Grado en Ingeniería Informática

Trabajo de Final de Grado

Desarrollo de un entorno de pruebas automatizado para un sistema de monitorización de dispositivos IoT

Autor:
Pablo Pitarch Lim

Supervisor:
José Luis Martínez Pérez
Tutor académico:
Sergi Trilles Oliver

Fecha de lectura: 21 de junio de 2023
Curso académico 2022/2023

Resumen

Este documento presenta la memoria de un trabajo de final de grado, consistente en el desarrollo de un entorno de pruebas automatizado para una plataforma existente del sistema de monitorización de edificios de *Smart Building* de la empresa IoTsens. En esta memoria se detalla la planificación, análisis, diseño e implementación de todo el entorno de pruebas.

El entorno de pruebas tiene como objetivo asistir a los desarrolladores a asegurar la calidad del software y ayudar a analizar el rendimiento de las aplicaciones propias de la empresa. Mediante los reportes proporcionados por las diferentes herramientas que conforman el entorno, cualquier miembro del equipo tiene la posibilidad de acceder a los resultados de las pruebas y realizar un análisis sobre la calidad y la seguridad del código.

Para el desarrollo de este entorno, debido a que la aplicación está desarrollada en Angular y Java Springboot, se utilizó Jasmine para el *frontend* y JUnit para el *backend*. Para realizar la integración continua y así automatizar las pruebas, se hizo uso de GitLab, que posteriormente se vinculó con SonarQube para realizar el análisis del código. Para obtener una planificación adecuada se hizo uso de una metodología ágil tipo Scrum.

Palabras clave

Pruebas, automatización, monitorización.

Keywords

Testing, automation, monitoring.

Índice general

Índice de figuras

Índice de cuadros

Índice de códigos

Capítulo 1

Introducción

En este capítulo se va a ofrecer una explicación de la empresa en la que se realiza el proyecto, la razón por la que se realiza y los objetivos y alcance tanto del proyecto como del producto, También se da una descripción del proceso de desarrollo de este proyecto.

1.1. Contexto y motivación del proyecto

Desde el año 2013, IoTsens [1] ofrece soluciones globales enfocadas en el Internet de las Cosas (IoT, en sus siglas en inglés) [2], Big Data e Inteligencia Artificial con el objetivo de conectar el mundo físico con el digital. Su propósito es conseguir un entorno sostenible mediante el uso de tecnologías y soluciones innovadoras que den como resultado un progreso hacia una sociedad más avanzada y limpia. En la Figura 1.1 se muestra el logotipo de la empresa.

Figura 1.1: Logotipo de IoTsens.

IoTsens forma parte de Grupo Gimeno, un grupo empresarial de Castellón, diversificado, con 150 años de experiencia y comprometido con el desarrollo sostenible. Desde sus comienzos, con la creación de Facsa, ha crecido y evolucionado adaptándose a las nuevas tecnologías y necesidades, posicionándose como una de las empresas pioneras en los sectores en los que proporciona sus servicios. En el caso de IoTsens, comenzó siendo una empresa que daba servicio a empresas del Grupo Gimeno, pero con el tiempo, ha ido creciendo hasta posicionarse como una empresa pionera tanto a nivel nacional como internacional.

IoTsens cuenta con una amplia gama de productos y soluciones para diferentes sectores y ámbitos de aplicación. Entre estos se encuentran:

- **Smart City**: solución que proporciona información sobre entornos urbanos para crear

ecosistemas digitales.

- **Smart Water**: solución integral para la gestión del agua con el fin de digitalizar las instalaciones y conseguir una mayor eficiencia en su gestión.
- **Smart Building**: solución para la gestión y control de edificios a través de la recopilación de información relevante procedente de diferentes áreas de gestión con el fin de aumentar la eficiencia, la seguridad y la accesibilidad.
- **Smart Irrigation**: solución integral para la automatización de las zonas de riego con el fin de obtener la información necesarias para la toma inteligente de decisiones en cultivos y jardines resultando en optimización de recursos y eficiencia en su gestión.

Este proyecto surge en la necesidad de mejorar el proceso de pruebas de software de estas soluciones, en concreto, la plataforma *Smart Building*. En el desarrollo de software, es esencial realizar pruebas exhaustivas para garantizar un producto final de calidad. Sin embargo, si estas pruebas se realizan manualmente pueden resultar tediosas y consumir mucho tiempo, y debido a la existencia de las diferentes verticales, es difícil realizar un seguimiento constante y fiable de todo el software de la empresa.

Es por eso que la principal motivación de este proyecto es la automatización del proceso de pruebas de software. Gracias a un sistema automatizado se pueden realizar pruebas más rápidas y eficientes, aumentando la eficiencia del equipo en general. Además, esta automatización permite reducir errores humanos y aumentar la precisión de las pruebas. Es importante destacar que con esto no solo se mejora la eficiencia del equipo, sino que también se mejora la calidad del software. Al realizar pruebas más exhaustivas y precisas, se pueden identificar y solucionar problemas antes de que el software se lance al mercado, lo que a su vez reduce el riesgo de errores y aumenta la satisfacción del cliente.

1.2. Objetivo y alcance del proyecto

El objetivo del proyecto es desarrollar y configurar una herramienta y metodología de testing automático para desarrollos de aplicaciones web. El proyecto busca como subobjetivos:

- Conocer y entender la importancia del testing en los ciclos de desarrollo.
- Conocer las herramientas de testing en entornos web.
- Adquirir experiencia en la utilización de herramientas de test.
- Incorporar el testing dentro de la integración continua.

El proyecto también busca identificar el problema o necesidad que se quiere abordar, definir el alcance del mismo, establecer los objetivos generales y específicos y definir los criterios de evaluación.

1.3. Objetivo y alcance del producto

El objetivo principal del producto resultante de este proyecto es proporcionar una solución automatizada para el proceso de pruebas de software dentro de la empresa. Esta solución debe ser fácil de usar, fiable y eficiente, lo que permitirá al equipo de desarrollo enfocarse en otras tareas críticas.

1.3.1. Alcance funcional

El alcance funcional incluye las características y funcionalidades del producto final. Estas se pueden resumir en lo siguiente:

- Creación y ejecución automatizada de casos de prueba.
- Generación de informes detallados y análisis de resultados.
- Personalización de las pruebas y la interfaz de usuario para adaptarse a las necesidades de la empresa.
- Monitorización continua del software para detectar problemas y errores en tiempo real.
- Integración con herramientas de control de versiones y gestión de proyectos.

1.3.2. Alcance organizativo

El alcance organizativo de un proyecto enfocado al testing automatizado se centra en la implementación de un conjunto de herramientas y prácticas para mejorar la eficiencia y la calidad de los procesos de pruebas en el desarrollo de software, lo que a su vez afecta a la organización y al equipo de desarrollo.

A nivel organizativo, el producto final tiene como objetivo reforzar todas las tareas de desarrollo y mantenimiento de software. Gracias a las pruebas automatizadas y la cobertura de código, se puede realizar un seguimiento de todas las partes del código que se han probado y se puede conocer si realmente funcionan de la manera esperada. Este proceso tiene diversos beneficios a nivel organizativo:

- **Mejora de calidad de software**: gracias a la implementación de pruebas de software, se obtiene una cobertura más completa del código y se pueden identificar los errores más temprano en el proceso de desarrollo.
- **Aumento en el rendimiento del equipo**: la implementación de un proceso de testing automatizado permite al equipo de desarrollo centrarse en la implementación de nuevas características y mejoras, en lugar de dedicar tiempo y esfuerzo a pruebas manuales repetitivas.

- **Mejor colaboración y comunicación**: la integración de herramientas como GitLab y SonarQube permite una mayor colaboración y comunicación entre los miembros del equipo de desarrollo y otros equipos de la organización. Además, gracias a las herramientas de cobertura de código se puede realizar un seguimiento de las pruebas que se ejecutan y las partes del código que se prueban en todo momento. Esto permite una gestión más eficiente del código y una mayor visibilidad de la calidad del software.

- **Reducción de costes y tiempo de desarrollo**: la automatización de los procesos de testing reduce el tiempo y los costos asociados a las pruebas manuales. Esto permite que los desarrolladores dediquen más tiempo a la implementación de nuevas características y a la mejora del código.

1.3.3. Alcance informático

El alcance informático de un proyecto enfocado al testing automatizado se centra en la implementación de un conjunto de herramientas y técnicas para mejorar la eficiencia y la calidad de los procesos de pruebas en el desarrollo de software.

La plataforma *Smart Building* en la que se va basar el proyecto, está desarrollada mediante el uso de **Angular** para el *frontend*, y **Java Springboot** para el *backend*. Es por ello que las herramientas de testing seleccionadas se deben adecuar a estas tecnologías.

Jasmine y Karma son herramientas de testing para Typescript que permiten realizar pruebas unitarias y de integración en el *frontend*. Ambas están integradas en Angular por defecto por lo que son las herramientas ideales para este proyecto. Jasmine es una suite de testing que sigue la metodología Behavior Driven Development (BDD). Tiene cosas muy buenas como que no requiere un DOM para hacer los tests y la sintaxis es bastante sencilla de entender [3]. Mientras, Karma ejecuta las pruebas en diferentes navegadores y sistemas operativos. Ambas herramientas se integran perfectamente con el sistema de construcción y gestión de paquetes de Node.js, lo que hace que la configuración sea fácil y eficiente.

Para el *backend*, se utilizó **JUnit**, un marco de pruebas unitarias para Java. JUnit proporciona una API simple pero poderosa para definir y ejecutar pruebas en código Java. Se integra perfectamente con Maven, la herramienta de construcción y gestión de dependencias de Java, lo que hace que la configuración sea sencilla y fácil de mantener. También se decidió implementar tests con **Cucumber** para implementar pruebas de aceptación en el *backend*. Cucumber es una herramienta de automatización de pruebas de software que se utiliza para escribir pruebas de aceptación en lenguaje natural. Con Cucumber, los desarrolladores y los equipos de pruebas pueden colaborar en la definición y escritura de pruebas de aceptación que se ejecutan de manera automatizada.

En conjunto, el uso de JUnit y Cucumber permite realizar pruebas exhaustivas en el *backend* del software, tanto a nivel de unidad como de aceptación, lo que ayuda a detectar y solucionar problemas antes de que se desplieguen en producción.

Para medir la cobertura de código, se usaron **Jacoco** e **Istanbul**. Jacoco es una herramienta de análisis de cobertura de código para Java que proporciona métricas detalladas sobre la

cobertura de instrucciones, ramas, líneas y métodos en el código. Istanbul, por otro lado, es una herramienta de análisis de cobertura de código para TypeScript que proporciona información detallada sobre la cobertura de instrucciones, ramas y líneas en el código. Ambas herramientas se integran fácilmente con los marcos de pruebas unitarias correspondientes (JUnit para Jacoco e Jasmine para Istanbul).

Se utilizó **GitLab** para automatizar el proceso de testing. GitLab proporciona integración continua (CI/CD) para el proyecto, lo que significa que las pruebas se ejecutan automáticamente en cada cambio de código. Además, se utilizó **SonarQube** para visualizar los resultados de las pruebas y de la cobertura de código. SonarQube es una plataforma de análisis de código estático que proporciona información detallada sobre la calidad del código y las posibles áreas problemáticas.

En cuanto a la planificación del proyecto, se hizo uso de **Redmine**, que es la herramienta de organización utilizada por la empresa. Redmine es una herramienta de gestión de proyectos que permite planificar, controlar y monitorear los proyectos en tiempo real, permitiendo una mejor toma de decisiones y una mayor eficiencia en la ejecución de tareas.

1.4. Descripción detallada del desarrollo del proyecto

El proyecto se comenzó realizando un estudio del software existente. Se analizó tanto la plataforma web, como el código tanto del *backend* como del *frontend*. Una vez se entendió el funcionamiento de la plataforma de *Smart Building*, se identificaron las herramientas y tecnologías que se iban a utilizar. Una vez se instalaron las dependencias necesarias, se comenzó con la implementación.

Se decidió comenzar por el *frontend* debido a la experiencia previa en el uso de Angular y algunas dificultades encontradas para testear en el *backend*. Esto permitió que una vez se completaron las pruebas del *frontend* y se comenzara con el *backend*, se tuviera una experiencia en el testing y la resolución de errores encontrados durante la implementación que probablemente no se hubiera podido realizar al principio del desarrollo de este proyecto.

Una vez terminada la implementación de los tests, se procedió con la automatización de estos. Hubo complicaciones con la configuración de ejecución de las dependencias que se encargan de crear los reportes, pero una vez se solucionó ese problema, con ayuda del administrador de sistemas, se configuró GitLab y SonarQube.

Cuando el vertical de *Smart Building* ya funcionaba correctamente y las pruebas se pasaban automáticamente, se pasó a configurar el entorno de otro vertical, el de *Smart Water*. Básicamente se repitió el proceso de instalación de dependencias, con la única diferencia de que por problemas de compatibilidad con las versiones, se tuvieron que probar otras que permitiesen la integración completa. Este proceso fue mucho más sencillo que en *Smart Building* debido a que ya se poseía la experiencia previa de haber tenido que pasar por los mismos problemas anteriormente.

1.5. Estructura de la memoria

En este apartado se va a dar detalles sobre la memoria. Los contenidos son los siguientes:

- **Capítulo 2 - Planificación del proyecto:** se explica la metodología que se ha utilizado, la planificación y seguimiento del estado del proyecto, y una estimación de los recursos y costes del proyecto.
- **Capítulo 3 - Análisis del sistema:** se realiza un estudio sobre las necesidades y el funcionamiento del sistema.
- **Capítulo 4 - Diseño del sistema:** se explica la metodología de testing utilizada y la configuración del entorno de pruebas
- **Capítulo 5 - Implementación y pruebas:** se explica cómo se han implementado las pruebas y se realiza un análisis de los resultados obtenidos de estas.
- **Capítulo 6 - Conclusiones:** se presentan las conclusiones a las que se ha llegado tras el transcurso del proyecto y los conocimientos adquiridos durante la estancia de prácticas.

Capítulo 2

Planificación del proyecto

En este capítulo, se va a explicar la metodología seguida para la planificación del proyecto. Con ello, se va a realizar un seguimiento del progreso del proyecto incluyendo las tareas realizadas. También se va a hacer una estimación de los costes del desarrollo del proyecto y los posibles riesgos que pueden surgir durante la realización del mismo.

2.1. Metodología

Para el proyecto, se optó por una metodología ágil, específicamente, la metodología Scrum. Esta metodología se enfoca en dividir el proyecto final en tareas más pequeñas, conocidas como sprints, y como el proyecto actual se puede dividir fácilmente en etapas más pequeñas e independientes, se organizaron por funcionalidades necesarias. Si se encontraba algún imprevisto que impedía el avance de las tareas relevantes, se trabajaba en objetivos secundarios hasta que se solucionara el problema, lo que permitía mantener un flujo de trabajo constante.

Aunque no se siguió estrictamente la metodología Scrum, se implementaron algunos de sus principios. Por ejemplo, se realizaban reuniones diarias de 15 minutos con todas las partes implicadas para comprobar el progreso del proyecto, el estado de las tareas asignadas y asegurarse de que se cumplían los objetivos a un ritmo adecuado. Estas reuniones permitían detectar problemas en las implementaciones de algunas funcionalidades y reaccionar rápidamente si alguna parte del proyecto no se estaba implementando al ritmo adecuado, o cambiar los requisitos del proyecto sin provocar cambios radicales en las funcionalidades ya implementadas.

Además, cada dos semanas se realizaba una reunión individual con el supervisor del proyecto para informar del progreso realizado, problemas encontrados y discutir posibles soluciones. La empresa utilizaba la herramienta Redmine para organizar el trabajo mediante el uso de tickets, lo que permitía la asignación y seguimiento de los objetivos entre todas las partes involucradas. De esta manera, se garantizaba la gestión efectiva y el seguimiento adecuado de todo el proyecto.

2.2. Planificación temporal del proyecto

Una pila de producto es una herramienta perteneciente a una metodología ágil, como Scrum, que es la que se utiliza para este proyecto. Básicamente es una lista ordenada del trabajo que el cliente considera necesario para desarrollar el producto. Esta lista contiene tareas que se dividen en detalles como funcionalidades, mejoras, tecnologías y corrección de errores necesarios para el proyecto. La pila de trabajo nunca se considera completa ya que está en constante crecimiento y evolución, y se modifica durante el desarrollo del proyecto, adaptándose a los imprevistos y nuevas necesidades.

En esta sección, se describirá la estructura de la pila inicial del producto que consistía en 4 sprints. Los sprints se han dividido en subtareas, similar al sistema de tickets utilizado en la empresa. Como no es un proyecto desarrollado para un cliente en particular, los sprints no se basaron en los requisitos del usuario, sino en las subtareas. Cada tarea tendrá una puntuación asignada según su importancia y complejidad al realizarla.

T01 - Formación en las herramientas

Tareas: Aprender el funcionamiento de Angular y SpringBoot con tutoriales y proyectos de prueba. Aprender la implementación y ejecución de los tests en un proyecto vacío.

Requisitos: Obtener conocimientos necesarios para poder desarrollar el producto de software.

Se busca adquirir conocimientos sobre las herramientas principales que se utilizarán en el desarrollo del producto de software. Específicamente, se debe aprender cómo trabajar con Angular y SpringBoot a través de tutoriales y proyectos de prueba. Además, es importante comprender cómo implementar y ejecutar pruebas en un proyecto vacío.

T02 - Entendimiento del producto

Tareas: Analizar el funcionamiento de la plataforma de *Smart Building* y su código

Requisitos: Conocer en profundidad el funcionamiento de *Smart Building* para ser capaz de detectar los tests necesarios.

Se debe realizar un análisis exhaustivo del funcionamiento de la plataforma *Smart Building* y estudiar su código. El objetivo es comprender en profundidad cómo funciona el producto y familiarizarse con su estructura. Este conocimiento es necesario para identificar los tests requeridos y asegurarse de que se cubren todos los aspectos importantes del software.

T03 - Configuración de herramientas de *frontend*

Tareas: Añadir las dependencias y modificar los archivos de configuración necesarios para poder ejecutar los tests de Jasmine mediante Karma.

Requisitos: Poder ejecutar los tests para Angular.

En esta etapa, se centra en la configuración de las herramientas necesarias para ejecutar pruebas en el *frontend*. Se deben añadir las dependencias necesarias y realizar modificaciones en los archivos de configuración para poder ejecutar los tests de Jasmine utilizando Karma. Esto permitirá probar el código escrito en Angular y asegurarse de que funciona correctamente.

T04 - Configuración de herramientas del *backend*

Tareas: Añadir las dependencias y modificar los archivos de configuración necesarios para poder ejecutar los tests de JUnit.

Requisitos: Poder ejecutar tests para Java Springboot

En esta etapa, se enfoca en la configuración de las herramientas necesarias para ejecutar pruebas en el *backend*. Se deben añadir las dependencias necesarias y realizar modificaciones en los archivos de configuración para poder ejecutar los tests de JUnit. Esto permitirá probar el código escrito en Java con SpringBoot y verificar su funcionamiento correcto.

T05 - Implementación tests *frontend*

Tareas: Desarrollar y ejecutar pruebas unitarias y de integración utilizando Jasmine y Karma en el código de Angular.

Requisitos: Garantizar el correcto funcionamiento de los componentes del *frontend*.

Se debe desarrollar y ejecutar pruebas tanto unitarias como de integración en el código del *frontend* utilizando Jasmine y Karma. El objetivo es verificar el correcto funcionamiento de los componentes desarrollados en Angular y asegurarse de que cumplen con los requerimientos establecidos.

T06 - Implementación tests *backend*

Tareas: Desarrollar y ejecutar pruebas unitarias y de integración utilizando JUnit en el código de Java Springboot.

Requisitos: Garantizar el correcto funcionamiento de los controladores del *backend*.

Se debe desarrollar y ejecutar pruebas tanto unitarias como de integración en el código del *backend* utilizando JUnit. El objetivo es asegurarse de que los controladores del *backend* funcionen correctamente y cumplan con las especificaciones.

T07 - Tests de aceptación

Tareas: Desarrollar y ejecutar pruebas de aceptación Cucumber para verificar el correcto funcionamiento del software en un entorno real.

Requisitos: Tener pruebas de aceptación que sean fácil de entender por parte del cliente y que prueben la funcionalidad de la aplicación final.

Se deben desarrollar y ejecutar pruebas de aceptación utilizando Cucumber. Estas pruebas

permiten verificar el correcto funcionamiento del software en un entorno real y asegurarse de que cumple con los requisitos del cliente. Es importante que las pruebas de aceptación sean comprensibles para el cliente y cubran todas las funcionalidades importantes de la aplicación final.

T08 - Cobertura de código

Tareas: Utilizar Istanbul y Jacoco para medir la cobertura de código de las pruebas desarrolladas y garantizar que se están evaluando todas las funcionalidades del software.

Requisitos: Cada vez que se ejecutan las pruebas, se debe actualizar el fichero de cobertura de código, mostrando la versión más actualizada del estado del código.

Se deben desarrollar y ejecutar pruebas de aceptación utilizando Cucumber. Estas pruebas permiten verificar el correcto funcionamiento del software en un entorno real y asegurarse de que cumple con los requisitos del cliente. Es importante que las pruebas de aceptación sean comprensibles para el cliente y cubran todas las funcionalidades importantes de la aplicación final.

T09 - Automatización

Tareas: Automatizar la ejecución de las pruebas desarrolladas para garantizar que se están evaluando todas las funcionalidades del software de forma continua y eficiente.

Requisitos: Las pruebas se deben ejecutar cada vez que se haga un *commit* gracias a la integración continua de GitLab.

Se busca automatizar la ejecución de las pruebas desarrolladas. Esto permite garantizar que se evalúan todas las funcionalidades del software de manera continua y eficiente. Se debe configurar la integración continua de GitLab para que las pruebas se ejecuten automáticamente cada vez que se realiza un *commit*.

T10 - Análisis de código

Tareas: Realizar análisis estático del código utilizando SonarQube para detectar posibles errores, vulnerabilidades o malas prácticas en el código.

Requisitos: Poder visualizar las estadísticas de calidad del software en SonarQube.

Se utiliza SonarQube para realizar análisis estático del código. Esto permite detectar posibles errores, vulnerabilidades o malas prácticas en el código. Es necesario visualizar las estadísticas de calidad del software en SonarQube para obtener información relevante sobre la calidad del código y poder tomar las acciones necesarias para mejorarlo.

T11 - Expansión del testing en la empresa

Tareas: Definir e implementar políticas y estrategias de testing en la empresa para garantizar la calidad del software desarrollado en todos los proyectos.

Requisitos: Poder escribir pruebas de software en otros verticales.

Se busca establecer políticas y estrategias de testing en la empresa con el objetivo de garantizar la calidad del software desarrollado en todos los proyectos. Esto implica definir estándares de calidad, establecer procesos de pruebas y promover buenas prácticas de testing en todos los equipos de desarrollo. Además, se debe ser capaz de escribir pruebas de software en otros proyectos de la empresa, lo que requiere una comprensión sólida de las técnicas de testing y la capacidad de adaptarse a diferentes contextos.

T12 - Documentación

Tareas: Realizar la documentación acerca de cómo configurar y ejecutar los tests.

Requisitos: Proporcionar la documentación necesaria para que cualquier miembro del equipo sea capaz de utilizar el producto.

Se debe crear documentación detallada que explique cómo configurar y ejecutar las pruebas desarrolladas. Esta documentación servirá como referencia para el equipo de desarrollo y otros interesados, asegurando que los tests se puedan ejecutar correctamente en diferentes entornos y circunstancias.

2.3. Seguimiento del proyecto

2.3.1. Primer sprint

Tareas del sprint:

- T01 - Formación en las herramientas.
- T02 - Entendimiento del producto.

Subtareas del sprint:

- Comprender los requisitos de la solución de *Smart Building* y elaborar un plan de pruebas detallado que cubra todos los aspectos clave de la solución.
- Diseño de casos de prueba basados en los requisitos del proyecto.

El objetivo de este sprint era adecuarse a las herramientas utilizadas en el proyecto y comprender los requisitos y la funcionalidad de *Smart Building*.

La primera semana se dedicó a la formación de herramientas y entender la plataforma de *Smart Building*. Para ello, se crearon unos proyectos de prueba y se siguieron ciertos tutoriales

recomendados por la empresa para la familiarización con las herramientas y la metodología de trabajo utilizadas en la empresa.

Una vez se disponía de los conocimientos necesarios, se comenzaron a evaluar las necesidades de la plataforma y de esta manera poder diseñar los casos de prueba requeridos. Estos casos de prueba no eran definitivos debido a que todavía no se conocía al completo el funcionamiento de las herramientas de testing.

2.3.2. Segundo sprint

Tareas del sprint:

- T03 - Configuración de herramientas de *frontend*.
- T05 - Implementación tests *frontend*.
- T08 - Cobertura de código.

Subtareas del sprint:

- Investigación y selección de herramientas adecuadas para el proyecto.
- Instalación y configuración de herramientas de desarrollo, como el editor de código y el sistema de control de versiones.
- Configuración de la infraestructura necesaria para la compilación y ejecución de las pruebas.
- Identificación y diseño de pruebas para el *frontend*.
- Implementación de pruebas unitarias para el *frontend*.
- Identificación de áreas de la aplicación con una baja cobertura de código.
- Diseño y implementación de pruebas para aumentar la cobertura de código en las áreas identificadas.

El objetivo del segundo sprint fue configurar las herramientas de *frontend* y mejorar la calidad del código a través de la identificación y resolución de áreas con baja cobertura de código, así como la implementación de pruebas de unidad.

Para lograr estas metas, se llevó a cabo una investigación exhaustiva para seleccionar las herramientas adecuadas para el proyecto. A continuación, se procedió a la instalación y configuración de estas herramientas, que incluyen un editor de código y un sistema de control de versiones. Se configuró la infraestructura necesaria para la compilación y empaquetado de la aplicación.

Luego, se identificaron las áreas del *frontend* que necesitaban mejoras en la cobertura de código y se diseñaron pruebas para aumentar la cobertura en estas áreas. Se implementaron pruebas unitarias para el *frontend* con el objetivo de mejorar la calidad del código y asegurarse de que la aplicación funcione según lo esperado.

Posteriormente, también se enfocó en la identificación y resolución de áreas con baja cobertura de código, para lo cual se diseñaron y ejecutaron pruebas específicas. El resultado final fue una mejora significativa en la cobertura de código en estas áreas.

2.3.3. Tercer sprint

Tareas del sprint:

- T04 - Configuración de herramientas del *backend*.
- T06 - Implementación tests *backend*.
- T08 - Cobertura de código.

Subtareas del sprint:

- Configuración del entorno de desarrollo para el *backend*, incluyendo la instalación de todas las herramientas necesarias.
- Implementación de pruebas de integración y pruebas unitarias en el *backend*.
- Aumento de la cobertura de código en el *backend* mediante la identificación de áreas con baja cobertura y la creación de pruebas adicionales.

El objetivo del tercer sprint era asegurar la calidad y la estabilidad del *backend* de *SmartBuilding*, para lo cual se trabajó en la configuración del entorno de desarrollo del *backend* y en la implementación de pruebas para validar su correcto funcionamiento.

Para la configuración del entorno, se instalaron todas las herramientas necesarias para el desarrollo del *backend*, como el servidor de aplicaciones y el sistema de gestión de bases de datos.

Posteriormente, se diseñaron y ejecutaron pruebas de integración y pruebas unitarias para validar el correcto funcionamiento del *backend* y garantizar su estabilidad. Estas pruebas también permitieron identificar áreas con baja cobertura de código, lo que llevó a la creación de pruebas adicionales para aumentar la cobertura y mejorar la calidad del código.

Al finalizar el sprint, se logró aumentar significativamente la cobertura de código en el *backend*, lo que permitió mejorar la calidad y estabilidad de *SmartBuilding* en su conjunto.

2.3.4. Cuarto sprint

Tareas del sprint:

- T07 - Tests de aceptación.
- T09 - Automatización.
- T10 - Análisis de código.
- T11 - Expansión del testing en la empresa.
- T12 - Documentación.

Subtareas del sprint:

- Diseño y ejecución de pruebas de aceptación para validar la funcionalidad completa del producto.
- Automatización de pruebas de regresión para mejorar la eficiencia y la consistencia del testing.
- Análisis de código para identificar áreas de mejora y posibles problemas de seguridad.
- Expansión del testing en la empresa mediante la creación de guías y recursos para que otros equipos puedan implementar las mejores prácticas de testing.
- Documentación de los procesos y los resultados del testing para facilitar la colaboración y el mantenimiento del producto.

El objetivo del cuarto sprint fue asegurar la calidad de *Smart Building* en su conjunto y permitir su expansión y mantenimiento a largo plazo. Para lograrlo, se trabajó en la creación de pruebas de aceptación para validar la funcionalidad completa del producto y en la automatización de pruebas de regresión para mejorar la eficiencia del testing.

Además, mediante el uso de SonarQube, se realizó un análisis exhaustivo del código para identificar áreas de mejora y posibles problemas de seguridad, y se crearon guías y recursos para que otros equipos puedan implementar las mejores prácticas de testing y mejorar la calidad de sus productos.

Finalmente, se documentaron todos los procesos y resultados del testing para facilitar la colaboración y el mantenimiento de *Smart Building* a largo plazo. De esta manera, se aseguró la calidad del producto y se sentaron las bases para su expansión y mejora continua en el futuro.

2.3.5. Calendario

Para alcanzar las trescientas horas asignadas al desarrollo del proyecto, se dedicaron cinco horas cada dıa por parte del alumno, para su desarrollo. Se empezó la estancia de prácticas el día

veintiuno de noviembre, y se completó el veintiséis de febrero. Por razones externas como puede ser la asistencia a clase, hay algunos días que se acudieron menos horas, pero se compensaron extendiendo la estancia en prácticas. A continuación se muestra un resumen de la distribución de las horas agrupado por semanas.

Noviembre (50 horas):

- Semana del 21 al 25: 25 horas
- Semana del 28 al 2: 25 horas

Diciembre (75 horas):

- Semana del 5 al 9: 15 horas
- Semana del 12 al 16: 25 horas
- Semana del 19 al 23: 20 horas
- Semana del 26 al 27: 10 horas

Enero (114 horas):

- Semana del 2 al 5: 20 horas
- Semana del 9 al 13: 25 horas
- Semana del 16 al 20: 20 horas
- Semana del 23 al 27: 25 horas
- Semana del 30 al 3: 24 horas

Febrero (61 horas):

- Semana del 6 al 10: 24 horas
- Semana del 13 al 16: 19 horas
- Semana del 20 al 23: 18 horas

2.4. Costes

En este apartado se describirán los costes aproximados relacionados con el desarrollo del proyecto y los recursos utilizados.

2.4.1. Recursos humanos

Para llevar a cabo este proyecto, se estableció un equipo formado por un programador junior, un administrador de sistemas y el supervisor del proyecto, que actuó como jefe de proyecto. Esta estrategia de equipo permitió mantener los costos y los recursos en el proyecto a un nivel óptimo, y se logró llevar a cabo el desarrollo de manera eficiente y efectiva. La colaboración entre el programador junior y el supervisor de la empresa, como jefe de proyecto, permitió una comunicación y colaboración efectivas, y la flexibilidad para adaptarse a cualquier imprevisto que surgiera durante el desarrollo. El administrador de sistemas se encargó de asegurarse de que los servidores y los sistemas necesarios para el proyecto funcionaran de manera eficiente y efectiva, y estuvieran disponibles para el equipo de desarrollo en todo momento. Su aporte permitió mantener la estabilidad y la disponibilidad de los recursos del proyecto, lo que contribuyó significativamente al éxito del mismo.

2.4.2. Recursos tecnológicos

En cuanto a los costes relacionados con el software, se utilizó principalmente software gratuito y de código abierto, con la excepción de la licencia de IntelliJ y la cuenta de Microsoft que se recibe al entrar en la empresa. En general, la mayoría de las herramientas utilizadas eran gratuitas y de acceso público, lo que permitió un ahorro significativo en costes de licencias.

En cuanto a los costes relacionados con el hardware, la empresa proporcionó todos los dispositivos necesarios. Se proporcionó un ordenador de oficina con los periféricos necesarios y se utilizaron los servidores que debían ser monitorizados, que alojaban las bases de datos de la empresa y almacenaban los repositorios y proyectos de la empresa.

En resumen, se logró reducir los costes en software gracias al uso de herramientas de código abierto y gratuito, y los costes relacionados con el hardware fueron proporcionados por la empresa.

2.4.3. Costes totales

Para calcular los costes totales del proyecto, es necesario sumar los costes de los recursos tecnológicos y los recursos humanos. Además, es necesario tener en cuenta los costes indirectos como la luz, el agua o el alquiler. Respecto a los recursos humanos, se deben incluir los costes del sueldo del programador junior y el jefe de proyecto. La Tabla 2.1 muestra los valores totales:

Rol	Horas diarias	Sueldo/hora (€)	Coste total (€)
Programador junior	5	11,52	3456,00
Jefe de proyecto	0,5	17,31	519,30
Administrador de sistemas	0,5	14,24	427,20

Cuadro 2.1: Costes humanos.

Los salarios se han obtenido consultando la plataforma de búsqueda y comparación de ofertas de trabajo, *Indeed* [4]. Esta plataforma proporciona estadísticas sobre los salarios medios para

un puesto y ubicación específicos. Los costes de los recursos humanos se calcularon en función de las horas diarias dedicadas durante el proyecto.

Para determinar los costes de los recursos tecnológicos, es necesario calcular los gastos asociados con el hardware y software utilizados en el proyecto. Aunque la empresa ya disponía de estos recursos con anterioridad y no se requirió ninguna inversión adicional para la realización del proyecto, se llevarán a cabo los cálculos como si se hubiera realizado una inversión (consultar Tabla 2.2).

Hardware	Coste (€)
HP ProDesk 400 G3 SFF	158,00
HP ProDisplay P222va x2	89x2 = 178,00
IntelliJ Idea Ultimate	499,00

Cuadro 2.2: Costes tecnológicos.

A continuación, vamos a agregar los costes indirectos al cálculo de los costes totales. Para estimar los costes indirectos (Tabla 2.3), se considerarán los gastos de luz, agua y alquiler durante el periodo de prácticas. Los datos para la estimación se han tomado de *Enalquiler* [5].

Gasto	Coste (€)
Luz y agua	300x3 = 900,00
Alquiler	839x3 = 2.517,00

Cuadro 2.3: Costes indirectos.

El coste total de los recursos se puede ver en la Tabla 2.4. Además de los costes de los recursos humanos, se deben agregar los costes de contratación y seguridad social, que representan aproximadamente el 20 % del coste original.

Recurso	Coste (€)
Costes humanos	4.402,50
Costes tecnológicos	835,00
Costes indirectos	3.417,00
Costes totales	8.654,50

Cuadro 2.4: Costes totales.

2.5. Riesgos

En esta sección, se describirán los posibles riesgos que podrían surgir durante el desarrollo del proyecto. El equipo es pequeño y consta solo de un programador. De esta forma, los riesgos del proyecto dependen principalmente del rendimiento individual de este miembro del equipo. No se esperan riesgos relacionados con la asignación de tareas o factores externos, ya que todos los sistemas a monitorizar funcionaban correctamente al inicio del proyecto y la implementación del proyecto depende en gran medida de una única persona.

Para llevar a cabo el análisis de riesgos, se ha elaborado una lista de todos los riesgos identificados y su categoría correspondiente (consultar Tabla 2.5). Los riesgos generales pueden afectar

Código	Descripción	Tipo de riesgo
R01	Falta de experiencia con las tecnologías del proyecto	General
R02	Retrasos en el desarrollo del proyecto	General
R03	Configuración incorrecta del entorno de testing	Específico
R04	Incompatibilidad entre versiones de software	Específico
R05	Datos de prueba no representativos	Específico

Cuadro 2.5: Lista de riesgos identificados.

al proyecto en cualquier momento de su desarrollo, mientras que los específicos se asocian con problemas puntuales. En la Tabla 2.6 se identifican las posibles causas que podrían desencadenar los riesgos mencionados anteriormente, y en la Tabla 2.7 se describen las medidas de prevención y contingencia adoptadas para mitigar los riesgos identificados en las tablas previas.

R01 - Falta de experiencia
Magnitud: Alta **Descripción:** El desarrollador carece de los conocimientos necesarios **Impacto:** Retrasos o bloqueos en el avance del proyecto **Indicadores:** Acumulación de tareas
R02 - Retrasos en el desarrollo
Magnitud: Alta **Descripción:** Las tareas no se cumplen en el tiempo estimado **Impacto:** Retraso en el comienzo de tareas planificadas más adelante **Indicadores:** Demasiado tiempo dedicado a ciertas tareas
R03 - Configuración incorrecta del entorno
Magnitud: Alta **Descripción:** El sistema automatizado no funciona como debería **Impacto:** Los tests no se pueden comprobar automáticamente **Indicadores:** Al realizar un *commit*, no se ejecutan correctamente las pruebas
R04 - Incompatibilidad entre versiones
Magnitud: Baja **Descripción:** Existen incompatibilidades entre las versiones del software o de las dependencias añadidas al proyecto **Impacto:** Se producen errores al compilar o ejecutar el proyecto **Indicadores:** Al instalar el software o las dependencias, el IDE o el sistema indica que se produce un error
R05 - Datos de prueba no representativos
Magnitud: Media **Descripción:** Los datos de prueba introducidos no proporcionan pruebas adecuadas **Impacto:** Es posible que el resultado de las pruebas sea incorrecto sin saberlo **Indicadores:** Las pruebas no se comportan como deberían

Cuadro 2.6: Análisis de riesgos.

Código	Plan de prevención	Plan de contingencia
R01	Dedicar tiempo a aprender a utilizar las herramientas y tecnologías antes de empezar	Comenzar con tareas simples, y subir la dificultad con el paso del tiempo
R02	Estimar el tiempo adecuado en función a las capacidades del desarrollador	Reasignar el tiempo estimado en caso de que se observe un retraso en el seguimiento de las tareas
R03	Establecer una guía antes de comenzar a instalar las dependencias	Realizar pruebas de validación exhaustivas antes de implementar el entorno de testing.
R04	Mantener una gestión de versiones adecuada y documentar las dependencias entre los componentes de software.	Establecer un proceso de prueba y validación exhaustivo para garantizar la compatibilidad entre las diferentes versiones de software antes de su implementación.
R05	Definir criterios claros para seleccionar y generar datos de prueba representativos de los casos de uso y escenarios del sistema.	Definir criterios claros para seleccionar y generar datos de prueba representativos de los casos de uso y escenarios del sistema.

Cuadro 2.7: Prevención y contingencia de riesgos.

Capítulo 3

Análisis del sistema

En este capítulo, se detallará una de las primeras fases del desarrollo de software: la definición de los requisitos. Durante esta fase, se lleva a cabo un análisis detallado de los requisitos del sistema que se va a desarrollar. Estos requisitos son fundamentales para el desarrollo del proyecto, ya que afectarán su planificación e implementación.

Los requisitos describen la funcionalidad que el proyecto final debe proporcionar. En el caso de haberse utilizado una metodología ágil, como es el caso, los requisitos se han agrupado en conjuntos de funcionalidades independientes para facilitar su planificación y desarrollo.

3.1. Análisis del software existente

Para empezar, es necesario hacer un análisis de la estructura del software sobre el cual se van a realizar las pruebas. La plataforma de *Smart Building* está compuesta por diferentes secciones que funcionan conjuntamente proporcionando un servicio al cliente.

Las diferentes secciones son las siguientes:

- **Buildings:** Son la base de la plataforma. Se puede indicar la ubicación en un mapa y pueden incluir *widgets*. Están compuestos por pisos (Floors). En la Figura 3.1 se puede observar su estructura.
- **Floors:** Están compuestos por zonas (Zones).
- **Zones:** Contienen dispositivos (Devices).
- **Devices:** Se reparten por los edificios para recopilar datos. Estos datos posteriormente se convierten en KPIs (Key Performance Indicator).
- **KPIs:** Son indicadores clave de rendimiento. Se calculan para dar al cliente una idea del estado de su edificio.

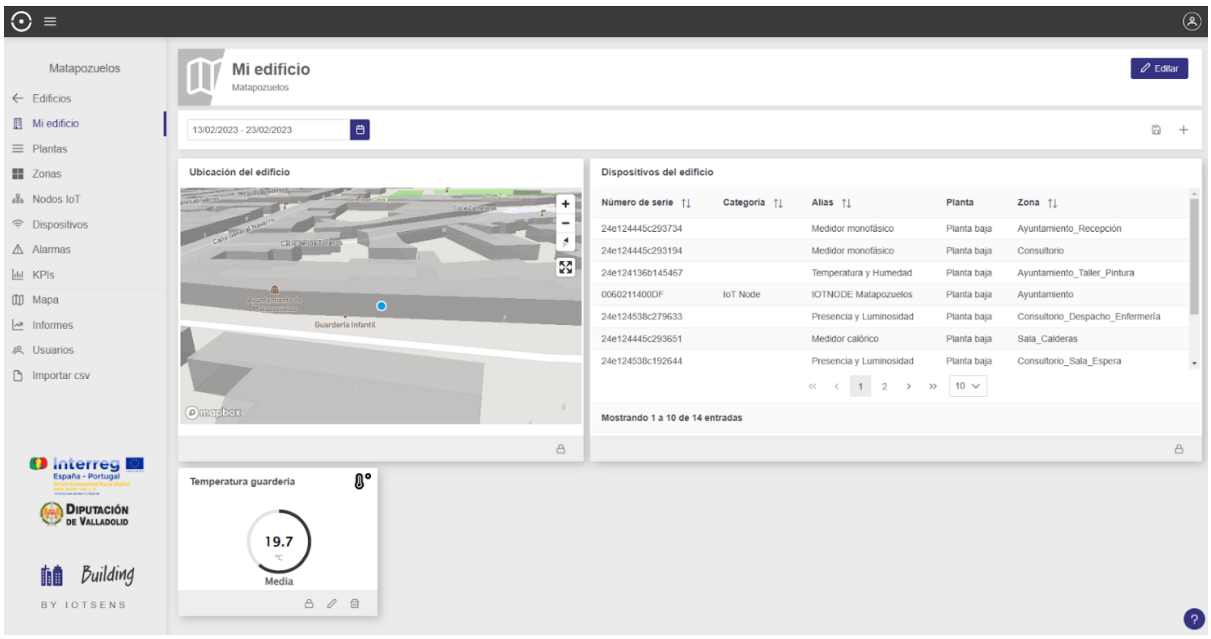

Figura 3.1: Dashboard de un edificio de Smart Building.

Las acciones que puede realizar el usuario y por lo tanto se han realizado pruebas para ello son las siguientes:

- **Crear nuevo:** mediante el uso de un formulario, el usuario puede crear un elemento, personalizando sus propiedades.
- **Editar:** mediante el uso de formularios, el usuario puede modificar propiedades de elementos creados previamente.
- **Ver listado:** el usuario puede observar un listado con todos los elementos de una clase, pudiendo observar y acceder a su información.
- **Ver información:** el usuario puede observar la información detallada de cada uno de los elementos.

Además, cada edificio tiene la posibilidad de crear unos *widgets* personalizados. Estos *widgets* permiten ver rápidamente los datos recopilados por los dispositivos. Existen diversos tipos de *widgets*. Entre ellos se encuentran gráficas, tablas o iconos.

3.2. Definición de requisitos

3.2.1. Historias de usuario

En esta sección se detallarán los requisitos del proyecto exclusivamente desde la perspectiva de los usuarios finales. Ya que es una herramienta que va a ser utilizada dentro de la empresa, se

incluye a los desarrolladores, el propietario del producto y el administrador de sistemas, junto con el cliente. Debido a que se ha utilizado una metodología ágil en su desarrollo, se emplearán Historias de Usuario (HU) para definir los requisitos finales.

Las HU no se han incluido en la definición de tareas de la planificación temporal ya que no deberían afectar el desarrollo de las diversas funcionalidades del proyecto, sino que simplemente servirían como una guía para los componentes del proyecto con los que el usuario final no interactúa.

A continuación, se presentarán todas las HU que se deben implementar para completar la funcionalidad requerida del proyecto.

Épica 1: Desarrollo de pruebas

- HU01: como desarrollador, quiero poder configurar un entorno de testing local para mi aplicación, de modo que pueda ejecutar pruebas y validar cambios antes de hacer un *commit*.
- HU02: como propietario del producto, quiero poder visualizar informes detallados de las pruebas de la aplicación, de modo que pueda evaluar la calidad del producto y tomar decisiones informadas sobre su lanzamiento.
- HU03: como usuario final, quiero tener acceso a una documentación clara y detallada sobre cómo utilizar la aplicación, de modo que pueda entender cómo funciona y qué esperar de ella.

Épica 2: Análisis del sistema

- HU04: como propietario del producto, quiero poder automatizar la ejecución de pruebas de regresión para mi aplicación, de modo que pueda detectar problemas temprano y minimizar el riesgo de fallos en producción.
- HU05: como propietario del producto, quiero poder visualizar informes detallados de las pruebas de la aplicación, de modo que pueda evaluar la calidad del producto y tomar decisiones informadas sobre su lanzamiento.
- HU06: como administrador de sistemas, quiero poder configurar entornos de prueba y producción separados, de modo que pueda realizar pruebas exhaustivas sin afectar el rendimiento o la disponibilidad de la aplicación en producción.

Épica 3: Cliente

- HU07: como cliente, quiero que la aplicación sea probada exhaustivamente, de modo que pueda estar seguro de que funcionará correctamente.
- HU08: como usuario final, quiero tener acceso a una documentación clara y detallada sobre cómo utilizar la aplicación, de modo que pueda entender cómo funciona y qué esperar de ella.

3.2.2. Diagrama de casos de uso

El diagrama de casos de uso (UCD) es una herramienta que se utiliza para representar los procesos y sistemas de un proyecto. Permite visualizar las relaciones existentes entre los distintos elementos del sistema, así como la forma en que interactúan entre ellos. A continuación se puede observar el UCD para este proyecto en la Figura 3.2.

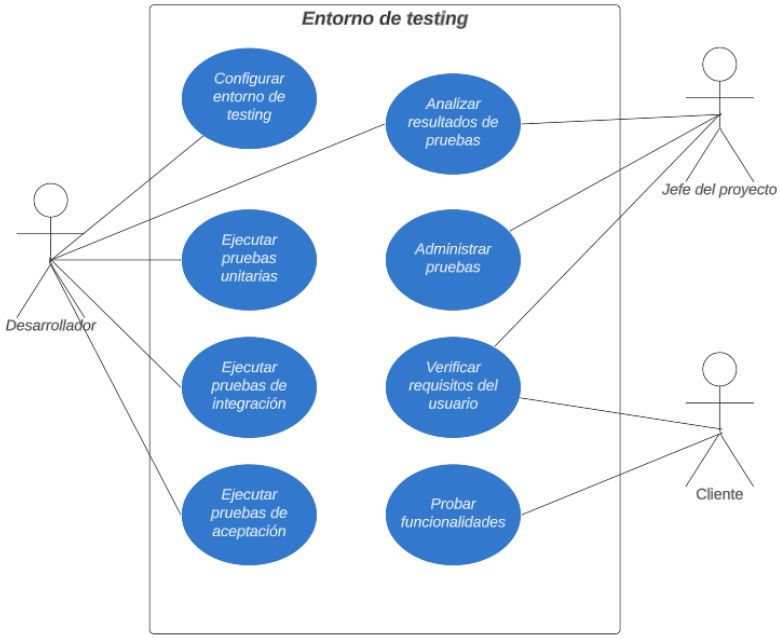

Figura 3.2: Diagrama de casos de uso.

3.3. Análisis de requisitos

A partir de todos los casos de uso anteriormente especificados, se pueden obtener los requisitos funcionales de la aplicación. De esta manera, el proyecto puede segmentarse por funcionalidades que serán accesibles para el usuario desde la interfaz principal una vez que se haya iniciado sesión exitosamente.

Estos requisitos funcionales son esenciales para el desarrollo del proyecto, ya que definen las funcionalidades que la aplicación debe proporcionar y cómo estas se relacionan entre sí. Además, permiten establecer una planificación adecuada para el desarrollo y la implementación de las funcionalidades.

- Configuración de pruebas: el sistema debe permitir la configuración de diferentes tipos de pruebas, con el fin de evaluar el rendimiento y la calidad de los dispositivos.
- Automatización de pruebas: el sistema debe permitir la automatización de las pruebas, a fin de reducir el tiempo y los costos de pruebas manuales.

- Análisis de código: el sistema debe permitir el análisis de código fuente de los dispositivos, con el fin de detectar posibles errores y vulnerabilidades.
- Integración con herramientas de control de versiones: el sistema debe permitir la integración con herramientas de control de versiones, como Git, para facilitar la gestión de los cambios y las versiones del código.
- Generación de informes: el sistema debe permitir la generación de informes detallados sobre los resultados de las pruebas, con el fin de facilitar la identificación de problemas y la toma de decisiones.

Capítulo 4

Diseño del sistema

En este capítulo se van a abordar los conceptos aplicados en el diseño del sistema. Para ello, se van a definir los conceptos teóricos para proporcionar un mejor entendimiento de las técnicas y herramientas utilizadas.

4.1. Metodología de testing

4.1.1. Software testing

En este apartado se va a hablar sobre los diferentes tipos de pruebas (unitarias, de integración, de aceptación, etc.), las técnicas de prueba (cajas negras, cajas blancas etc.) y cómo se van a aplicar en el entorno de pruebas que se está desarrollando.

Tipos de pruebas

Las pruebas **unitarias** son aquellas que se realizan en el nivel más bajo de la jerarquía del software, es decir, en las unidades individuales de código. Estas pruebas, al ser predecibles, permiten verificar el correcto funcionamiento de cada unidad de código de forma aislada y detectar errores en las fases tempranas de desarrollo. Además, utilizan los principios FIRST:

- Fast: ejecución rápida, si tardan terminaremos por no repetirlas.
- Independent: cada prueba unitaria no debe depender de otras.
- Repeatable: deben poder repetirse (predecibles, automáticas y rápidas).
- Self-validating: deben poder validarse automáticamente.
- Timely: deben escribirse junto con el código que deben probar.

Por otro lado, las pruebas de **integración** se llevan a cabo para verificar que los diferentes componentes del software funcionan correctamente cuando se integran entre sí. Las pruebas de integración permiten detectar errores que no se pueden encontrar en las pruebas unitarias, ya que se simulan situaciones más complejas y reales.

También existen pruebas de **aceptación**, que son aquellas que se realizan para verificar que el software cumple con los requisitos especificados por el cliente o usuario final, pero no se van a abordar en este proyecto por motivos de falta de tiempo y falta de interés por parte de la empresa.

Técnicas de prueba

Existen varias técnicas de prueba y a continuación se van a explicar las que se han utilizado principalmente en este proyecto. Estas son las pruebas de caja blanca y de caja negra.

Las pruebas de **caja blanca** se realizan teniendo en cuenta el conocimiento interno de la estructura y el funcionamiento del código fuente. En otras palabras, el tester tiene acceso al código fuente y entiende su lógica para diseñar las pruebas. En el proyecto actual, se podrían realizar pruebas de caja blanca utilizando Jasmine y JUnit. Estas pruebas se enfocan en probar el correcto funcionamiento de las diferentes partes del código, incluyendo la lógica, las condiciones, las estructuras de control de flujo y la seguridad.

Por otro lado, las pruebas de **caja negra** se realizan sin tener en cuenta el conocimiento interno del código fuente. El tester solo tiene en cuenta las entradas y las salidas esperadas del software para diseñar las pruebas. Estas pruebas se enfocan en probar el comportamiento general del software, su capacidad de respuesta y la funcionalidad esperada.

Ambas técnicas son importantes y complementarias en la realización de pruebas de software. Las pruebas de caja blanca permiten verificar que cada una de las partes del código funciona correctamente, mientras que las pruebas de caja negra permiten comprobar que el software en general cumple con los requisitos y expectativas del usuario final.

Uso de stubs

Durante el desarrollo de pruebas de software, el uso de stubs es muy útil para facilitar la prueba de las diferentes partes de una aplicación. En el caso de Angular y Java Springboot, es posible utilizar stubs para simular el comportamiento de componentes o servicios que aún no han sido implementados o que dependen de recursos externos que no están disponibles en el momento de las pruebas.

En el caso de Angular, los stubs se pueden implementar como componentes o servicios falsos llamados *spies* [6] que proporcionan una respuesta predefinida cuando son llamados por otro componente o servicio. Por ejemplo, si un componente depende de un servicio para obtener datos de un servidor externo, se puede crear un stub de servicio que devuelva una respuesta de prueba, sin necesidad de esperar a que el servidor externo proporcione los datos reales. Esto

permite probar el componente de manera aislada, sin depender de factores externos que pueden ser difíciles de controlar. En el Código 1 se puede ver un ejemplo del funcionamiento de un spy.

```
spyOn(zoneService, "getZoneByUuid").and.returnValue(of(zoneRes))
```

Código 1: Ejemplo de mock en Jasmine.

En Java Springboot, los stubs se pueden implementar utilizando Mockito, una biblioteca de pruebas que permite simular el comportamiento de objetos y servicios [7]. Por ejemplo, si una clase de servicio depende de otra clase que aún no ha sido implementada, se puede crear un stub de la clase faltante utilizando Mockito, lo que permitirá que la clase de servicio pueda ser probada sin errores. La manera en la que se utiliza es utilizando la estructura Given, When, Then. En el caso que se ilustra solamente se hace uso de When y Then. De esta forma, dados (Given) unos datos, cuando (When) el código llega a un punto determinado, entonces (Then) se sustituye la ejecución de esa sentencia por un resultado dado por el tester. Un ejemplo de esto se puede observar en el Código 2.

```
when(deviceRepository.findDeviceByUuid(deviceUuid))
    .thenReturn(Optional.of(device));
```

Código 2: Ejemplo de mock en Java.

4.1.2. Desarrollo dirigido por pruebas

El desarrollo dirigido por pruebas (Testing Driven Development o TDD) es una metodología de desarrollo de software que se basa en la realización de pruebas desde el principio del proceso de desarrollo [8]. En TDD, el programador primero escribe una prueba para un pequeño trozo de funcionalidad, luego escribe el código para hacer que la prueba pase y, finalmente, refactoriza el código para que sea más eficiente y mantenible (Figura 4.1.2).

En el proyecto actual, se utilizan técnicas de TDD en el desarrollo de software. Al seguir esta metodología, los programadores escriben las pruebas antes de escribir el código, lo que ayuda a definir claramente los requisitos y la funcionalidad esperada. Además, al hacerlo, el desarrollador se asegura de que el software que está creando sea fácilmente testeable, lo que puede reducir la cantidad de errores que se introducen en el software.

Otra ventaja del TDD es que permite a los desarrolladores diseñar y desarrollar software de manera modular, lo que hace que sea más fácil de mantener y ampliar. Al escribir pruebas para cada pieza de funcionalidad del software, se aseguran de que cada componente del software funcione de manera independiente y cumpla con sus especificaciones.

4.1.3. Desarrollo dirigido por comportamiento

El desarrollo dirigido por comportamiento (Behaviour Driven Development o BDD) es una metodología de desarrollo de software que se centra en el comportamiento del software desde el punto de vista del usuario final. En BDD, los desarrolladores escriben pruebas que describen

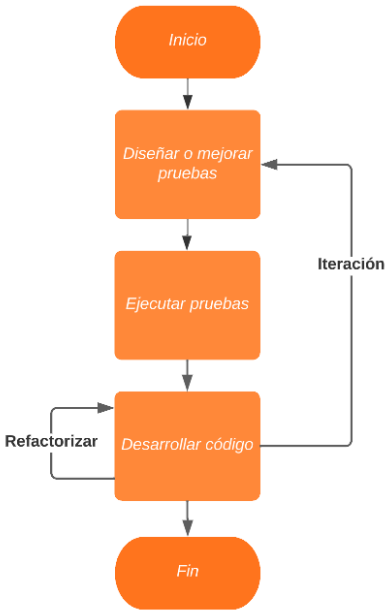

Figura 4.1: Diagrama de flujo de metodología TDD.

cómo se debe comportar el software en diferentes situaciones, en lugar de centrarse en los detalles de implementación.

Al seguir esta metodología, se comienza por escribir pruebas que describen el comportamiento esperado del software en diferentes situaciones. Estas pruebas se describen en un lenguaje natural que los usuarios pueden entender fácilmente. Luego, se implementa el código necesario para hacer que esas pruebas pasen.

Al utilizar BDD, los desarrolladores se centran en la funcionalidad del software y en cómo se comporta en diferentes situaciones. Esto ayuda a garantizar que el software que están desarrollando cumpla con los requisitos y expectativas del usuario final. Además, al escribir las pruebas en un lenguaje natural, los usuarios y otros miembros del equipo de desarrollo pueden entender fácilmente lo que se espera del software, lo que ayuda a garantizar que el software se desarrolle de manera eficiente y eficaz.

Otra ventaja del BDD es que ayuda a los desarrolladores a identificar los requisitos del software antes de comenzar a escribir el código. Al escribir pruebas que describen el comportamiento esperado del software, los desarrolladores pueden comprender mejor lo que se espera del software y pueden identificar cualquier requisito faltante o incompleto.

4.1.4. Cobertura de código

La cobertura de código se refiere a la medida en que el código fuente de un programa es ejecutado por un conjunto de pruebas. Es una métrica importante en el desarrollo de software ya que puede proporcionar una indicación de la calidad y efectividad de las pruebas realizadas.

En otras palabras, la cobertura de código mide la proporción del código que se ejecuta durante la realización de las pruebas.

El uso de pruebas de cobertura de código es importante para garantizar que el software sea robusto y libre de errores. Si una parte del código no se ejecuta durante las pruebas, puede haber errores ocultos en esa parte del código que no se detecten hasta que se presente un problema en producción. La cobertura de código también ayuda a los desarrolladores a identificar áreas del código que necesitan una mayor atención y mejora.

Hay diferentes formas de medir la cobertura de código, como la cobertura de línea, la cobertura de rama y la cobertura de condición:

- La cobertura de línea mide la cantidad de líneas de código ejecutadas por las pruebas en comparación con la cantidad total de líneas de código del programa.
- La cobertura de rama mide la cantidad de ramas de control de flujo (por ejemplo, if-else) que se ejecutan durante las pruebas en comparación con la cantidad total de ramas.
- La cobertura de condición mide la cantidad de condiciones lógicas (por ejemplo, expresiones booleanas) que se ejecutan durante las pruebas en comparación con la cantidad total de condiciones.

Es importante tener en cuenta que una alta cobertura de código no garantiza la ausencia de errores en el software, pero proporciona una indicación de la calidad y efectividad de las pruebas realizadas. Una buena práctica es establecer un umbral de cobertura mínimo y asegurarse de que todas las pruebas pasen antes de enviar el código a producción.

4.2. Entorno de testing

La configuración del entorno de testing es una parte crucial para garantizar la calidad del software desarrollado. Es por ello que es muy importante elegir las herramientas correctas y conseguir un funcionamiento fluido entre ellas.

Para configurar el entorno de testing, se siguieron los siguientes pasos:

1. Se añadieron las dependencias necesarias para el *testing*, en este caso consiste en las herramientas del *backend*, debido a que Jasmine y Karma están integradas por defecto en Angular. En el *backend* se añadieron las dependencias necesarias para JUnit y Cucumber en el archivo pom.xml que es el archivo de configuración de Maven. La principal dificultad fue escoger una versión que estuviese lo suficientemente actualizada para tener las funcionalidades necesarias, pero que a la vez fuese compatible con el resto de dependencias y versiones.
2. Para poder ejecutar las pruebas desde la integración continua, se incluyó el plugin necesario para posteriormente poder ejecutarlas.

3. Se añadieron las dependencias necesarias para la cobertura de código, en este caso fueron Istanbul para el *frontend*, para la cual era necesario activar la cobertura de código en la configuración de Karma. Por otro lado, en el *backend* se añadieron las dependencias de Jacoco.

4. Fue necesario crear las clases de *testing*, según las necesidades del sistema. En el caso del *frontend*, Angular genera ficheros de prueba para cada componente que se crea, por lo que simplemente era necesario identificar qué componente se quería testear. Para el *backend*, se creó una carpeta de test, en la que se fueron creando clases para los controladores que se testeaban.

Estos pasos permiten configurar un entorno de *testing* que permite la ejecución de tests que facilitan la detección y corrección de errores, vulnerabilidades y defectos de código en el proyecto.

4.3. Automatización

Para la automatización de la ejecución y análisis en el sistema de monitorización de dispositivos, se han llevado a cabo las siguientes tareas:

1. Configuración de la CI en GitLab: se ha configurado la integración continua en GitLab para el proyecto [9]. De esta forma, cada vez que se realiza un push al repositorio, GitLab inicia automáticamente el proceso de construcción y pruebas.

2. Configuración del plugin de SonarQube: se ha configurado el plugin de SonarQube en el archivo pom.xml del proyecto para enviar los datos de análisis de calidad de código a SonarQube. Se ha indicado la URL del servidor de SonarQube y el token de autenticación correspondiente.

3. Modificación del archivo .gitlab-ci.yml: con asistencia del administrador de sistemas, se ha modificado el archivo .gitlab-ci.yml en la raíz del proyecto para definir el pipeline de CI en GitLab. En el archivo .gitlab-ci.yml, se ha definido el proceso que debe seguir GitLab para construir y probar el proyecto. Este proceso incluye la instalación de Maven y Java manualmente en el script. El pipeline ejecuta los siguientes comandos de Maven:

    - `mvn clean compile`: limpia y compila el proyecto.
    - `mvn test`: ejecuta las pruebas.
    - `mvn failsafe:verify`: ejecuta las pruebas de Cucumber.
    - `mvn jacoco:report`: genera los informes de JaCoCo.
    - `mvn sonar:sonar`: envía los datos a SonarQube.

4.4. Ciclo de funcionamiento

En la Figura 4.2 se puede observar cómo funcionan las tecnologías usadas en el proyecto para desarrollar el producto final.

1. Para empezar, es necesario analizar la plataforma de *Smart Building* y entender las necesidades de este producto de software. Esto se obtiene realizando un análisis tanto de la plataforma web, como del código de la aplicación. Para ello se observa el *frontend* desarrollado en *Angular* y el *backend* desarrollado en *Java Springboot*.
2. Una vez se conocen las funcionalidades que se han de probar, se pueden desarrollar las pruebas de software, tanto de *frontend* mediante *Jasmine* como de *backend* mediante *JUnit*.
3. Estas pruebas son ejecutadas mediante la integración continua de GitLab, asegurando que las pruebas se pasan cada vez que se realiza una modificación a la plataforma.
4. Además, mediante las pruebas desarrolladas, se generan reportes de cobertura de código mediante Istanbul y Jacoco utilizados para analizar el software.
5. Mediante SonarQube se analizan estos resultados, que están disponibles para todo el equipo desde la nube. Se obtienen factores que miden la calidad y la seguridad del código.
6. Gracias al análisis realizado, se intentan mejorar las características de la plataforma *Smart Building*.

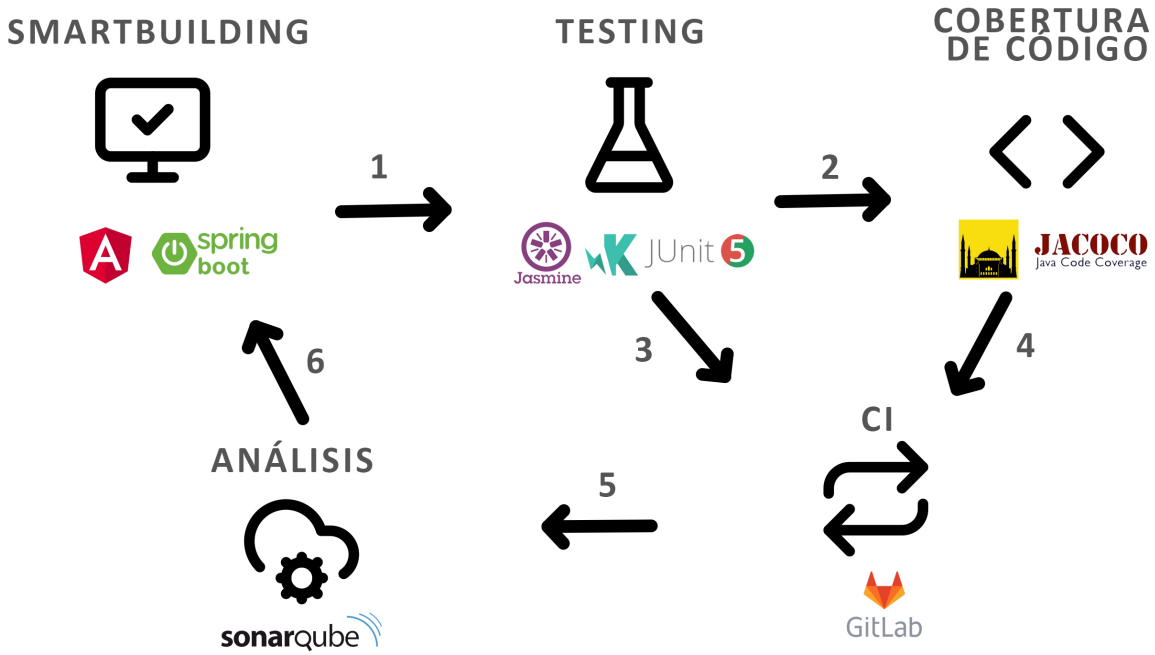

Figura 4.2: Tecnologías usadas.

Capítulo 5

Implementación y pruebas

En este capítulo se va a explicar en detalle el proceso de implementación de las pruebas, su ejecución y análisis final.

5.1. Estructura del código

Se va a empezar definiendo qué tests se han hecho y cómo se han estructurado las diferentes clases para tener una organización adecuada de los tests.

Primero, los tests se pueden dividir según el ámbito al que van dirigidos.

5.1.1. Estructura de *frontend*

Debido a la forma en la que está compuesta la aplicación, se centraron los tests en 5 partes. Cada una de estas partes tiene una serie de acciones que el usuario puede realizar. Como ejemplo, se va a utilizar los dispositivos (*Devices*).

- *Cargar el dispositivo*: comprueba que se carga el dispositivo correctamente cuando se accede a la vista de detalles.
- *Cargar el plano*: comprueba que se carga el plano con la ubicación del dispositivo correctamente.
- *Filtros*: comprueba que en la lista de dispositivos, se aplican los filtros correctamente.
- *Ordenación*: verifica que en la lista de dispositivos, se aplica la asignación de orden correctamente.
- *Eliminar dispositivo*: comprueba que se elimina el dispositivo correctamente cuando el usuario pulsa el botón correspondiente.

- *Crear dispositivo*: verifica que se siguen correctamente todos los pasos necesarios para crear un dispositivo y que éste se crea correctamente.
- *Editar dispositivo*: verifica que se siguen correctamente todos los pasos necesarios para editar un dispositivo y que éste se edita correctamente.
- *Tratamiento de errores*: para las diversas acciones anteriores, verifica que se muestre el mensaje de error adecuado en caso que sea necesario.

5.1.2. Estructura de *backend*

En cuanto a la estructura de los tests del *backend*, se centró en los controladores que permiten comunicar los distintos servicios con las llamadas de API-REST del *frontend*. En la Figura 5.1 se puede observar el listado de clases de tests.

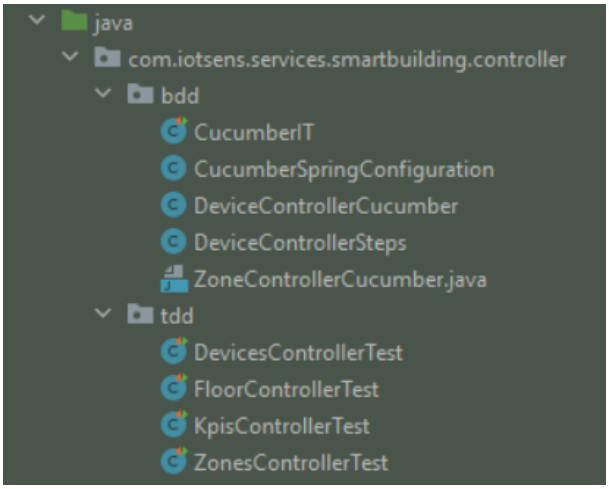

Figura 5.1: Listado de clases de tests en el *backend*.

TDD

Se realizaron los tests para los siguientes controladores: *Floors*, *Zones*, *Devices* y *KPIs*. Los tipos de pruebas eran los mismos para los cuatro controladores, pero adaptados a las necesidades y funcionalidad de cada uno de ellos. A continuación se va a realizar un desglose de los escenarios de prueba implementados en el controlador de dispositivos:

- *getDeviceTest()*: este método se utiliza para obtener información detallada de un dispositivo en particular. Proporciona datos específicos sobre el dispositivo, como su estado, características y configuración.
- *getDeviceCatalogTest()*: este método se utiliza para obtener información de la lista de dispositivos disponibles.

- *createDeviceTest()*: con este método, se puede crear un nuevo dispositivo en el sistema. Se proporciona la información necesaria para configurar un nuevo dispositivo, como su nombre, tipo y características.

- *editDeviceTest()*: este método se utiliza para editar la configuración de un dispositivo existente.

- *deleteDeviceTest()*: con este método, es posible eliminar un dispositivo específico del sistema. Al utilizar este método, se borra permanentemente un dispositivo y toda su información asociada.

- *deleteDevicesTest()*: similar al método anterior, pero en lugar de eliminar un solo dispositivo, este método permite eliminar varios dispositivos al mismo tiempo. Se proporciona una lista de identificadores de dispositivos a eliminar.

- *getDeviceSummaryTest()*: este método devuelve un resumen de un dispositivo en particular. Proporciona información más detallada sobre un dispositivo específico, incluyendo su estado y mediciones recientes.

- *getDeviceSummariesTest()*: este método obtiene un resumen de todos los dispositivos registrados en el sistema. Proporciona una visión general de los dispositivos, incluyendo información básica y su estado actual.

- *getDeviceMeasurementTest()*: este método se utiliza para obtener las mediciones específicas de un dispositivo. Proporciona datos relacionados con las mediciones o sensores que el dispositivo puede estar capturando en tiempo real.

- *getDevicesByZonesTest()*: con este método, se pueden obtener todos los dispositivos registrados en una zona específica del edificio.

- *getDeviceMeasurementsTest()*: este método permite obtener todas las mediciones registradas de un dispositivo en particular. Proporciona datos históricos de las mediciones capturadas por el dispositivo en un período de tiempo determinado.

- *getDevicesByTenantTest()*: este método obtiene todos los dispositivos asociados a un inquilino o arrendatario específico en el sistema. Es útil para gestionar dispositivos relacionados con un usuario o grupo en particular.

BDD

Para los tests de Cucumber, se implementó solamente los tests para los dispositivos. Lo que se buscaba era realizar una prueba de que realmente se podía llegar a implementar en un futuro. Para ello, fue necesario crear varias clases de configuración. Estas son CucumberIT y CucumberSpringConfiguration y se pueden observar en el Código 3. Se trata de clases vacías y su función es indicar mediante anotaciones su configuración y permitir al sistema entender que se trata de pruebas de Cucumber y que de esta forma las trate como se debe. A continuación, se pueden observar ambas clases y sus anotaciones.

```
@RunWith(Cucumber.class)
@CucumberOptions(features = {"src/test/resources/features"},
    plugin = {"pretty", "html:target/cucumber-html-report.html",
    "json:target/cucumber.json"},
    glue = "com.iotsens.services.smartbuilding.controller.bdd")
public class CucumberIT {
}

@CucumberContextConfiguration
@SpringBootTest(classes = Application.class)
public class CucumberSpringConfiguration {
}
```

Código 3: Clases de configuración.

5.2. Descripción técnica de la implementación

En esta sección se van a explicar las pautas que se han seguido para implementar de manera correcta y eficiente los tests.

5.2.1. Implementación *frontend*

Jasmine proporciona un conjunto de funciones y estructuras para escribir y ejecutar pruebas en el código de Angular. Estas funciones proporcionan una sintaxis fácil de entender y de organizar [10].

La función `describe()` se trata de una especificación, que es un conjunto de pruebas que verifican el comportamiento de una unidad de código específica, como un componente, un servicio o una directiva de Angular. En el Código 4 se puede observar una implementación de esta función.

```
describe("delete device", () => {
    ...
}
```

Código 4: Implementación de la función describe().

La función `beforeEach()` sirve para ejecutar código antes de cada prueba y se utiliza para configurar el estado inicial necesario para las pruebas, como la creación de instancias de componentes o servicios. Se puede observar un ejemplo en el Código 5.

```
beforeEach(() => {
    component.rootForm = "DEVICE"
    service.deviceForm  = service.createDeviceForm()

    // @ts-ignore
    spyOn(component, "setFloor").and.callFake(() => {
      component.deviceForm.controls.floorUuid.setValue("floor")
      component.deviceForm.controls.floor.setValue(floor)
    })
  });
```

Código 5: Implementación de la función beforeEach().

La función `it()` se utiliza para definir una prueba individual dentro de una especificación. Como se puede ver en el Código 6, proporciona un contexto claro y descriptivo para cada prueba y se utiliza en combinación con la función `expect()` para establecer expectativas y verificar el comportamiento esperado.

```
it('should create', () => {
    expect(component).toBeTruthy();
  });
```

Código 6: Implementación de la función it().

En Jasmine, los **mocks** se llaman **spies**. Se utilizan para simular o controlar el comportamiento de dependencias externas de un componente o servicio durante las pruebas. Un mock es un objeto simulado que proporciona respuestas predefinidas a las llamadas de funciones. Un spy es una función falsa que registra información sobre las llamadas realizadas a ella, como cuántas veces se llamó y con qué argumentos. Se puede observar un ejemplo de la implementación en el Código 7

```
spy = spyOn(router, 'navigate')
...
expect(spy).toHaveBeenCalledWith(["/buildings/1/floors/1/details"])
```

Código 7: Implementación de un spy.

Finalmente, se va a mostrar un ejemplo un escenario de prueba. Se trata de la eliminación de un dispositivo. Primero, se declara la función `describe()` con el nombre de "delete devices", indicando que esta sección de pruebas se centra en probar la funcionalidad de eliminar dispositivos.

Dentro de esta función, se definen algunas variables y configuraciones necesarias para realizar la prueba:

- `let messages: CommonMessagesService;`: declara una variable que se utilizará para si-

mular mensajes enviados por el sistema.

- `beforeEach()`: aquí se configuran valores iniciales para las variables `messages` y la lista de `component.devices`, que es una lista de dispositivos. La variable `messages` se inicializa con una instancia de su servicio utilizando la función `TestBed.inject()`.

A continuación, existe otro `describe()` llamado "delete individual device" para diferenciarlo de las pruebas enfocadas a la eliminación de varios dispositivos. Dentro de esta sección, se dispone de otra función `beforeEach()` que realiza más configuraciones para esta sección en concreto. Entre ellas, se utiliza `spyOn()` para crear un espía del método de `getDevicesByBuildingUUid` y se configura para que éste devuelva un objeto observable.

Posteriormente, se define una prueba individual mediante `it()` llamada *deletes a device correctly* para verificar que la eliminación del dispositivo se realiza correctamente. Este escenario de prueba se establece como asíncrono mediante `fakeAsync()` para poder controlar la ejecución de la prueba. Dentro de esta prueba se realizan las siguientes acciones:

- Se crea una instancia de `DeleteResponse` y se configura la propiedad `delResponse.deleted` a *true*.
- La propiedad `delResponse.found` se establece en *true* para indicar que el sistema ha encontrado el dispositivo a eliminar.
- `component.floorsLoaded` se establece en *true* para indicar el estado necesario para la eliminación del dispositivo.
- Se espía el método `showSuccessfullyDeletedMessage` del servicio `messages`. A este no se le indica un valor a devolver, provocando que no se realice la ejecución de este método debido a que es irrelevante para el escenario actual y así evitar que su resultado interfiera con la prueba.
- Se utiliza `spyOn()` para espiar el método `deleteDevice()` del servicio `service` y se establece un valor de respuesta para que emita la instancia `delResponse` creada anteriormente.
- Se llama al método `deleteDevice()` del componente `component` pasando el `uuid` del primer dispositivo de `component.devices`
- Se hace uso de la función `tick()` para simular la finalización de cualquier operación asíncrona que pueda ocurrir durante este proceso.
- Se realizan las siguientes comprobaciones:
  - Se espera que el primer dispositivo de `component.devices` sea igual al segundo de `devices`. Esto verifica que se ha eliminado el dispositivo correctamente y los dispositivos restantes se han movido una posición.
  - Se espera que el método `showSuccessfullyDeletedMessage` haya sido llamado.

A continuación, en el Código 8 se puede observar el ejemplo que se acaba de explicas:

```
describe("delete devices", () => {
  let messages: CommonMessagesService
  beforeEach(() => {
    messages = TestBed.inject(CommonMessagesService)
    component.devices = devices
  })

  describe("delete individual device", () => {

    beforeEach(() => {
      let response: DevicesResponse = new DevicesResponse()
      let array: Device[] = []
      array.push(devices[1])
      array.push(devices[2])
      response.devices = array

      spyOn(service, "getDevicesByBuildingUuid").and.returnValue(of(response))
    })

    it('deletes a device correctly', fakeAsync(() =>{
      let delResponse: DeleteResponse = new DeleteResponse()
      delResponse.deleted = true
      delResponse.found = true
      component.floorsLoaded = true

      spy = spyOn(messages, "showSuccessfullyDeletedMessage")

      spyOn(service, "deleteDevice").and.returnValue(of(delResponse))

      component.deleteDevice(component.devices[0].uuid)

      tick()

      expect(component.devices[0]).toEqual(devices[1])
      expect(spy).toHaveBeenCalled()
    }));
```

Código 8: Implementación de pruebas de eliminar dispositivos.

5.2.2. Implementación *backend*

TDD

Para crear tests con JUnit, primero hay que crear la clase. En este caso, debido a que el proyecto utiliza Java SpringBoot, se ha de anotar la clase con el *Runner* adecuado. En el Código 9 se puede observar la implementación de una clase de pruebas.

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class DevicesControllerTest {
    ...
}
```

Código 9: Clase de DeviceControllerTest.

Posteriormente, se ha de crear los *mocks* adecuados. Para ello, se han de instanciar mediante la anotación de `@MockBean` como en el Código 10.

```
@MockBean
private DeviceCreationService deviceCreationService;
```

Código 10: Implementación de un @MockBean.

También es necesario crear las conexiones con el controlador que se va a testear mediante la anotación de `@Autowired`. Se puede ver en el Código 11.

```
@Autowired
private DeviceRepository deviceRepository;
@Autowired
private BuildingRepository buildingRepository;
```

Código 11: Implementación de un @Autowired.

Debido a que se van a realizar varios tests y todos estos requieren de una configuración previa, se utiliza la función `setup()` anotada con `@Before`, para que antes de ejecutar cada test, el sistema realizara esta configuración previa para que el entorno sea el mismo en cada escenario. En ella se pueden realizar acciones como crear servicios o objetos que se van a utilizar posteriormente en los tests. Se puede observar un ejemplo en el Código 12

```
@Before
public void setup() {
    building = new Building();
    building.setTenantUuid("tennantuuid");
    building.setCode("buildingcode");
    building.setOrganizationUuid("organizationUuid");
}
```

Código 12: Implementación de un @Before.

Una vez llegado a los escenarios de prueba, como se puede ver en el Código 13, es importante indicar cada escenario con la anotación de `@Test`, para que el sistema sepa que se trata de una prueba que ha de ejecutar. Además, en ocasiones, es necesario utilizar la anotación `@WithMockUser`, debido a que el método que se intentaba testear requería de algún rol de usuario. De esta forma el sistema obtiene un usuario "falso".

```
    @Test
public void getDevice() {
    GetDevice.GetDeviceRequest getDeviceRequest = new
        GetDevice.GetDeviceRequest(deviceUuid);

    when(deviceRepository.findDeviceByUuid(deviceUuid))
        .thenReturn(Optional.of(device));

    GetDevice.GetDeviceResponse getDeviceResponse =
        getDevice.execute(getDeviceRequest);

    assertEquals(deviceDto.getUuid(), getDeviceResponse.getDevice()
        .getUuid());
}
```

Código 13: Implementación de un escenario de prueba.

BDD

A continuación se describe la implementación técnica de la implementación de BDD a través de Cucumber paso a paso mediante un ejemplo, que se puede observar en el Código 14.

Para empezar, hay que definir la acción a testear en el archivo **.feature** como el de la Figura 5.2. La prueba comienza con la declaración de una característica (Feature) llamada "user wants to get device", que define el comportamiento deseado. Dentro de la característica, se define un escenario (*Scenario Outline*) llamado "client makes call to get a device", que representa una situación específica para probar. En el escenario, se utiliza el paso *Given* para establecer el contexto inicial de la prueba. En este caso, se declara que el cliente tiene un identificador único universal (UUID) que se proporciona como parámetro («uuid¿"). A continuación, se utiliza el paso *When* para simular la acción del cliente de hacer una solicitud de obtener un dispositivo (`getDevice request`). Por último, se utiliza el paso *Then* para verificar si el cliente recibe el objeto `deviceDto` correcto correspondiente al UUID proporcionado («uuid¿"). Se proporcionan ejemplos (*Examples*) para ejecutar el mismo escenario varias veces con diferentes valores de UUID. Los valores se especifican en una tabla, donde cada fila representa un conjunto diferente de valores para el UUID.

```
  Feature: user wants to get device
Scenario Outline: client makes call to get a device
  Given client has an uuid: "<uuid>"
  When client makes a getDevice request
  Then client receives the correct deviceDto "<uuid>"

  Examples:
    | uuid       |  |
    | uuid       |  |
    | device     |  |
    | deviceuuid |  |
```

Código 14: Implementación de una *Feature* de Cucumber.

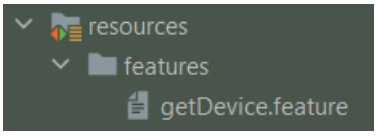

Figura 5.2: Archivo .feature para Cucumber.

Posteriormente, se ha de realizar la implementación de estos tres pasos mediante lenguaje Java. Se puede observar la implementación de estos pasos en el Código 15.

- Método `client\_has\_uuid(String uuid)`: Este método se anota con `@Given` y recibe un parámetro `uuid` de tipo String. Su función es almacenar el valor del UUID proporcionado en la variable `deviceUuid`.

- Método `client\_makes\_getDevice\_request()`: Este método se anota con `@When` y no tiene parámetros. Su función es realizar una solicitud `getDevice` al controlador llamado `deviceControllerCucumber` utilizando el UUID almacenado en `deviceUuid`. El resultado de la solicitud se almacena en la variable `deviceDtoResponse`.

- Método `clientRecievesTheCorrectDeviceDto(String device)`: Este método se anota con `@Then` y recibe un parámetro device de tipo String. Su función es validar si el objeto `deviceDtoResponse` recibido del controlador coincide con el UUID proporcionado y si es igual al objeto `deviceDto` esperado. Para hacer esto, se configura el UUID del objeto `deviceDto` con el valor proporcionado en `device`, y luego se utilizan las aserciones `assertEquals` para verificar si el UUID y el objeto completo son iguales entre `deviceDto` y `deviceDtoResponse`.

```
    @Given("client has an uuid: {string}")
public void client_has_uuid(String uuid) {
    this.deviceUuid = uuid;
}


@When("client makes a getDevice request")
public void client_makes_getDevice_request() {
    deviceDtoResponse = deviceControllerCucumber.getDevice(deviceUuid);
}


@Then("client receives the correct deviceDto {string}")
public void clientRecievesTheCorrectDeviceDto(String device) {
    deviceDto.setUuid(device);
    assertEquals(deviceDto.getUuid(), deviceDtoResponse.getUuid());
    assertEquals(deviceDto, deviceDtoResponse);
}
```

Código 15: Implementación de los pasos de Cucumber.

5.3. Ejecución y análisis

En esta sección, se va a realizar una explicación de la ejecución de las pruebas tanto del *frontend* como del *backend*. También se va a comentar acerca del análisis de la cobertura de código con Istanbul y Jacoco y el análisis estático de código mediante SonarQube.

Karma e Istanbul

En este apartado, se abordará el tema de la ejecución de pruebas, centrándonos específicamente en dos herramientas ampliamente utilizadas en el ámbito de las pruebas de software: Karma e Istanbul. En el proceso de desarrollo de software, las pruebas desempeñan un papel crucial para garantizar la calidad y la funcionalidad del código. Con el fin de optimizar el tiempo y los recursos utilizados en la ejecución de pruebas, existen diversas técnicas y herramientas disponibles.

Lo primero, es asegurarse de que el entorno de Karma está configurado como se espera. Esto incluye especificar el navegador en el que se van a realizar las pruebas. En este caso, mediante el fichero *karma.conf.js*, se especifica en qué navegador se ejecutan las pruebas, siendo en este caso Chrome.

Existe la posibilidad de filtrar pruebas que no sea necesario ejecutar. Esto se consigue añadiendo una "x" delante de la función `it()` o la función `describe()`. En el Código 16 se puede ver cómo se implementa un escenario de prueba excluido mediante `xit()`. En el caso de

que la "x" se insertase delante de la función `describe()`, se excluirían todas las especificaciones que se encuentran dentro de esa *suite* de pruebas.

```
    xit('adds device', function () {
        component.deviceForm.controls.alias.setValue("alias")
component.deviceForm.controls.serialNumber.setValue("serialNumber")
        component.deviceForm.controls.zoneUuid.setValue("zone")
        expect(component.validForm).toBeTruthy()
      })
```

Código 16: Implementación de xit().

También es posible filtrar las pruebas para elegir cuáles ejecutar. Añadiendo un "f" delante de la función `it()` o `describe()` se consigue ejecutar solamente las pruebas que tengan este indicador. En el Código 17 se puede ver cómo se implementa un escenario de prueba específico mediante `fit()`. La capacidad de filtrar las pruebas y ejecutar solo aquellas que se deseen es una funcionalidad útil al trabajar con tests. Esto permite seleccionar pruebas individuales o grupos de pruebas para una ejecución selectiva y eficiente.

```
    fit('filter when floors are loaded and selected', function () {
      component.floorsLoaded = true
      component.floorSelected  = true

      component.applyFilters()

      expect(spy).toHaveBeenCalledWith("1", 0, 10, component.currentFilters)
      expect(component.devices[0]).toEqual(devices[2])
    });
```

Código 17: Implementación de fit().

La capacidad de filtrar y seleccionar pruebas específicas para su ejecución brinda flexibilidad a los desarrolladores y les permite realizar pruebas más rápidas y enfocadas en áreas específicas del código, lo que puede mejorar la eficiencia del proceso de desarrollo y garantizar una mayor confiabilidad en la funcionalidad probada.

Para arrancar Karma y ejecutar los tests en el *frontend*, es necesario utilizar el comando `ng test`. Con esto se abre una ventana del navegador en la que se observan los tests que se han ejecutado que podemos ver en la Figura 5.3. En verde están los que pasan correctamente, en rojo los que no, y en amarillo los que se han excluido de la ejecución. En caso de que salte algún error durante la ejecución de los tests, aparece un cuadro con el mensaje del error en cuestión.

Una vez realizada la ejecución de las pruebas, se genera automáticamente el análisis de cobertura de código de Istanbul [11]. Se puede observar un ejemplo en la Figura 5.4. En él se puede observar las líneas que han sido ejecutadas por Karma al ejecutar los tests. En rojo aparecen las líneas que no se han ejecutado. Cuando aparece una línea de color amarillo, significa que es una sentencia condicional que se ha ejecutado parcialmente, por ejemplo, en un *if*, puede ser que solamente se haya probado el caso en el que la condición es *true*. Este análisis proporciona una visión detallada de la cobertura alcanzada por las pruebas, permitiendo identificar áreas

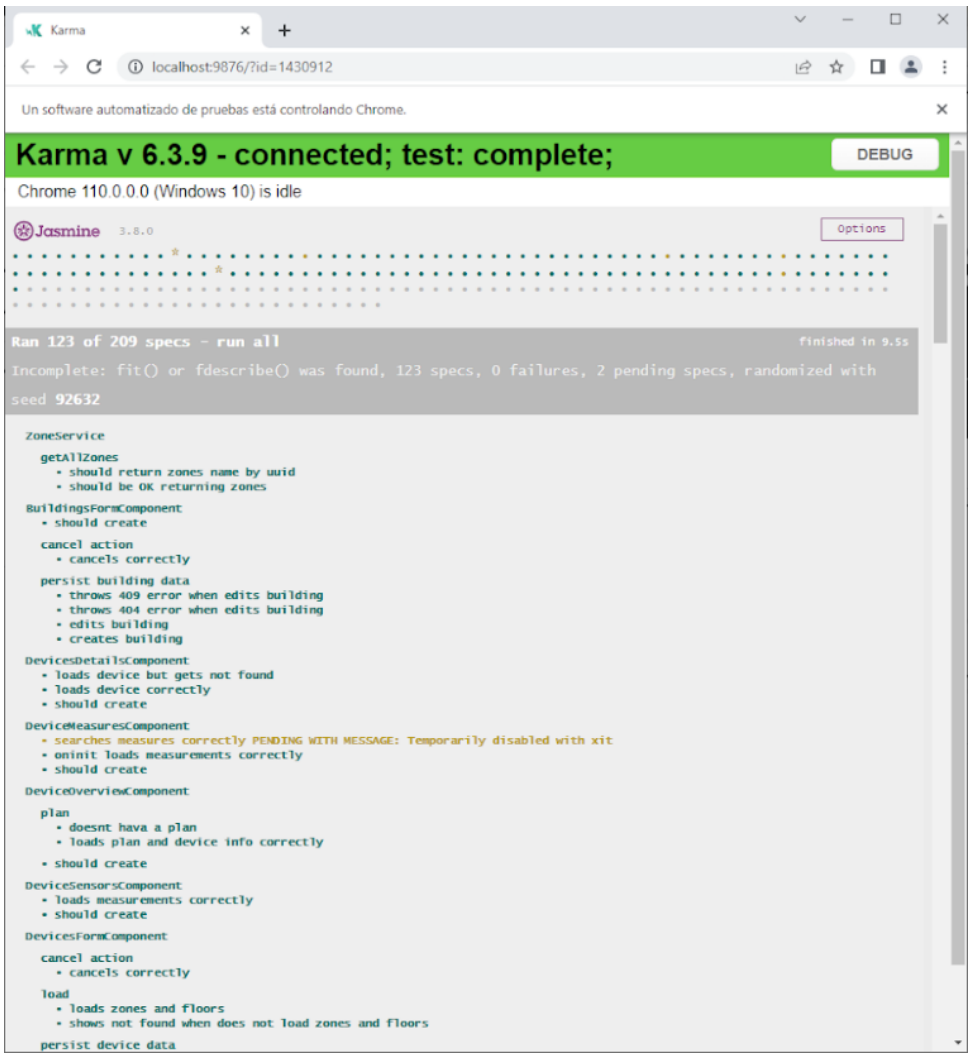

Figura 5.3: Ejecución de tests con Karma.

de código que no han sido ejercitadas adecuadamente y áreas que solo han sido parcialmente probadas.

Al ejecutar los tests desde la línea de comandos, Karma proporciona un resumen de la cobertura de código como el de la Figura 5.5, que se va actualizando en tiempo real, a medida que se van modificando las pruebas, siempre que el comando `ng test` esté activo.

Este resumen de cobertura proporciona información valiosa sobre el estado actual de las pruebas y la cantidad de código que ha sido cubierta. Permite identificar rápidamente las áreas que han sido probadas y aquellas que aún requieren atención. La cobertura de código se muestra en porcentajes, lo que permite evaluar de manera cuantitativa la calidad de las pruebas realizadas.

Es importante destacar que esta funcionalidad de actualización en tiempo real del resumen de cobertura brinda una retroalimentación inmediata, lo que facilita la detección temprana de posibles brechas en la cobertura de código y ayuda a mejorar la calidad de las pruebas realizadas.

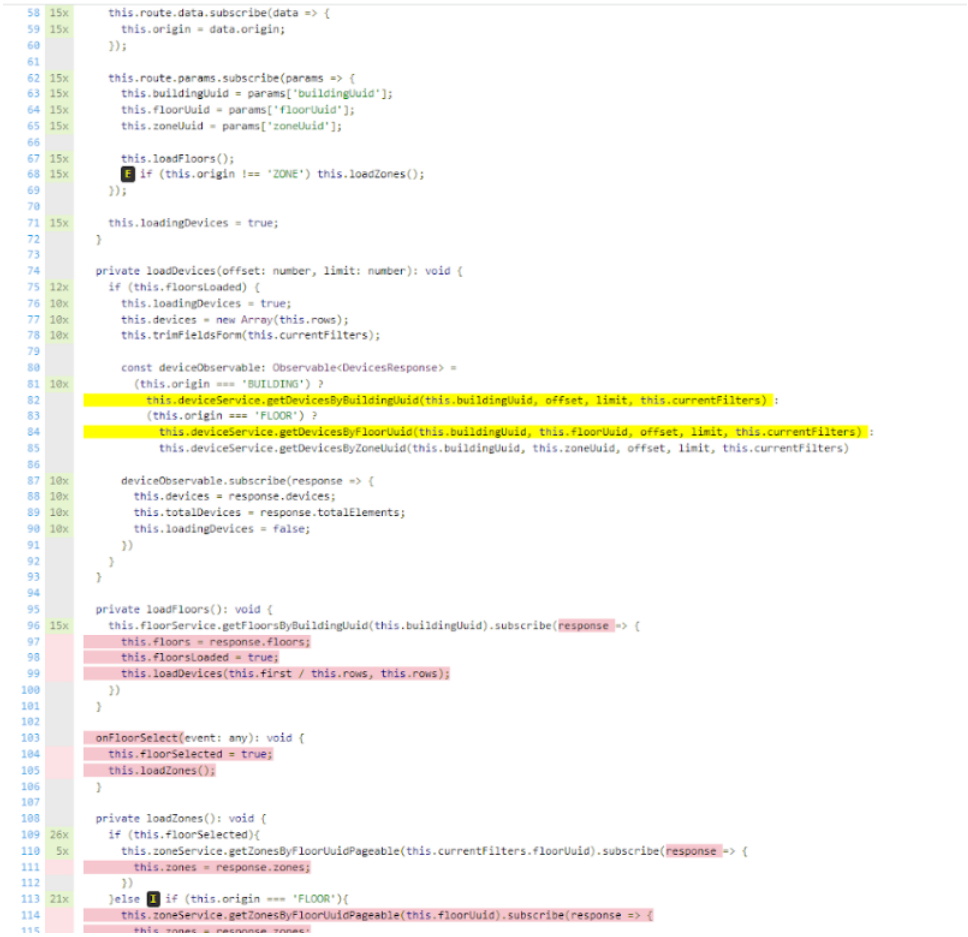

Figura 5.4: Cobertura de código de Istanbul.

JUnit y Jacoco

JUnit y Jacoco son las dos herramientas utilizadas en el desarrollo de este proyecto para realizar pruebas de *backend* y evaluar la cobertura de código. En este apartado, se explorarán dos formas de ejecutar estas pruebas.

La primera forma es pulsar el botón verde que aparece al lado de cada clase o prueba en particular al visualizar el proyecto mediante un IDE como es Intellij. Al seleccionar esta opción, no se generará la cobertura con Jacoco, pero se mostrará el resultado correspondiente. Este resultado proporciona información sobre el éxito o fracaso de las pruebas y puede ser útil para un análisis rápido de la funcionalidad. El resultado que se verá es el que se observa en la Figura 5.6.

La segunda forma de ejecutar las pruebas de *backend* es mediante el comando `mvn test`. Al utilizar este comando, el sistema ejecutará todos los tests configurados y, gracias a la configuración previa, generará el reporte de cobertura de código con Jacoco como el de la Figura 5.7. Este reporte detallado muestra la cobertura de cada clase y método, lo que permite identificar las áreas del código que no han sido probadas adecuadamente y que podrían requerir una mayor atención.

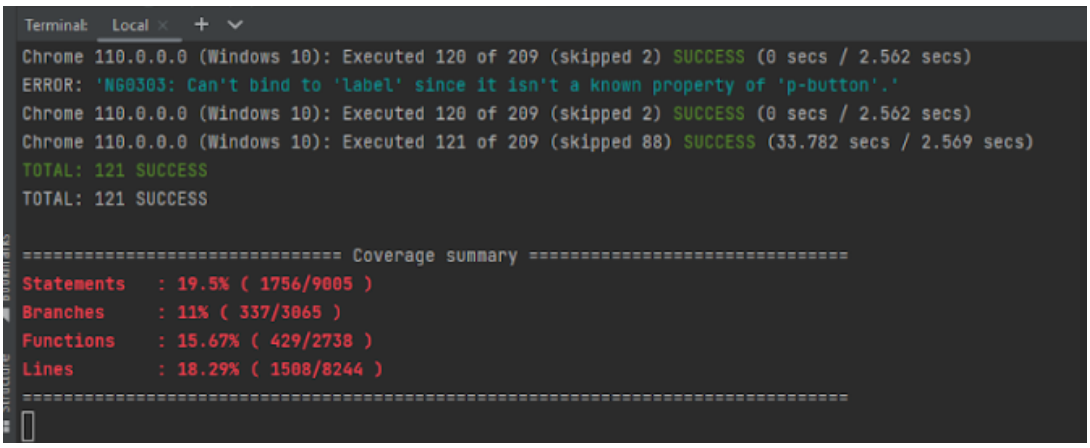

Figura 5.5: Ejecución de tests de Karma en IntelliJ.

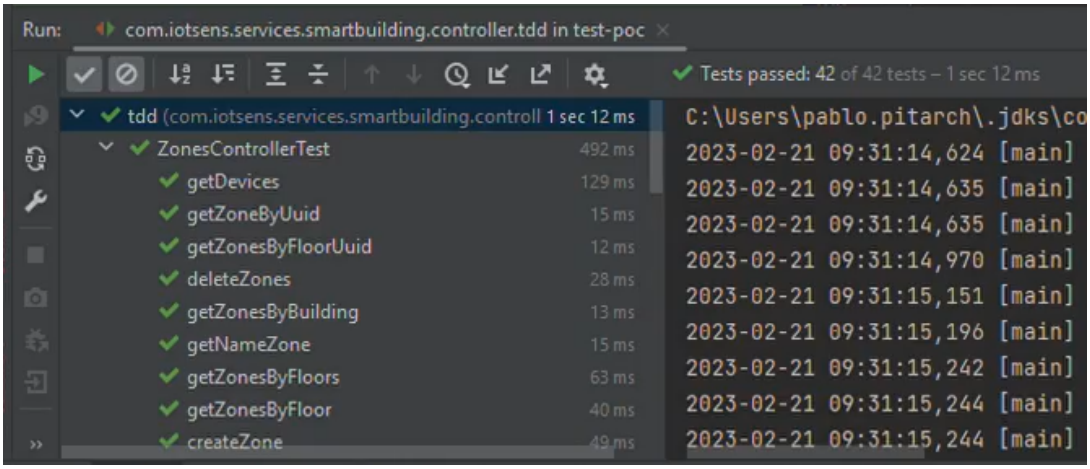

Figura 5.6: Ejecución de tests en JUnit.

El reporte de cobertura generado por Jacoco es una herramienta valiosa para evaluar la calidad y la efectividad de las pruebas de *backend*. Proporciona información detallada sobre la cobertura de cada clase y método, lo que permite identificar las áreas del código que no han sido probadas adecuadamente. Esta visibilidad adicional permite a los desarrolladores identificar posibles puntos débiles en el código y enfocar sus esfuerzos en mejorar la cobertura de pruebas en esas áreas.

Además, el reporte de cobertura también es útil para garantizar que se están cumpliendo los objetivos de cobertura establecidos para el proyecto. Al analizar el porcentaje de cobertura alcanzado, los equipos de desarrollo pueden determinar si se están probando adecuadamente todas las funcionalidades y si existen áreas críticas del sistema que requieren una mayor atención en términos de pruebas.

```
import com.iotsens.services.smartbuilding.dto.filter.DeviceFilterDto;
import com.iotsens.services.smartbuilding.mapper.DeviceMapper;
import com.iotsens.services.smartbuilding.model.Device;
import com.iotsens.services.smartbuilding.utils.PaginationUtils;
import lombok.AllArgsConstructor;
import lombok.Data;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.domain.Specification;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.stream.Collectors;

@Service
@AllArgsConstructor
public class GetDevices {

    private final DeviceRepository deviceRepository;
    private final DeviceMapper deviceMapper;
    private final DeviceSpecifications deviceSpecifications;

    public GetDevicesResponse execute(GetDevicesRequest getDevicesRequest) {
        DeviceFilterDto filters = getDevicesRequest.getDeviceFilter();
        Pageable pageable = PaginationUtils.pageableFrom(getDevicesRequest.getOffset(), getDevicesRequest.getLimit(), filters.getSortField(), filters.getSortOrder());
        Specification<Device> specification = deviceSpecifications.searchByFilterDto(filters, getDevicesRequest.getBuildingUuid(), null);
        Page<Device> devices = deviceRepository.findAll(specification, pageable);
        List<DeviceDto> devicesMapped = devices.getContent().stream()
                .map(deviceMapper::mapToDeviceDtoList)
                .collect(Collectors.toList());
        return new GetDevicesResponse(devicesMapped, devices.getTotalElements());
    }

    @AllArgsConstructor
    @Data
    public static class GetDevicesRequest {
        private Integer offset;
        private Integer limit;
        private String buildingUuid;
        private DeviceFilterDto deviceFilter;
    }

    @AllArgsConstructor
    @Data
    public static class GetDevicesResponse {
        private List<DeviceDto> devices;
        private long totalElements;
    }
}
```

Figura 5.7: Cobertura de código de Jacoco.

SonarQube

La ejecución de las secciones anteriores también se puede realizar de manera automatizada. Mediante la integración continua de GitLab, se puede conseguir ejecutar las instrucciones necesarias para pasar los tests cada vez que un miembro del equipo realiza un *commit*. De esta forma, no es necesario ejecutar las pruebas manualmente, y se podrá conocer si existe algún error antes de realizar un *merge*. Además de ejecutar las pruebas, también se generan los informes de Istanbul y Jacoco. Dada la configuración actual, el informe de Jacoco es utilizado por SonarQube para obtener un resultado más preciso y con información más útil para el equipo de desarrollo.

Durante el análisis, SonarQube [12] identifica problemas comunes de calidad del código, como la duplicación, la complejidad excesiva, la falta de comentarios, el incumplimiento de estándares de codificación y la presencia de vulnerabilidades de seguridad conocidas.

Una vez completado el análisis, SonarQube genera informes detallados que muestran los problemas encontrados y proporciona métricas para medir la calidad del código. Al revisar los informes de SonarQube, el equipo de desarrollo puede identificar los problemas de calidad del código y tomar medidas para corregirlos. Esto puede implicar refactorizar el código para eliminar la duplicación o reducir la complejidad, agregar comentarios para mejorar la legibilidad, asegurarse de que el código cumpla con los estándares establecidos y abordar las vulnerabilidades de seguridad conocidas. Se pueden observar los resultados del análisis en las Figuras 5.8 y 5.9.

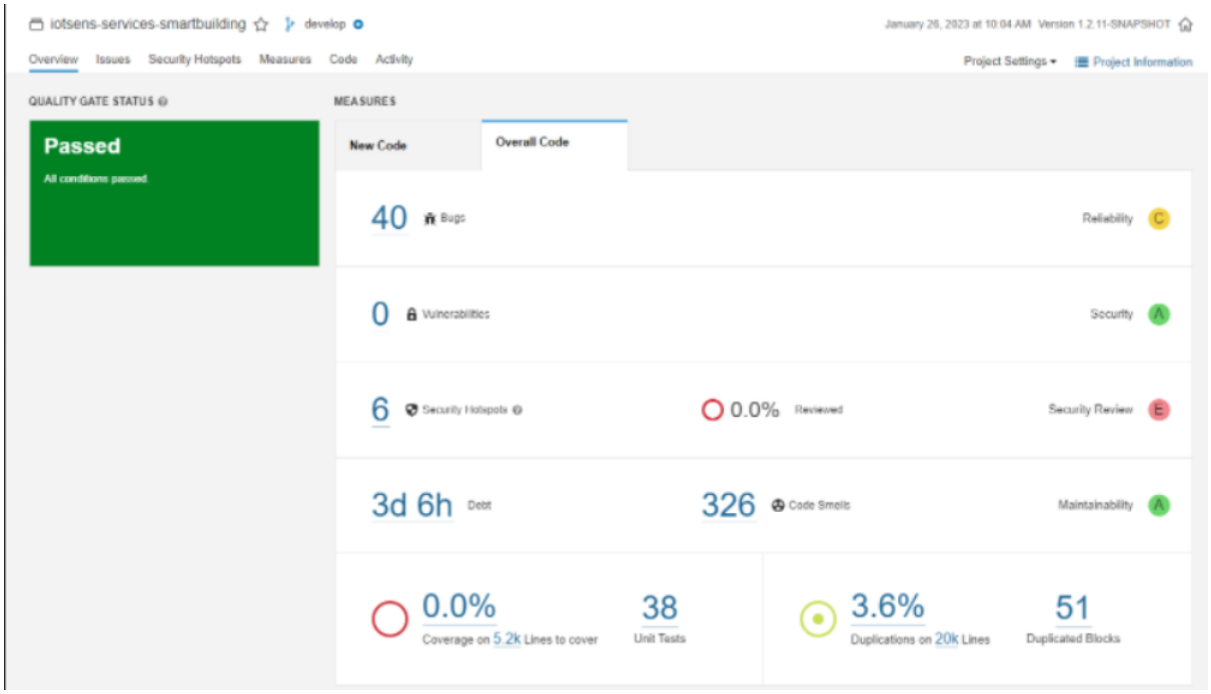

Figura 5.8: Análisis de SonarQube en Smart Building.

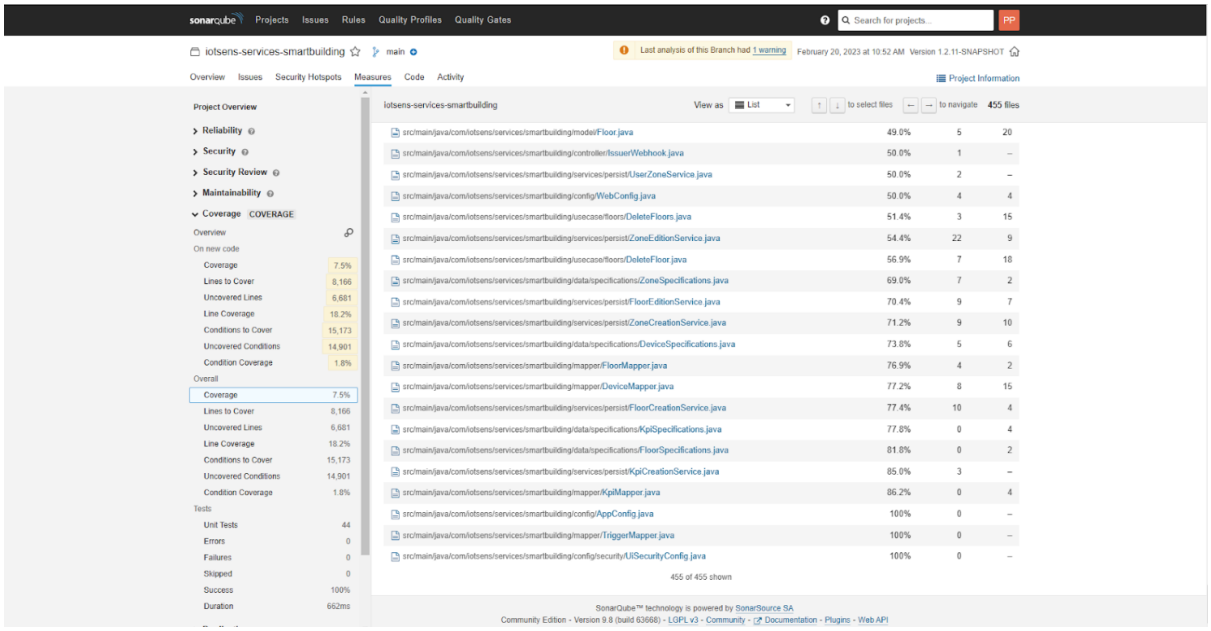

Figura 5.9: Análisis de cobertura de SonarQube en Smart Building.

Capítulo 6

Conclusiones

El objetivo de este proyecto ha sido configurar un entorno de pruebas automatizado para un software ya existente, además de implementar las pruebas necesarias para verificar el correcto funcionamiento de dicho software. Se han logrado cumplir otros subobjetivos que se esperaban, habiendo sido capaz de entender la importancia del testing y lo útil que resulta tener un buen entorno de pruebas dentro de un equipo de trabajo. El producto final cumplió con los requisitos especificados dentro del plazo establecido. Además, se llegó a expandir en otras soluciones de la empresa, indicativo del éxito de este producto. Ha servido a la empresa como base para iniciarse en el mundo del testing, cosa que era inexistente hasta la llegada de este proyecto.

Es importante destacar que este proyecto no ha sido un TFG convencional, ya que se ha centrado principalmente en la configuración y validación de un entorno de pruebas automatizado para un software existente. La validación, que es una etapa fundamental dentro de cualquier proyecto, ha sido el foco principal de este trabajo. Sin embargo, a pesar de no seguir el formato tradicional de un TFG, el proyecto ha sido igualmente relevante y ha brindado a la empresa beneficios tangibles, como la mejora de la calidad del software y la introducción de un proceso de pruebas más eficiente.

Fue un proceso que tuvo sus complicaciones, poniendo a prueba mi capacidad de resolución de problemas, especialmente a la hora de configurar el entorno, pero que al final conseguí sacar adelante. Una vez terminado el proyecto, ver que el flujo del sistema funciona sin problemas se hace muy satisfactorio. Además, he aprendido mucho sobre la importancia del testing en un proyecto de software, especialmente cuando el proyecto comienza a escalar y las pruebas manuales son inviables. También cabe destacar la importancia de realizar pruebas a la vez que se desarrolla el resto de software. De esta forma, es posible verificar que lo que se está desarrollando funciona correctamente y se está avanzando en la dirección adecuada. De lo contrario, realizar pruebas una vez se ha terminado el desarrollo puede resultar complicado de organizar y de obtener un resultado eficiente.

En cuanto a la organización del proyecto, mediante la metodología ágil seguida y las herramientas de gestión de proyectos, se ha obtenido una mejora en la gestión del tiempo y las tareas a realizar. Por un lado, Redmine sirvió de gran ayuda a la hora de asignar tareas y realizar un seguimiento de ellas, mientras que mediante el uso de Teams se pudo establecer una comunica-

ción con el resto de compañeros de departamento que sirvió de gran ayuda, tanto mediante el chat como mediante las reuniones.

Para este proyecto, los conocimientos obtenidos durante el grado han sido de gran ayuda, especialmente las asignaturas que forman parte del itinerario escogido, en este caso Ingeniería del Software. Las herramientas, lenguajes de programación y patrones de diseño utilizados en el proyecto ya habían sido estudiados durante la etapa universitaria o tenían una estructura muy similar, lo que facilitó la adaptación a las nuevas tecnologías. Sin embargo, se realizaron mejoras significativas en el proceso de implementación para optimizar el rendimiento y la eficiencia del sistema.

A nivel personal, esta experiencia ha supuesto desafíos, pero también ha sido gratificante. He adquirido habilidades para trabajar en equipo en un entorno profesional, adaptarme a situaciones cambiantes y abordar problemas complejos de manera creativa y decidida. He fortalecido mis aptitudes colaborativas y mejorado mis habilidades de programación para el desarrollo de proyectos.

Además, es importante destacar que la experiencia de trabajar en el desarrollo de un proyecto y en esta empresa ha sido sumamente positiva. Mis compañeros de trabajo han brindado asistencia en los desafíos que surgían durante el proceso, y la empresa me ha brindado la oportunidad de trabajar en un proyecto con impacto real. Mediante este proyecto, he adquirido conocimientos en tecnologías relevantes para el desarrollo de software y aplicaciones web, lo cual me abrirá muchas puertas en el futuro.

En cuanto a posibles trabajos futuros relacionados con este proyecto, una opción interesante sería seguir mejorando y ampliando las capacidades del entorno de pruebas automatizado. Se podrían implementar nuevas pruebas y funcionalidades que cubran casos más complejos y variados, incluyendo pruebas de rendimiento, pruebas de seguridad y pruebas de usabilidad. Esto garantizaría que el software se encuentra en un estado óptimo en términos de calidad y rendimiento.

Bibliografía

[1] Iotsens. https://www.iotsens.com/compania/. [Consulta: 15 de mayo de 2023].

[2] IoT. Internet de las cosas. https://es.wikipedia.org/wiki/Internet_de_las_cosas. [Consulta: 30 de mayo de 2023].

[3] Digital55. Cómo usar testing en angular con jasmine y karma. https://digital55.com/blog/como-usar-testing-angular-jasmine-karma/. [Consulta: 5 de mayo de 2023].

[4] Indeed.com. ¿cuánto se gana como uno programador/a junior en españa? https://es.indeed.com/career/programador-junior/salaries. [Consulta: 5 de mayo de 2023].

[5] Enalquiler. Evolución del alquiler en castellón. https://www.enalquiler.com/precios/precio-alquiler-vivienda-castellon-castello_21-14-12492-0.html. [Consulta: 30 de mayo de 2023].

[6] Angular. Testing services. https://angular.io/guide/testing-services. [Consulta: 30 de mayo de 2023].

[7] Baeldung. Mockito.mock() vs @mock vs @mockbean. https://www.baeldung.com/java-spring-mockito-mock-mockbean. [Consulta: 5 de mayo de 2023].

[8] Daniel Díaz Suarez. Tdd, bdd & test de aceptación. https://www.adictosaltrabajo.com/2013/05/27/tdd-bdd-test-de-aceptacion/. [Consulta: 9 de mayo de 2023].

[9] GitLab. Gitlab ci/cd. https://docs.gitlab.com/ee/ci/. [Consulta: 10 de mayo de 2023].

[10] molily. Test suites with jasmine. https://testing-angular.com/test-suites-with-jasmine/#test-suites-with-jasmine. [Consulta: 9 de mayo de 2023].

[11] Abha Gupta. Increase your code coverage using istanbul. https://medium.com/walmartglobaltech/do-you-have-100-code-coverage-10c09a44832b. [Consulta: 9 de mayo de 2023].

[12] SonarQube. Analyzing source code. https://docs.sonarqube.org/latest/analyzing-source-code/overview/. [Consulta: 16 de mayo de 2023].