

***Facultad
de
Ciencias***

**HAPI SECURITY: APLICACIÓN PARA EL
ANÁLISIS DE DISPOSITIVOS IoT EN BASE A
SU SEGURIDAD Y SOSTENIBILIDAD**
**Hapi Security: Mobile app for the analysis of
IoT devices based on their security and
sustainability**

**Trabajo de Fin de Grado
para acceder al**

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Mario Ingelmo Diana

Director: Carlos Blanco Bueno

Co-Director: Juan Maria Rivas Concepcion

Julio – 2023

Índice

Resumen	5
Palabras clave:	5
Abstract	6
Key Words:	6
1. Introducción	7
1.1. Objetivo	8
2. Herramientas y metodología utilizada	8
2.1. Tecnologías y Herramientas	8
2.1.1. Git y GitHub	9
2.1.2. Spring Boot, Maven y Java	9
2.1.3. Eclipse	10
2.1.4. Microsoft Azure	10
2.1.5. Android Studio y Java	11
2.2. Metodología	11
2.3. Planificación del trabajo	12
2.3.1. Preparación previa al desarrollo	13
2.3.2. Desarrollo y despliegue del software	13
2.3.3. Desarrollo de la memoria	14
3. Desarrollo del servicio	14
3.1. Análisis de requisitos del servicio	14
3.1.1. Idea tras el servicio	14
3.1.2. Requisitos funcionales	14
3.1.3. Requisitos no funcionales	15
3.2. Diseño del servicio	16
3.2.1. Diseño arquitectónico	16
3.2.2. Diseño REST	17
3.2.3. Diseño de la base de datos	17
3.3. Implementación del servicio	17
3.3.1. Implementación de la repository layer	18
3.3.1.1. Entidades	18
3.3.1.2. Interfaces de repositorio	19
3.3.1.3. Enumerados	20
3.3.1.4. Listas	20
3.3.2. Implementación de la service layer	20
3.3.3. Implementación de la controller layer	22
3.3.4. Implementación de la seguridad y la configuración	25
3.4. Pruebas del servicio	27
3.5. Despliegue del servicio	27

4.	Desarrollo de la aplicación móvil	28
4.1.	Análisis de requisitos de la aplicación	28
4.1.1.	Idea tras la aplicación	28
4.1.2.	Diseño del logotipo	30
4.1.3.	Requisitos funcionales	30
4.1.4.	Requisitos no funcionales	31
4.2.	Diseño de la aplicación	31
4.2.1.	Diseño arquitectónico	31
4.2.1.1.	Diseño del modelo	33
4.2.1.2.	Diseño de la vista	33
4.2.1.3.	Diseño del presentador	34
4.3.	Implementación de la aplicación	34
4.3.1.	Implementación de la conexión con el servicio REST	34
4.3.2.	Implementación de la base de datos	35
4.3.3.	Implementación del menú inicial	36
4.3.4.	Implementación del dispositivo en detalle	37
4.3.5.	Implementación de la sección de buscador	38
4.3.6.	Implementación de la sección de escanear	41
4.3.7.	Implementación de la sección de favoritos	42
4.3.8.	Implementación de la sección de compartir	43
4.4.	Pruebas de la aplicación	44
4.4.1.	Pruebas unitarias	44
4.4.2.	Pruebas de integración e interfaz	45
4.4.3.	Pruebas de sistema	47
4.4.4.	Pruebas de aceptación	47
4.4.5.	Análisis de la calidad	48
5.	Conclusiones y trabajo futuro	48
5.1.	Conclusiones	48
5.2.	Trabajo futuro	49
	Bibliografía	50

Resumen

Actualmente, utilizamos un gran número y variedad de dispositivos inteligentes conectados a la red (IoT) tanto en entornos domésticos como laborales, sanitarios, de transporte, etc. Aunque dichos dispositivos manejan infraestructuras críticas e información sensible, no suelen contar con un buen nivel de seguridad. Por otro lado, al ser dispositivos que están continuamente conectados, la sostenibilidad también es un aspecto importante a tener en cuenta. Este proyecto se centra en ayudar a los usuarios a mejorar la infraestructura de su entorno (hogar, oficina, etc.) mediante el uso de dispositivos más adecuados.

Para ello, en este proyecto se desarrolla una aplicación móvil (Hapi Security) que permite conocer cómo de seguro y sostenible es un dispositivo IoT concreto, a qué se deben sus problemas de seguridad y sostenibilidad, y si existen alternativas mejores. Cada dispositivo tiene asociada una calificación de seguridad (de 0 a 100) y de sostenibilidad (de A a G) y una serie de aspectos que presenta e influyen de forma positiva o negativa en dichas calificaciones. La aplicación ha sido desarrollada para Android utilizando Java y presenta funcionalidades para permitir búsquedas y filtrados de dispositivos, escanear códigos de barras, gestionar favoritos y compartir la aplicación.

Como apoyo a dicha aplicación, se desarrolla un servicio REST utilizando Spring Boot con información sobre el catálogo de dispositivos IoT (alineada con ENISA) junto con sus calificaciones de seguridad y sostenibilidad y el listado de aspectos que influyen positiva o negativamente en ellas.

Palabras clave:

Aplicación móvil, Dispositivos IoT, Seguridad, Sostenibilidad, Servicio REST, Java, Android Studio.

Abstract

Currently, we use a substantial number and variety of smart devices connected to the network (IoT) both in home and work environments, health, transportation, etc. Although these devices handle critical infrastructure and sensitive information, they often do not have a good level of security. On the other hand, being devices that are continuously connected, sustainability is also an important aspect to consider. This project focuses on helping users to improve the infrastructure of their environment (home, office, etc.) by using more suitable devices.

To do this, in this project a mobile application (Hapi Security) is developed that allows knowing how safe and sustainable a specific IoT device is, the reasons behind that rating, and if there are better alternatives. Each device is associated with a safety rating (from 0 to 100) and sustainability (from A to G) and a series of aspects that it presents and influences positively or negatively in said ratings. The application has been developed for Android using Java and presents functionalities to allow searching and filtering of devices, scanning barcodes, managing favorites, and sharing the application.

In support of said application, a REST service is developed using Spring Boot with information about the IoT device catalog (aligned with ENISA) together with its security and sustainability ratings and the list of aspects that positively or negatively influence them.

Key Words:

Mobile application, IoT devices, Security, Sustainability, REST service, Java, Android Studio.

1. Introducción

Hoy en día vivimos rodeados de dispositivos IoT, los entornos IoT desempeñan un papel cada vez más importante, estos dispositivos hacen de puente entre el mundo físico y el digital, incluyen todo tipo de funciones de computación, almacenamiento y comunicación que les permite gestionar objetos en el mundo físico y proporcionan servicios en numerosas áreas como la salud, el suministro de energía, el transporte, la automatización industrial o el hogar inteligente. Todo ello sumado a su constante conexión a Internet supone un riesgo en la seguridad de estos y en la seguridad de sus usuarios.

Prueba de esto es la cantidad de ataques que se detectan a dispositivos de este tipo. *Kaspersky*, conocida compañía internacional en el sector de la ciberseguridad, hizo públicos los siguientes datos sobre ataques a sus honeypots (Software que imita un dispositivo IoT vulnerable) en 2021: “En el primer semestre de 2021, el número de intentos de infección totales alcanzó los 1.515.714.259, mientras que durante los seis meses anteriores fueron 639.155.942” [1]. A la vista de estos datos observamos como el aumento de los ataques, solamente en “señuelos” de la empresa *Kaspersky*, casi se triplican, lo que nos da una idea general de lo que puede suponer a nivel global donde actualmente hay alrededor de 7 mil millones de dispositivos IoT conectados a la red y se estima un crecimiento hasta los 27 mil millones en 2025 [2].

A la vista de estos datos, podemos observar la gran importancia que tiene la seguridad en los dispositivos IoT. La falta de seguridad en estos dispositivos puede tener consecuencias considerables. Por un lado, sus áreas de aplicación suelen corresponder a infraestructuras críticas, en las que la interrupción o el compromiso de estos sistemas puede tener consecuencias devastadoras, que van desde interrupciones en los servicios públicos hasta riesgos para la seguridad y la vida humana. Por otro lado, recopilan y procesan grandes cantidades de datos sensibles, como información personal, datos de salud o datos empresariales confidenciales. Por ello, la falta de seguridad en estos sistemas puede resultar en fugas de datos, robo de información personal o financiera, y posibles daños a la reputación de las organizaciones. Sin embargo, su rápida evolución y adopción ha llevado a que muchos de ellos se diseñen y se lancen al mercado sin una atención adecuada a estos aspectos, generando así un gran número de vulnerabilidades que pueden ser explotadas. Además la gran variedad de dispositivos de este tipo dificulta la estandarización de unas medidas de seguridad consistentes y eficaces. Adicionalmente, este es un aspecto al que poca gente presta atención y cuyos datos son de difícil acceso.

También es importante la sostenibilidad de estos dispositivos, tanto la eficiencia energética al estar conectados continuamente, como el proceso de fabricación de este (si es respetuoso con el medio ambiente), o la reparabilidad del dispositivo. Es un aspecto más ignorado que la seguridad, pero también es vital, puesto que puede ayudarnos a ahorrar ya sea en la factura de la luz o en reparaciones, y puede ayudar a cuidar del planeta.

A la vista de la dificultad de obtener datos sobre la seguridad y sostenibilidad de estos dispositivos, ya que la cantidad de estos es enorme y abarca muchas categorías

diferentes y para cada categoría muchos fabricantes, se desarrolla una aplicación móvil (Hapi Security) donde poder consultar la seguridad de los diferentes dispositivos IoT del mercado, además de la sostenibilidad y las listas con los aspectos tanto positivos como negativos de seguridad y sostenibilidad, de manera que el usuario tenga fácil acceso a los mismos y pueda valorar diferentes opciones a la hora de comprar dispositivos IoT en materia de seguridad y sostenibilidad.

El nombre de la aplicación proviene del dios egipcio Hapi, dios encargado de la inundación anual del río Nilo, que proveía de suelo fértil para los cultivos y así de alimento al pueblo egipcio. Por eso, era considerado por muchos el dios de la seguridad, al encargarse de “proteger” a los egipcios dándoles una tierra donde cultivar y así mantener la vida y la economía.

El presente TFG se ha desarrollado en el marco del proyecto ALBA: mejora de la ciberseguridad y su sostenibilidad en Beneficio de la sociedad y de las personas, financiado por el Ministerio de Ciencia e Innovación dentro de la convocatoria de Proyectos de Transición Ecológica y Transición Digital. Además de estar alineado con los Objetivos de Desarrollo Sostenible (ODS) al tratar temas como la sostenibilidad de los dispositivos IoT y contribuir en estos.

1.1. Objetivo

Como objetivo principal de este proyecto se plantea el desarrollo de una aplicación móvil (llamada Hapi Security) donde los usuarios puedan consultar información sobre la seguridad y sostenibilidad de dispositivos IoT. Dicha aplicación les permitirá buscar dispositivos, observar sus calificaciones en cuanto a seguridad y sostenibilidad y a qué se deben (qué aspectos presentan que afectan positiva o negativamente en la seguridad y sostenibilidad). De esta forma, los usuarios podrán tomar decisiones más informadas a la hora de comprar un dispositivo nuevo o de reemplazar alguno de sus dispositivos actuales.

Para la consecución de dicho objetivo se plantea el desarrollo de un servicio donde almacenar y obtener los datos de los dispositivos: categorías, dispositivos, información de detalle, calificaciones de seguridad y sostenibilidad asociadas, aspectos que influyen positiva o negativamente en la seguridad y/o sostenibilidad, etc. La aplicación móvil utilizará este servicio a la vez que proporcionará funcionalidades de búsqueda, filtrado, gestión de favoritos, etc.

2. Herramientas y metodología utilizada

Este apartado recoge tanto la metodología y la planificación del trabajo seguida, como las tecnologías y herramientas utilizadas.

2.1. Tecnologías y Herramientas

Las tecnologías y herramientas que se han utilizado son las siguientes:

2.1.1. Git y GitHub

Git es un sistema avanzado de control de versiones (como el “control de cambios” de Microsoft Word) distribuido. Git permite “rastrear” el progreso de un proyecto a lo largo del tiempo ya que hace “capturas” del mismo a medida que evoluciona y los cambios se van registrando. Esto permite ver qué cambios se hicieron, quién los hizo y por qué, e incluso volver a versiones anteriores [4].

Por otra parte, GitHub es un servidor de alojamiento en línea o repositorio remoto para albergar proyectos basados en Git que permite la colaboración entre diferentes usuarios o con uno mismo. Un repositorio es un directorio donde desarrollar un proyecto que contiene todos los archivos necesarios para el mismo [4].



Imagen 1. Logos de Git y GitHub

Se usarán tanto Git por línea de comandos, para ir almacenando los cambios y poder llevar así un control de versiones del proyecto, como GitHub para almacenar el repositorio en la nube. Estas tecnologías son mundialmente conocidas y utilizadas, y se utilizan en el proyecto para facilitar la colaboración, trazabilidad y manejo de código y documentos.

2.1.2. Spring Boot, Maven y Java

Para el desarrollo del servicio se ha tomado la decisión de utilizar Spring Boot, desarrollado por la empresa Spring y que está disponible para usarse en Java, Kotlin o Groovy. En mi caso, al ser el lenguaje más dominado Java, se ha utilizado este, además de utilizar Maven a la hora de manejar el empaquetamiento del servicio.

Java Spring Boot (Spring Boot) es una herramienta que acelera y simplifica el desarrollo de microservicios y aplicaciones web con Spring Framework gracias a tres funciones principales:

- Configuración automática.
- Un enfoque de configuración que facilita los pasos a dar.
- La capacidad de crear aplicaciones autónomas.

Estas características, combinadas, conforman una herramienta que le permite configurar una aplicación basada en Spring con el mínimo de instalación y configuración [5].



Imagen 2. Logo de Spring Boot

Hoy en día, la mayoría de las empresas piden conocimientos sobre cómo implementar microservicios con Spring, por lo que, además de parecerme la opción más cómoda después de valorar varias aprendidas en la asignatura de *Servicios Software*, también me pareció la que más flexibilidad podía otorgarme y más proyección a futuro podía tener.

2.1.3. Eclipse

Para el desarrollo del servicio mencionado anteriormente, gracias a la gran cohesión que tiene tanto con Java (es uno de los entornos más utilizados a nivel mundial para el desarrollo de software Java) como con Spring Boot y Maven, se ha decidido utilizar como entorno de desarrollo Eclipse IDE for Enterprise Java and Web Developers.

```
GeneralController.java ❷
1 package es.unican.hapisecurity.REST_TFGMarioIngelmoDiana.controllerLayer;
2
3 import java.net.URI;
4
5 @RestController
6 @RequestMapping("REST_TFGMarioIngelmoDiana")
7 public class GeneralController {
8
9     @Autowired
10    private AuthenticationManager authenticationManager;
11
12    @Autowired
13    private UserDetailsServiceImpl usuarioDetailsService;
14
15    @Autowired
16    private GestionTokens gestion;
17
18    @Autowired
19    private GeneralService servicio;
20
21 }
```

Imagen 3. Ejemplo de uso de Eclipse en el servicio implementado

2.1.4. Microsoft Azure

Microsoft Azure es una plataforma desarrollada por Microsoft compuesta por más de 200 productos y servicios en la nube [6]. Entre los servicios que ofrece encontramos desde Bases de Datos en la nube, hasta la creación de máquinas virtuales o el despliegue de aplicaciones Spring.



Imagen 4. Logo de Microsoft Azure

Gracias a estas características que ofrece y a la disponibilidad de una licencia de estudiante, se ha decidido utilizar para usar una de sus bases de datos para el servicio y poder desplegarlo en una máquina virtual.

2.1.5. Android Studio y Java

Android Studio es el entorno de desarrollo oficial que se usa en el desarrollo de aplicaciones Android [7]. Android Studio admite desarrollo en dos lenguajes, Java y Kotlin, en este caso se va a desarrollar la aplicación en Java, puesto que ya he desarrollado más aplicaciones Android en el mismo lenguaje y el dominio que tengo de este lenguaje es considerable a diferencia del dominio que tengo de Kotlin.

Además Android Studio ofrece todo tipo de facilidades a la hora de desarrollar aplicaciones Android, dispone de compilación flexible basada en Gradle, un emulador de dispositivos Android donde emular una gran cantidad de dispositivos con diferentes versiones de Android y otras muchas funcionalidades que son de gran ayuda.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    // Creo la view para la actividad principal con el xml correspondiente
    this.view = inflater.inflate(R.layout.fragment_buscador, container, attachToRoot: false);

    if (Red.isNetworkAvailable(this.requireContext())) {
        presenter = new BuscadorPresenter( view: this, categoriaSeleccionada, String.valueOf(valorSeguridad),
    } else {
        presenter = new BuscadorPresenter( view: this, categoriaSeleccionada, String.valueOf(valorSeguridad),
    }

    this.init();
    return view;
}
```

Imagen 5. Ejemplo de uso de Android Studio en la aplicación

2.2. Metodología

La metodología seguida ha sido la iterativa incremental. Esta metodología consiste en dividir el proyecto en diferentes iteraciones o ciclos. En cada iteración el producto se va actualizando de manera que se desarrolla hasta llegar a un producto final que cumpla con los requisitos y objetivos marcados [3]. Se ha elegido esta metodología porque así las funcionalidades se desarrollan de una en una, dado que en mi opinión, esto beneficia el correcto desarrollo del producto total al pulir cada una de las funcionalidades en la

iteración correspondiente y poder ir usando esas implementaciones en el desarrollo de las siguientes a esta.



Imagen 6. Representación de la metodología iterativa incremental

En este proyecto cada iteración se ha dividido en 4 partes:

- Requisitos.
- Diseño.
- Implementación.
- Pruebas.

2.3. Planificación del trabajo

La planificación de este proyecto puede dividirse en tres apartados principales: preparación previa al desarrollo, desarrollo y despliegue del software, y desarrollo de la memoria del trabajo.

A continuación se añade un diagrama de Gantt con la planificación del trabajo, en la que se incluyen las iteraciones realizadas.

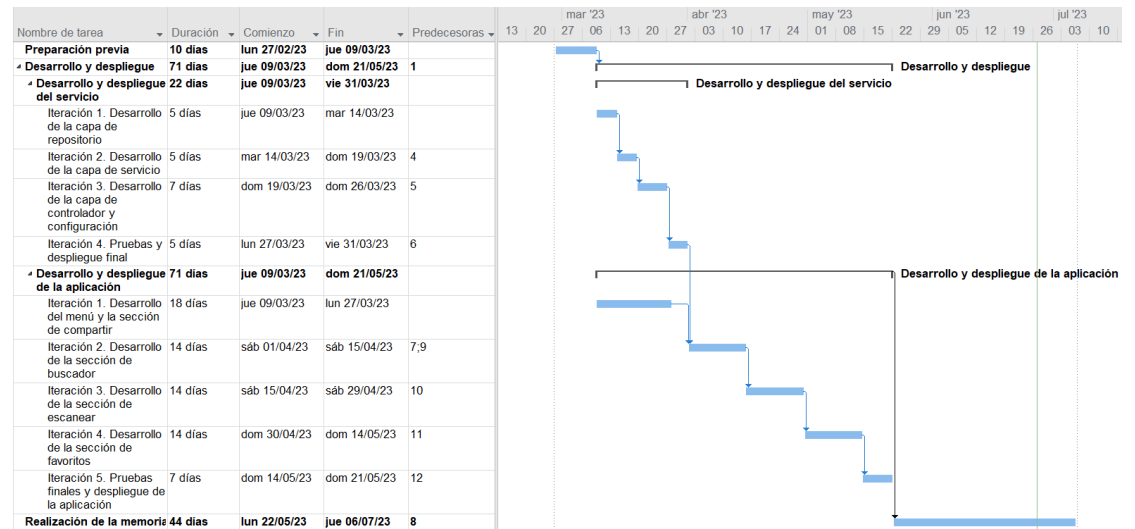


Imagen 7. Diagrama de Gantt

2.3.1. Preparación previa al desarrollo

Este apartado es clave puesto que es el inicio de todo. Que este apartado nazca con buen pie es de suma importancia, ya que se determinan los objetivos principales del proyecto y qué herramientas se van a utilizar para lograrlos. Por lo que una buena elección de objetivos y herramientas ayuda en las diferentes etapas del proyecto.

Tras una pequeña reunión inicial donde intercambiamos los directores del proyecto y yo las ideas que teníamos, pusimos en consonancia los objetivos a desarrollar para este proyecto, llegando a un acuerdo y rellenando un documento con estos, para ir revisándolos y ver que todo se completaba adecuadamente.

En otra reunión donde se siguió perfilando la idea del proyecto también se evaluaron las herramientas de desarrollo y se estableció que tecnologías utilizar para el desarrollo de las diferentes partes del proyecto. Estas se explicarán más adelante.

2.3.2. Desarrollo y despliegue del software

Este apartado corresponde con lo hablado anteriormente en la metodología. Se ha dividido en iteraciones, incluyendo cada una de las fases mencionadas anteriormente (requisitos, diseño, implementación y pruebas), para una vez finalizado todo el trabajo, realizar el despliegue final del servicio y de la aplicación móvil.

A continuación se detallan las fases en las que se divide cada iteración:

- **Requisitos:** Se establecen los objetivos a desarrollar en cada iteración del software (requisitos), estos pueden dividirse en dos:
 - o **Requisitos funcionales:** Son requisitos sobre las funcionalidades que nuestro sistema deberá implementar y ofrecer a los usuarios, tales como poder buscar, filtrar, escanear, etc...
 - o **Requisitos no funcionales:** Son atributos de calidad del sistema, tales como la seguridad, el rendimiento, la mantenibilidad, etc...
- **Diseño:** Se analizan los requisitos definidos anteriormente y se plantea una solución sobre cómo poder implementarlos en consonancia con los ya implementados. Esto implica generar o ampliar el diseño a alto nivel (arquitectura) así como el diseño detallado del sistema.
- **Implementación:** Basándose en el diseño creado en el apartado anterior, este es implementado en el software de manera que añada toda la funcionalidad nueva establecida en los objetivos de la iteración.
- **Pruebas:** Tras realizar la implementación del software en el apartado anterior se pasa al testeo de los cambios realizados, ya sea funcionalidad nueva o actualizada, tanto individualmente, como en conjunto. Por ello, se realizan cuatro tipos de pruebas diferentes que explicaremos más adelante: unitarias, integración, sistema y aceptación.

2.3.3. Desarrollo de la memoria.

Aunque durante las diferentes etapas de desarrollo se ha ido generando documentación del sistema, la memoria como tal se ha realizado una vez todo el software ha sido desarrollado, testado y desplegado. Esta decisión podría haber sido completamente diferente y haberlo hecho en paralelo con el desarrollo, pero en mi caso me decanté por un desarrollo de esta posterior, para así centrar todos mis esfuerzos en el correcto desarrollo del servicio y la aplicación.

3. Desarrollo del servicio

Este capítulo describe todas las etapas referentes al desarrollo del servicio (requisitos, diseño, implementación, pruebas y despliegue), dejando para el siguiente capítulo la descripción referente al desarrollo de la aplicación.

3.1. Análisis de requisitos del servicio

En este apartado se analizarán los requisitos que se han determinado para el servicio a desarrollar. Se dará una idea sobre qué se busca con el servicio y se analizarán tanto los requisitos funcionales como los no funcionales.

3.1.1. Idea tras el servicio

La idea tras el servicio es la de construir y desplegar un servicio REST que dé funcionalidad a la hora de obtener los dispositivos con sus respectivos datos asociados. En resumen, debe permitir:

- Obtener los datos de los dispositivos o de un solo dispositivo y sus características de seguridad y sostenibilidad asociadas.
- Filtrar la lista de los dispositivos.
- Ordenar la lista de los dispositivos.
- Obtener las diferentes características o una sola de ellas.
- Crear o modificar dispositivos.
- Crear características nuevas.

Además, la seguridad se puntuará de 0 a 100 y la sostenibilidad de A a G. Y las categorías de los dispositivos estarán alineadas con ENISA (Agencia de la Unión Europea para la Ciberseguridad) de manera que sigan un criterio unificado por una agencia europea.

3.1.2. Requisitos funcionales

A continuación se presenta una tabla con los diferentes requisitos funcionales identificados:

ID	Descripción
RF1	El usuario podrá obtener una lista con todos los dispositivos junto con su información asociada (foto, nombre, marca, categoría, precio, descripción, puntuación de seguridad, puntuación de sostenibilidad y lista de características que influyen positiva o negativamente en su seguridad o sostenibilidad).
RF2	El usuario podrá obtener una lista con todos los dispositivos y su información asociada, filtrada por la categoría de los dispositivos que indique.
RF3	El usuario podrá obtener una lista con todos los dispositivos su información asociada, filtrada por la seguridad mínima que indique.
RF4	El usuario podrá obtener una lista con todos los dispositivos su información asociada, filtrada por la sostenibilidad mínima que indique.
RF5	El usuario podrá obtener una lista con todos los dispositivos su información asociada, filtrada por los tres parámetros de los requisitos anteriores de manera conjunta o combinados.
RF6	El usuario podrá obtener una lista con todos los dispositivos su información asociada, ordenada alfabéticamente, por puntuación de seguridad de mejor a peor o por puntuación de sostenibilidad de mejor a peor.
RF7	El usuario podrá obtener una lista con todos los dispositivos su información asociada, ordenada y filtrada por cualquiera de las opciones de los requisitos anteriores.
RF8	El usuario podrá obtener un dispositivo indicando su id.
RF9	El usuario podrá obtener una lista con todas las características que influyen positiva o negativamente en la seguridad o sostenibilidad.
RF10	El usuario podrá obtener una característica que influye positiva o negativamente en la seguridad o sostenibilidad indicando su id.
RF11	El usuario deberá identificarse para realizar modificaciones en los datos.
RF12	El usuario podrá obtener un token de seguridad identificándose (administrador).
RF13	El administrador podrá añadir nuevos dispositivos.
RF14	El administrador podrá modificar dispositivos ya añadidos.
RF15	El administrador podrá añadir nuevas características.

Tabla 1. Requisitos funcionales del servicio

3.1.3. Requisitos no funcionales

A continuación se presenta una tabla con los diferentes requisitos no funcionales identificados:

ID	Clasificación	Descripción
RNF1	Seguridad	El servicio deberá estar protegido utilizando JSON Web Tokens.
RNF2	Usabilidad, Mantenibilidad	El servicio deberá poder ser utilizado desde otras aplicaciones que lo necesiten.
RNF3	Portabilidad	El servicio deberá estar desplegado en un entorno de acceso global.
RNF4	Fiabilidad	El servicio deberá estar disponible (en funcionamiento) cuando se realice una llamada al mismo

RNF5	Mantenibilidad	El servicio deberá poder añadir o modificar funcionalidades sin impactar en el resto del servicio.
------	----------------	--

Tabla 2. Requisitos no funcionales del servicio

3.2. Diseño del servicio

Podemos dividir el diseño del servicio en dos partes. Primero el diseño arquitectónico del servicio, con la arquitectura que se ha seguido y una explicación de cada capa, y segundo, el diseño del servicio REST con los recursos, URIs, métodos HTTP y códigos de respuesta HTTP.

3.2.1. Diseño arquitectónico

El servicio se ha diseñado usando la arquitectura típica de los servicios de Spring Boot. Esta arquitectura está basada en capas, donde encontramos que se descompone en capa de control, capa de servicio y capa de repositorio.

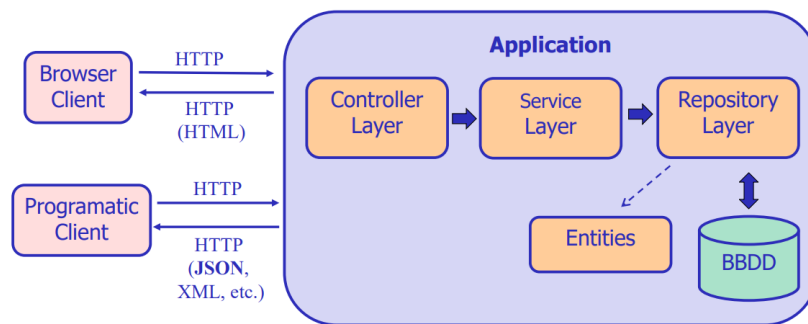


Imagen 8. Arquitectura típica de las aplicaciones Spring Boot

A continuación se describen las capas de las que dispone el servicio:

- **Controller Layer:** Es la capa encargada de recibir las solicitudes HTTP y como su nombre indica, controlarlas. Esta capa puede estar formada por una o más clases anotadas con `@RestController`, que son las clases encargadas de, dependiendo de la URI solicitada por el cliente, procesar la petición y devolver una respuesta a este.
- **Service Layer:** Es la capa encargada de hacer de conexión entre la capa de control y la capa de repositorio. Facilita el manejo de diferentes funcionalidades desde los controladores al simplificar el código a la hora de, por ejemplo, conseguir un dispositivo, actualizarlo, etc, al no interactuar la capa de control directamente con el repositorio.
- **Repository Layer:** Es la capa encargada de comunicarse con la base de datos y obtener las entidades correspondientes. En este servicio se ha implementado con JPA que ayuda a la hora de gestionar las entidades y la base de datos mediante anotaciones e interfaces predefinidas.

3.2.2. Diseño REST

A la vista de los requisitos, se han definido los recursos, URIs, métodos y respuestas HTTP, y se ha generado el siguiente diseño REST para el servicio:

RECURSO	URI	MÉTODOS	RESPUESTAS HTTP
Token JWT	/REST_TFGMarioIngelmoDiana/token	POST	200, 400, 401, 403, 500
Lista Dispositivos	/REST_TFGMarioIngelmoDiana/dispositivos	GET - categoría - seguridad - sostenibilidad - ordenar	200, 400, 404, 500, 503
Dispositivo	/REST_TFGMarioIngelmoDiana/dispositivos/{id}	GET, PUT	200, 201, 400, 401, 403, 404, 500, 503
Lista Características	/REST_TFGMarioIngelmoDiana/caracteristicas	GET, POST	200, 201, 400, 401, 403, 404, 500, 503
Característica	/REST_TFGMarioIngelmoDiana/caracteristicas/{id}	GET	200, 400, 404, 500, 503

Tabla 3. Diseño REST del servicio

3.2.3. Diseño de la base de datos

La base de datos se ha modelado de manera que se disponga de dispositivos con sus atributos y que tengan asociados las características correspondientes dependiendo de si son positivas o negativas y de si afectan a seguridad o sostenibilidad.

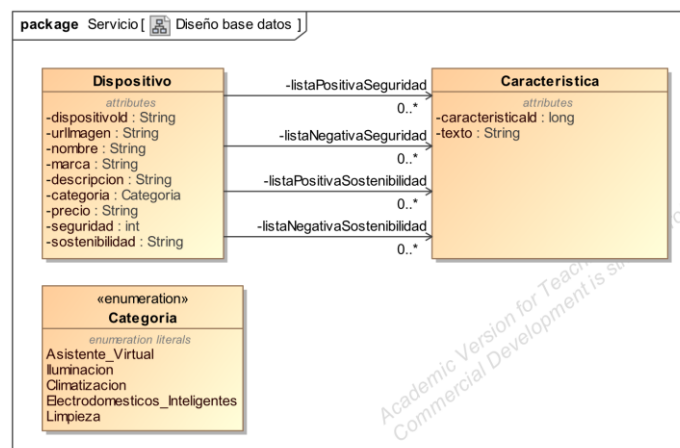


Imagen 9. Diseño base de datos

3.3. Implementación del servicio

Tomando como referencia el diseño tanto arquitectónico como REST del servicio, el siguiente paso es implementarlo, para ello, se ha creado un proyecto con Spring Boot Initializr con las siguientes dependencias:

- **JPA:** Esta dependencia sirve para manejar las entidades en la base de datos que se utilice en el servicio. Esto es, maneja automáticamente la base de datos mediante anotaciones, anotando las clases como entidades (@Entity) y permitiendo así su mapeo a la base de datos seleccionada, así como también a sus atributos y relaciones con otras clases.
- **Web:** Esta dependencia proporciona herramientas para desarrollar aplicaciones web en Spring Boot.
- **MySQL:** Esta dependencia sirve para implementar la base de datos del servicio en MySQL, se encarga de conectar el servicio con una base de datos MySQL local o en la nube. En este caso, la base de datos se ha implementado en Azure, como se comentará en la implementación de la repository layer.
- **JWT:** Esta dependencia sirve para implementar seguridad con JSON Web Tokens y así poder restringir el acceso a los usuarios a diferentes funcionalidades como modificar dispositivos, añadir características, etc.

La implementación se irá explicando por capas, siendo la última a explicar la capa de controlador donde también se comentará la implementación del diseño REST. Adicionalmente se explicará la configuración y la seguridad JWT.

3.3.1. Implementación de la repository layer

Esta capa está formada por dos entidades (Característica.java y Dispositivo.java), dos interfaces de repositorios (CaracteristicaRepository.java y DispositivoRepository.java), dos enumerados (Categoria.java y TipoOrdenar.java) y dos clases para las listas de características y dispositivos (ListaCaracteristicas.java y ListaDispositivos.java).

Cabe destacar que esta capa realiza la conexión con la base de datos, es la encargada de comunicarse con esta y realizar los cambios pertinentes u obtener los datos. Para esto se modifica el archivo application.properties de manera que escribiendo unas pocas líneas con la url de la base de datos, el usuario y la contraseña, se pueda realizar una conexión con esta.

La base de datos para este servicio ha sido implementada con las herramientas de Azure. Se ha creado una base de datos MySQL en Azure y se ha vinculado con el servicio, de manera que gracias a las anotaciones JPA, la base de datos, alojada en la nube, cree las tablas automáticamente y guarde valores al cargarlos. Se ha tomado esta decisión puesto que es la opción más profesional y habitual frente a la de tenerlo en local donde se despliegue el servicio.

3.3.1.1. Entidades

Tenemos las dos entidades principales del servicio anotadas con JPA, estas entidades son las que se almacenan y se obtienen de la base de datos. Gracias a las anotaciones en las mismas, este mapeo es automático, por un lado tenemos Dispositivo.java, que representa el elemento principal del servicio, que es el dispositivo con sus diferentes datos y características asociadas:

```

@Entity
public class Dispositivo {

    @Id
    private String id;
    private String urlImagen;
    private String nombre;
    private String marca;

    @Size(max = 750)
    private String descripcion;
    private Categoria categoria;
    private String precio;
    private int seguridad;
    private String sostenibilidad;

    @ManyToMany
    private List<Caracteristica> listaPositivaSeguridad;

    @ManyToMany
    private List<Caracteristica> listaNegativaSeguridad;

    @ManyToMany
    private List<Caracteristica> listaPositivaSostenibilidad;

    @ManyToMany
    private List<Caracteristica> listaNegativaSostenibilidad;
}

```

Imagen 10. Clase Dispositivo.java

Por otro lado, tenemos Característica.java, entidad que almacena diferentes características que influyen sobre la seguridad y sostenibilidad, tanto positiva como negativamente. Un ejemplo de característica que influye negativamente en la seguridad sería el uso de contraseñas predeterminadas débiles.

```

@Entity
public class Caracteristica {

    @Id
    @GeneratedValue
    private Long id;
    private String texto;
}

```

Imagen 11. Clase Caracteristica.java

3.3.1.2. Interfaces de repositorio

Tenemos los dos repositorios principales que sirven de comunicación entre la base de datos y el servicio, estos extienden de JpaRepository y se indica la clase de la que va a ser repositorio y el tipo de id. Por defecto heredan todos los métodos CRUD, pero pueden añadirse más si se cree necesario. Primero tenemos DispositivoRepository.java que representa el repositorio de los dispositivos.

```

public interface DispositivoRepository extends JpaRepository<Dispositivo, String> {
}

```

Imagen 12. Interfaz DispositivoRepository.java

También tenemos CaracteristicaRepository.java que representa el repositorio de las características.

```

public interface CaracteristicaRepository extends JpaRepository<Caracteristica, Long> {
}

```

Imagen 13. Interfaz CaracteristicaRepository.java

3.3.1.3. Enumerados

Pasamos con los enumerados, tenemos dos diferentes, uno es como se ha visto en la clase Dispositivo.java, para indicar la categoría a la que pertenece el dispositivo. De momento hay cinco categorías diferentes, pero al final del trabajo hablaremos sobre la posibilidad de ampliar esta cantidad. El enumerado tiene de nombre Categoria.java.

```
public enum Categoria {  
    Asistente_Virtual, Iluminacion, Climatizacion, Electrodomesticos_Inteligentes, Limpieza  
}
```

Imagen 14. Enumerado Categoria.java

El otro enumerado aunque no sea directamente de la capa de repositorio se ha añadido aquí y sirve para clasificar a la hora de ordenar, cuál de los tres métodos se quiere utilizar. El enumerado tiene de nombre TipoOrdenar.java.

```
public enum TipoOrdenar {  
    Alfabetico, Seguridad, Sostenibilidad  
}
```

Imagen 15. Enumerado TipoOrdenar.java

3.3.1.4. Listas

Finalmente nos encontramos con las dos listas, que aunque no se utilicen en el repositorio, al estar relacionadas directamente con las dos entidades se han colocado en esta capa. Estas listas sirven para devolver las listas de dispositivos y características de manera más ordenada y de manera que a la hora de mapear en la aplicación resultase más sencillo. Las clases son ListaDispositivos.java y ListaCaracteristicas.java.

```
public class ListaDispositivos {  
  
    private List<Dispositivo> dispositivos;
```

Imagen 16. Clase ListaDispositivos.java

```
public class ListaCaracteristicas {  
  
    private List<Caracteristica> caracteristicas;
```

Imagen 17. Clase ListaCaracteristicas.java

3.3.2. Implementación de la service layer

Esta capa está formada únicamente por una clase llamada GeneralService.java, esta clase es un servicio que sirve de puente entre la capa de control y la de repositorio, no es estrictamente necesaria y en lógicas sencillas se puede omitir, pero en este caso se ha implementado porque facilita la legibilidad y complejidad del código de la capa de control.

Con una anotación `@Service` se indica que es una clase de servicio que se conecta a uno o más repositorios. Los repositorios se obtienen con la anotación `@Autowired` y a partir de ahí se puede trabajar con normalidad llamando a los métodos de cada uno de los repositorios para realizar las operaciones pertinentes.

```
@Service
public class GeneralService {

    @Autowired
    private DispositivoRepository repositorioDispositivos;

    @Autowired
    private CaracteristicaRepository repositorioCaracteristicas;
```

Imagen 18. Clase GeneralService.java

En el caso de este servicio se han implementado siete métodos que realizan las siguientes funciones:

El método `dispositivos()` coge del repositorio de dispositivos todos los dispositivos y los devuelve.

```
public List<Dispositivo> dispositivos() {
    return repositorioDispositivos.findAll();
}
```

Imagen 19. Método dispositivos() de la clase GeneralService.java

El método `dispositivoPorId(String id)` coge del repositorio de dispositivos el dispositivo del que se pasa el id, si no lo encuentra, devuelve null.

```
public Dispositivo dispositivoPorId(String id) {
    Optional<Dispositivo> dispositivoOptional = repositorioDispositivos.findById(id);
    if (dispositivoOptional.isEmpty()) {
        return null;
    }
    return dispositivoOptional.get();
}
```

Imagen 20. Método dispositivoPorId(String id) de la clase GeneralService.java

El método `creaDispositivo(Dispositivo d)` recibe como parámetro un dispositivo, comprueba que no exista el id del dispositivo y lo crea, si ya existe, devuelve null.

```
public Dispositivo creaDispositivo(Dispositivo d) {
    Optional<Dispositivo> optional = repositorioDispositivos.findById(d.getId());
    if (!optional.isEmpty())
        return null;
    return repositorioDispositivos.saveAndFlush(d);
}
```

Imagen 21. Método creaDispositivo(Dispositivo d) de la clase GeneralService.java

El método `actualizaDispositivo(Dispositivo d)` recibe como parámetro un dispositivo, comprueba que exista y lo actualiza, en caso de no existir, devuelve null.

```

public Dispositivo actualizaDispositivo(Dispositivo d) {
    Optional<Dispositivo> optional = repositorioDispositivos.findById(d.getId());
    if (optional.isEmpty())
        return null;
    return repositorioDispositivos.saveAndFlush(d);
}

```

Imagen 22. Método actualizaDispositivo(Dispositivo d) de la clase GeneralService.java

El método características() coge del repositorio de características todas las características y las devuelve.

```

public List<Caracteristica> caracteristicas() {
    return repositorioCaracteristicas.findAll();
}

```

Imagen 23. Método características() de la clase GeneralService.java

El método característicaPorId(Long id) coge del repositorio de características la característica de la que se pasa el id, si no la encuentra, devuelve null.

```

public Caracteristica caracteristicaPorId(Long id) {
    Optional<Caracteristica> carcateristicaOptional = repositorioCaracteristicas.findById(id);
    if (carcateristicaOptional.isEmpty()) {
        return null;
    }
    return carcateristicaOptional.get();
}

```

Imagen 24. Método caracteristicaPorId(Long id) de la clase GeneralService.java

El método creaCaracteristica(Caracteristica d) recibe como parámetro una característica, comprueba que la característica no tenga id definido y la crea, si tiene id definido, devuelve null. Esto se debe a que el id de la característica es autogenerado, por lo que tiene que estar vacío.

```

public Caracteristica creaCaracteristica(Caracteristica d) {
    if (d.getId() != null)
        return null;
    return repositorioCaracteristicas.saveAndFlush(d);
}

```

Imagen 25. Método creaCaracteristica(Caracteristica d) de la clase GeneralService.java

3.3.3. Implementación de la controller layer

Esta capa está formada por un único controlador llamado GeneralController.java que es el encargado de hacer de RestController, por ello se anota la clase con @RestController indicando al servicio que esta clase es la encargada de recibir las peticiones y dependiendo del path hacer unos u otro métodos.

El path general para todas las direcciones en este controlador se añade con la anotación @RequestMapping y se ha utilizado como se puede ver en la Tabla 3 “REST_TFGMarioIngelmoDiana”.

En este controlador se obtienen con la anotación @Autowired tanto el servicio comentado en la capa anterior (GeneralService.java) como otras tres clases que se

explicarán en el apartado siguiente de seguridad. Por esto, el método getToken, aunque pertenezca al controlador, se mencionará y explicará en el siguiente punto.

```
@RestController
@RequestMapping("REST_TFGMarioIngelmoDiana")
public class GeneralController {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private UserDetailsServiceImpl usuarioDetailsService;

    @Autowired
    private GestionTokens gestion;

    @Autowired
    private GeneralService servicio;
```

Imagen 26. Clase GeneralController.java

Este controlador, para cumplir con lo especificado en la Tabla 3, dispone de siete métodos para cubrir todas las peticiones que se han definido. En este apartado se explicarán seis de ellas, dejando para después el POST de getToken():

El método getDispositivos(...) se encarga de gestionar las peticiones GET con el path “REST_TFGMarioIngelmoDiana/dispositivos” y conseguir todos los dispositivos del servicio, filtrando por categoría, seguridad y sostenibilidad si se indica y ordenando en caso de solicitarse. En caso de no haber dispositivos devuelve un NOT FOUND.

```
@GetMapping("/dispositivos")
public ResponseEntity<ListaDispositivos> getDispositivos(
    @RequestParam(value = "categoria", required = false) String categoria,
    @RequestParam(value = "seguridad", required = false) String seguridad,
    @RequestParam(value = "sostenibilidad", required = false) String sostenibilidad,
    @RequestParam(value = "ordenar", required = false) String ordenar) {
    List<Dispositivo> dispositivos = servicio.dispositivos();
    if (dispositivos.isEmpty()) {
        return ResponseEntity.notFound().build();
    }
    if (categoria != null && !categoria.equals("Todas")) {
        dispositivos = dispositivos.stream().filter(d -> d.getCategoria() == Categoria.valueOf(categoria))
            .collect(Collectors.toList());
    }
    if (seguridad != null && Integer.valueOf(seguridad) >= 0 && Integer.valueOf(seguridad) <= 100) {
        dispositivos = dispositivos.stream().filter(d -> d.getSeguridad() >= Integer.valueOf(seguridad))
            .collect(Collectors.toList());
    }
    if (sostenibilidad != null) {
        dispositivos = dispositivos.stream()
            .filter(d -> d.getSostenibilidad().matches("[A-" + sostenibilidad.toUpperCase() + "]"))
            .collect(Collectors.toList());
    }
    if (ordenar != null && (ordenar.equals("Alfabetico") || ordenar.equals("Seguridad") || ordenar.equals("Sostenibilidad"))) {
        switch (TipoOrdenar.valueOf(ordenar)) {
            case Alfabetico:
                dispositivos = dispositivos.stream().sorted(Comparator.comparing(Dispositivo::getNombre, String.CASE_INSENSITIVE_ORDER)).collect(Collectors.toList());
                break;
            case Seguridad:
                dispositivos = dispositivos.stream().sorted(Comparator.comparingInt(Dispositivo::getSeguridad).reversed()).collect(Collectors.toList());
                break;
            case Sostenibilidad:
                dispositivos = dispositivos.stream().sorted(Comparator.comparing(Dispositivo::getSostenibilidad)).collect(Collectors.toList());
                break;
            default:
                break;
        }
    }
    return ResponseEntity.ok(new ListaDispositivos(dispositivos));
}
```

Imagen 27. Método getDispositivos(...) de la clase GeneralController.java

El método getDispositivo(@PathVariable String id) se encarga de gestionar las peticiones GET con el path “REST_TFGMarioIngelmoDiana/dispositivos/{id}” y conseguir el dispositivo cuyo id se pasa en el path, devolviendo NOT FOUND si no existe ningún dispositivo con ese id.

```

@GetMapping("/dispositivos/{id}")
public ResponseEntity<ListaDispositivos> getDispositivo(@PathVariable String id) {
    Dispositivo d = servicio.dispositivoPorId(id);
    if (d == null) {
        return ResponseEntity.notFound().build();
    }
    List<Dispositivo> dispositivos = new LinkedList<Dispositivo>();
    dispositivos.add(d);
    return ResponseEntity.ok(new ListaDispositivos(dispositivos));
}

```

Imagen 28. Método getDispositivo(@PathVariable String id) de la clase GeneralController.java

De la misma manera que los dos métodos anteriores, tenemos lo mismo para las características, el método getCaracteristicas() se encarga de gestionar las peticiones GET con el path “REST_TFGMariIngelmoDiana/ caracteristicas” y el método getCaracteristica(@PathVariable String id) se encarga de gestionar las peticiones GET con el path “REST_TFGMariIngelmoDiana/caracteristicas/{id}”.

```

@GetMapping("/caracteristicas")
public ResponseEntity<ListaCaracteristicas> getCaracteristicas() {
    List<Caracteristica> caracteristicas = servicio.caracteristicas();
    if (caracteristicas.isEmpty()) {
        return ResponseEntity.notFound().build();
    }
    return ResponseEntity.ok(new ListaCaracteristicas(caracteristicas));
}

```

Imagen 29. Método getCaracteristicas() de la clase GeneralController.java

```

@GetMapping("/caracteristicas/{id}")
public ResponseEntity<Caracteristica> getCaracteristica(@PathVariable String id) {
    Caracteristica c = servicio.caracteristicaPorId(Long.valueOf(id));
    if (c == null) {
        return ResponseEntity.notFound().build();
    }
    return ResponseEntity.ok(c);
}

```

Imagen 30. Método getCaracteristica(@PathVariable String id) de la clase GeneralController.java

El método creaOReemplazaDispositivo(@PathVariable String id, @RequestBody Dispositivo d) se encarga de gestionar las peticiones PUT con el path “REST_TFGMariIngelmoDiana/dispositivos/{id}” y de crear o actualizar el dispositivo cuyo id se pasa en el path, en caso de que los ids no coincidan o que no se pueda crear o actualizar se devolverá un CONFLICT.

```

@GetMapping("/caracteristicas/{id}")
public ResponseEntity<Caracteristica> getCaracteristica(@PathVariable String id) {
    Caracteristica c = servicio.caracteristicaPorId(Long.valueOf(id));
    if (c == null) {
        return ResponseEntity.notFound().build();
    }
    return ResponseEntity.ok(c);
}

```

Imagen 31. Método creaOReemplazaDispositivo(@PathVariable String id, @RequestBody Dispositivo d) de la clase GeneralController.java

Y finalmente el método creaCaracteristica(@RequestBody Caracteristica c) se encarga de gestionar las peticiones POST con el path “REST_TFGMariIngelmoDiana/ caracteristicas” y de crear una nueva característica, en caso de que no se pueda crear, se devolverá un CONFLICT.


```

@PostMapping("/caracteristicas")
public ResponseEntity<Caracteristica> creaCaracteristica(@RequestBody Caracteristica c) {
    Caracteristica creado = servicio.creaCaracteristica(c);
    if (creado == null)
        return ResponseEntity.status(HttpStatus.CONFLICT).build();
    URI location = ServletUriComponentsBuilder.fromCurrentRequest().build().toUri();
    return ResponseEntity.created(location).body(creado);
}

```

Imagen 32. Método creaCaracteristica(@RequestBody Caracteristica c) de la clase GeneralController.java

3.3.4. Implementación de la seguridad y la configuración

Para implementar la seguridad se va a utilizar JSON Web Tokens (JWT), JWT es un estándar abierto que define un mecanismo para el intercambio seguro de información en forma de objetos JSON. La información es verificable y confiable porque está digitalmente firmada, además pueden ser firmados usando una llave pública o privada secreta [8]. Todo esto convierte a JWT en una forma muy completa y buena para implementar la seguridad en el servicio y poner un filtro de autorización en las funcionalidades deseadas.

Para ello se necesitan cubrir los siguientes pasos: 1. Verificar al usuario mediante sus credenciales. 2. Crear y devolver el token JWT. 3. Validar el token introducido por el usuario y aprobar la operación o denegarla.

Se han creado una serie de clases, con diferentes funcionalidades para cubrir esos pasos:

Primero se ha añadido en el controlador (GeneralController.java) un método getToken(@RequestBody Credenciales c) que se encarga de gestionar las llamadas POST al path "REST_TFGMarioIngelmoDiana/token", recoge las credenciales del usuario (usuario y clave), autentica al mismo, lo carga, genera el token y lo devuelve, estos pasos se explicarán a continuación.

```

@PostMapping("/token")
public ResponseEntity<String> getToken(@RequestBody Credenciales c) {
    if (c == null) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
    } else {
        authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(c.getUsuario(), c.getClave()));
        usuarioDetailsService.loadUserByUsername(c.getUsuario());
        String token = gestion.generaToken(c.getUsuario(), c.getClave());
        if (token == null) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
        } else {
            return ResponseEntity.ok(token);
        }
    }
}

```

Imagen 33. Método getToken(@RequestBody Credenciales c) de la clase GeneralController.java

Una vez las credenciales se han recogido, es necesario autenticar al usuario, esto se hace de manera automática utilizando el AuthenticationManager de la dependencia de seguridad de Spring. Después se carga el usuario, esto se hace con un @Service que implementa UserDetailsService, otra clase de la dependencia de seguridad de Spring.

Y finalmente se genera el token, para esto se ha creado un clase GestionTokens.java que es un @Service que sirve para generar y para validar un token. En este caso, se genera el token mediante el nombre y contraseña que había en las credenciales que el

usuario ha pasado en el método POST. Se comprueba que el nombre y contraseña coincidan y una vez verificado se firma la llave y se crea el token dándole una validez de 15 minutos. Una vez hecho esto, se devuelve al usuario.

```
public String generaToken(String nombre, String contra) {
    String token = null;
    if (nombre.equals(NOMBRE) && contra.equals(CONTRA)) {
        Key signingKey = new SecretKeySpec(DatatypeConverter.parseBase64Binary(SECRET_KEY),
            SignatureAlgorithm.HS256.getJcaName());
        keyGenerada = signingKey;
        Calendar cal = Calendar.getInstance();
        cal.add(Calendar.MINUTE, 15);
        Date date = cal.getTime();
        JwtBuilder builder = Jwts.builder().setSubject(NOMBRE).setExpiration(date).signWith(signingKey,
            SignatureAlgorithm.HS256);

        token = builder.compact();
    }
    return token;
}
```

Imagen 34. Método generaToken(String nombre, String contra) de la clase GestionTokens.java

Una vez el usuario obtiene su token, lo introduce en la siguiente petición que realice y se verificará si: 1. Su petición necesita autorización (las peticiones GET no las necesitan y el POST del token tampoco). 2. El token es válido.

Para comprobar todo lo mencionado hay que configurar el servicio de manera que lo personalizemos y ofrezca la funcionalidad que nosotros queremos, si este paso no es realizado, el servicio solicitará el token para cualquier petición a este.

Primero configuramos la seguridad web, para esto añadimos la clase WebSecurityConfig.java con la anotación @Configuration para que al lanzar el servicio, este la detecte como una configuración propia y la aplique. En esta clase, se crea un @Bean que se encargará de deshabilitar el csrf, que es el mecanismo de protección por defecto de Spring, para así permitir el uso de tokens JWT. También se encargará de autorizar determinadas llamadas sin autenticación y de aplicar una configuración personalizada añadiendo un filtro propio y así poder solicitar recursos restringidos por la seguridad. Esto último se hace modificando el cors, que es la política de seguridad de Spring y otro mecanismo de seguridad.

```
@Bean
SecurityFilterChain web(HttpSecurity http) throws Exception {
    http.csrf().disable()
        .authorizeHttpRequests((authorize) -> authorize.requestMatchers("/REST_TFGMarioIngelmoDiana/token")
            .permitAll().requestMatchers(HttpMethod.GET, "/*").permitAll()
            .requestMatchers(HttpMethod.POST, "/*").hasRole("ADMIN").requestMatchers(HttpMethod.PUT, "/*")
            .hasRole("ADMIN").requestMatchers(HttpMethod.DELETE, "/*").hasRole("ADMIN").anyRequest()
            .authenticated())
        .cors(withDefaults()).addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class)
        .sessionManagement((session) -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

    return http.build();
}
```

Imagen 35. Bean de la clase WebSecurityConfig.java para la configuración de la seguridad

El filtro propio (JwtRequestFilter.java) es el encargado de gestionar la obtención del token (lee la cabecera Authorization), también se encarga de saltar el proceso de autorización si la petición es GET o es el POST para conseguir el token y también valida el token una vez obtenido.

```

public boolean validaToken(String token) {
    try {
        Jwts.parserBuilder().setSigningKey(keyGenerada).build().parseClaimsJws(token);
        return true;
    } catch (JwtException e) {
        System.out.println("El token está mal");
    }
    return false;
}

```

Imagen 36. Método validaToken(String token) de la clase GestionTokens.java

Finalmente, para que las peticiones puedan hacerse desde cualquier dirección, hay que habilitar en el cors diferentes orígenes, métodos, etc. Esto se hace con la clase CorsConfig.java anotada con @Configuration y con un @Bean que se encarga de la configuración personalizada del cors.

```

@Bean
CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();
    configuration.setAllowedOrigins(Arrays.asList("*"));
    configuration.setAllowedMethods(Arrays.asList("*"));
    configuration.setAllowedHeaders(Arrays.asList("*"));
    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/*", configuration);
    return source;
}

```

Imagen 37. Bean de la clase CorsConfig.java para la configuración del cors

3.4. Pruebas del servicio

Se han realizado pruebas en Postman para probar cada una de las funcionalidades básicas del servicio, esto es, cada una de las peticiones que se pueden realizar al servicio, se han probado tanto casos de éxito a la hora de recuperar los dispositivos, como casos de error, por ejemplo que un id de un dispositivo no exista. También se ha probado que la seguridad con JWT funcione correctamente, se ha comprobado a generar el token con un usuario registrado y con uno sin registrar y se ha comprobado que en el segundo de los casos no genera el token y devuelve el fallo deseado. También se ha comprobado que el tiempo de validez del token es funcional (15 minutos) y que los métodos indicados, funcionan solo cuando se introduce un token válido.

3.5. Despliegue del servicio

Para el despliegue del servicio se ha utilizado Microsoft Azure, en mi cuenta de la universidad se ha creado un grupo de recursos donde también se ha añadido la base de datos que utiliza el servicio. Una vez terminado el servicio, este es empaquetado, primero se prueba su funcionamiento en local y después se despliega.

Para el despliegue se ha creado una máquina virtual Linux con Ubuntu 20.04, se ha creado una interfaz de red y se ha reservado una ip pública, asociándola a la máquina virtual, de manera que, se pueda acceder al servicio a través de esa ip y del puerto 8080. Una vez pasado a la máquina y ejecutado, el servicio queda desplegado en ese puerto, haciendo que siempre que la máquina virtual esté activa, el servicio esté disponible en la siguiente dirección: "http://51.137.100.222:8080/". Pudiendo así hacer

peticiones como el GET de los dispositivos en la siguiente dirección: “http://51.137.100.222:8080/REST_TFGMarioIngelmoDiana/dispositivos”.

Para agilizar el procedimiento a la hora de arrancar la máquina virtual y que no sea necesario acceder a esta y arrancarlo manualmente, se ha creado un servicio que se encarga de arrancar el servicio REST automáticamente al iniciar.

```
[Unit]
Description=Servicio para arrancar el jar
After=network.target

[Service]
ExecStart=/usr/bin/java -jar /home/azureuser/REST_TFGMarioIngelmoDiana.jar

[Install]
WantedBy=default.target
```

Imagen 38. Servicio en la máquina virtual para el despliegue automático del servicio REST

4. Desarrollo de la aplicación móvil

En este capítulo se describen las etapas de desarrollo de la aplicación Android: análisis de requisitos, arquitectura, diseño, implementación y pruebas.

4.1. Análisis de requisitos de la aplicación

Se van a analizar tanto la idea tras la aplicación, como los pasos iniciales que se dieron en cuanto al diseño de interfaces (mockups) y también los requisitos, tanto funcionales como no funcionales.

4.1.1. Idea tras la aplicación

La idea a la hora de desarrollar la aplicación es darle al usuario una manera de poder comprobar y comparar la seguridad y sostenibilidad de diferentes dispositivos IoT que pueda utilizar en su hogar o pueda estar interesado en adquirir.

Aunque hoy en día la seguridad y sostenibilidad son aspectos de gran importancia, también son aspectos a los que los usuarios finales no suelen prestar la atención suficiente. Esto puede ser por desconocimiento de los riesgos a los que está expuesto o las repercusiones de sostenibilidad, ahorro energético, etc. O simplemente por no disponer de información sobre la seguridad o sostenibilidad de sus dispositivos. Por ejemplo, cuando vas a comprar un dispositivo de este estilo siempre tienes las mismas características: color, tamaño, precio, almacenamiento, ram, consumo, etc. Pero pocas veces o ninguna encuentras características relacionadas con la seguridad y la sostenibilidad, por esto se ha decidido desarrollar la aplicación.

Para hacer la experiencia más cómoda y sencilla al usuario se han propuesto unos objetivos básicos que se explican de manera resumida:

- Disponer de filtros y de un buscador de dispositivos.

- Disponer de una sección donde guardar los dispositivos favoritos.
- Disponer de la posibilidad de escanear los códigos de barras de los productos y acceder a sus características.
- Disponer de una interfaz donde mostrar las características del dispositivo más en detalle.
- Disponer de una interfaz moderna y amigable con el usuario.

Para mostrar las ideas principales que se manejaban y cuál era la idea sobre el diseño se van a presentar unos mockups realizados a mano alzada para luego compararlos con el diseño final de la aplicación.

Primero tenemos el mockup de lo que iba a ser la sección principal de la aplicación, con el filtro y el buscador, que una vez buscado por nombre, mostrase una lista de resultados coincidentes tanto con el filtro como con el texto del buscador.

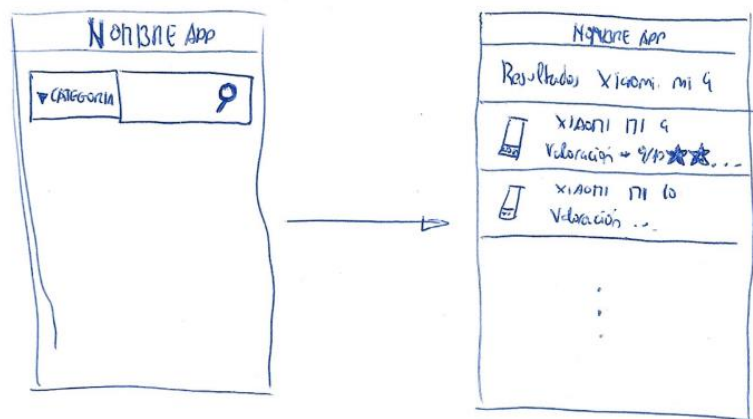


Imagen 39. Mockup de la idea sobre la sección principal

También tenemos el mockup sobre lo que sería la sección donde se mostrase un dispositivo específico en detalle al ser seleccionado.

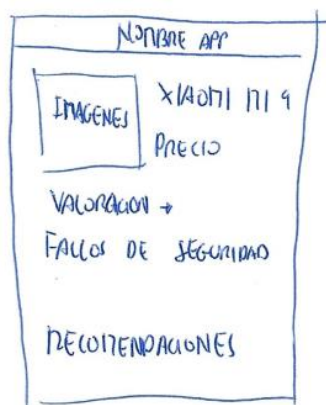


Imagen 40. Mockup de la idea sobre la sección del detalle de un dispositivo

Aunque la versión final difiere en ciertos aspectos de estos mockups, estos sirven para mostrar cuál era el enfoque inicial y poder comprobar cómo, durante el desarrollo de la aplicación, ciertos aspectos han ido cambiando, mientras que otros se han mantenido.

4.1.2. Diseño del logotipo

Para el diseño del logotipo de la aplicación se ha tomado como referencia una imagen del dios Hapi en blanco y negro, dios que como ya se mencionó anteriormente da nombre a esta aplicación. Y se ha añadido un candado sobre su mano que da la clave sobre la seguridad y alrededor una circunferencia con el nombre de la aplicación.



Imagen 41. Logotipo de la aplicación

4.1.3. Requisitos funcionales

A continuación se presenta una tabla con los diferentes requisitos funcionales identificados:

ID	Descripción
RF1	La aplicación dispondrá de menú lateral de navegación.
RF2	La aplicación dispondrá de una sección de buscador.
RF3	La aplicación dispondrá de una sección de escanear.
RF4	La aplicación dispondrá de una sección de favoritos.
RF5	La aplicación dispondrá de una sección de compartir.
RF6	La sección de buscador se abrirá por defecto al abrir la aplicación.
RF7	La sección de buscador mostrará una lista con todos los dispositivos ordenados por orden alfabético al abrirse.
RF8	La sección de buscador dispondrá de un botón de filtros donde establecer la categoría, la seguridad y sostenibilidad mínimas y la forma de ordenar la lista de dispositivos.
RF9	La sección de buscador dispondrá de un buscador de texto que filtrará los resultados en función al texto coincidente con el nombre o la marca de los dispositivos.
RF10	La lista resumen de dispositivos mostrará la imagen, nombre, marca, seguridad y sostenibilidad del dispositivo.
RF11	En la lista resumen de dispositivos se permitirá seleccionar uno para ver una vista de detalle del dispositivo.
RF12	La vista en detalle de un dispositivo permitirá añadir o eliminar un dispositivo de favoritos.
RF13	La vista en detalle de un dispositivo mostrará imagen, nombre, marca, categoría, precio, puntuación de seguridad, puntuación de sostenibilidad, descripción, lista con las características de seguridad positivas y negativas y

	lista con las características de sostenibilidad positivas y negativas del dispositivo.
RF14	La sección de escáner permitirá escanear códigos de barras y abrir la vista en detalle del dispositivo en cuestión.
RF15	La sección de favoritos mostrará una lista resumen con los dispositivos que se hayan añadido a favoritos.
RF16	La sección de compartir permitirá compartir, mediante diferentes medios, un mensaje para invitar al uso de la aplicación.

Tabla 4. Requisitos funcionales de la aplicación

4.1.4. Requisitos no funcionales

A continuación se presenta una tabla con los diferentes requisitos no funcionales identificados:

ID	Clasificación	Descripción
RNF1	Portabilidad	La aplicación podrá utilizarse en cualquier dispositivo Android con una versión igual o superior a la API 28 (Android Pie 9.0).
RNF2	Mantenibilidad	La aplicación estará modularizada de manera que cambios en una sección principal no afecten a otra.
RNF3	Usabilidad	La aplicación dispondrá de una interfaz amigable con el usuario, que facilite su uso.
RNF4	Rendimiento	El tamaño del archivo apk para instalar la aplicación no superará los 100 MB.
RNF5	Rendimiento	La aplicación deberá coexistir con el resto de las aplicaciones del sistema Android, de manera que no consuma más de 300MB de RAM y un 25% de la CPU permitiendo así, el uso de otras aplicaciones si esta está activa.

Tabla 5. Requisitos no funcionales de la aplicación

4.2. Diseño de la aplicación

En esta sección se describe la arquitectura utilizada en la aplicación así como otros detalles de diseño, para en secciones posteriores pasar a describir la implementación.

4.2.1. Diseño arquitectónico

Para el diseño de la aplicación se ha utilizado el patrón Modelo-Vista-Presentador (MVP), el cuál es aplicable al desarrollo de aplicaciones Android y reduce la complejidad al modularizar el código en [9]:

- **Modelo:** El modelo se encarga del acceso a los datos que se van a utilizar en la aplicación, ya sea una base de datos, una memoria caché o una API REST. En este caso, se encarga de la conexión con el servicio desarrollado para la obtención de los datos del dispositivo y también de gestionar la base de datos para los favoritos.
- **Vista:** La vista se encarga de la gestión de la interfaz de usuario, de lo que se le muestra y lo que puede hacer el usuario interactuando con el dispositivo.

Muestra las diferentes actividades y los diferentes fragmentos, con sus respectivos componentes.

- **Presentador:** El presentador se encarga de hacer de puente entre el modelo y la vista. Transmite al modelo las peticiones que el usuario realiza a través de la vista, ya sea obtener datos, almacenar en favoritos, etc, y se encarga de devolver a la vista esos datos que pide al modelo. Esto permite separar la interfaz de usuario de la lógica de la aplicación.

A continuación se presenta el diagrama de arquitectura de la aplicación y el diagrama de especificación de las interfaces.

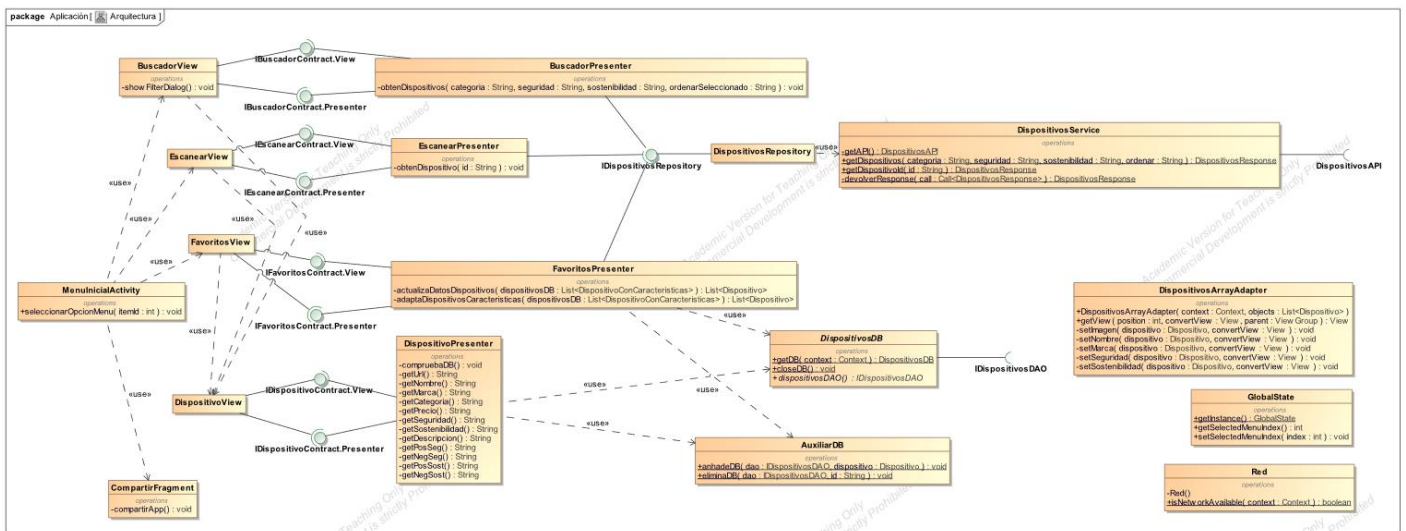


Imagen 42. Diagrama de arquitectura de la aplicación

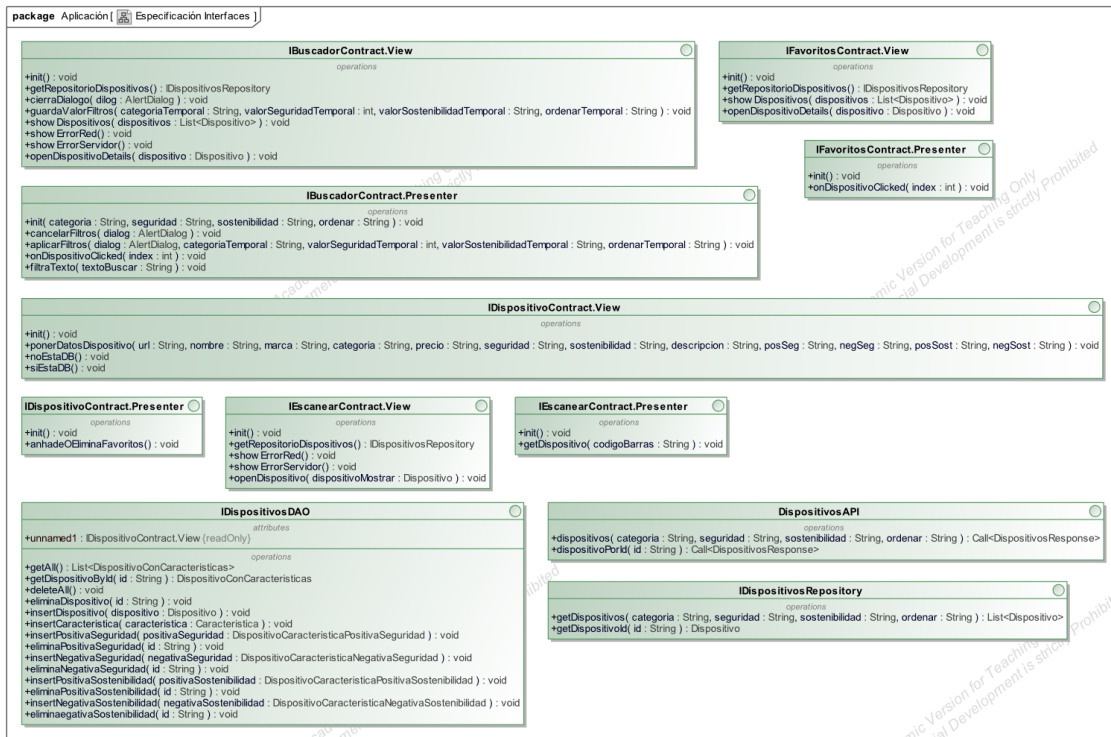


Imagen 43. Diagrama de especificación de las interfaces de la aplicación

4.2.1.1. Diseño del modelo

El modelo se ha diseñado de manera que contenga la conexión con el servicio, la gestión de la base de datos y el dominio de la aplicación. Para esto se ha diseñado un grupo de dos interfaces (IDispositivosRepository.java y DispositivosAPI.java) y dos clases (DispositivosRepository.java y DispositivosService.java) que se encargan de la conexión con el servicio mediante llamadas que se realizan desde los presentadores. También se ha diseñado un grupo de dos clases (DispositivosDB.java y AuxiliarDB.java) y una interfaz (IDispositivosDAO.java) que se encargan de gestionar la base de datos de la aplicación para almacenar los dispositivos favoritos. Y finalmente, el dominio, que se presenta a continuación, en el que están recogidas categorías de dispositivos, dispositivos concretos y características que influyen positiva o negativamente en la seguridad o sostenibilidad de dichos dispositivos.

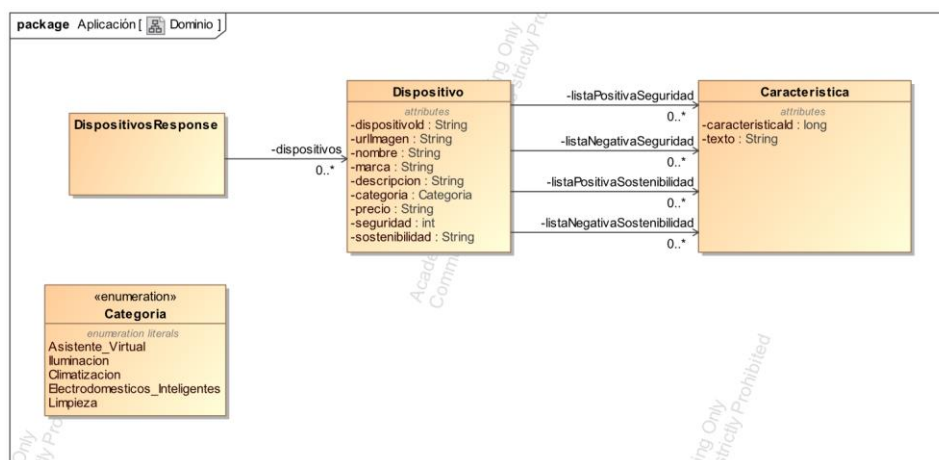


Imagen 44. Diagrama con el dominio de la aplicación

4.2.1.2. Diseño de la vista

La parte de interfaces de usuario se ha diseñado siguiendo la estrategia de actividades y fragmentos, los cuales proporcionan una mayor modularidad, flexibilidad y facilidad de uso en el desarrollo de aplicaciones, lo que contribuye a una mejor experiencia de usuario y a un código más limpio y mantenible.

Por un lado, una actividad corresponde con el menú inicial, que dispone de un menú lateral de navegación con las funcionalidades principales: buscador, escanear, favoritos y compartir. Cada una de ellas corresponde con un fragmento.

Por otro lado, otra actividad es la encargada de mostrar los datos de cada dispositivo en detalle.

Cada actividad y fragmento contará con su clase view y su clase presenter, excepto la sección de compartir, que al tener una lógica muy simple que no necesita de datos se ha implementado directamente, al igual que el menú inicial que gestiona la navegación y el despliegue de los diferentes fragmentos y tampoco necesita datos.

4.2.1.3. Diseño del presentador

La parte de los presentadores ha sido diseñada, como ya se ha comentado, de manera que hay cuatro presentadores. Cada uno se encarga de una funcionalidad determinada y hace de puente entre su vista y el modelo.

Los presentadores del buscador, de escanear y de favoritos se han diseñado de manera que hagan de puente entre su vista y el repositorio de llamadas al servicio, de manera que sus vistas puedan obtener los datos de los dispositivos del servicio. Además, el presentador de favoritos también tiene (al igual que el de detalle de dispositivo) acceso a la gestión de la base de datos, de manera que puedan hacer de puente entre su vista y la base de datos de la aplicación donde se almacenan los dispositivos favoritos.

4.3. Implementación de la aplicación

A raíz de la arquitectura generada, se va a explicar la implementación de la siguiente manera: primero se explicará cómo se ha implementado la conexión con el servicio REST desarrollado, seguido de la gestión de la base de datos interna para los favoritos. Después se explicará la implementación del menú inicial con el menú lateral y el dispositivo en detalle y, finalmente, se explicarán las secciones de buscador, escanear, favoritos y compartir.

4.3.1. Implementación de la conexión con el servicio REST

Para la conexión con el servicio y la obtención de los datos de los dispositivos se ha utilizado Retrofit. Retrofit es una biblioteca de Android desarrollada por Square que facilita la implementación de peticiones HTTP en las aplicaciones Android. Es muy popular y utilizada debido a su simplicidad y facilidad de uso [10]. Para la deserialización de los datos obtenidos del servicio se ha utilizado Gson, librería encargada de deserializar JSON convirtiendo los datos en las clases correspondientes, es decir dispositivos y características.

Para utilizar Retrofit se han creado dos interfaces y tres clases. Se ha creado la interfaz `IDispositivoRepository.java` y una clase que la implementa `DispositivosRepository.java`. Esta clase es la utilizada por los presenters desarrollados para hacer peticiones para obtener los dispositivos sin necesidad de gestionar las llamadas a la API que genera Retrofit. Dispone de dos métodos: `getDispositivos(String categoría, String seguridad, String sostenibilidad, String ordenar)` que devuelve una lista con los dispositivos filtrados y ordenados y `getDispositivoId(String id)` que devuelve el dispositivo cuyo id se ha pasado.

```

@Override
public List<Dispositivo> getDispositivos(String categoria, String seguridad, String sostenibilidad, String ordenar) {
    DispositivosResponse response = DispositivosService.getDispositivos(categoria, seguridad, sostenibilidad, ordenar);
    return response != null ? response.getDispositivos() : null;
}

Mario Ingelmo Diana
@Override
public Dispositivo getDispositivoId(String id) {
    DispositivosResponse response = DispositivosService.getDispositivoId(id);
    return response != null ? response.getDispositivos().get(0) : null;
}

```

Imagen 45. Implementación de los métodos getDispositivos y getDispositivoId en la clase DispositivosRepository.java

Estos métodos hacen uso de otra clase Retrofit, DispositivosService.java que se encarga de, utilizando como URL base la seleccionada en DispositivosServiceConstants.java (esta clase sirve para poder cambiar la url base dependiendo de si se están haciendo pruebas o no), crear una API con los métodos de la interfaz DispositivosAPI.java y así poder hacer las llamadas correspondientes al servicio, tanto para conseguir la lista de dispositivos como un solo dispositivo por id.

```

@GET("REST_TFGMarioIngelmoDiana/dispositivos")
Call<DispositivosResponse> dispositivos(@Query("categoria") String categoria, @Query("seguridad")
    String seguridad, @Query("sostenibilidad") String sostenibilidad, @Query("ordenar") String ordenar);

1 usage  Mario Ingelmo Diana
@GET("REST_TFGMarioIngelmoDiana/dispositivos/{id}")
Call<DispositivosResponse> dispositivoPorId(@Path("id") String id);

```

Imagen 46. Implementación de las llamadas en la interfaz DispositivosAPI.java

4.3.2. Implementación de la base de datos

Para la implementación de la base de datos interna de la aplicación para la gestión de los favoritos se ha utilizado Room. Room es una librería desarrollada por Android, que añade una capa de abstracción entre una base de datos SQLite y la aplicación, facilitando así el trabajo en esta.

Para esto se han anotado las diferentes entidades a almacenar con sus respectivas anotaciones y se han creado una serie de clases “especiales” para poder realizar el mapeado en la base de datos correctamente. Se ha tenido que almacenar el dispositivo como un dispositivo con características, donde se ha embebido el dispositivo sin las listas de características y se han añadido por separado, creando una unión mediante tablas intermedias (una para las características positivas de seguridad, otra para las negativas y lo mismo para la sostenibilidad) que almacenan el id del dispositivo y el id de la característica, permitiendo así almacenarlo y posteriormente obtenerlo, ya que, no se puede almacenar un dispositivo con las características directamente dentro de este. Resultando todo esto en un cambio del dominio para el almacenamiento en Room, como se muestra a continuación.

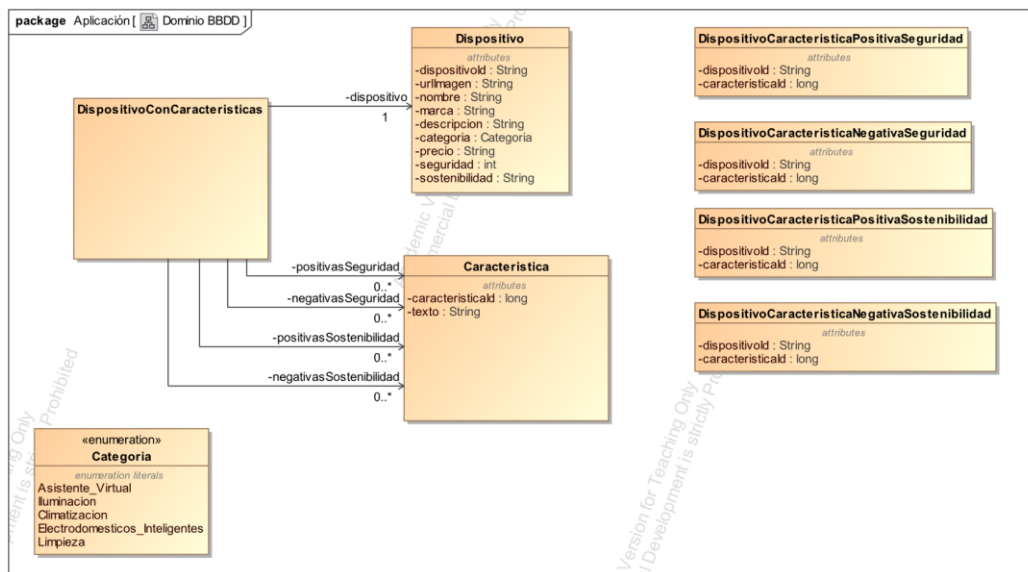


Imagen 47. Diagrama con el dominio de la aplicación para la BBDD

Para la gestión de la base de datos se ha creado una interfaz con las diferentes llamadas que se pueden realizar (conseguir todos los dispositivos, seleccionar un dispositivo por id, añadir una característica, etc.) llamada IDispositivosDAO.java y una clase abstracta DispositivosDB.java que es la propia base de datos anotada con @Database y que extiende de RoomDatabase y es la encargada de abrir y cerrar la base de datos y de devolver al resto de la aplicación IDispositivosDAO.java para su uso.

4.3.3. Implementación del menú inicial

Se ha creado una actividad inicial que se va a encargar de desplegar los diferentes fragmentos desarrollados para las secciones de buscador, escanear, favoritos y compartir. A esta actividad inicial se le ha añadido un menú lateral para navegar entre los diferentes fragmentos que contiene: el logo y nombre de la aplicación, el nombre del desarrollador y cuatro botones que indican mediante una imagen y un texto, que sección representan. Además, se resalta la sección en la que se está, que al iniciar la aplicación es el buscador.

La clase desarrollada para la gestión del menú es MenuInicialActivity.java que además de inicializar la actividad, también tiene un método que se encarga de gestionar el cambio entre fragmentos. Para la gestión del resaltado se cuenta con una clase llamada GlobalState.java que solo puede tener una instancia y almacena la posición del menú seleccionada.

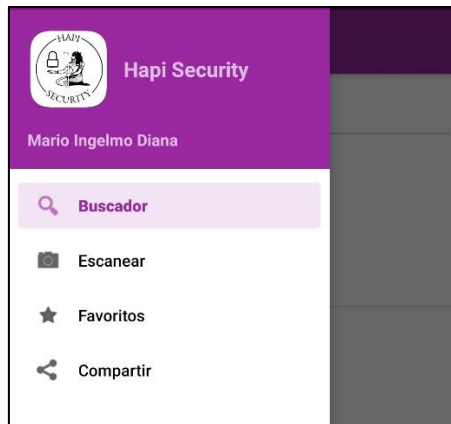


Imagen 48. Menú inicial y lateral de la aplicación

4.3.4. Implementación del dispositivo en detalle

Se ha creado una actividad para mostrar los datos de un dispositivo concreto. Esta actividad es lanzada desde las secciones de buscador, escanear y favoritos y muestra los datos del dispositivo que se selecciona o en el caso del escáner, que se escanea su código de barras.

Esta actividad cuenta tanto con los datos del dispositivo que se muestran (imagen, nombre, marca, categoría, precio, puntuación de seguridad, puntuación de sostenibilidad, descripción y listas con los aspectos positivos y negativos de seguridad y sostenibilidad) como con una imagen de una estrella que está o no marcada dependiendo de si el dispositivo está añadido en la lista de favoritos y que al pulsar sobre ella se puede añadir o eliminar de favoritos.

Para que esto sea posible se ha creado una vista y un presentador (DispositivoView.java y DispositivoPresenter.java). La vista se encarga de inicializar la actividad y de recibir el dispositivo a mostrar y se pasa al presentador. El presentador se encarga de obtener todos los datos necesarios del dispositivo que se ha pasado y mandárselos a la vista para que los plasme en la interfaz. También se encarga de gestionar si está o no el dispositivo en favoritos y en caso de que se pulse sobre la estrella, la vista avisa al presentador y este se encarga de añadir o eliminar de favoritos el dispositivo. Esto se hace a través de una clase auxiliar llamada AuxiliarDB.java que se encarga de añadir o eliminar de la base de datos. A continuación, se muestran varias imágenes de la interfaz.


← Dispositivo	← Dispositivo	← Dispositivo
<div data-bbox="304 412 400 472"></div> <div data-bbox="515 371 552 409">☆</div> <p>Nombre: Amazon Echo</p> <p>Marca: Amazon</p> <p>Categoría: Asistente Virtual</p> <p>Precio: 30 €</p> <p>Seguridad: 70 / 100</p> <p>Sostenibilidad: B</p> <p>Descripción: El Echo Dot es un altavoz inteligente que se controla con la voz y que usa el Alexa Voice Service. Gracias a su diseño, es ideal para cualquier habitación. Simplemente pídele música, las noticias o información. También puedes llamar a cualquiera que tenga un dispositivo Echo, la app Alexa o Skype, así como controlar dispositivos de Hogar digital con la voz.</p>	<p>Lista Características Seguridad:</p> <p>Positivas:</p> <ul style="list-style-type: none"> - Autenticación fuerte de usuario: Requerir una autenticación sólida, como la autenticación de dos factores, para acceder al dispositivo. - Cifrado de datos: La encriptación de datos transmitidos y almacenados, lo que dificulta su acceso no autorizado. - Notificaciones de seguridad en tiempo real: Alertas en tiempo real de cualquier actividad sospechosa o comportamiento anormal del dispositivo. - Detección de intrusiones: La capacidad de detectar y responder a intentos de intrusión y ataques maliciosos. - Compatibilidad con protocolos de seguridad estándar: Soporte de protocolos de seguridad estándar, como SSL/TLS y AES, para proteger la comunicación de red. <p>Negativas:</p> <ul style="list-style-type: none"> - Vulnerabilidades conocidas no corregidas: No solucionar vulnerabilidades de seguridad conocidas 	<p>Lista Características Sostenibilidad:</p> <p>Positivas:</p> <ul style="list-style-type: none"> - Eficiencia energética: Los dispositivos IoT que son energéticamente eficientes pueden ayudar a reducir el consumo de energía y, por lo tanto, reducir la huella de carbono. - Diseño modular: Los dispositivos IoT que están diseñados con módulos intercambiables y actualizables pueden extender su vida útil y reducir la cantidad de residuos electrónicos. - Uso de materiales sostenibles: Los dispositivos IoT que utilizan materiales sostenibles, como plásticos reciclados o materiales de origen biológico, pueden reducir su impacto ambiental. - Tecnologías de bajo consumo: Las tecnologías de bajo consumo, como Bluetooth de baja energía (BLE) o Zigbee, pueden ayudar a reducir el consumo de energía de los dispositivos IoT. - Ciclo de vida prolongado: Los dispositivos IoT que están diseñados para durar más tiempo pueden reducir la cantidad de residuos electrónicos y la necesidad de fabricar nuevos dispositivos

Imagen 49. Interfaz del dispositivo Amazon Echo en detalle



Imagen 50. Interfaz del dispositivo Amazon Echo en detalle (añadido a favoritos)

4.3.5. Implementación de la sección de buscador

Es el fragmento inicial de la aplicación y está compuesto por su vista (BuscadorView.java) y su presentador (BuscadorPresenter.java). Estas dos clases son las encargadas de inicializar el fragmento, cargar la lista con los dispositivos obtenidos desde el servicio y dar funcionalidad a los filtros y al buscador como se explica a continuación.

De base, nada más arrancar el fragmento, desde el presentador se hace una llamada para obtener los dispositivos. Puesto que no se sabe si el usuario busca filtrarlos o quiere la lista completa, se obtienen los datos de todos los dispositivos sin restricción alguna y ordenados alfabéticamente. Esto significa que la categoría indicada es "Todas", la seguridad es "0" y la sostenibilidad es "G".

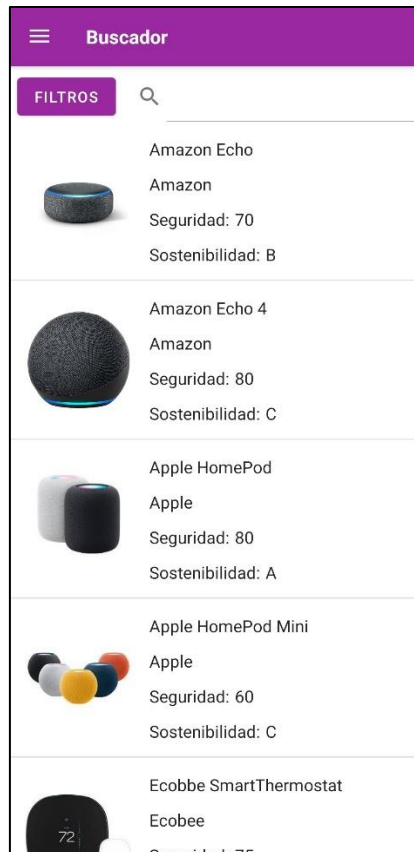


Imagen 51. Interfaz del fragmento buscador al iniciar

Una vez iniciado el fragmento podemos observar los diferentes dispositivos que se nos muestran deslizando por la lista, donde encontramos el nombre, marca, puntuación de seguridad y puntuación de sostenibilidad y también, podemos pulsar sobre ellos y que se nos abra la vista en detalle. Además, disponemos del buscador y de un botón de filtros.

El buscador es un `SearchView` que se encarga de leer el texto que el usuario introduce. Este texto es enviado al presentador, que mediante su método `filtraTexto(String textoBuscar)` se encarga de buscar coincidencias entre el nombre y marca de los dispositivos, actualizando la lista con los dispositivos coincidentes. El texto se puede borrar de golpe o ir borrando poco a poco y el buscador siempre irá actualizando la lista. En caso de que no haya ninguna coincidencia, se mostrará un mensaje indicando que no hay dispositivos con las características indicadas.

```
public void filtraTexto(String textoBuscar) {
    if (textoBuscar.isBlank()) {
        obtenerDispositivos(categoriaSeleccionada, String.valueOf(valorSeguridad), valorSostenibilidad, ordenarSeleccionado);
    }
    if (dispositivosOriginales != null && !dispositivosOriginales.isEmpty()) {
        List<Dispositivo> dispositivosTexto = new LinkedList<>();
        for (Dispositivo d : dispositivosOriginales) {
            if (d.getNombre().toLowerCase(Locale.ROOT).contains(textoBuscar.toLowerCase(Locale.ROOT)) || d.getMarca().toLowerCase(Locale.ROOT).contains(textoBuscar.toLowerCase(Locale.ROOT))) {
                dispositivosTexto.add(d);
            }
        }
        dispositivosMostrados = dispositivosTexto;
        view.showDispositivos(dispositivosTexto);
    }
}
```

Imagen 52. Método `filtraTexto` de la clase `BuscadorPresenter.java`



Imagen 53. Interfaz del fragmento buscador al buscar “apple”

Pasando con los filtros, encontramos que al pulsar el botón de “Filtros” se despliega un AlertDialog compuesto por cuatro secciones: un Spinner donde seleccionar entre un conjunto de opciones la categoría, una SeekBar donde deslizando se selecciona el valor de seguridad deseado (0-100), otra SeekBar donde realizar la misma opción con la sostenibilidad (A-G) y un último Spinner donde seleccionar el ordenamiento que se quiere (Alfabético, Seguridad o Sostenibilidad).

Dependiendo de los valores seleccionados, la vista mandará estos al presentador que se encargará de realizar una llamada al servicio para obtener la lista correspondiente en el orden correspondiente. Si los filtros han sido muy restrictivos y no hay dispositivos con esas características se indicará con un mensaje.

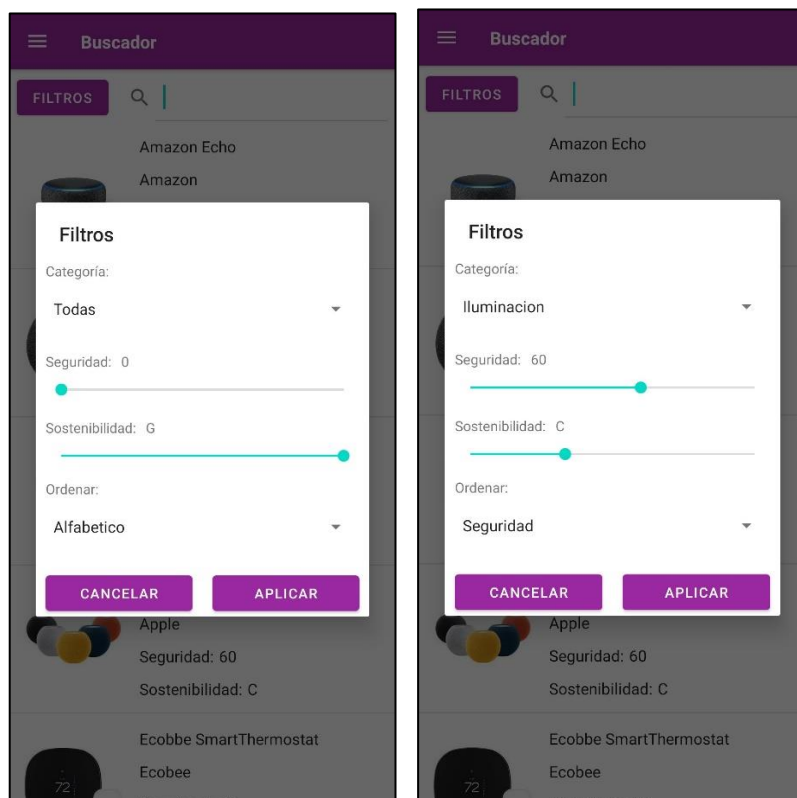


Imagen 54. Interfaz de los filtros del buscador base y modificados

Y a continuación se muestra el resultado de utilizar el filtro anterior y de utilizar un filtro excesivamente restrictivo:

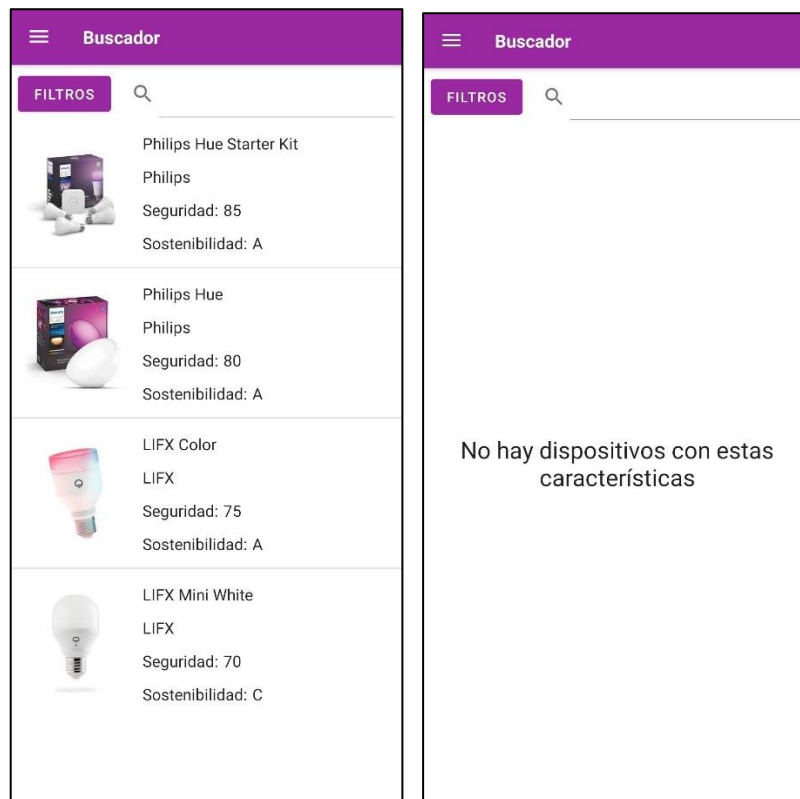


Imagen 55. Interfaz del fragmento buscador al filtrar y al filtrar con demasiada restricción

4.3.6. Implementación de la sección de escanear

Se ha creado un fragmento que se encarga de mostrar un lector de código de barras y es gestionado por una vista y un presentador (EscanearView.java y EscanearPresenter.java). La vista es la encargada de inicializar el fragmento y poner al lector de código de barras a leer. Este lector de código de barras ha sido implementado con la librería ZXing que implementa un lector para aplicaciones Android.

Cada vez que se detecta un código de barras se pasa el código al presentador, que se encarga de intentar conseguir el dispositivo a través de la clase IDispositivoRepository.java (explicada anteriormente) utilizando como id el código leído. Si el código corresponde con el id de algún dispositivo, este se obtiene del servicio y se abre la vista en detalle. Si no coincide, se muestra un mensaje de error y se reanuda la lectura del lector. A continuación se muestra la interfaz.



Imagen 56. Interfaz del fragmento de escanear

4.3.7. Implementación de la sección de favoritos

Para la implementación de los favoritos se ha creado un fragmento que muestra una lista con los dispositivos que se han añadido a favoritos previamente. Está formada por su vista (`FavoritosView.java`) que se encarga de inicializar el fragmento y mostrar la lista de dispositivos y por su presentador (`FavoritosPresenter.java`) que se encarga de obtener los dispositivos de la base de datos de la aplicación.

En el proceso de obtener los dispositivos de la base de datos y si se dispone de red, el presentador intenta actualizar los datos de los dispositivos almacenados con los dispositivos del servicio buscándolos por su id. De esta manera se asegura, que los datos están actualizados a la última versión y que si los dispositivos reciben actualizaciones y sus valores cambian, el usuario pueda observarlos actualizados aun teniéndolos en favoritos.

En caso de que no haya dispositivos favoritos agregados, aparecerá un mensaje indicándoselo al usuario.

```

/**
 * Metodo para actualizar los dispositivos de la base de datos que se encuentren en el repositorio
 * @param dispositivosDB la lista de dispositivos en la base de datos
 * @return lista con los dispositivos actualizados y convertidos para mostrar
 */
usage  ▲ Mario Ingelmo Diana
private List<Dispositivo> actualizaDatosDispositivos(List<DispositivoConCaracteristicas> dispositivosDB) {
    List<Dispositivo> listaDevolver = new LinkedList<>();
    List<DispositivoConCaracteristicas> listaNoEstanRepositorio = new LinkedList<>();

    for (DispositivoConCaracteristicas d : dispositivosDB) {
        Dispositivo dispositivo = repositorioDispositivos.getDispositivoId(d.getDispositivo().getDispositivoId());
        // Compruebo si se ha podido traer el dispositivo actualizado del repositorio,
        // sino se añade a otra lista para añadirlo sin actualizar posteriormente
        if (dispositivo != null) {
            AuxiliarDB.eliminaDB(dao, dispositivo.getDispositivoId());
            AuxiliarDB.anhadeDB(dao, dispositivo);
            listaDevolver.add(dispositivo);
        } else {
            listaNoEstanRepositorio.add(d);
        }
    }
    // Si algún dispositivo no ha podido actualizarse, lo convierto y lo añado a la lista a mostrar
    if (!listaNoEstanRepositorio.isEmpty()) {
        List<Dispositivo> listaAnhadirADevolver = adaptaDispositivosCaracteristicas(listaNoEstanRepositorio);
        listaDevolver.addAll(listaAnhadirADevolver);
    }

    return listaDevolver;
}

```

Imagen 57. Método actualizaDatosDispositivos de la clase FavoritosPresenter.java encargado de actualizar los datos de los dispositivos favoritos

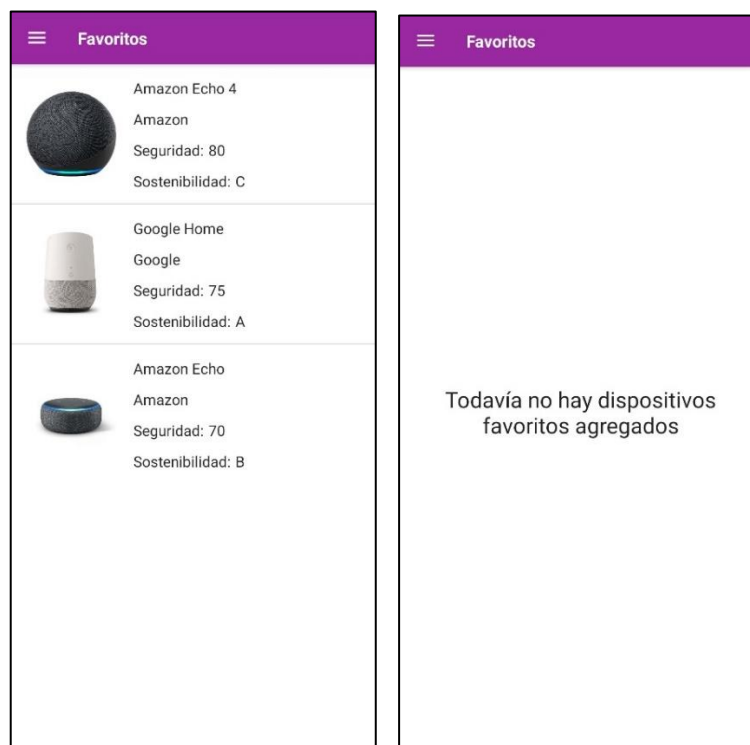


Imagen 58. Interfaz del fragmento favoritos con dispositivos agregados y sin agregar

4.3.8. Implementación de la sección de compartir

Para implementar esta sección se ha creado un fragmento que se gestiona con la clase CompartirFragment.java. Esta clase es la encargada de mostrar el texto que aparece en el fragmento y dar funcionalidad al botón de compartir, que al pulsar sobre este inicia un Intent donde mandar un mensaje a través de diferentes medios.

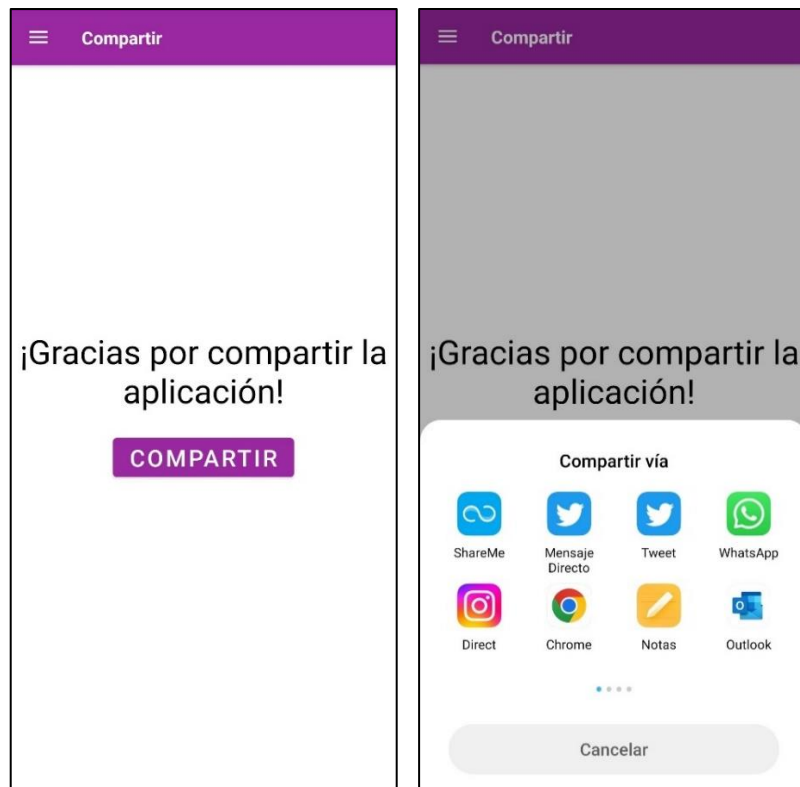


Imagen 59. Interfaz del fragmento de compartir

4.4. Pruebas de la aplicación

En esta sección se analizan los diferentes tipos de pruebas que se han realizado (unitarias, integración e interfaz, sistema y aceptación) y también un análisis de la calidad del código.

4.4.1. Pruebas unitarias

Las pruebas unitarias sirven para probar diferentes componentes de manera aislada, de tal forma que se compruebe su funcionalidad independientemente del resto de la aplicación. Utilizando Junit y Robolectric, se han probado diferentes funcionalidades de la aplicación.

Se han probado diferentes funcionalidades para comprobar que: un dispositivo se cree bien y sus datos también. Que la lista de favoritos se inicie correctamente sin dispositivos favoritos añadidos, con uno añadido y con varios y también que el dispositivo se actualice correctamente. Y también que el dispositivo en detalle inicialice correctamente usando un dispositivo sin datos y otro con datos. A continuación se muestra una prueba unitaria donde se comprueba que se inicializa el presentador del dispositivo en detalle correctamente al pasarle un dispositivo con datos.

```

/**
 * Test donde se prueba a inicializar el presenter con un dispositivo con datos
 */
@Mario Ingelmo Diana
@Test
public void initDispositivoPresenterTest() {
    when(db.dispositivosDAO()).thenReturn(dao);

    when(c1.getTexto()).thenReturn( value: "Característica 1");
    when(c2.getTexto()).thenReturn( value: "Característica 2");
    when(c3.getTexto()).thenReturn( value: "Característica 3");
    when(c4.getTexto()).thenReturn( value: "Característica 4");

    LinkedList<Caracteristica> lista1 = new LinkedList<>();
    lista1.add(c1);
    LinkedList<Caracteristica> lista2 = new LinkedList<>();
    lista2.add(c2);
    LinkedList<Caracteristica> lista3 = new LinkedList<>();
    lista3.add(c3);
    LinkedList<Caracteristica> lista4 = new LinkedList<>();
    lista4.add(c4);

    when(d2.getDispositivoId()).thenReturn( value: "54326534");
    when(d2.getUrlImagen()).thenReturn( value: "http://www.aaa.es");
    when(d2.getNombre()).thenReturn( value: "Dispositivo Test");
    when(d2.getMarca()).thenReturn( value: "Test");
    when(d2.getDescripcion()).thenReturn( value: "Descripcion Test");
    when(d2.getCategoria()).thenReturn(Categoria.Climatizacion);

    when(d2.getPrecio()).thenReturn( value: "100");
    when(d2.getSeguridad()).thenReturn( value: 90);
    when(d2.getSostenibilidad()).thenReturn( value: "A");
    when(d2.getListPositivaSeguridad()).thenReturn(lista1);
    when(d2.getListNegativaSeguridad()).thenReturn(lista2);
    when(d2.getListPositivaSostenibilidad()).thenReturn(lista3);
    when(d2.getListNegativaSostenibilidad()).thenReturn(lista4);

    presenter = new DispositivoPresenter(view, d2, db);
    presenter.init();

    verify(d2, times( wantedNumberOfInvocations: 2)).getUrlImagen();
    verify(d2, times( wantedNumberOfInvocations: 2)).getNombre();
    verify(d2, times( wantedNumberOfInvocations: 2)).getMarca();
    verify(d2, times( wantedNumberOfInvocations: 2)).getDescripcion();
    verify(d2, times( wantedNumberOfInvocations: 2)).getCategoria();
    verify(d2, times( wantedNumberOfInvocations: 2)).getPrecio();
    verify(d2, times( wantedNumberOfInvocations: 2)).getSeguridad();
    verify(d2, times( wantedNumberOfInvocations: 2)).getSostenibilidad();
    verify(d2, times( wantedNumberOfInvocations: 2)).getListPositivaSeguridad();
    verify(d2, times( wantedNumberOfInvocations: 2)).getListNegativaSeguridad();
    verify(d2, times( wantedNumberOfInvocations: 2)).getListPositivaSostenibilidad();
    verify(d2, times( wantedNumberOfInvocations: 2)).getListNegativaSostenibilidad();

    verify(view, times( wantedNumberOfInvocations: 1)).ponerDatosDispositivo( url: "http://www.aaa.es", nombre: "Dispositivo Test",
    marca: "Test", categoria: "Climatizacion", precio: "100 €", seguridad: "90 / 100", sostenibilidad: "A", descripcion: "Descripcion Test",
    posSeg: "\t- Característica 1\n", negSeg: "\t- Característica 2\n", posSort: "\t- Característica 3\n",
    negSort: "\t- Característica 4\n");
}

```

Imagen 60. Test unitario del dispositivo en detalle

4.4.2. Pruebas de integración e interfaz

Después de realizar pruebas unitarias y probar el funcionamiento de cada componente de manera aislada, se realizan pruebas de integración. Estas consisten en probar en conjunto, el funcionamiento de varios componentes.

Se han realizado pruebas de diferentes conjuntos de componentes para comprobar que: la inicialización de la lista del buscador obteniendo los datos de un repositorio estático es correcta, que el funcionamiento del buscador al obtener los dispositivos funciona o que la inicialización de la lista de favoritos una vez se han añadido dispositivos a la base de datos también es correcta. A continuación se muestra una prueba de integración donde se comprueba el funcionamiento del buscador con el resto de los componentes.

```

/**
 * Test donde se comprueba que el buscador funciona y la lista se reduce
 */
* Mario Ingelmo Diana
@Test
public void buscadorFiltrarListaTest() {
    ArgumentCaptor<List<Dispositivo>> lista = ArgumentCaptor.forClass(List.class);
    presenter = new BuscadorPresenter(view, categoria: "Todas", seguridad: "G", sostenibilidad: "G", ordenar: "Alfabetico", red: true);
    verify(view, times( wantedNumberOfInvocations: 1)).showDispositivos(lista.capture());
    assertEquals( expected: 21, lista.getValue().size());

    presenter.filtrarTexto( textoBuscar: "Apple");
    verify(view, times( wantedNumberOfInvocations: 2)).showDispositivos(lista.capture());
    assertEquals( expected: 2, lista.getValue().size());

    Dispositivo d1 = lista.getValue().get(0);
    assertEquals( expected: "Apple HomePod", d1.getNombre());
    assertEquals( expected: "Apple", d1.getMarca());
    assertEquals( expected: 80, d1.getSeguridad());
    assertEquals( expected: "A", d1.getSostenibilidad());

    Dispositivo d2 = lista.getValue().get(1);
    assertEquals( expected: "Apple HomePod Mini", d2.getNombre());
    assertEquals( expected: "Apple", d2.getMarca());
    assertEquals( expected: 60, d2.getSeguridad());
    assertEquals( expected: "C", d2.getSostenibilidad());
}

```

Imagen 61. Test de integración del buscador

Respecto a las pruebas de interfaz, en aplicaciones Android estas se realizan para comprobar el correcto funcionamiento de la aplicación en un dispositivo emulado. Para ello se utiliza la librería Espresso que ayuda a emular las diferentes acciones que un usuario podría realizar, como por ejemplo deslizar, pinchar, escribir, etc. Además se hace uso de un repositorio estático, de manera que los datos que se obtengan siempre sean los mismos y las comprobaciones sean fiables.

Se han realizado pruebas de interfaz de diferentes actividades y fragmentos de la aplicación, probando la funcionalidad general de esta, se ha probado la navegación con el menú lateral, el uso de la sección de buscador con el uso del buscador, el uso de la sección de favoritos, añadiendo y eliminando dispositivos de la base de datos y también el uso del dispositivo en detalle, entre otros. A continuación se muestra una prueba de interfaz donde se escribe en el buscador, se comprueba que se actualiza la lista, se borra el texto y se comprueba que la lista vuelve a su estado original.

```

@Test
public void pruebaBuscadorTest() {
    // Compruebo que se ha abierto el fragment buscador y que la lista tiene los dispositivos
    onView(withId(R.id.lvDispositivos)).check(matches(hasElements()));
    onData(anything()).inAdapterView(withId(R.id.lvDispositivos))
        .atPosition(0).onChildView(withId(R.id.tvName))
        .check(matches(withText("Amazon Echo"))));
    // Escribo apple en el buscador y compruebo que los dispositivos que quedan son los que deberían
    onView(allOf(instanceOf(SearchView.class), withId(R.id.svBuscador)))
        .perform(typeText( stringToBeTyped: "Apple")).perform(pressKey(KeyEvent.KEYCODE_ENTER));
    // Compruebo que la lista tiene los 2 dispositivos de apple
    onView(withId(R.id.lvDispositivos)).check(matches(hasElements()));
    onData(anything()).inAdapterView(withId(R.id.lvDispositivos))
        .atPosition(0).onChildView(withId(R.id.tvName))
        .check(matches(withText("Apple HomePod"))));
    onData(anything()).inAdapterView(withId(R.id.lvDispositivos))
        .atPosition(1).onChildView(withId(R.id.tvName))
        .check(matches(withText("Apple HomePod Mini"))));
    // Borro el buscador y compruebo que los elementos se resetean a todos
    onView(allOf(instanceOf(SearchView.class), withId(R.id.svBuscador)))
        .perform(pressKey(KeyEvent.KEYCODE_DEL)).perform(pressKey(KeyEvent.KEYCODE_DEL))
        .perform(pressKey(KeyEvent.KEYCODE_DEL)).perform(pressKey(KeyEvent.KEYCODE_DEL))
        .perform(pressKey(KeyEvent.KEYCODE_DEL)).perform(pressKey(KeyEvent.KEYCODE_ENTER));
    // Compruebo que la lista tiene los dispositivos y el primero coincide otra vez
    onView(withId(R.id.lvDispositivos)).check(matches(hasElements()));
    onData(anything()).inAdapterView(withId(R.id.lvDispositivos))
        .atPosition(0).onChildView(withId(R.id.tvName))
        .check(matches(withText("Amazon Echo"))));
}

```

Imagen 62. Test de interfaz del buscador

4.4.3. Pruebas de sistema

A la hora de realizar este tipo de pruebas se busca comprobar el funcionamiento del sistema, dando especial importancia al cumplimiento de los requisitos no funcionales. Para ello, se han realizado diferentes comprobaciones que muestren que los requisitos no funcionales se han cumplido.

Para cumplir con el RNF1 (Portabilidad) se ha indicado en Android Studio que la aplicación se desarrolla con la API 28 o superior, esto hace que la aplicación se pueda correr en aproximadamente el 84% de los dispositivos Android.

El RNF2 (Mantenibilidad) trata sobre la modularidad de la aplicación y eso se ha cumplido al aplicar el patrón MVP en el desarrollo de esta. El RNF3 (Usabilidad) que trata sobre el fácil uso de la aplicación se ha cumplido al seguir las recomendación de Android de diseño y al utilizar componente comunes que los usuarios conocen.

Y pasamos finalmente con los requisitos no funcionales de rendimiento. El RNF4 que trata sobre el tamaño del APK se ha cumplido, al pesar este archivo 15MB, siendo bastante inferior al límite marcado de 100MB. Y el RNF5 sobre el consumo de RAM y CPU de la aplicación también se ha cumplido, para comprobar esto se ha utilizado el emulador, con un dispositivo bastante sencillo, un pixel 2 con solamente 256MB de RAM y una CPU poco potente. Los resultados obtenidos con el profiler de Android Studio, herramienta propia que sirve para analizar el rendimiento de un dispositivo son muy buenos. Nos encontramos con un consumo de RAM estable de 65MB y con un consumo de CPU muy bajo como se mostrará a continuación, reduciéndose además al 1% cuando la aplicación está en segundo plano.

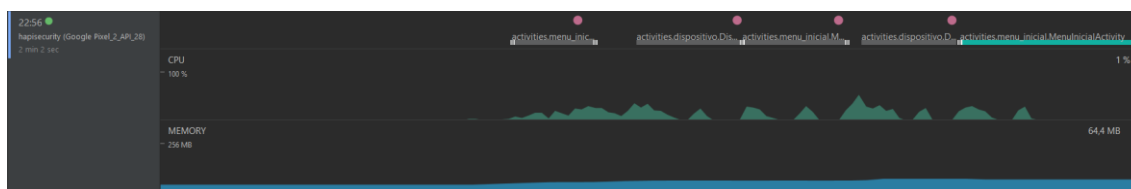


Imagen 63. Gráficas del profiler de Android Studio

4.4.4. Pruebas de aceptación

Para la realización de las pruebas de aceptación se ha distribuido la aplicación a diferentes usuarios con diferentes dispositivos Android y se les ha pedido que utilicen la aplicación durante unos días.

Estos usuarios no han reportado fallos durante su uso a lo largo de los días y al terminar ese periodo han mostrado su contento respecto a la facilidad de uso de la aplicación. Cabe destacar lo interesante que sería realizar este pequeño estudio con un conjunto de usuarios mayor y más variado.

4.4.5. Análisis de la calidad

Para analizar la calidad del código desarrollado se ha utilizado la herramienta SonarCloud, que es un servicio de análisis de la calidad de código en la nube de la empresa SonarQube. El resultado obtenido ha sido satisfactorio y es el siguiente.

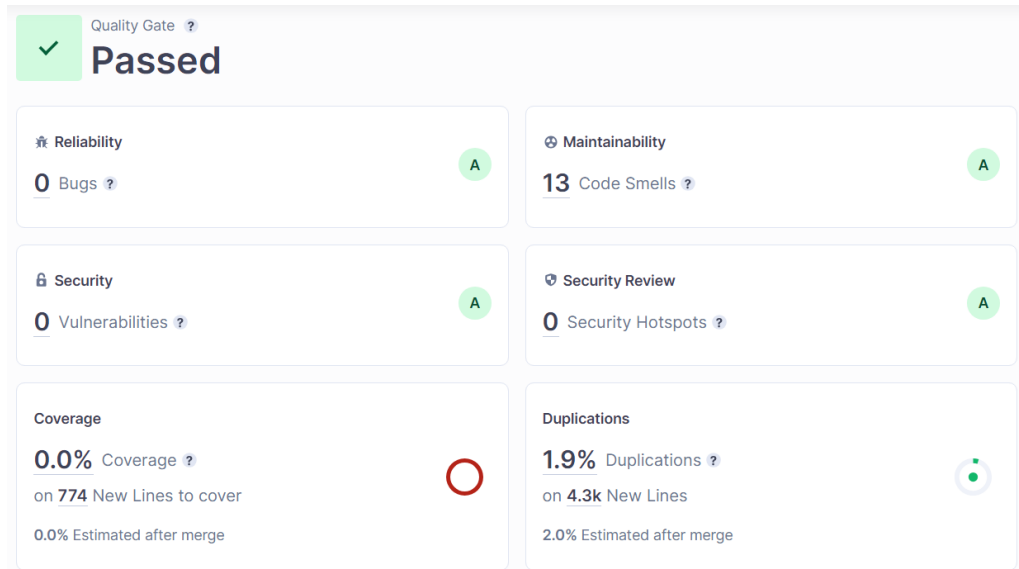


Imagen 64. Análisis de la calidad del código

5. Conclusiones y trabajo futuro

En este último apartado del trabajo se analiza si los objetivos marcados se han cumplido y también se habla sobre el trabajo futuro a realizar en el servicio y la aplicación.

5.1. Conclusiones

El objetivo principal del proyecto era el de crear una aplicación donde los usuarios pudieran consultar la información sobre la seguridad y sostenibilidad de dispositivos IoT. Para poder cumplir con este objetivo, primero había que diseñar un servicio de donde obtener los datos y empezaremos hablando de este.

El objetivo de diseñar un servicio para obtener los datos de los diferentes dispositivos IoT se ha cumplido, se ha diseñado un servicio seguro, algo muy importante y que da servicio desde cualquier red, de manera que cualquier persona puede utilizarlo y puede utilizarse en otros proyectos mediante llamadas HTTP. Además de cubrir con los datos necesarios de los diferentes dispositivos.

El objetivo principal, que era el desarrollo de la aplicación, también se ha cumplido. El producto final cumple con los requisitos acordados y se han implementado todas las funcionalidades, de modo que los usuarios ahora disponen de una aplicación, sencilla e intuitiva, con la que consultar la seguridad y sostenibilidad de dispositivos IoT.

Como conclusiones a nivel personal, remarcar que estoy muy contento con cómo se ha desarrollado el proyecto y con el producto final, aún con todos los contratiempos que

han surgido. He aprendido mucho sobre la importancia de la seguridad y sostenibilidad de los dispositivos IoT y sobre lo descuidado que está este tema.

5.2. Trabajo futuro

Para finalizar me gustaría hablar sobre ideas a realizar en el futuro. Ningún trabajo es perfecto y como en todos los casos, aquí también se podría ampliar el trabajo realizado, dando pie a una funcionalidad más extensa.

Lo primero a realizar sería una ampliación de la base de datos en cuanto al número de dispositivos y categorías que hay actualmente almacenados. Al existir tantos tipos diferentes de dispositivos IoT, en este proyecto se han introducido los dispositivos y categorías más básicos.

Además de la idea comentada anteriormente, también se debería añadir en el servicio llamadas para eliminar dispositivos y modificar o eliminar características, pudiendo eliminar características o dispositivos obsoletos y modificar características ya añadidas.

Ampliar las pruebas realizadas sobre el servicio sería otro de los trabajos a realizar en el futuro, de manera que se implementen de manera más directa, por ejemplo con JUnit.

Otra idea sería hacer que la seguridad y sostenibilidad se auto calculasen, esto se haría añadiendo un baremo en las características, de manera que mediante una fórmula matemática y sabiendo la “puntuación” de cada característica, la seguridad y sostenibilidad se auto calculen.

Si la aplicación y el servicio escalasen a nivel internacional, sería interesante la posibilidad de obtener los datos traducidos a cualquier idioma, de manera que una persona en cualquier país pudiera usar tanto el servicio como la aplicación.

Y finalmente como idea para la aplicación se podría añadir un recomendador, que en función de qué dispositivo busques o estés viendo, te dé opciones con mejor seguridad, mejor sostenibilidad, etc. Es algo que actualmente se puede realizar fijando una categoría y mostrando los resultados ordenados, pero sería interesante que al estar viendo un determinado dispositivo aparezca una recomendación del mejor o mejores de esa categoría. Por ejemplo, el más seguro, el más sostenible y el mejor ponderando esos dos valores.

Bibliografía

- [1] Kaspersky. (18 de octubre de 2021). El número de ataques a dispositivos IoT se duplica en un año. Recuperado el 19 de junio de 2023.
https://www.kaspersky.es/about/press-releases/2021_el-numero-de-ataques-a-dispositivos-iot-se-duplica-en-un-ano
- [2] Cordeiro, M. (8 de febrero de 2023). Número de dispositivos IoT conectados alcanzará 22 mil millones para 2025. Recuperado el 19 de junio de 2023.
<https://dplnews.com/numero-de-dispositivos-iot-conectados-alcanzara-22-mil-millones-para-2025/>
- [3] Albaladejo, X. (27 de septiembre de 2008). Desarrollo iterativo e incremental. Recuperado el 20 de junio de 2023.
<https://proyectosagiles.org/desarrollo-iterativo-incremental/>
- [4] Astigarraga, J., & Cruz-Alonso, V. (2022). ¿ Se puede entender cómo funcionan Git y GitHub!. Ecosistemas, 31(1), 2332-2332.
- [5] IBM. ¿Qué es Java Spring Boot?. Recuperado el 20 de junio de 2023.
<https://www.ibm.com/es-es/topics/java-spring-boot>
- [6] Azure. ¿Qué es Azure?. Recuperado el 21 de junio de 2023.
<https://azure.microsoft.com/es-es/resources/cloud-computing-dictionary/what-is-azure/>
- [7] Android Developers. (9 de mayo de 2023). Introducción a Android Studio. Recuperado el 21 de junio de 2023.
<https://developer.android.com/studio/intro>
- [8] JWT. Introduction to JSON Web Tokens. Recuperado el 22 de junio de 2023.
<https://jwt.io/introduction>
- [9] KeepCoding Team. (10 de abril de 2023). ¿Qué es el Modelo-Vista-Presentador (MVP)?. Recuperado el 26 de junio de 2023.
<https://keepcoding.io/blog/que-es-el-modelo-vista-presentador/>
- [10] Leiva, A. (25 de octubre de 2016). Cómo conectar tu aplicación de Android a una API con Retrofit y Kotlin. Recuperado el 28 de junio de 2023.
<https://devexpert.io/retrofit-android-kotlin/>