Universidad de Cantabria

Facultad
de
Ciencias

HAPI SECURITY: DESARROLLO DE UNA APP MÓVIL PARA COMPARAR LA SEGURIDAD EN DISPOSITIVOS IoT

(Hapi Security: Development of a mobile app to compare the security of IoT devices)

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

La idea de desarrollar esta aplicación nace hablando con el tutor del proyecto, Carlos Blanco, sobre un proyecto otorgado a la universidad **(indicar nombre proyecto)**. Carlos busca crear, en consonancia con el proyecto y unos estándares de la ENISA (Agencia de la Unión Europea para la Ciberseguridad), una aplicación donde poder comparar la seguridad y sostenibilidad de distintos dispositivos IoT que podamos tener en nuestros hogares, de manera que el usuario pueda tomar decisiones basándose en datos que comúnmente son de difícil acceso y así poder saber que tan seguro y sostenible es el producto que ha adquirido o quiere adquirir.

A la vista de la idea inicial y el encanto que tiene para mí el desarrollar una aplicación de este estilo, decido seguir para adelante con la idea, creando así Hapi Security. Hapi Security es una aplicación para dispositivos móviles, desarrollada en Java, que cuenta con diferentes funcionalidades, dispone de filtros y un buscador, de un escáner de códigos de barras, de favoritos para guardar tus dispositivos favoritos y de una sección donde compartir la aplicación con la gente que te rodea. Además de obviamente, tener los datos de los dispositivos de IoT más comunes en diferentes secciones y sus respectivas puntuaciones en seguridad y sostenibilidad.

Para el desarrollo de Hapi Security se ha tenido que dividir el proyecto en dos partes: La primera, donde se ha desarrollado un servicio REST con Spring Boot de donde poder tomar los datos de los dispositivos IoT desde la aplicación. Y la segunda, la propia aplicación desarrollada en Android Studio usando como lenguaje Java y usando el servicio desarrollado anteriormente para obtener los datos de los dispositivos.

Palabras clave:

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

Abstract

The idea of developing this application arises from a conversation with the project supervisor, Carlos Blanco, regarding a project assigned to the university **(specify project name)**. Carlos aims to create, in line with the project and ENISA standards (European Union Agency for Cybersecurity), an application that allows users to compare the security and sustainability of different IoT devices found in their homes. This way, users can make informed decisions based on data that is often difficult to access, enabling them to determine the level of security and sustainability of a product they have acquired or wish to acquire.

Considering the initial idea and my personal enthusiasm for developing an application of this nature, I decide to proceed with the concept, thus creating Hapi Security. Hapi Security is a Java-based mobile application that offers various functionalities. It includes filters and a search function, a barcode scanner, a favorites feature to save preferred devices, and a section for sharing the application with people in your surroundings. Additionally, it provides comprehensive data on the most common IoT devices in different sections, along with their corresponding security and sustainability ratings.

The development of Hapi Security required splitting the project into two parts: First, a REST service was created using Spring Boot to retrieve IoT device data for the application. Second, the application itself was developed using Android Studio, using Java as the programming language, and utilizing the previously developed service to obtain device data.

Key Words:

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

1. Introducción

Hoy en día vivimos rodeados de dispositivos IoT, desde los asistentes virtuales, pasando por la iluminación y terminando en los electrodomésticos inteligentes entre muchos otros campos, muchas de las cosas que nos rodean disponen de una conexión a Internet y eso supone un riesgo en la seguridad de estos y en tu seguridad.

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, pero lamentablemente, es un aspecto al que poca gente presta atención y cuyos datos son de difícil acceso.

Por todo ello y gracias al otorgue del proyecto ***INSERTAR NOMBRE*** a la universidad, decidí desarrollar Hapi Security, una aplicación móvil 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.

1.1. Objetivo

El objetivo principal es darle al usuario una aplicación móvil donde poder comparar la seguridad y sostenibilidad de diferentes dispositivos IoT ayudándole a la hora de decidir que dispositivo comprar. También que pueda buscar los dispositivos de los que ya dispone, mediante un buscador o escaneando el código de barras del dispositivo y tener una sección donde guardar sus dispositivos favoritos.

Para conseguirlo, se debe desarrollar el proyecto en dos partes: Una parte donde se crea y despliega un servicio donde almacenar y obtener los datos de los dispositivos y otra parte donde desarrollar la aplicación móvil que recoja los datos y se los muestre al usuario. En este documento se recogen estas dos partes, así como los requisitos, el diseño e implementación de estas.

2. Materiales y metodología utilizada

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

2.1. 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.

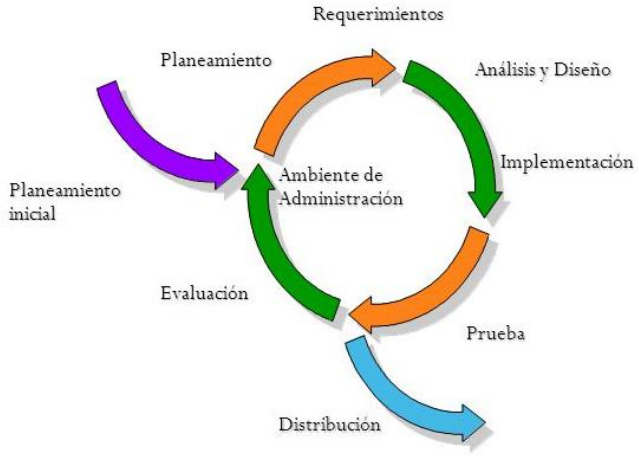

Imagen 1. Representación de la metodología iterativa incremental

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

- Requisitos.
- Análisis y Diseño.
- Implementación.
- Pruebas.

2.2. 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.

2.2.1. 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 logarlos. Por lo que una buena elección de objetivos y herramientas ayuda en las diferentes etapas del proyecto.

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

También fue sencillo el tema de las herramientas, en la misma reunión mencionada anteriormente establecimos que tecnologías utilizar para el desarrollo de las diferentes partes del proyecto. Estas se explicarán más adelante.

2.2.2. Desarrollo y despliegue del software

Este apartado corresponde con lo hablado anteriormente en la metodología. Se ha dividido en iteraciones y cada una de sus partes mencionadas anteriormente, para una vez finalizado todo el trabajo, realizar el despliegue final del servicio y de la aplicación móvil.

Ahora explicaré un poco más en detalle las partes 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:
    - **Requisitos funcionales:** Son requisitos sobre las funcionalidades que nuestro sistema deberá implementar y ofrecer a los usuarios, tales como poder buscar, filtrar, escanear, etc…
    - **Requisitos no funcionales:** Son requisitos sobre las diferentes propiedades del sistema, tales como la seguridad, el rendimiento, la mantenibilidad, etc…

- **Análisis y Diseño:** Se analizan los requisitos definidos anteriormente y se plantea una solución sobre como poder implementarlos en consonancia con los ya implementados, esto implica generar o ampliar el diseño para el software (Arquitectura), así como también tomar decisiones en cuanto al tema gráfico en la aplicación (logo, relación de colores, etc).

- **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, actualice funcionalidad ya implementada o realice cambios gráficos en la aplicación.

- **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, interfaz y aceptación.

2.2.3. Desarrollo de la memoria.

La memoria se ha realizado una vez todo el software ha sido realizado, testeado 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.

2.3. Tecnologías y Herramientas

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

2.3.1. Git y GitHub

Git es un sistema avanzado de control de versiones (como el “control de cambios” de Microsoft Word) distribuido (Ram 2013; Blischak et al. 2016). 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 (Perez-Riverol et al. 2016; Galeano 2018). Un repositorio es un directorio donde desarrollar un proyecto que contiene todos los archivos necesarios para el mismo [4].

Imagen 2. Logos de Git y GitHub

Se usarán tanto Git con su bash, para ir almacenando los cambios y poder llevar así un control de versiones del proyecto, como GitHub para almacenar el repositorio en la nube. Esto se realiza así por si en un futuro se añadieran más personas al proyecto, facilitar la trazabilidad y el manejo del código y los documentos. Ya que estas tecnologías son mundialmente conocidas y utilizadas.

2.3.2. Spring Boot, Maven y Java

Para el desarrollo del servicio implementado se ha tomado la decisión de utilizar Spring Boot software 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 obstinado
- 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 3. 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 variabilidad podía otorgarme y más proyección a futuro podía tener.

2.3.3. 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

```
1  package es.unican.hapisecurity.REST_TFGMarioIngelmoDiana.controllerLayer;
2
3  import java.net.URI;
32
33 @RestController
34 @RequestMapping("REST_TFGMarioIngelmoDiana")
35 public class GeneralController {
36
37     @Autowired
38     private AuthenticationManager authenticationManager;
39
40     @Autowired
41     private UserDetailsServiceImpl usuarioDetailsService;
42
43     @Autowired
44     private GestionTokens gestion;
45
46     @Autowired
47     private GeneralService servicio;
48
```

Imagen 4. Ejemplo de uso de Eclipse en el servicio implementado

2.3.4. 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 5. 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, cosas que se explicarán más adelante.

2.3.5. 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.

```java
@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 6. Ejemplo de uso de Android Studio en la aplicación

3. Desarrollo del servicio

Los apartados de desarrollo van a dividirse en dos: Por una parte en el punto 3 estará el desarrollo del servicio con sus respectivas partes y en el punto 4 estará el desarrollo de la aplicación con sus respectivas partes. Comenzamos con el desarrollo del servicio.

3.1. Análisis de requisitos del servicio

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

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 y características. Para esto se busca tener las siguientes características de manera resumida:

- 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.
- Que disponga de seguridad.

3.1.2. Requisitos funcionales

A continuación se presenta una tabla con los diferentes requisitos funcionales que se han obtenido:

ID	Descripción
RF1	El usuario podrá obtener una lista con todos los dispositivos y sus características.
RF2	El usuario podrá obtener una lista con todos los dispositivos y sus características filtrada por la categoría de los dispositivos que indique.
RF3	El usuario podrá obtener una lista con todos los dispositivos y sus características filtrada por la seguridad mínima que indique.
RF4	El usuario podrá obtener una lista con todos los dispositivos y sus características filtrada por la sostenibilidad mínima que indique.
RF5	El usuario podrá obtener una lista con todos los dispositivos y sus características 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 y sus características 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 y sus características 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 de seguridad y sostenibilidad.
RF10	El usuario podrá obtener una característica 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 que se han obtenido:

ID	Clasificación	Descripción
RNF1	Seguridad	El servicio deberá estar protegido utilizando JSON Web Tokens.
RNF2	Usabilidad, Mantenibilidad	El servicio deberá poder se 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.

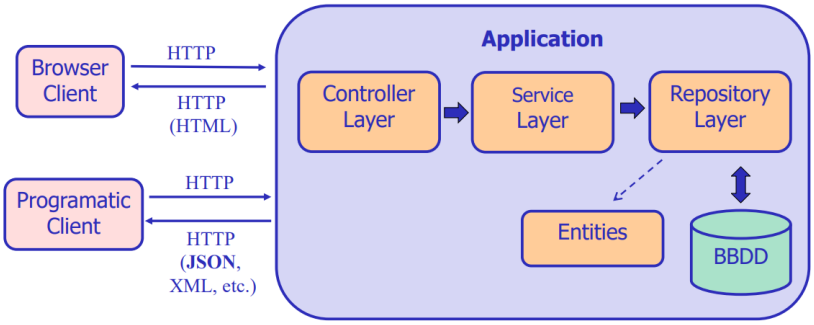

Imagen 7. Arquitectura típica de las aplicaciones Spring Boot

Viendo las capas de las que dispone el servicio, voy a explicarlas una a una para entender su funcionalidad:

- **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 desde 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 está implementando 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

3.3. 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 mi caso, la base de datos se ha implementado en Azure, como se comentará más adelante.
- **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 una de 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 se cree que aunque pueda ser algo más lento a tenerlo de manera local en donde el servicio se despliegue, si es algo más profesional y habitual.

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:

```java
@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 8. Clase Dispositivo.java

Por otro lado, tenemos Característica.java, entidad que almacena las diferentes características que existen según la ENISA sobre seguridad y sostenibilidad, tanto buenas como malas. Un ejemplo de característica mala de seguridad es: Contraseñas predeterminadas débiles: Contraseñas que son fáciles de adivinar o que no se pueden cambiar fácilmente.

```java
@Entity
public class Caracteristica {

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

Imagen 9. 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 10. Interfaz DispositivoRepository.java

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

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

}
```

Imagen 11. Interfaz CaracteristicaRepository.java

3.3.1.3. Enumerados

Pasamos con los enumerados, tenemos dos diferentes, un 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 12. 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 13. 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 caracteristicas 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 14. Clase ListaDispositivos.java

```java
public class ListaCaracteristicas {

    private List<Caracteristica> caracteristicas;
```

Imagen 15. 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.

```java
@Service
public class GeneralService {

    @Autowired
    private DispositivoRepository repositorioDispositivos;

    @Autowired
    private CaracteristicaRepository repositorioCaracteristicas;
```

Imagen 16. 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.

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

Imagen 17. 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.

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

Imagen 18. 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 19. 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 20. Método actualizaDispositivo(Dispositivo d) de la clase GeneralService.java

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

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

Imagen 21. Método caracteristicas() de la clase GeneralService.java

El método caracteristicaPorId(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 22. 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 23. 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 24. 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(Varios request params) 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.

```java
@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 25. 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.

```java
@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 26. 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_TFGMarioIngelmoDiana/ caracteristicas” y el método getCaracteristica(@PathVariable String id) se encarga de gestionar las peticiones GET con el path “REST_TFGMarioIngelmoDiana/caracteristicas/{id}”.

```java
@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 27. Método getCaracteristicas() de la clase GeneralController.java

```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 28. 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_TFGMarioIngelmoDiana/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.

```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 29. 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_TFGMarioIngelmoDiana/caracteristicas" y de crear una nueva característica, en caso de que no se pueda crear, se devolverá un CONFLICT.

```java
@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 30. 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 31. 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 32. 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 personalicemos 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, para así permitir el uso de tokens JWT, también se encargará de autorizar determinadas llamadas sin autenticación y de aplicar una configuración de cors personalizada añadiendo un filtro propio y así poder solicitar recursos restringidos por la seguridad, que se explicará a continuación.

```
@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 33. 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 34. . 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 35. Bean de la clase CorsConfig.java para la configuración del cors

3.4. Pruebas del servicio

Este apartado es uno de los puntos a tratar en el trabajo futuro. Se han realizado pruebas al servicio, pero no tantas como se querría y esto ha sido debido a la extensión del proyecto completo.

Se han realizado pruebas 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 mete un token válido.

3.5. Despliegue del servicio

Para el despliegue del servicio se ha utilizado Microsoft Azure, 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, también se ha creado una interfaz de red y se ha reservado una ip pública, asociándola a la máquina virtual, de manera que, a través de esa ip, se pueda acceder al servicio a través del puerto 8080, ya que, una vez pasado a la máquina y ejecutado, el servicio queda desplegado en ese puerto, haciendo así 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 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 36. Servicio en la máquina virtual para el despliegue automático del servicio REST

4. Desarrollo de la aplicación

En este apartado se explicarán los diferentes apartados que ha tenido el desarrollo de la aplicación Android, desde el diseño hasta la implementación.

4.1. Análisis de requisitos de la aplicación

4.2. Diseño de la aplicación

4.3. Implementación de la aplicación

4.4. Pruebas de la aplicación

5. Conclusiones y trabajo futuro

Trabajo futuro: Ampliar la base de datos, ampliar las categorías que se cubren, hacer que la seguridad y sostenibilidad sean auto calculados dependiendo de un baremo con un dato en las caracteristicas, eliminar dispositivos, modificar o eliminar caracteristicas.

BIBLIOGRAFIA

[1] https://www.kaspersky.es/about/press-releases/2021_el-numero-de-ataques-a-dispositivos-iot-se-duplica-en-un-ano

[2] https://dplnews.com/numero-de-dispositivos-iot-conectados-alcanzara-22-mil-millones-para-2025/#:~:text=El%20experto%20particip%C3%B3%20en%20el,millones%20de%20dispositivos%20IoT%20conectados

[3] 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] https://www.ibm.com/es-es/topics/java-spring-boot#%C2%BFQu%C3%A9+es+Java+Spring+Boot%3F

[6] https://azure.microsoft.com/es-es/resources/cloud-computing-dictionary/what-is-azure/

[7] https://developer.android.com/studio/intro?hl=es-419

[8] https://jwt.io/introduction