continuación se detalla con un poco más de detalle el objetivo y funcionamiento de los mismos:

- Modelo: Rol que representa todas la clases que se encargan del mantenimiento y gestión de los datos de la aplicación. En el caso del proyecto desarrollado esta parte contiene cierta dificultad adicional, pues los datos a mantener se encuentran en dos sistemas bastante distintos, MySQL y Elasticsearch.
- Vista: Rol que representa todos los componentes encargados de la visualización de datos. En este caso se corresponde íntegramente con la parte *front-end* de la aplicación, implementada mediante el *framework* Angular.
- Controlador: Se encarga de hacer de mediador entre la vista y el modelo. Indica al modelo que tiene que realizar cambios una vez el usuario ha interaccionado con la vista.

Este patrón permite desacoplar completamente la vista del modelo, sin tener que modificar el último. Esto es útil, por ejemplo, para tener más de una vista interactuando con el mismo modelo, y también permite modificar el modelo sin tener que modificar la vista al mismo tiempo.

5.3.2. Patrón DAO

Tal y como se ha expuesto anteriormente en la Figura 4.6, las aplicaciones cuentan con una capa de persistencia de datos, compuesta por una serie de objetos encargados del almacenamiento y consulta de datos.

Estos objetos DAO (*Data Access Object*) proporcionan una interfaz de acceso a los datos, independizando la persistencia del resto de componentes del sistema. De hecho, en el caso del proyecto desarrollado la capa de servicio no es consciente en ningún momento de si los datos se están almacenando en un SGBD relacional o no-relacional, lo que indica hasta qué punto llega el nivel de abstracción entre las distintas capas.

Se recomienda definir un DAO por cada tipo de objeto de negocio. En el caso de este proyecto se han implementado objetos DAO en el nuevo micro-servicio. No ha sido necesario incluir nuevos DAO en el resto de micro-servicios modificados, pues éstos únicamente llaman al nuevo micro-servicio de auditorías mediante el cliente API del mismo. En total se han implementado dos clases DAO:

- AuditESDAO: Incluye métodos para insertar, buscar y eliminar (este último de forma asíncrona) nuevos registros de auditoría. También incluye otro método auxiliar para obtener el número de registros existentes. Estas operaciones se lanzan contra un motor Elasticsearch.

- ObjectTypeDAO: Incluye métodos para insertar, modificar y eliminar tipos de objeto. Estas operaciones se lanzan contra MySQL mediante JPA.

5.3.3. Patrón *Builder*

Este patrón permite crear una gran variedad de objetos complejos desde un mismo objeto fuente, compuesto por una gran variedad de propiedades cuyo valor se asigna mediante un conjunto de llamadas a operaciones dedicadas a ello.

Por tanto, mediante el uso del patrón *Builder* es posible crear instancias de una clase de forma totalmente desacoplada. A continuación se enseña, a modo de ejemplo, cómo ha sido utilizado este patrón para la construcción de los objetos utilizados en el cliente para construir las llamadas de búsqueda que se realizan hacia el micro-servicio:

```java
public class AuditRequestBuilder {

    private AuditRequest auditRequest = new AuditRequest();

    public static AuditRequestBuilder anAuditsRequest() { return new AuditRequestBuilder(); }

    public AuditRequestBuilder withObjectType(String objectType) {
        auditRequest.setObjectType(objectType);
        return this;
    }

    public AuditRequestBuilder withObjectIds(List<String> objectIds) {
        auditRequest.setObjectIds(objectIds);
        return this;
    }

    public AuditRequestBuilder withUserName(String userName) {
        auditRequest.setUserName(userName);
        return this;
    }

    public AuditRequestBuilder withApp(String app) {
        auditRequest.setApp(app);
        return this;
    }
    //...
```

Figura 5.2: Ejemplo de implementación del patrón *Builder* en el cliente del micro-servicio de auditorías.

Al abstraer el proceso de creación es posible crear representaciones diferentes en el propio proceso de construcción. La Figura 5.3 muestra un ejemplo de uso de este patrón, en concreto la creación de un nuevo objeto de la clase AuditRequest, la encargada de construir las peticiones desde el cliente al micro-servicio de auditorías.

```java
AuditRequest request = AuditRequestBuilder.anAuditsRequest()
        .withAction(Action.CREATE)
        .withApp("ms-audit")
        .withObjectType("SENSOR")
        .withFromDate(date.getTime())
        .withUntilDate(date.getTime())
        .build();
```

Figura 5.3: Ejemplo de uso del patrón *Builder* en el cliente del micro-servicio de auditorías.

5.3.4. Patrón *Strategy*

El patrón *Strategy* permite mantener un conjunto de algoritmos entre los que el cliente puede elegir para llevar a cabo una tarea, pudiendo éste ser cambiado dinámicamente según las necesidades del cliente.

Aunque a lo largo del proyecto se ha usado en varias ocasiones, una de las más representativas es en la construcción de la respuesta por parte del cliente, que debe convertir la propia respuesta del micro-servicio de auditorías en objetos que puedan ser usados en el resto de aplicaciones.

Un ejemplo de implementación de este patrón es el mostrado en la Figura 5.4.

```
public interface DeserializationStrategy <T> {
    T getResult(String body) throws IOException;
}

public class PaginatedDeserializationStrategy <T> implements DeserializationStrategy<PaginatedResult<T>> {

    private final T stateClass;
    private ObjectMapper objectMapper;

    public PaginatedDeserializationStrategy(ObjectMapper objectMapper, T stateClass) {
        this.objectMapper = objectMapper;
        this.stateClass = stateClass;
    }

    @Override
    public PaginatedResult<T> getResult(String body) throws IOException {

        JavaType type = TypeFactory
            .defaultInstance()
            .constructParametricType(
                ListResponse.class, stateClass.getClass());

        ListResponse ioTSensResponse = objectMapper.readValue(body, type);
        return new PaginatedResult<>(ioTSensResponse.getData(), ioTSensResponse.getTotal());
    }
}
```

Figura 5.4: Ejemplo de uso del patrón *Strategy* en el cliente del micro-servicio de auditorías.

Mediante el uso de este patrón, el cliente es capaz de seguir una estrategia para *deserializar* respuestas que contienen más de un elemento, o respuestas que sólo contengan un elemento, cuyo método necesario se encuentra implementado en otra clase distinta.

Otro uso de este patrón es la decisión de los datos de los objetos que van a ser auditados. Mediante diferentes estrategias es posible incluir unos datos u otros de cada tipo de objeto, como por ejemplo el nivel de profundidad que se quiere auditar, o si incluir o no sus identificadores.

5.4. Interfaz de usuario

Las auditorías deben poder ser consultadas por los administradores de IoTsens. Se decidió implementar la interfaz en su plataforma de *Smart City*, una aplicación web encargada de centralizar las diferentes operaciones que se pueden realizar con los dispositivos IoT de la empresa, además de contar con diversas herramientas de administración, entre las que se incluye este proyecto.

Esta aplicación cuenta con una interfaz web desarrollada en Angular, que utiliza documentos HTML y hojas de estilo CSS para la visualización, y TypeScript para su lógica. Como se ha comentado anteriormente, esto proporciona un mayor dinamismo al no tener que recargar constantemente la página completa para mostrar nuevos datos.

Se han utilizado como guía las siguientes reglas de oro del diseño de interfaces [8]:

1. **Mantener la consistencia**: Las secuencias de acciones, terminología, colocación de entradas de texto y menús, colores, fuentes, etc. deben ser consistentes a lo largo de toda la aplicación. Afortunadamente, la empresa cuenta con componentes reutilizables y una hoja de estilos bastante completa que han facilitado esta labor.

2. **Facilitar la universalidad**: Se deben reconocer las necesidades de los distintos usuarios, facilitando la transformación del contenido. Estas diferencias pueden encontrarse entre usuarios de distinta experiencia, con discapacidades, con diferentes habilidades tecnológicas, etc.

3. **Ofrecer *feedback* informativo**: El sistema debe proporcionar información por cada acción de los usuarios. En el caso de acciones pequeñas la información puede ser más escueta, pero en caso de acciones de gran importancia la información debería ser más sustancial.

4. **Diseñar grupos de acciones cerrados**: Las secuencias de acciones deben organizarse en grupos con un inicio y final, permitiendo ofrecer *feedback* informativo cuando se complete uno de estos grupos de acciones.

5. **Prevenir errores**: El sistema debe estar diseñado de forma que los usuarios no puedan cometer errores graves. Esto se puede conseguir mediante limitaciones en los propios *input* de los formularios. No obstante, si se produce un error el usuario debe ser informado mediante un mensaje que indique claramente qué ha ocurrido, ofreciendo al mismo tiempo una solución.

6. **Permitir revertir acciones**: Tanto como sea posible, las acciones de los usuarios deben ser reversibles, animando a los usuarios a explorar acciones que no les resulten familiares.

7. **Dar control al usuario**: Los usuarios más experimentados suelen tener la necesidad de sentirse en control de la interfaz, y que ésta responda a sus acciones.

8. **Reducir la carga de memoria**: La media de información que los seres humanos podemos almacenar en un corto plazo de tiempo es de alrededor de 7 elementos. Al reducir esta carga, el usuario se sentirá más en control y mejorará su experiencia.

Aunque la interfaz desarrollada corresponde únicamente a una página, el seguir los anteriores principios ha resultado muy útil, resultando en una interfaz intuitiva, agradable estéticamente y sencilla de utilizar aún sin mucha experiencia.

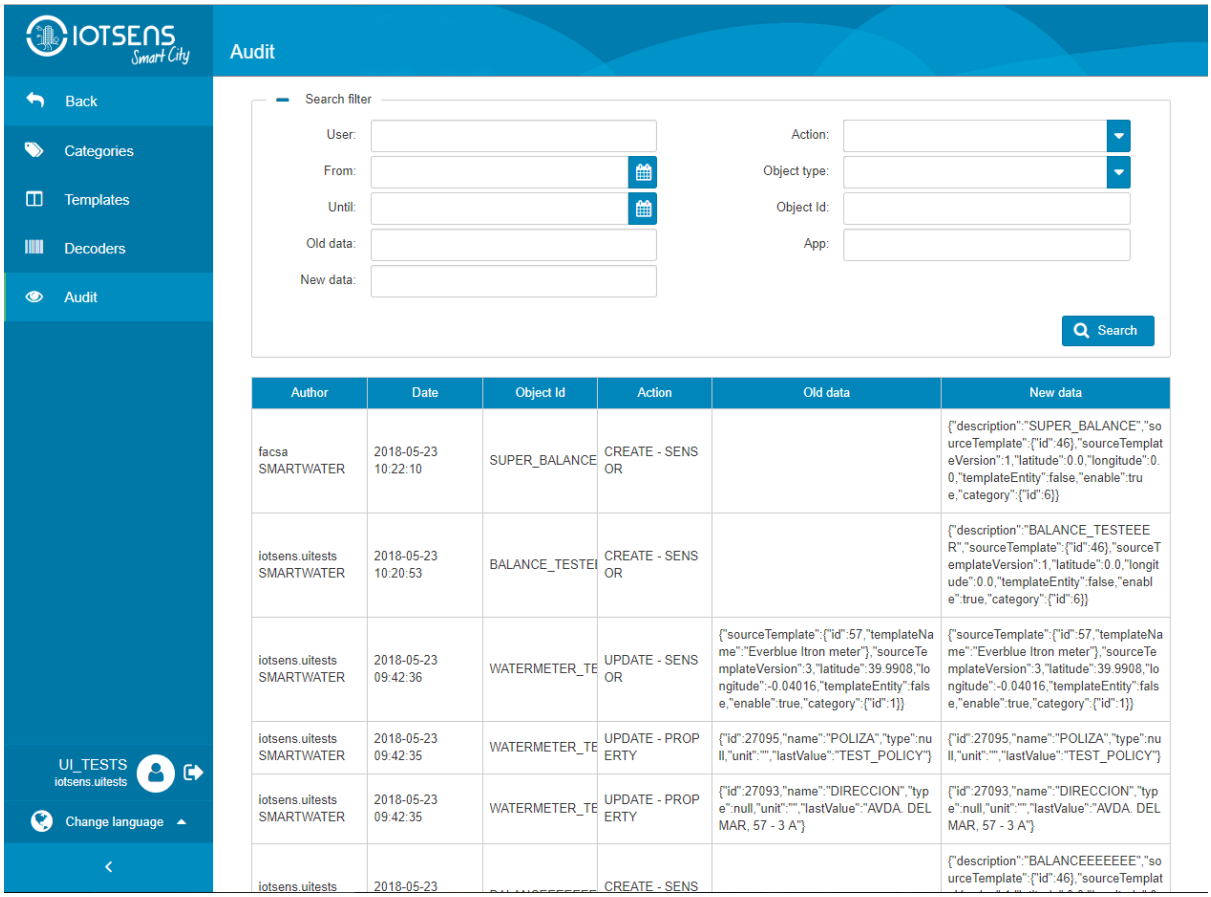


Figura 5.5: Pantalla de administración de auditorías.

La Figura 5.5 muestra el diseño final de la pantalla de auditorías, encargada de mostrar un listado con todos los registros existentes. Además, en la parte superior se pueden filtrar estos registros según diversos parámetros.

A continuación se detalla más profundamente las partes de las que está compuesta la interfaz.

5.4.1. Listado

En la parte inferior de la pantalla se encuentra el listado de auditorías. Éste nos permite obtener información sobre las acciones que se han producido en la plataforma. De izquierda a derecha podemos observar la aplicación y usuarios con los que se ha producido la acción, la fecha y hora, el identificador del objeto modificado, la acción producida y sobre qué tipo de objeto, y los datos antiguos y nuevos que contiene ese objeto en particular.

Los datos de los objetos se muestran en formato JSON, permitiendo una visualización rápida de los diferentes atributos de los objetos. Cabe mencionar que se encuentra planificada una mejora en esta visualización, que permitirá visualizar dichos datos de una forma más amigable.

Para proporcionar una mejor experiencia al usuario la tabla sólo carga la paginación en la

que se encuentra, recibiendo menos datos desde el servidor y, por tanto, siendo necesario menos tiempo de espera al entrar por primera vez.

Para ello se utiliza el *limit* y *offset* que se han definido como parámetros en la llamada de búsqueda de auditorías.

Author	Date	Object Id	Action	Old data	New data
iotsens.uitests SMARTWATER	2018-05-28 09:06:16	WATERMETER_TEST_UPDATE	UPDATE - SENSOR	{"sourceTemplate":{"id":57,"templateName":"Everblue Itron meter"},"sourceTemplateVersion":3,"latitude":39.94765,"longitude":-0.04016,"templateEntity":false,"enable":true,"category":{"id":1}}	{"sourceTemplate":{"id":57,"templateName":"Everblue Itron meter"},"sourceTemplateVersion":3,"latitude":39.94765,"longitude":-0.16136,"templateEntity":false,"enable":true,"category":{"id":1}}
iotsens.uitests SMARTWATER	2018-05-28 09:06:16	WATERMETER_TEST_UPDATE	UPDATE - PROPERTY	{"id":27105,"name":"POLIZA","type":null,"unit":"","lastValue":"TEST_POLICY"}	{"id":27105,"name":"POLIZA","type":null,"unit":"","lastValue":"TEST_POLICY"}
iotsens.uitests SMARTWATER	2018-05-28 09:06:16	WATERMETER_TEST_UPDATE	UPDATE - PROPERTY	{"id":27106,"name":"DIRECCION","type":null,"unit":"","lastValue":"AVDA. DEL MAR, 58 - 3 A"}	{"id":27106,"name":"DIRECCION","type":null,"unit":"","lastValue":"AVDA. DEL MAR, 58 - 3 A"}
iotsens.uitests SMARTWATER	2018-05-28 09:06:14	WATERMETER_TEST_UPDATE	UPDATE - SENSOR	{"sourceTemplate":{"id":57,"templateName":"Everblue Itron meter"},"sourceTemplateVersion":3,"latitude":39.9908,"longitude":-0.04016,"templateEntity":false,"enable":true,"category":{"id":1}}	{"sourceTemplate":{"id":57,"templateName":"Everblue Itron meter"},"sourceTemplateVersion":3,"latitude":39.94765,"longitude":-0.04016,"templateEntity":false,"enable":true,"category":{"id":1}}
iotsens.uitests SMARTWATER	2018-05-28 09:06:14	WATERMETER_TEST_UPDATE	UPDATE - PROPERTY	{"id":27105,"name":"POLIZA","type":null,"unit":"","lastValue":"TEST_POLICY"}	{"id":27105,"name":"POLIZA","type":null,"unit":"","lastValue":"TEST_POLICY"}
iotsens.uitests SMARTWATER	2018-05-28 09:06:14	WATERMETER_TEST_UPDATE	UPDATE - PROPERTY	{"id":27106,"name":"DIRECCION","type":null,"unit":"","lastValue":"AVDA. DEL MAR, 58 - 3 A"}	{"id":27106,"name":"DIRECCION","type":null,"unit":"","lastValue":"AVDA. DEL MAR, 58 - 3 A"}
iotsens.uitests SMARTWATER	2018-05-28 09:06:11	WATERMETER_TEST_UPDATE	UPDATE - SENSOR	{"sourceTemplate":{"id":57,"templateName":"Everblue Itron meter"},"sourceTemplateVersion":3,"latitude":39.9908,"longitude":-0.04016,"templateEntity":false,"enable":true,"category":{"id":1}}	{"sourceTemplate":{"id":57,"templateName":"Everblue Itron meter"},"sourceTemplateVersion":3,"latitude":39.9908,"longitude":-0.04016,"templateEntity":false,"enable":true,"category":{"id":1}}
iotsens.uitests SMARTWATER	2018-05-28 09:06:11	WATERMETER_TEST_UPDATE	UPDATE - PROPERTY	{"id":27105,"name":"POLIZA","type":null,"unit":"","lastValue":"TEST_POLICY"}	{"id":27105,"name":"POLIZA","type":null,"unit":"","lastValue":"TEST_POLICY"}
iotsens.uitests SMARTWATER	2018-05-28 09:06:11	WATERMETER_TEST_UPDATE	UPDATE - PROPERTY	{"id":27106,"name":"DIRECCION","type":null,"unit":"","lastValue":"AVDA. DEL MAR, 57 - 3 A"}	{"id":27106,"name":"DIRECCION","type":null,"unit":"","lastValue":"AVDA. DEL MAR, 58 - 3 A"}
iotsens.uitests SMARTWATER	2018-05-28 09:06:08	WATERMETER_TEST_UPDATE	UPDATE - SENSOR	{"description":"WATERMETER_TEST_UPDATED","sourceTemplate":{"id":57,"templateName":"Everblue Itron meter"},"sourceTemplateVersion":3,"latitude":0.0,"longitude":0.0,"templateEntity":false,"enable":true,"category":{"id":1}}	{"sourceTemplate":{"id":57,"templateName":"Everblue Itron meter"},"sourceTemplateVersion":3,"latitude":39.9908,"longitude":-0.04016,"templateEntity":false,"enable":true,"category":{"id":1}}
iotsens.uitests SMARTWATER	2018-05-28 09:06:08	WATERMETER_TEST_UPDATE	UPDATE - PROPERTY	{"id":27105,"name":"POLIZA","type":null,"unit":"","lastValue":"XXXX"}	{"id":27105,"name":"POLIZA","type":null,"unit":"","lastValue":"TEST_POLICY"}
iotsens.uitests SMARTWATER	2018-05-28 09:06:08	WATERMETER_TEST_UPDATE	UPDATE - PROPERTY	{"id":27106,"name":"DIRECCION","type":null,"unit":"","lastValue":"XXXX"}	{"id":27106,"name":"DIRECCION","type":null,"unit":"","lastValue":"AVDA. DEL MAR, 57 - 3 A"}
iotsens.uitests SMARTWATER	2018-05-28 09:06:08	WATERMETER_TEST_UPDATE	CREATE - SENSOR		{"description":"WATERMETER_TEST_UPDATED","sourceTemplate":{"id":57},"sourceTemplateVersion":3,"latitude":0.0,"longitude":0.0,"templateEntity":false,"enable":true,"category":{"id":1}}
iotsens.uitests SMARTWATER	2018-05-28 09:06:06	NEW_NODE2_UNIQUE_ID	UPDATE - SENSOR	{"description":"NEW_NODE2_UNIQUE_ID","sourceTemplate":{"id":15,"templateName":"Gateway"},"sourceTemplateVersion":1,"latitude":39.98467,"longitude":-0.0274,"templateEntity":false,"enable":true,"category":{"id":4}}	{"sourceTemplate":{"id":15,"templateName":"Gateway"},"sourceTemplateVersion":1,"latitude":39.98467,"longitude":-0.0274,"templateEntity":false,"enable":true,"category":{"id":4}}
iotsens.uitests SMARTWATER	2018-05-28 09:06:02	NEW_NODE1_UNIQUE_ID	UPDATE - SENSOR	{"sourceTemplate":{"id":15,"templateName":"Gateway"},"sourceTemplateVersion":1,"latitude":39.98467,"longitude":-0.0274,"templateEntity":false,"enable":true,"category":{"id":4}}	{"sourceTemplate":{"id":15,"templateName":"Gateway"},"sourceTemplateVersion":1,"latitude":39.98467,"longitude":-0.0274,"templateEntity":false,"enable":true,"category":{"id":4}}

1 2 3 4 5 6 7 8 9 10 15 Total results: 16283

Figura 5.6: Listado de auditorías.

Se decidió mostrar el objeto completo en lugar de únicamente aquéllos campos cambiados, aportando al administrador más información del contexto. No obstante, esto se puede configurar en el propio código, auditando dinámicamente mediante el uso de *JSON Views* cuando se considere que el objeto contiene demasiada información por sí mismo.

5.4.2. Filtrado

Los campos mostrados en el apartado anterior pueden ser filtrados fácilmente mediante el formulario superior, aunque éste puede ser ocultado en caso de que no se necesite.

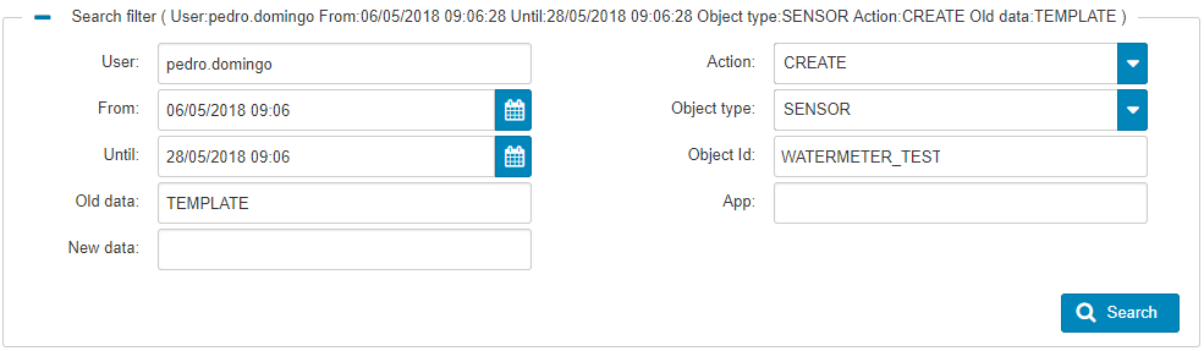


Figura 5.7: Listado de auditorías.

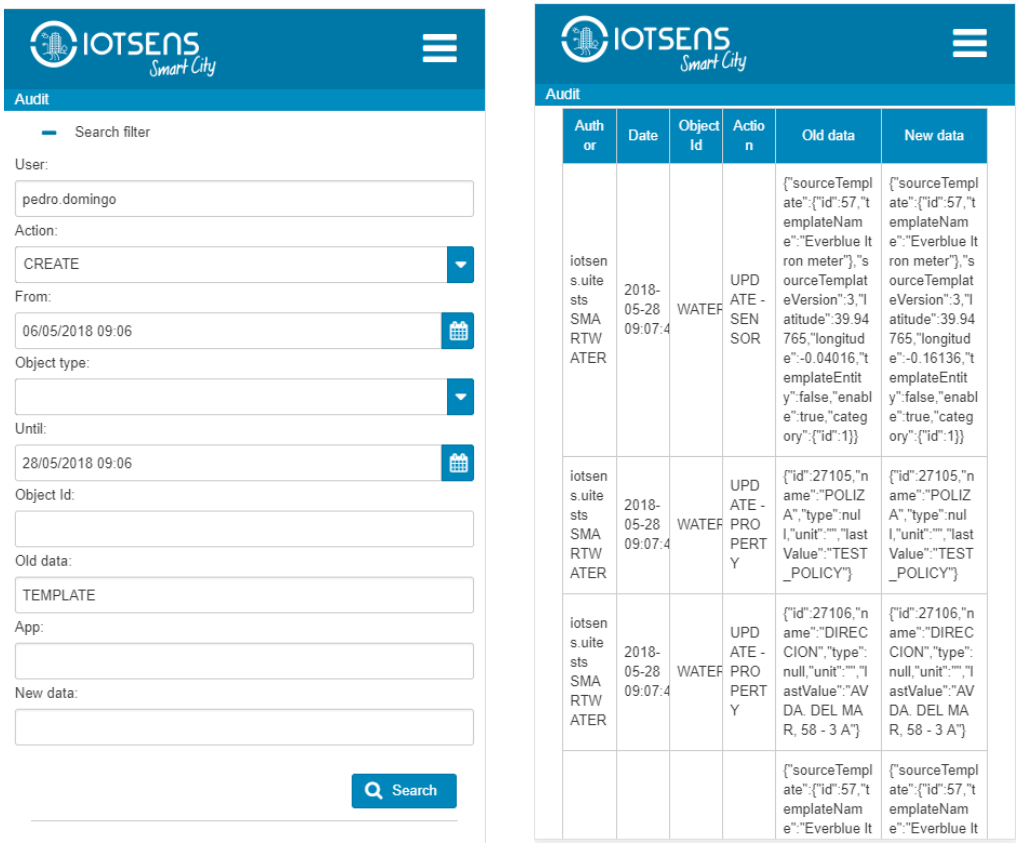
Los filtros se añaden a la propia URL de la página, de forma que también es posible filtrar desde la propia barra de navegación si se incluyen a mano los campos y valores, o directamente entrando desde un enlace que haya sido compartido.

Además, tal y como se muestra en la Figura 5.7, los filtros se muestran como texto en la parte superior del panel de filtrado, ayudando al usuario a comprender cuáles son los campos por los que se está filtrando los registros.

5.4.3. Dispositivos móviles

Aunque no se ha desarrollado una aplicación nativa para dispositivos móviles OSX o Android, se ha adaptado la vista para que se ajuste a los dispositivos desde los que se está visualizando.

Esto es posible mediante *CSS Media Queries*, que permiten definir las formas de visualización dependiendo del tamaño de la ventana. Además, el *layout* del *framework* Bootstrap ayuda significativamente en esta tarea, pues muchas de estas acciones las realiza automáticamente.



(a) (b)

Figura 5.8: Visualización de auditorías en dispositivos móviles

Actualmente se encuentra prevista una mejora en este aspecto que permita visualizar de mejor forma los datos de los registros.

Capítulo 6

Verificación y validación

La fase de verificación y validación es el conjunto de procesos que comprueban, analizan y se aseguran de que el software desarrollado concuerda con su especificación, cumpliendo además con las necesidades de los clientes.

Es una de las fases más importantes, pues la ejecución de unas pruebas correctamente definidas puede encontrar muchos errores en la programación y funcionalidad del código que de no encontrarse pasarían a formar parte del resultado final. Encontrar dichos errores tempranamente evita tener que invertir tiempo y recursos en solucionar los fallos posteriormente.

A lo largo del proyecto se han utilizado cuatro tipos de pruebas de software: Pruebas de unidad, pruebas de integración, pruebas de interfaz de usuario y pruebas de aceptación.

6.1. Pruebas unitarias

Las pruebas unitarias se utilizan para comprobar que el *software* funciona de forma correcta, descomponiendo los métodos en comportamientos comprobables que se pueden ejecutar de forma individual y aislada.

Son una parte vital en el flujo de trabajo de desarrollo de *software* que, en caso de definirlas previamente a la implementación del código, permiten obtener una visión más clara y detallada de cómo va a ser ese código a implementar.

Hoy por hoy las pruebas unitarias cuentan con más importancia si cabe gracias a metodologías como TDD, que han sido adoptadas por multitud de desarrolladores debido a las grandes ventajas que aporta el diseño de software para que pase esas pruebas. No obstante, en este caso no se ha utilizado dicha metodología debido a la gran cantidad de esfuerzo y tiempo que supone su adopción, sólo se ha seguido en casos puntuales que cuentan con mayor complejidad.

Para el desarrollo de las pruebas unitarias se ha utilizado la herramienta JUnit, un *framework* de Java pensado para este propósito. En ciertos casos también se ha utilizado la herramienta

Mockito, que permite simular el comportamiento de instancias de otras clases del proyecto, proporcionando la independencia necesaria como para que las pruebas unitarias no se vean afectadas por comportamientos externos.

Con el propósito de simular las llamadas REST desde el cliente se ha utilizado la herramienta MockWebServer. Esto permite encolar llamadas a un servidor simulado, que devuelve respuestas *mock* que en este caso se han definido en un fichero aparte.

El siguiente ejemplo intenta mostrar el funcionamiento de estas tecnologías y el proceso de pruebas en conjunto, ilustrándolo mediante los test unitarios del cliente API. En primer lugar se debe definir cuál será la respuesta que queremos simular, definiéndola en un fichero JSON auxiliar. En este caso se tomará como base la respuesta de la Figura 6.1.


```
{
  "success": true,
  "msg": null,
  "data": [
    {
      "id": "685c4b57-4114-4a42-9373-3d81c28d6557",
      "app": "IOTFRONT",
      "userName": "pedro.domingo",
      "timestamp": 1521203741382,
      "objectType": "SENSOR",
      "objectId": "SENSOR_UNIQUE_ID_2",
      "action": "CREATE",
      "oldData": null,
      "newData": "{prueba: \\\"sin id\\\"}"
    }
  ],
  "total": 1
}
```


Figura 6.1: Respuesta *mock* usada en los test unitarios.

Las ejecuciones de los distintos test unitarios pueden llegar a tener partes en común, como el inicio y cierre del servidor *mock* o la asignación de credenciales de usuario. Dichas partes en común pueden extraerse a métodos externos que pueden ser marcados mediante las anotaciones `@Before` y `@After`, que permiten que dichos métodos se ejecuten antes y después de cada test.

```
@Before
public void setUp() throws URISyntaxException {
    mockWebServer = new MockWebServer();
    credentials = new Credentials(A_USER, AN_APP);
    String baseUrl = mockWebServer.url("/v1/").url().toURI().toString();

    AuditsApiClientFactory factory = new AuditsApiClientFactory();
    apiClient = factory.getAuditsApiClient(baseUrl);
}

@After
public void tearDown() throws IOException {
    mockWebServer.shutdown();
}
```

Figura 6.2: Ejemplo de métodos auxiliares *Before* y *After*.

Finalmente se encuentra la definición del propio test. Para ello, se utiliza la anotación `@Test` en el método deseado. Seguidamente se construye la petición contra el *mockserver*, recibiendo la respuesta que se había definido, y tras la que se puede comprobar que la ejecución ha sido correcta.

```
@Test
public void searchAuditsCall() throws IOException, URISyntaxException, InterruptedException {
    enqueueResponse("/audits.json");

    AuditRequest request = AuditRequestBuilder.anAuditsRequest()
            .withApp(APP)
            .withUserName(USER_NAME)
            .withFromDate(new Date(DATE))
            .withUntilDate(new Date(DATE))
            .withObjectType(OBJECT_TYPE)

    PaginatedResult<AuditDTO> result = apiClient.searchAudits
            (credentials, request);
    RecordedRequest recordedRequest = mockWebServer.takeRequest();
    AuditDTO audit = result.getCurrentResults().get(0);

    String app = recordedRequest.getHeader("IOT-Requester-Application");
    String user = recordedRequest.getHeader("IOT-Authorized-User");
    assertThat(app, equalTo(AN_APP));
    assertThat(user, equalTo(A_USER));
    assertThat(result.getTotalCount(), equalTo(1L));
    assertThat(audit.getId(), equalTo(AUDIT_ID));
    assertThat(audit.getUserName(), equalTo(USER_NAME));
    assertThat(audit.getAction(), equalTo(ACTION));
    assertThat(audit.getApp(), equalTo(APP));
    assertThat(audit.getNewData(), equalTo(NEW_DATA));
    assertThat(audit.getOldData(), equalTo(null));
    assertThat(audit.getTimestamp(), equalTo(new Date(DATE)));
    assertThat(audit.getObjectId(), equalTo(OBJECT_ID));
    assertThat(audit.getObjectType(), equalTo(OBJECT_TYPE));
}
```

Figura 6.3: Ejemplo de test unitario.

El uso de pruebas unitarias ha permitido encontrar errores que de otra forma hubiesen pasado al producto final, por lo que se ha intentado utilizarlas frecuentemente. No obstante, tal y como se ha comentado anteriormente, en muchos casos la implementación de estas pruebas

supone una inversión de tiempo excesiva, por lo que se han evitado en los casos más sencillos y por ende menos propensos a errores.

6.2. Pruebas de integración

Tras conseguir la correcta ejecución de todas las pruebas unitarias se procede a realizar las pruebas de integración, encargadas de verificar que los componentes colaboran entre sí de forma satisfactoria.

Esto cuenta con mucha importancia en la empresa, que cuenta con un servicio de integración continua encargado de automatizar estas tareas.

El servidor Jenkins se encarga de descargar el código cada vez que detecta un cambio en la rama *develop* de algún repositorio en el servidor de GitLab, compila dicho código y ejecuta los test que se han definido en el proyecto. A continuación, si se ha compilado correctamente, genera un informe con el estado de la ejecución, en el que se indica si se han pasado los test.

Además, la empresa cuenta con un proyecto que se encuentra configurado para que Jenkins lo ejecute cada media hora. En este proyecto se encuentran definidos los test de interfaz de las distintas aplicaciones. Si los test fallan, se envía un correo electrónico a los responsables de la aplicación para que éstos sean conscientes de la existencia de errores y puedan solucionarlos cuanto antes.

6.3. Pruebas de interfaz

Las pruebas de interfaz permiten verificar las interacciones del usuario con la aplicación, asegurando su correcta navegación y comportamiento.

Para ello, tal y como se ha comentado en la sección anterior, la empresa cuenta con un proyecto encargado de las pruebas de interfaz, que se ejecuta automáticamente en el servidor Jenkins en el entorno de pre-producción, que a su vez elabora un informe por cada ejecución.

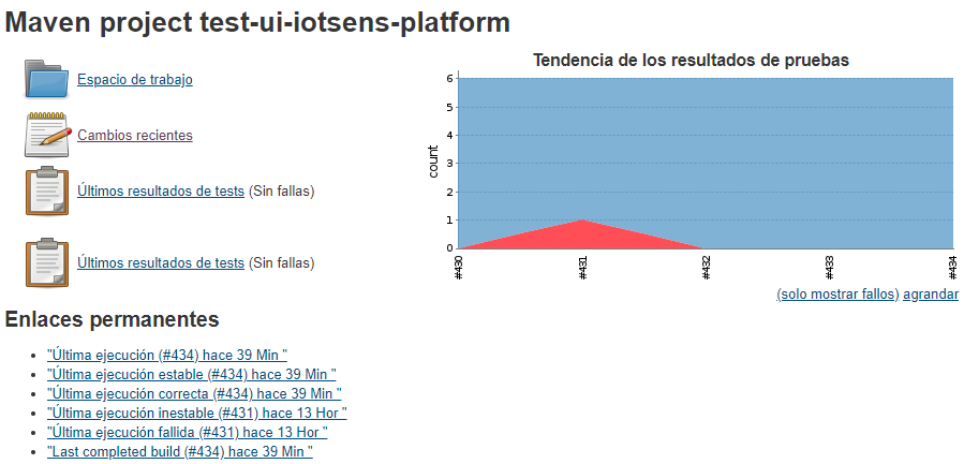


Figura 6.4: Gráfica de ejecución de test de interfaz en Jenkins.

Las pruebas se encuentran implementadas utilizando la herramienta Selenium, que permite simular las acciones de un usuario en un navegador web. Se puede visualizar la secuencia de acciones de forma gráfica, aunque en este caso Jenkins requiere el uso de la opción *headless* de Google Chrome, que permite la simulación sin la necesidad que se abra la ventana del navegador.

Se ha ampliado el proyecto existente con las pruebas de la nueva pantalla de auditorías, simulando la entrada de texto en los formularios, la correcta paginación, etc.

```
public void loadPage() {
    waitForIoTsensAppLoadAndNGComponentReady("app-deployment");
    checkCorrectNavigationSection("audits");
    waitForSpinner();

    WebElement table = waitForElement(By.tagName("p-datatable"));

    if (table != null) {
        dataTable = new AngularDataTable(table, uniqueColumn);
    }
}
public Integer countTotalItems() {
    WebElement totalItems = getDriver()
            .findElements(By.className("table-total-items"))
            .get(0)
            .findElement(By.tagName("strong"));

    getFluentWait().until(ExpectedConditions.attributeToBeNotEmpty(totalItems, "innerHTML"));
    return Integer.parseInt(totalItems.getAttribute("innerHTML"));
}

public boolean isAuditInTable(String objectId) {
    filterByObjectId(objectId);
    if (countTotalItems() == 0) {
        return false;
    } else {
        String searchedId = dataTable.getColumnValueForEntry(objectId, uniqueColumn);
        return objectId.equals(searchedId);
    }
}
```

Figura 6.5: Ejemplo de test de interfaz.

La Figura 6.5 muestra un ejemplo de implementación de un test de interfaz gráfica lanzado contra la pantalla de auditorías. Por claridad se enseña únicamente la implementación de las secuencias de pasos, omitiendo configuración y comprobaciones de JUnit, que se encuentran implementadas en clases distintas.

6.4. Pruebas de aceptación

Las pruebas de aceptación sirven para asegurarse de que la aplicación cumple con la funcionalidad establecida por los requisitos funcionales del sistema. Debido a la metodología de trabajo, la validación de dichas pruebas de aceptación se ha realizado una vez se ha comprobado el código, antes de indicar que la tarea pasa de estar "En pruebas" a "Hecho".

Aunque normalmente las realizan un conjunto de usuarios finales en este caso, por las características del desarrollo, las ha realizado el propio alumno, los distintos revisores de código de cada tarea, y el Scrum Manager.

La lista de pruebas de aceptación que se han llevado a cabo se encuentra adjunta en el Anexo C.

Capítulo 7

Conclusiones

La realización de la estancia en prácticas y el desarrollo del proyecto me han permitido obtener un nuevo punto de vista sobre el sector del desarrollo de *software*, en concreto sobre el desarrollo dentro del ámbito del *Internet de las cosas*. La primera impresión que obtuve fue la observación de una gran diferencia entre el mundo universitario, más centrado en la teoría, y el mundo de la empresa, que no olvida los aspectos teóricos pero es capaz de aplicarlos en conjunto con la práctica.

Las sensaciones transmitidas en la empresa han sido excelentes, ayudándome a mejorar tanto la fluidez y calidad del trabajo como mis aptitudes personales, un valor muy importante en el entorno laboral.

Además, considero que este último punto cuenta con gran importancia en el lugar de trabajo. Tras la estancia en prácticas me ha quedado claro que no sólo es importante la calidad del código, sino el ambiente en el que se desarrolla. En este caso, cuando el buen ambiente se suma a las metodologías ágiles utilizadas, que al proporcionar una planificación incremental reducen la sensación de incertidumbre, todos los desarrolladores se encuentran motivados y son capaces de conseguir un trabajo de calidad.

Desde un punto de vista técnico el proyecto me ha permitido formarme en herramientas y tecnologías que desconocía hasta el momento, tales como Angular, la arquitectura de microservicios o bases de datos no relacionales como Elasticsearch, entre otras.

En referencia al proyecto, éste se terminó inicialmente en un plazo de tiempo menor al planificado. No obstante, se decidió implantar el nuevo servicio en más aplicaciones de la empresa, lo que hizo que el número de horas dedicadas creciese sustancialmente.

Aunque el proyecto no proporciona a la empresa un beneficio económico directo, sí que ofrece más control a los administradores y desarrolladores. Esto ha fomentado la implantación del mismo en los entornos de desarrollo y producción, encontrándose actualmente en uso. Además, tras la finalización de la estancia en prácticas la empresa decidió incluir en su siguiente *sprint* nuevas mejoras y ampliaciones al proyecto, tales como:

- Internacionalización de las descripciones de los tipos de objeto.
- Mejora de la visualización JSON de los datos anteriores y posteriores de los datos auditados para que éstos se puedan visualizar con más espacio, formateados y con colores.
- Implantación del servicio en el resto de aplicaciones y servicios de la empresa.

Bibliografía

[1] Apache lucene. https://lucene.apache.org/. [Consulta: 28 de Mayo de 2018].

[2] Beginning ajax.

[3] Lora alliance. https://lora-alliance.org/. [Consulta: 28 de Mayo de 2018].

[4] Página web de iotsens. http://www.iotsens.com/. [Consulta: 14 de Mayo de 2018].

[5] Página web del grupo gimeno. http://www.grupogimeno.com/en. [Consulta: 14 de Mayo de 2018].

[6] Sueldos en programador/a junior en españa. https://www.indeed.es/salaries/Programador/a-junior-Salaries. [Consulta: 28 de Mayo de 2018].

[7] Robert C. Martin ("Uncle Bob"). The principles of ood. http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod. [Consulta: 17 de Mayo de 2018].

[8] Catherine Plaisant Ben Shneiderman. Design the user interface. *Addison Wesley; 4th edition*, 2005.

[9] Rosa Alarcon Cesare Pautasso, Erik Wilde. Rest: Advanced research topics and practical applications. *Lasa, Carmen et al*, 2014.

[10] Julian Agyeman Duncan McLaren. Sharing cities: A case for truly smart and sustainable cities. *MIT Press*, 2015.

[11] James Lewis, Martin Fowler. Microservices, a definition of this new architectural term. https://martinfowler.com/articles/microservices.html. [Consulta: 15 de Mayo de 2018].

[12] Carmen et al Lasa. Métodos Ágiles. scrum, kanban, lean. *ANAYA*, 2017.

[13] Daniel SHaischt. A traditional, stateless web application and it's ajaxified counterpart on the right site. https://commons.wikimedia.org/wiki/File:Ajax-vergleich-en.svg. [Consulta: 25 de Junio de 2018].

[14] Daqiang Zhang, Huansheng Ning, Kevin S. Xu, Feiyu Lin, and Laurence Tianruo Yang. Internet of things. *J. UCS*, 2012.

Anexo A

Requisitos funcionales

AUDIT-1 Insertar nuevos registros de auditoría	
ID	AUDIT-1
Nombre	Insertar nuevos registros de auditoría
Actor principal	Usuario
Actores secundarios	Administrador
Descripción	Este caso de uso representa la funcionalidad que permite que se auditen automáticamente las acciones que producen los usuarios del sistema.
Nivel de complejidad	Media
Relaciones	AUDIT-2
Precondición	El usuario debe estar autenticado ante el sistema. El tipo de objeto a auditar debe estar definido en el sistema.
Trigger	Un usuario realiza cualquier acción que modifique algún dato de la plataforma.
Secuencia normal	
1	El usuario accede a la plataforma de la empresa.
2	El usuario inserta un nuevo sensor.
3	El sistema guarda los datos del usuario, la acción realizada y las modificaciones en el objeto.
Excepciones	
Frecuencia esperada	Varias veces al día
Importancia	Muy alta
Prioridad	Corto plazo
Comentarios	El sistema guardará el id de objeto, tipo, acción, aplicación, datos antiguos y datos nuevos.

Cuadro A.1: AUDIT-1 Insertar nuevos registros de auditoría

AUDIT-2 Listar todos los registros	
ID	AUDIT-2
Nombre	Listar todos los registros
Actor principal	Administrador
Actores secundarios	Usuario
Descripción	Este caso de uso representa la funcionalidad que permite al administrador listar todos los registros de auditoría almacenados en el sistema.
Nivel de complejidad	Media
Relaciones	AUDIT-1, AUDIT-3
Precondición	El administrador debe estar autenticado ante el sistema.
Trigger	El administrador accede a la sección de listado de registros de auditoría.
Secuencia normal	
1	El administrador accede a la plataforma de la empresa.
2	El administrador accede a la sección de listado de auditorías.
3	El sistema muestra todos los registros actuales.
Excepciones	*No existe ningún registro en ese momento.*
1	El sistema muestra un mensaje indicando que no hay registros disponibles.
Frecuencia esperada	Varias veces al día
Importancia	Muy alta
Prioridad	Corto plazo
Comentarios	-

Cuadro A.2: AUDIT-2 Listar todos los registros

AUDIT-3 Filtrar registros de auditoría	
ID	AUDIT-3
Nombre	Filtrar registros de auditoría
Actor principal	Administrador
Actores secundarios	Usuario
Descripción	Este caso de uso representa la funcionalidad que permite al administrador filtrar los registros existentes mediante diferentes criterios.
Nivel de complejidad	Alta
Relaciones	AUDIT-2, AUDIT-4
Precondición	El administrador debe estar autenticado ante el sistema.
Trigger	El administrador inicia una búsqueda tras introducir un filtro.
Secuencia normal	
1	El administrador accede a la plataforma de la empresa.
2	El administrador accede a la sección de listado de auditorías.
3	El administrador introduce un criterio de búsqueda en alguno de los filtros disponibles.
4	El sistema actualiza el listado con aquellos registros que cumplan las condiciones establecidas.
Excepciones	*No existe ningún registro que cumpla las condiciones.*
1	El sistema muestra un mensaje indicando que no hay registros disponibles.
Frecuencia esperada	Varias veces al día
Importancia	Muy alta
Prioridad	Corto plazo
Comentarios	El sistema permitirá el filtrado mediante todos o algunos de los siguientes parámetros: id del registro, nombre de usuario, aplicación, fecha mínima, fecha máxima, tipo de objeto, lista de identificadores de objetos, acción, datos antiguos o datos nuevos.

Cuadro A.3: AUDIT-3 Filtrar registros de auditoría

AUDIT-4 Limitar la cantidad de resultados de la lista mediante un límite y un offset.	
ID	AUDIT-4
Nombre	Limitar la cantidad de resultados de la lista mediante un límite y un offset.
Actor principal	Administrador
Actores secundarios	Usuario
Descripción	Este caso de uso representa la funcionalidad que permite al administrador filtrar los registros existentes mediante diferentes criterios.
Nivel de complejidad	Media
Relaciones	AUDIT-2, AUDIT-3
Precondición	El administrador debe estar autenticado ante el sistema.
Trigger	El administrador cambia el límite y *offset* por defecto.
Secuencia normal	
1	El administrador accede a la plataforma de la empresa.
2	El administrador accede a la sección de listado de auditorías.
3	El administrador cambia el límite y *offset* o la paginación de la lista de auditorías
4	El sistema actualiza el listado con aquellos registros que cumplan las condiciones establecidas.
Excepciones	-
Frecuencia esperada	Varias veces al día
Importancia	Media
Prioridad	Largo plazo
Comentarios	-

Cuadro A.4: AUDIT-4 Limitar la cantidad de resultados de la lista mediante un límite y un offset.

AUDIT-5 Eliminar todos los registros de las auditorías de un objeto determinado.	
ID	AUDIT-5
Nombre	Eliminar todos los registros de las auditorías de un objeto determinado.
Actor principal	Administrador
Actores secundarios	Usuario
Descripción	Este caso de uso representa la funcionalidad que permite al administrador eliminar los registros de las auditorías de un objeto determinado.
Nivel de complejidad	Media
Relaciones	AUDIT-1
Precondición	El administrador debe estar autenticado ante el sistema. El objeto del que se quieren eliminar los registros debe haber existido en el sistema.
Trigger	El administrador cambia el límite y *offset* por defecto.
Secuencia normal	
1	El administrador accede a la plataforma de la empresa.
1	El administrador ejecuta la acción de eliminar registros de un objeto, indicando el id.
1	El sistema elimina correctamente todas las acciones auditadas realizadas sobre un objeto.
Excepciones	*El id del objeto no existe* El sistema muestra un mensaje de error indicando que el objeto no existe.
Frecuencia esperada	Varias veces al día
Importancia	Media
Prioridad	Largo plazo
Comentarios	-

Cuadro A.5: AUDIT-5 Eliminar todos los registros de las auditorías de un objeto determinado.

AUDIT-6 Añadir nuevos tipos de objeto.	
ID	AUDIT-6
Nombre	Añadir nuevos tipos de objeto.
Actor principal	Administrador
Actores secundarios	-
Descripción	Este caso de uso representa la funcionalidad que permite al administrador añadir nuevos tipos de objeto que serán auditados.
Nivel de complejidad	Media
Relaciones	AUDIT-1, AUDIT-2, AUDIT-3
Precondición	El administrador debe estar autenticado ante el sistema. No puede existir otro tipo de objeto con el mismo nombre.
Trigger	El administrador quiere insertar un nuevo tipo de objeto.
Secuencia normal	
1	El administrador se autentica en la plataforma de la empresa.
2	El administrador inserta un nuevo tipo de objeto.
3	El sistema guarda el nuevo tipo de objeto y a partir de ahora se podrán auditar acciones del usuario con él.
Excepciones	*Ya existe un tipo de objeto con el nombre especificado.* El sistema muestra un mensaje de error indicando que el tipo de objeto ya existe.
Frecuencia esperada	Varias veces al año
Importancia	Alta
Prioridad	Corto plazo
Comentarios	El tipo de objeto contendrá un identificador, nombre y descripción.

Cuadro A.6: AUDIT-6 Añadir nuevos tipos de objeto.

AUDIT-7 Consultar los tipos de objeto existentes.	
ID	AUDIT-7
Nombre	AUDIT-7 Consultar los tipos de objeto existentes.
Actor principal	Usuario
Actores secundarios	- Administrador
Descripción	Este caso de uso representa la funcionalidad que permite al administrador listar los tipos de objeto existentes en el sistema.
Nivel de complejidad	Baja
Relaciones	AUDIT-1, AUDIT-3, AUDIT-6
Precondición	El administrador debe estar autenticado ante el sistema.
Trigger	El administrador quiere listar los tipos de objeto disponibles.
Secuencia normal	
1	El administrador se autentica en la plataforma de la empresa.
2	El administrador indica que quiere listar los tipos de objeto.
3	El sistema muestra los tipos de objeto disponibles.
Excepciones	-
Frecuencia esperada	Varias veces al día
Importancia	Media
Prioridad	Medio plazo
Comentarios	-

Cuadro A.7: AUDIT-7 Consultar los tipos de objeto existentes.

Anexo B

Diagramas de clases

A continuación se muestran los diagramas de clases que han resultado de la implementación final de las distintas partes del proyecto. Dichos diagramas además incluyen las dependencias entre las distintas clases.

Los diagramas se han generado una vez acabado el desarrollo mediante la herramienta de creación de diagramas de IntelliJ, el IDE utilizado durante el transcurso de la estancia.

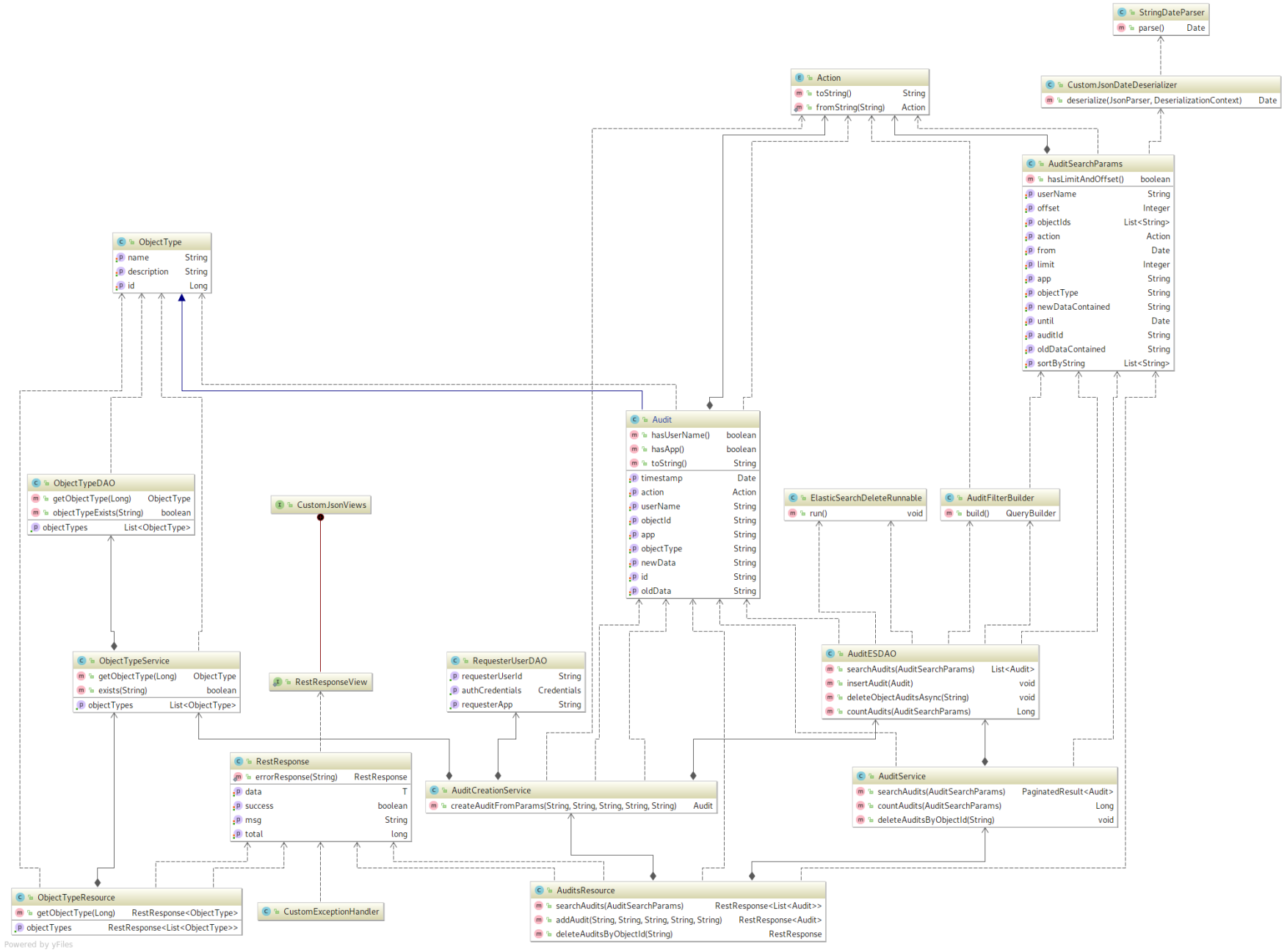


Figura B.1: Diagrama de clases UML del micro-servicio de auditorías.

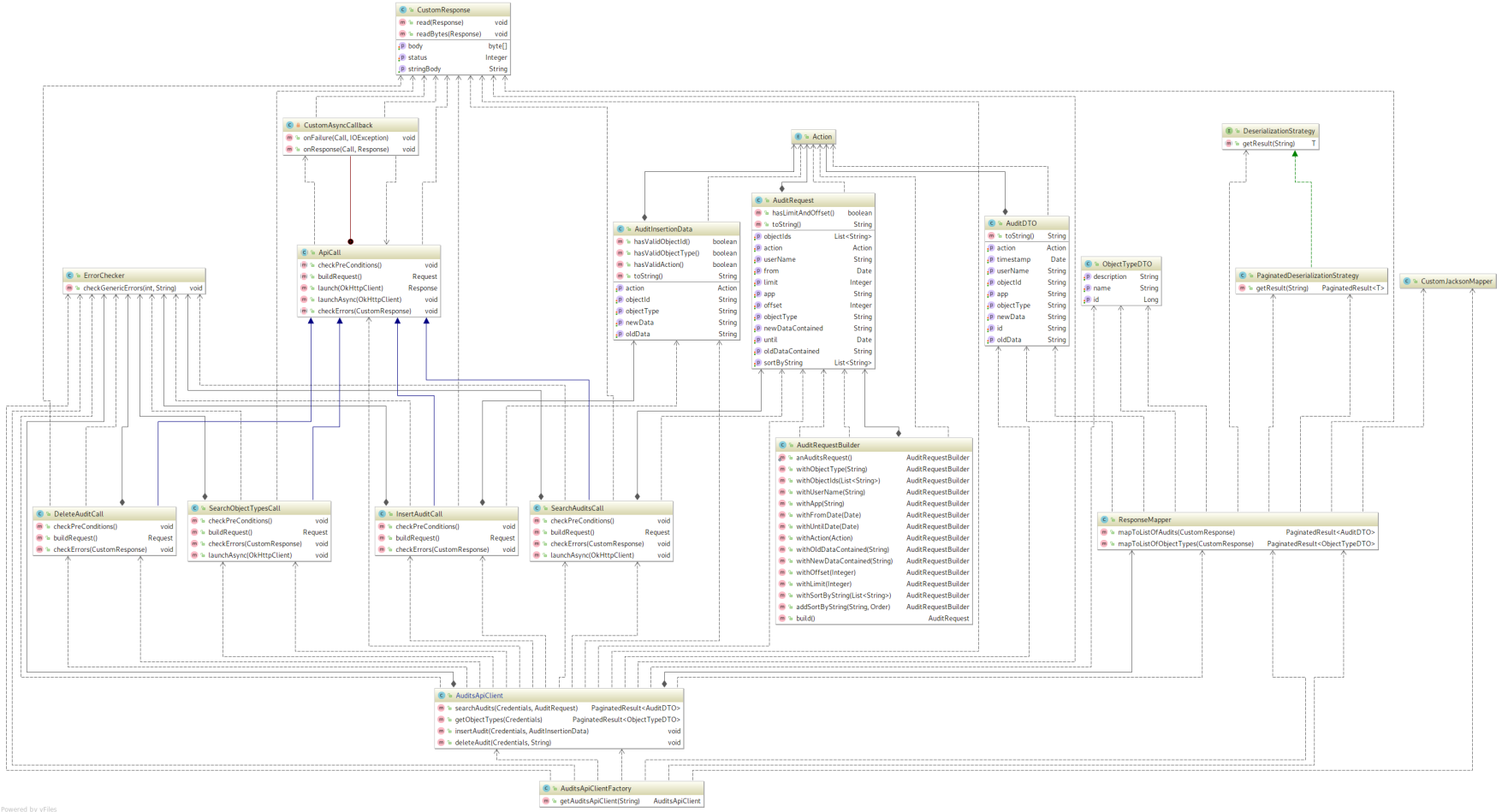


Figura B.2: Diagrama de clases UML del cliente del micro-servicio de auditorías.

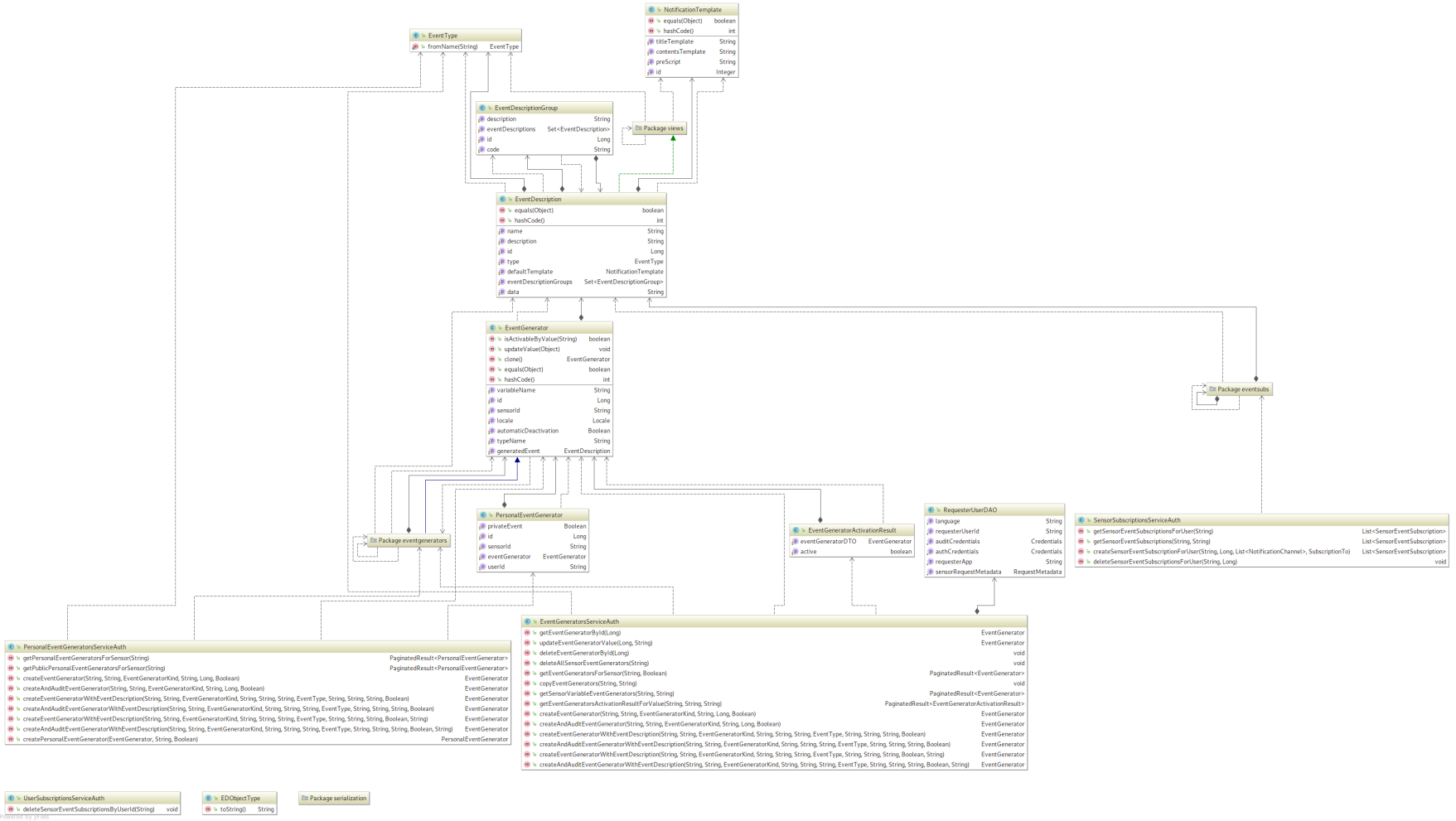


Figura B.3: Diagrama de clases UML del micro-servicio de Event Definitions.

Anexo C

Pruebas de aceptación

- Inserción de registros (AUDIT-1)
  - Por cada acción auditada se añade un nuevo registro en base de datos.
  - Los registros contienen los datos del usuario y la aplicación.
  - El estado inicial y final del objeto auditado se ha guardado en formato JSON y cuenta con toda la información necesaria.
  - Si la llamada no contiene todos los datos necesarios se muestra información del error en la respuesta.
  - Si el tipo de objeto proporcionado no existe, se devuelve un mensaje de error indicando el motivo.
- Listado de registros (AUDIT-2, AUDIT-4)
  - El listado es capaz de mostrar todos los registros.
  - Si se produce un error ajeno a la aplicación la pantalla muestra un mensaje de error con información de la excepción producida.
  - La tabla carga dinámicamente los registros en función de la paginación.
  - La tabla muestra el número de filas por página que se le indica, cargándolo dinámicamente.
  - Los registros se encuentran ordenados por fecha de forma descendente.
  - Si aún no se ha auditado ninguna acción la tabla muestra un mensaje indicando que aún no hay datos que mostrar.
- Filtrado de registros (AUDIT-3, AUDIT-4, AUDIT-7)
  - El seleccionable de tipo de objeto muestra todos los tipos de objeto existentes en el sistema.
  - Todas las entradas de formulario funcionan correctamente permitiendo filtrar los registros que cumplan todas las condiciones de los campos no-vacíos.
  - Las fechas se muestran en el formato deseado.
  - Si se produce una excepción ajena a la aplicación, se muestra un mensaje de error con información de la excepción.

- Eliminación de registros de un objeto determinado (AUDIT-5)
  - Se eliminan los registros de un objeto correctamente.
  - Si el identificador del objeto no existe, se devuelve un mensaje de error que lo indica.
- Inserción de nuevos tipos de objeto (AUDIT-6)
  - Si se proporcionan datos válidos, se inserta correctamente el tipo de objeto en el sistema.
  - En el caso de proporcionar un nombre que ya exista, se devuelve un mensaje de error indicando esto mismo.