

Grado Ingeniería de Sistemas Audiovisuales
2018-2019

Trabajo Fin de Grado

“Diseño e implementación de un microservicio con Spring”

Jesús Rienda Iáñez

Tutor/es

Carmen Pelaez Moreno

Leganés, 2019



[Incluir en el caso del interés en su publicación en el archivo abierto]

Esta obra se encuentra sujeta a la licencia Creative Commons **Reconocimiento - No Comercial - Sin Obra Derivada**

RESUMEN

En este trabajo fin de grado se ha realizado un estudio sobre las posibilidades existentes en el desarrollo de aplicaciones web. Desde el desarrollo partiendo de cero hasta el uso de software de terceros como son los framework para facilitar el desarrollo y centrarse en la funcionalidad.

Para el correcto diseño de nuestra arquitectura ha sido necesario descomponer e investigar cada integrante de esta. Por una parte se ha investigado sobre los patrones de diseño existentes y sus funcionalidades, también sobre tipos de arquitecturas, framework y además sobre las bases de datos.

Para afianzar todo lo desarrollado en este trabajo se ha implementado un microservicio basado en listas de reproducción de Spotify. Utilizando para ello las últimas tecnologías de desarrollo como Spring Boot, Spring Data y una base de datos muy potente como MongoDB. Este microservicio hace de intermediario con la base de datos, filtrando por ejemplo las listas de reproducción que contienen canciones que ha escuchado un usuario.

Palabras clave: Microservicio, aplicaciones web, SOA, API, Swagger, MongoDB, Lombok, Spring Boot

DEDICATORIA

ÍNDICE GENERAL

1. INTRODUCCIÓN.	1
1.1. Planteamiento del problema.	1
1.2. Objetivos	1
1.3. Solución propuesta.	2
1.4. Estructura trabajo	2
2. ESTADO DEL ARTE.	3
2.1. Patrones de diseño	4
2.2. Patrones de arquitectura	6
2.3. Frameworks.	7
2.3.1. Frameworks de java para microservicios.	8
2.4. Bases de datos	9
2.4.1. Pros y contras bases de datos relacionales	10
2.4.2. Pros y contras bases de datos no relacionales	10
2.5. Microservicios	11
2.5.1. Pros y contras de los microservicios	11
2.5.2. Ejemplos de empresas que utilizan microservicios	12
2.6. Comparación de SOA con Microservicios	13
2.6.1. Ejemplo de visión monolítica vs visión de microservicio	15
3. JUSTIFICACIÓN DE LA SOLUCIÓN	17
4. DESARROLLO DE LA APLICACIÓN	19
5. MARCO REGULADOR	24
6. ENTORNO SOCIO-ECONÓMICO	25
6.1. Presupuesto	25
6.1.1. Mano de obra.	25
6.1.2. Servicios	25
6.1.3. Coste total.	26
6.2. Impacto socio-económico	26

7. CONCLUSIONES	27
7.1. Trabajos futuros	27
8. ANEXO I: DESPLIEGUE DE LA APLICACIÓN.	29
9. ANEXO II: DESCRIPCIÓN API MICROSERVICIO	31
BIBLIOGRAFÍA	33

ÍNDICE DE FIGURAS

2.1	Estructura arquitectura microservicios	4
2.2	Historia de los Web Frameworks	8
2.3	Arquitectura Monolítica frente Microservicios	13
2.4	Eficiencia microservicios frente SOA	14
2.5	Ejemplo de arquitectura monolitica(SOA)	15
2.6	Ejemplo arquitectura microservicios	16
3.1	API en swagger	18
4.1	Ejemplo configuración arquetipo	20
4.2	Dependencias Swagger necesarias	20
4.3	Clase configuración Swagger	20
4.4	Clase de configuración MongoDB	21
4.5	Estructura microservicio	22
8.1	Configuración compilación mediante Maven	30
8.2	Configuración JRE proyecto	30
8.3	Configuración Run despliegue	30
9.1	Recursos disponibles dentro del controller Listas	31
9.2	Recursos disponibles dentro del controller Usuarios	32
9.3	Json ejemplo lista reproducción	32
9.4	Json ejemplo usuario	32

ÍNDICE DE TABLAS

6.1	Coste personal.	25
6.2	Costes Microsoft Azure.	25
6.3	Costes Totales.	26

1. INTRODUCCIÓN

El mundo de la tecnología se basa en tendencias, hoy en día los microservicios están de moda, gracias a grandes compañías como Amazon, Ebay, Twitter, Paypal, Netflix, etc. Pero realmente es una tecnología que ha surgido gracias a la aparición de la nube(cloud) donde todas estas empresas se están llevando sus software

En la nube no necesitas servidores físicos, con el mantenimiento y la inversión que esto supone, sino que solo pagas por los recursos que necesitas en cada momento. Esto es una gran revolución para la informática, te permite gestionar los picos de actividad sin perder servicio.

1.1. Planteamiento del problema

Contamos con datos reales sobre listas de reproducción de Spotify en formato json los cuales contienen información de cada lista y las canciones que contiene. Por otra parte tenemos datos de usuarios con las canciones que escuchan.

Necesitamos almacenar estos datos en una base de datos para posteriormente consultarla, actualizarla o crear nuevos registros. Podríamos actuar directamente sobre la base de datos pero no sería admisible para un usuario final. Por tanto tendremos que crear un programa el cual haga las consultas a la bbdd y devuelva los datos validados en un formato optimo para mostrarlos por pantalla. Para ello Por lo que necesitamos crear un programa que al invocar nos devuelva los datos almacenados con un tratamiento específico y un formato definido. Necesitamos que sea sencillo y simple para el cliente que va a consumir dicho servicio. Por ejemplo uno de los tratamiento necesarios sería filtrar las listas en función de las canciones que escuche cada usuario, si ha escuchado mas de 3 canciones de una lista, deberíamos devolverla.

Este programa tendrá que tener una alta disponibilidad y escalarse cuando sea necesario para siempre tener unos tiempos de respuesta bajos. Ya que existe una gran cantidad de personas que van a consumir el programa al mismo tiempo coincidiendo con el lanzamiento de nuevas listas.

1.2. Objetivos

Diseñar una arquitectura de software adecuada para el problema, que sea capaz de soportar una gran cantidad de peticiones simultaneas sin dejar sin servicio a nadie.

Crear y rellenar una base datos con los ficheros de listas con los que contamos.

Implementar un microservicio que al pasarle un usuario y un numero de canciones

a coincidir devuelva las listas de reproducción que contengan al menos el numero de canciones que coincidan con las escuchadas por el usuario.

1.3. Solución propuesta

Como solución a nuestro problema vamos a diseñar una arquitectura de microservicios. Para ello implementamos un microservicio el cual, se va a desarrollar con java mediante el framework Spring Boot. La BBDD estará almacenada en Azure y sera CosmosDB Para llamar al microservicio creado utilizaremos Swagger.

1.4. Estructura trabajo

Este trabajo está estructurado de la siguiente forma:

- **Capítulo 1: Introducción.** En este capítulo se explica el problema, los objetivos del trabajo y la solución propuesta.
- **Capítulo 2: Estado del arte.** En este capítulo se analiza el estado a día de hoy del diseño de aplicaciones desde cero y en profundidad de microservicios.
- **Capítulo 3: Justificación de la solución.** Según lo visto en el capítulo anterior analizamos que es lo que más nos conviene para nuestros requerimientos.
- **Capítulo 4: Desarrollo de la aplicación.** Explicación desde cero de como crear y desarrollar el microservicio.
- **Capítulo 5: Implantación de la aplicación.** Pasos a seguir para poder implantar la aplicación en un entorno empresarial.
- **Capítulo 6: Marco regulador.** Regulación existente en el ámbito de los microservicios.
- **Capítulo 7: Entorno socio-económico.** En este punto se muestra el coste total que tendría el proyecto y el impacto que tiene.
- **Capítulo 8: Conclusiones.** Aquí describimos las conclusiones sacadas a raíz de la realización del trabajo y las posibles tareas a realizar en un futuro.
- **Anexo I: Despliegue de la aplicación** En este anexo se realiza un tutorial para desplegar nuestra aplicación en local.
- **Anexo II: Descripción API microservicio** En este anexo se detallan los recursos y servicios creados en nuestra aplicación.

2. ESTADO DEL ARTE

En la década de los 60 surgió lo que a día de hoy conocemos como arquitectura de software, esta fue tomando cada vez mas interés hasta que en la década de 1980 se integro totalmente el diseño en el desarrollo de software.

El Instituto de Ingeniera de Software la define como: "La arquitectura de software es una representación del sistema que ayuda a comprender cómo se comportará un programa.

La arquitectura del software sirve como modelo tanto para el sistema como para el proyecto que lo desarrolla. La arquitectura es la principal portadora de cualidades del sistema, como el rendimiento, la modificabilidad y la seguridad, ninguna de las cuales se puede lograr sin una visión arquitectónica unificadora. La arquitectura es un artefacto para el análisis temprano para asegurar que un enfoque de diseño proporcionará un sistema aceptable. Al construir una arquitectura efectiva, puede identificar los riesgos de diseño y mitigarlos al inicio del proceso de desarrollo."[1].

El sistema se divide en elementos de software también llamados módulos, con propiedades y relaciones existentes entre ellos. Las propiedades de estos elementos pueden ser de dos tipos, internas y externas.

Las **propiedades internas** son aquellas que definen el módulo, es decir, el lenguaje en el que está desarrollado y todos los detalles de la implementación de este como pueden ser:

- Entidades dinámicas en tiempo de ejecución como objetos e hilos.
- Entidades lógicas en tiempo de desarrollo como clases y módulos.
- Entidades físicas como nodos o carpetas.

Las **propiedades externas** son los contratos que existen entre módulos y que permiten a otros módulos establecer dependencias/conexiones entre ellos. Es de vital importancia que las interfaces que definen los contratos estén bien definidas para la perfecta integración de los elementos.

Es necesario dividir los requerimientos con los que va a contar el sistema en módulos y definir sus propiedades y como se relacionan entre si. Por ejemplo en la figura 2.1 podemos ver la estructura de un sistema basado en microservicios donde aparecen bases de datos, que serian un modulo donde habría que definir sus propiedades, tablas, colecciones de datos, etc; para cada microservicio habría que definir un API(Interfaz de programación de aplicaciones) en el que se indique la entrada y salida del modulo, el lenguaje en el que se va a desarrollar...; por ultimo tendríamos un módulo de presentación UI(Interfaz de usuario).

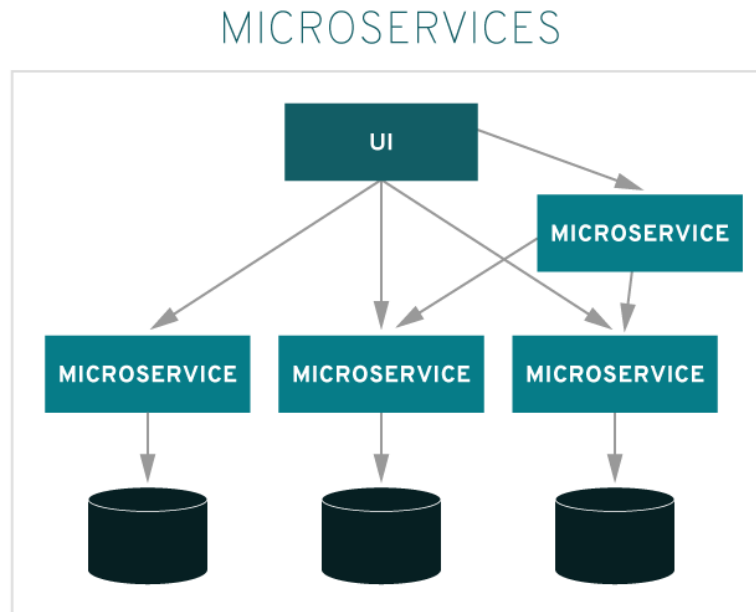


Fig. 2.1. Estructura arquitectura microservicios

2.1. Patrones de diseño

En el momento de diseño de una arquitectura existen dos formas de actuar, diseñar una arquitectura desde cero o buscar soluciones ya propuestas a nuestro problema. Si elegimos la segunda opción existen patrones de diseño, patrones arquitectónicos y frameworks.

Los patrones son una forma reutilizable de resolver problemas comunes. Los primeros definidos en arquitectura civil se encuentran en el libro *"A pattern language"*[2], en él se describen los patrones con un nombre, de tal manera que con hacer referencia a este, cualquier arquitecto será capaz de entender que solución se está usando sin tener que entrar en detalles. Sin embargo en la arquitectura de software no aparecieron hasta 1994 en un libro que a día de hoy sigue siendo un modelo de referencia[3].

Los patrones definidos en el libro [3] se dividen en función del problema a resolver:

- **Patrones creacionales:** estos son los utilizados para facilitar la creación de nuevos objetos. Los más conocidos son:
 - **Abstract Factory:** Contiene una interfaz que otorga la creación de objetos relacionados, sin tener que especificar cuáles son las implementaciones concretas.
 - **Factory Method:** Contiene un método de creación que delega en las subclases la implementación de dicho método.
 - **Builder:** Con un mismo proceso de construcción nos sirve para crear diferentes representaciones, separando la creación de un objeto complejo de su estructura.

- Singleton: Solo se permite la creación de una instancia de una clase en nuestro programa y se proporciona un acceso global a el.
 - Prototype: Un objeto se crea a partir de la clonación de otro objeto, es decir se crea basandose en unas "plantillas"
- Patrones estructurales: estos nos facilitan la modelización de nuestro software, definiendo la forma en que las clases se relacionan entre si. Los más conocidos son:
- Adapter: Mediante un objeto intermedio, se pueden comunicar dos clases con distinta interfaz.
 - Bridge: Se crea un puente entre la abstracción y la implementación, para puedan evolucionar independientemente.
 - Composite: Sirve para crear objetos contenidos en un arbol, donde todos los elementos emplean una misma interfaz.
 - Decorator: Se añade funcionalidad extra a un objeto (de forma dinámica o estática) sin cambiar su comportamiento.
 - Facade: Objeto que crea una interfaz para poder trabajar con otra parte más compleja. Un ejemplo podría ser crear una fachada para trabajar con una librería externa.
 - Flyweight: Para ahorrar memoria, gran cantidad de objetos comparten un objeto con las mismas propiedades.
 - Proxy: Clase que funciona como interfaz destinada a cualquier otra cosa: conexión a Internet, archivo en disco, etc.
- Patrones de comportamiento: Se usan para gestionar algoritmos, relaciones y responsabilidades entre objetos. Los más conocidos son:
- Command: Objetos que necesitan para ejecutarse, contener una acción y sus parámetros.
 - Chain of responsibility: Permite pasar solicitudes a lo largo de una cadena de receptores. Al recibir una solicitud, cada controlador decide procesar la solicitud o pasarla al siguiente de la cadena.
 - Interpreter: Define una representación y el mecanismo para poder evaluar una gramática. El árbol de sintaxis del lenguaje se modela mediante el patrón **Composite**.
 - Iterator: Nos permite movernos por los elementos de forma secuencial sin necesidad de conocer su implementación.
 - Mediator: Objeto que contiene un conjunto de objetos que interactúan y se comunican entre sí.
 - Memento: Permite restaurar un objeto a un estado anterior.

- Observer: Objetos que pueden unirse a una serie de eventos que otro objeto va a producir para estar informados cuando esto cambie.
- State: Modifica el comportamiento de un objeto en el tiempo de ejecución.
- Strategy: Selecciona el algoritmo que ejecuta ciertas acciones en tiempo de ejecución.
- Template Method: Nos permite conocer la forma del algoritmo.
- Visitor: Separa el algoritmo de la estructura de datos que se utilizará a la hora de ejecutarlo. Por lo que se pueden añadir nuevas opciones sin tener que ser modificadas.

2.2. Patrones de arquitectura

Por otra parte también existen los patrones arquitectónicos o arquetipos, los cuales tienen un nivel superior de abstracción (explicar). Estos al igual que los patrones de diseño, solucionan problemas recurrentes de una forma reutilizable.

Los patrones de arquitectura más usados son:

- Programación por capas: Se utiliza para estructurar programas que pueden descomponerse en subtarefas. Normalmente cuenta con tres capas, en una de ellas se implementa la lógica de negocio, en otra los datos y en otra la presentación de los datos ya tratados. Es uno de los arquetipos más usados a día de hoy.
- Cliente-servidor: Este patrón consta de dos partes: un servidor y múltiples clientes, el servidor será el que de servicio a diversos componentes del cliente y los clientes solicitarán servicio al servidor. En esta arquitectura se separan las capas en varias máquinas físicas, normalmente se utiliza 2 o 3 niveles. Si utilizáramos dos niveles tendríamos la capa de presentación en una máquina del cliente y el tratamiento de los datos con la base de datos en otra máquina. En la de 3 niveles se pondría cada capa en un servidor diferente con la mejora de escalabilidad.
- Arquitectura orientada a servicios: Es la que da soporte a los requerimientos del negocio mediante la creación de servicios. Un servicio corresponde a un requerimiento funcional del negocio, por ejemplo: Un cliente necesita saber el stock de un determinado producto, por lo que se crea un servicio que consulte en la base de datos la cantidad de producto disponible.
- Arquitectura de microservicios: Utilizada para crear aplicaciones usando un conjunto de pequeños servicios, los cuales se comunican entre sí pero se ejecutan de forma individual.
- Pipeline: Utilizado para organizar sistemas que procesan una sucesión de datos. Estos datos pasan a través de tuberías (que son combinación de comandos que se

ejecutan de forma simultanea, donde el resultado de la primera se envía de entrada para el siguiente). Las tuberías se usan para almacenar datos en buffer o para la sincronización de estos.

- **Arquitectura en pizarra:** Se utiliza para la resolución de problemas de los cuales se desconoce su estrategia. Está formado por tres componentes:
 - **Pizarra:** memoria que contiene todos los objetos.
 - **Fuente de conocimiento:** son módulos especializados.
 - **Componente de control:** encargado de seleccionar y ejecutar los módulos.
- **Arquitectura dirigida por eventos:** Maneja principalmente los eventos y está formado por cuatro componentes que son: fuente de evento , escucha de evento , canal y bus de evento.
- **Peer-to-peer:** Se llama pares a las componentes individuales y estos pueden funcionar como servidor, dando servicio a otros pares, o como cliente, pidiendo servicio a otros pares.
- **Modelo Vista Controlador:** Divide un programa interactivo en tres partes:
 - **Modelo:** está formado por los datos básicos y contiene la funcionalidad del programa.
 - **Vista:** maneja la visualización de la información.
 - **Controlador:** encargado de controlar la entrada (teclado y ratón) del usuario e informar al modelo y la vista de los cambios de acuerdo a los requerimientos.

Permite desacoplar los componentes y reutilizar código más eficiente.

2.3. Frameworks

Otra opción para no tener que diseñar una arquitectura desde cero sería la utilización de frameworks. Estos son estructuras de software ya implementadas en las que un programador puede apoyarse para desarrollar un proyecto propio. Los frameworks están implementados y siguen tanto los patrones de software como los patrones de arquitectura y suelen incluir ficheros de configuración, librerías, etc.

Existen frameworks para prácticamente todos los lenguajes de programación, con ellos solo se implementarían los requisitos funcionales del producto, eso sí, es importante valorar y elegir el framework que mejor se adapta al problema que vas a resolver.

Los frameworks de java más utilizados en los últimos tiempos son Spring y Struts.

Spring surgió en 2003 de la mano de Rod Johnson, la idea de este framework es agilizar y estandarizar la creación de proyectos de una misma arquitectura.

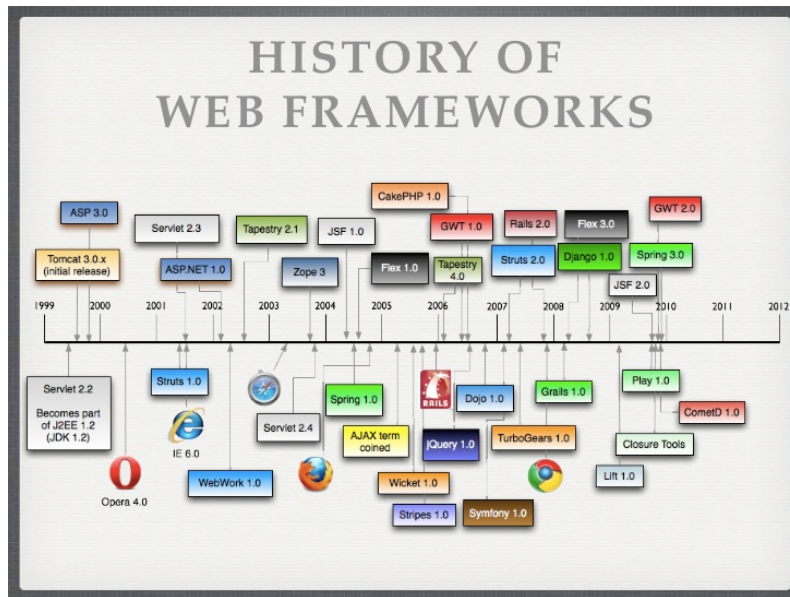


Fig. 2.2. Historia de los Web Frameworks

Struts nació en 2001 y está basado en la arquitectura MVC(Modelo vista controlador) el cual permite gracias a las anotaciones poder configurar este modelo. Posteriormente, en 2004, surgió Stripes con la misma arquitectura pero con la finalidad de ser más ligero y eficiente.

Existen framework de otros lenguajes de programación como por ejemplo Angular lanzado en septiembre de 2016. Está implementado en typescript y orientado al desarrollo del front-end de aplicaciones web. Este framework es propiedad de Google aunque es de código libre.

React es otro framework para el desarrollo de front-end. Contiene una serie de librerías de javascript para construir las aplicaciones web mas fácilmente. Este es propiedad de Facebook.

Ionic surgió en 2013, siempre ha ido de la mano con Angular, desde AngularJs hasta las ultimas versiones del mismo. Ionic está diseñado para desarrollar aplicaciones híbridas, según este artículo de next-u [4] "Las aplicaciones híbridas son aplicaciones móviles diseñadas en un lenguaje de programación web (HTML5, CSS o JavaScript), junto con un framework (Cordova) que permite adaptar la vista web a cualquier vista de un dispositivo móvil. Es decir, no son más que una aplicación construida para ser utilizada o implementada en distintos sistemas operativos móviles evitándonos la tarea de crear una aplicación para cada sistema operativo."

2.3.1. Frameworks de java para microservicios

Por otra parte existen bastantes que se basan en java y ofrecen una arquitectura de microservicios, estan han surgido de las grandes empresas del sector como Oracle, Spring, Eclipse estos son:

- **Spring Boot:** Es una capa de personalización sobre el famoso framework Spring, la cual ofrece un arquetipo de un microservicio básico, ampliable fácilmente con poco desarrollo. Por ejemplo Gustavo Peiretti en el artículo [5], propone una implementación de un patrón strategy desarrollado con Spring Boot.
- **Cricket:** Es un framework para implementación de arquitectura dirigida por eventos. Contiene un servidor http incorporado, serialización de Java a json automática, gestión de usuarios y accesos, bases de datos integradas como h2 y un planificador de eventos. Con todo ello se puede construir una aplicación fácilmente.
- **Dropwizard:** Es un framework para arquitectura de microservicios por capas, para utilizar este solo es necesario importarlo en la aplicación y usar todas las librerías que este incluye.
- **Eclipse MicroProfile:** En junio de 2018 consiguió cubrir todas las necesidades de la arquitectura. Cuenta al igual que Spring con una web donde obtener un arquetipo de la aplicación con las dependencias que se necesiten.
- **Helidon:** Es de las últimas incorporaciones en la arquitectura. Perteneció a la compañía Oracle la cual lo utilizó internamente bajo el nombre "Java for Cloudz" que finalmente se ha liberado para todo el público.

2.4. Bases de datos

En el proceso de diseño de una arquitectura otro módulo fundamental es el almacenamiento de los datos que vamos a utilizar, para ello usamos las bases de datos.

Estas se pueden definir como un conjunto de información relacionada que se encuentra agrupada o estructurada.

El concepto de bases de datos ha estado siempre ligado a la informática, estas son sistemas formados por grupos de datos almacenados en discos, que permiten la consulta, modificación y borrado de los mismos. Se crearon para poder almacenar grandes cantidades de información.

En la década de los 70, Edgar Frank Codd, científico informático inglés conocido por su participación en la teoría de bases de datos relacionales, definió el modelo relacional a través de su artículo "Un modelo relacional de datos para grandes bancos de datos compartidos"[6].

Larry Ellison, basándose en el trabajo de Edgar F. Codd, creó el "Relational Software System" más conocido como Oracle Corporation, el sistema de gestión de bases de datos relacional más conocido actualmente.

En la década de los 80 se desarrolló SQL (Structured Query Language) un lenguaje de acceso a bases de datos relacionales que permite realizar consultas y cambios de forma

sencilla. En los 90 se actualizó SQL y se agregaron: expresiones regulares, consultas recursivas, triggers y algunas características orientadas a objetos.

En la actualidad, las tres grandes compañías que dominan el mercado de las bases de datos son IBM, Microsoft y Oracle. En el campo de internet, la compañía que más información genera es Google.

No fue hasta la década de los 2000 cuando surgieron las bases de datos no relacionales (NoSQL) que son las que no contienen un identificador para relacionar un conjunto de datos con otro. La información se organiza en documentos y es útil cuando no tenemos claro lo que se va a almacenar.

La más exitosa en bases de datos no relacionales es MongoDB seguida por Redis, Elasticsearch y Cassandra.

2.4.1. Pros y contras bases de datos relacionales

Ventajas:

- Existe gran variedad de información para poder realizar cualquier tipo de desarrollo o consulta, gracias a sus años de madurez.
- Asegura que la información no se va a completar si a mitad de una operación realizada en base de datos ocurre un problema.
- Tiene los estándares bien definidos.
- Es sencillo a la hora de programar operaciones.

Inconvenientes:

- Estas bases de datos tienden a crecer demasiado por lo que aumenta el tiempo de respuesta.
- Se tendrá que reestructurar la base de datos de una empresa debido a que esta realice algún cambio en sus sistemas informáticos.
- No garantizan el buen funcionamiento si el sistema operativo donde se va a instalar no cumplen con los requerimientos mínimos.

2.4.2. Pros y contras bases de datos no relacionales

Ventajas:

- Adaptabilidad para crecimientos o cambios a la hora de almacenar la información, si se necesita un nuevo campo en la base, se agrega sobre el documento y el sistema sigue operando sin tener que añadir nuevas configuraciones.

- Crecimiento horizontal. Si la actuación de los servidores disminuye durante la operación, existe la posibilidad de instalar nuevos nodos que distribuyen la carga de trabajo, de ahí, crecimiento horizontal.
- No es necesario contar con servidores con gran cantidad de recursos disponibles para realizar operaciones, si no que pueden empezar con un número de recursos limitado y dependiendo de las necesidades ir creciendo sin tener que detener las operaciones.
- Cuenta con un algoritmo interno que reformule las consultas de los usuarios o de las aplicaciones programadas para no sobrecargar a los servidores.

Inconvenientes:

- Como no nos garantiza que si la operación falla se complete la información, esta en ocasiones es inconsistente.
- Dado que es un modelo bastante nuevo, se necesitan conocimientos elevados en el uso de esta herramienta, ya que las operaciones son limitadas.
- No tiene un estándar de lenguaje definido.
- No contiene una interfaz gráfica por lo que es necesario hacer todo mediante consola.

2.5. Microservicios

Una “arquitectura de microservicios” es útil para desarrollar una aplicación software dividiéndola en una serie de pequeños servicios, los cuales se ejecutan de forma independiente y se comunican entre sí, por ejemplo, a través de peticiones HTTP a sus API.

Tiene que haber un número mínimo de servicios encargados de gestionar los procesos en común. Cada microservicio es pequeño y se corresponde con un área de la aplicación. Cada uno es independiente, pudiéndose programar en lenguajes diferentes y su código debe poder ser implementado sin que afecte a los demás.

Cada microservicio no tiene definido ni el tamaño, ni cómo se tiene dividir la aplicación pero en el libro "Building Microservices", [7] se caracteriza un microservicio como “algo que a nivel de código podría ser reescrito en dos semanas”.

2.5.1. Pros y contras de los microservicios

Ventajas:

- Concede a los desarrolladores la libertad de implementar y desplegar los servicios de forma independiente.

- Los microservicios se pueden desarrollar con un equipo de trabajo mínimo.
- Para los diferentes módulos, existe la posibilidad de utilizar lenguajes diferentes.
- Con el uso de contenedores el desarrollo y despliegue de la app es mucho más rápido.
- Son fácilmente escalables.

Inconvenientes:

- Las pruebas pueden resultar complicados debido al despliegue distribuido.
- Con un gran número de servicios se puede dar lugar a grandes bloques de información para gestionar.
- Si contamos con un gran número de servicios, a la hora de integrarlos y gestionarlos puede resultar muy complicado.
- Esta tecnología suele incurrir en un alto consumo de memoria.
- Fragmentar una aplicación en diferentes microservicios puede llevar muchas horas de planificación.

2.5.2. Ejemplos de empresas que utilizan microservicios

Para poder contemplar hasta dónde ha llegado la arquitectura de microservicios observamos algunas grandes marcas que lo han implementado. Webs de aplicaciones a gran escala han decidido utilizar los microservicios en vistas de productos mucho más simples, efectivos y rápidos. Encontramos estas compañías:

- **Netflix:** Esta plataforma tiene una arquitectura generalizada que desde hace un par de años se pasó a los microservicios. A diario recibe una media de mil millones de llamadas a sus diferentes servicios y es capaz de adaptarse a más de 800 tipos de dispositivos mediante su API de streaming de vídeo.
- **Amazon:** No tiene soporte para tantos dispositivos como Netflix, pero tampoco es algo fundamental en su sector. Se cambió hace tres años a la arquitectura de microservicios siendo una de las primeras compañías que la implementan en producción. No hay cifra aproximada de la cantidad de solicitudes que pueden recibir a diario, pero no son pocas.
- **Ebay:** Es una de las empresas con mayor visión de futuro, siendo pionera en la adopción de tecnologías como Docker. Su aplicación principal comprende varios servicios autónomos, y cada uno ejecutará la lógica propia de cada área.

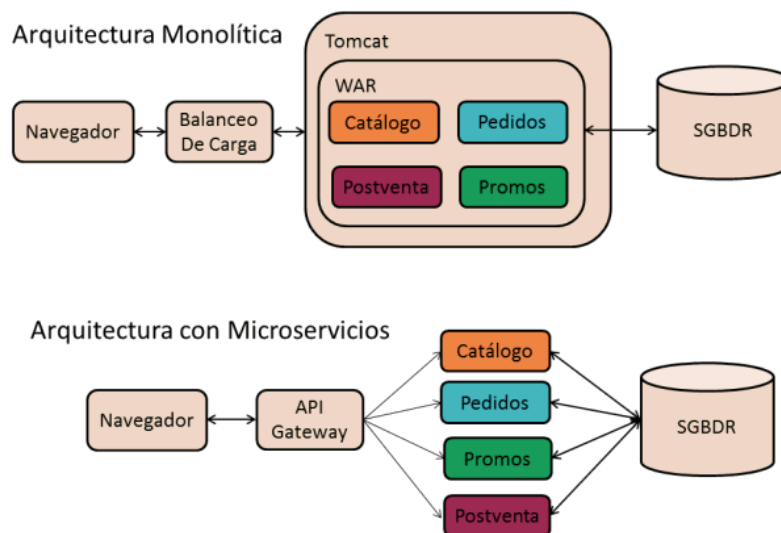


Fig. 2.3. Arquitectura Monolítica frente Microservicios

2.6. Comparación de SOA con Microservicios

Primero surgió la arquitectura orientada objetos y más adelante la orienta a componentes. Pero no fue hasta 1996 cuando se desarrolló por primera vez SOA (Service Oriented Architecture) “arquitectura orientada a servicios”. Con esta se desarrollaban todos los servicios necesarios y se empaquetaban en un WAR (Web Application Archive) para poder desplegarse en un servidor de aplicaciones, como tomcat, que se encontraba dentro de una máquina. Esto puede verse en la figura 2.3.

Todos estos servicios tenían que estar implementados en el mismo lenguaje y no se les podía asignar más recursos a cada uno de ellos. Era asignado a todo el conjunto, dividiendo el WAR en varias máquinas o réplicas de la misma. Para ello se necesitaba un balanceador de carga que determinaba que máquina iba a atender la petición.

Todo esto era suficiente para las empresas. A día de hoy cuando una aplicación monolítica (SOA) crece demasiado es difícil mantenerla y añadir nuevas funcionalidades, ya que cada línea modificada implica re-desplegar toda la aplicación, lo que puede llevar mucho tiempo pues en los despliegues están involucrados varios departamentos de la empresa que impide al equipo seguir desarrollando. También es complicado encontrar el origen de algún error en el código.

La necesidad por resolver todos estos problemas desembocó en la arquitectura de microservicios. La primera vez que se mencionó la palabra "microservicios" fue en 2011 en una conferencia sobre computación en la nube, donde el Dr. Peter Rogers[8] se refirió a ello para describir la arquitectura que estaban usando grandes empresas como Netflix, Facebook, Amazon o PayPal.

Los microservicios gestionan la complejidad granulando funcionalmente en un conjunto de servicios pequeños e independientes. Con esto se consigue que el equipo de

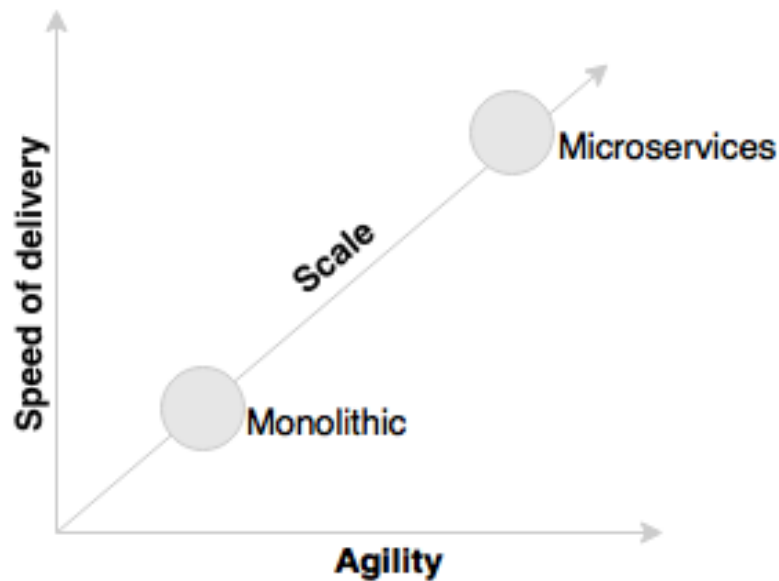


Fig. 2.4. Eficiencia microservicios frente SOA

desarrollo sea capaz de implementar varias funcionalidades a la vez, sin tocar el código de otra funcionalidad y desplegar cada módulo por separado tal y como se ve en la figura 2.3 donde cada microservicio está separado del resto, y puede o no tener una base de datos común.

El cambio más notable respecto a SOA es que los equipos de desarrollo tienen una mayor responsabilidad, lo que se traduce en una gran facilidad, ya que manejan todo lo siguiente: proceso de desarrollo, despliegues en distintos entornos, gestión de contenedores como Kubernetes, etc. Todo esto antes se tenía que realizar en otros departamentos de la empresa aumentando tiempos de producción. La arquitectura de microservicios resuelve todos los problemas que presenta SOA y cada vez es más popular, pero aún está en su base de inicio como se menciona en el artículo[9] y aun le queda mucho por mejorar y evolucionar.

Rajest RV en su libro "Spring Microservices"[10] nos muestra el gráfico 2.4 en el que se observa como es más rápido y ágil el desarrollo de aplicaciones con microservicios frente a aplicaciones tradicionales. Menciona que, "Los microservicios prometen más agilidad, velocidad de entrega y escala. En comparación con las aplicaciones monolíticas tradicionales."

En un futuro se deberían solucionar problemas debidos a estar poco restringidos, por ejemplo si cada microservicio se desarrollara con un lenguaje diferente no estaría del todo claro los protocolos que se usan en cada uno de ellos y de que la comunicación sea totalmente compatibles.

Otro aspecto a mejorar es la seguridad, ya que cuando se descompone una aplicación en cientos de microservicios aparecen dificultades en la depuración, monitoreo, auditoría y análisis forense de toda la aplicación. Los atacantes podrían aprovechar esta compleji-

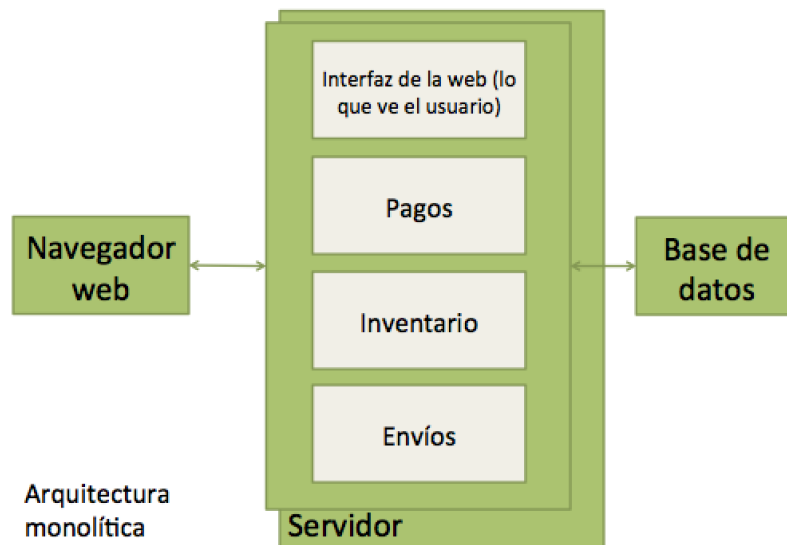


Fig. 2.5. Ejemplo de arquitectura monolítica(SOA)

dad para atacar.

Lo que sí es seguro es que han surgido para quedarse y cada vez más empresas están empezando a utilizar los microservicios.

2.6.1. Ejemplo de visión monolítica vs visión de microservicio

Accedemos a una página web de una tienda para realizar compras. Hay un servidor que está ejecutando el código de la aplicación y de vez en cuando se conecta a una base de datos para recuperar información. Cuando un usuario compra un producto vía web, se mostrarán los productos disponibles. La aplicación responderá según haya sido programada y mostrará el inventario, pagos o envíos esta información se empaqueta en un único archivo ejecutable. De ahí nombrarlo como arquitectura monolítica. Si se realiza un cambio en cualquiera de los módulos, se tendrá que subir a producción un código nuevo aunque los otros módulos no hayan sido modificados. Pero si la aplicación no es compleja, la subida será sencilla ya que en este caso tenemos un solo archivo ejecutable. En el caso de microservicios, en vez de tener un único ejecutable, cada componente del sistema es un único servicio que se comunicará con los otros a través de llamadas. Contamos con una interfaz web en la que interactúan los usuarios y por debajo los servicios de pagos, inventario y envíos. Con respecto a la visión monolítica tenemos: – Los microservicio pueden desplegarse de manera independiente: si realizamos un cambio en uno de los módulos, no se verán afectados los otros módulos y solo tendremos que subir ese módulo. – Es fácil de entender y dividir, pues las secciones del negocio están bien separadas. – Cada microservicio es multifuncional: tiene una parte de base de datos, de backend y es independiente de los demás.

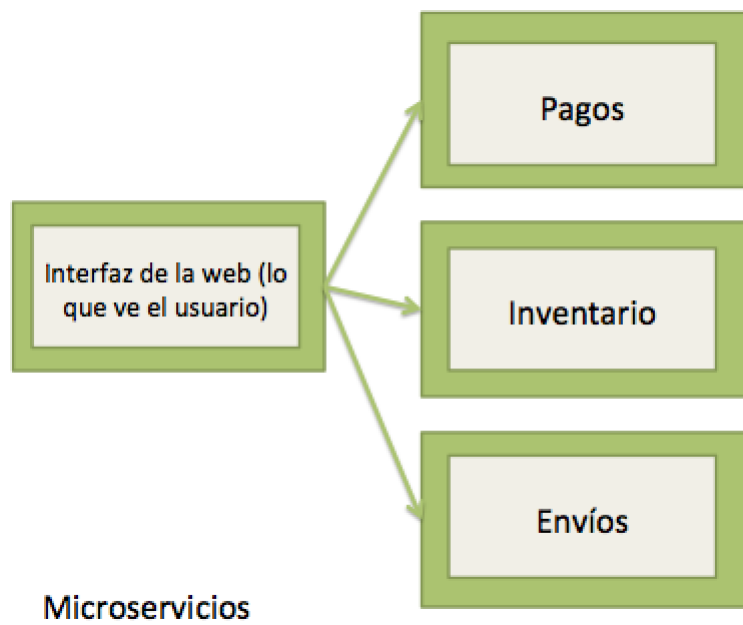


Fig. 2.6. Ejemplo arquitectura microservicios

3. JUSTIFICACIÓN DE LA SOLUCIÓN

Analizando los requisitos que tenemos nos quedaríamos con 3 tipos de arquitecturas a usar, una sería la arquitectura en capas, otra la arquitectura orientada a servicios(SOA) y por ultimo la arquitectura de microservicios.

Con cualquiera de ellas podríamos montar un servicio que hiciera lo que nosotros necesitamos, sin embargo la única que ofrece escalabilidad y alta disponibilidad es la arquitectura de microservicios. Además se adapta perfectamente a la filosofía de la arquitectura, ya que cada servicio tiene que ser pequeño. Al estar en la nube no necesitamos servidores físicos, contratas en cada momento lo que necesitas y se reducen gastos.

Para el desarrollo de la aplicación hemos elegido Spring Boot como framework de microservicios, tiene web para descargar un arquetipo, incluye todas las ventajas de Spring para el desarrollo en capas, que también vamos a necesitar y además es el más utilizado a día de hoy, por lo que es el que más soporte tiene.

Para la gestión de dependencias elegimos Maven ya que es el más rodado y totalmente dirigido para java. Este nos va a empaquetar la aplicación en un jar compilándola y añadiendo las dependencias que necesite.

Vamos a utilizar Lombok para evitar código repetitivo en las clases, como getter, setter y constructores. Mediante anotaciones conseguimos todo esto en tiempo de ejecución. Además también nos ofrece una implementación del patrón de diseño builder.

La comunicación con el microservicio va a ser mediante REST, ya que es un protocolo simple y muy eficaz para realizar las distintas operaciones(verbos) en base de datos: añadir, recuperar, actualizar y eliminar, esto en REST sería PUT, GET, POST, y DELETE.

Todo servicio Rest necesita un documento(API) donde se describa la entrada salida de él, en este caso vamos a utilizar anotaciones de Swagger para que este documento se autogenera, además nos ofrece la posibilidad de llamar a través de él, que es más visual que una llamada por Postman(programa para realizar llamadas rest) por ejemplo.

En la figura 3.1 vemos los métodos a los que podemos llamar y dentro de los métodos tenemos la entrada salida y una descripción de que hace cada uno.

Por otra parte para la base de datos ya que tenemos los datos en formato json lo ideal es una BBDD noSQL para guardar directamente el json sin tener que hacer una conversión a tablas. Además es adaptable a cambios para que sea más mantenible y puede establecerse en la nube al igual que el microservicio para que la comunicación sea muy rápida. Al estar en la nube se adapta fácilmente a los requisitos de cada momento. Por ejemplo si en una campaña se necesita realizar muchas consultas a base de datos se puede ampliar los núcleos de esta para que responda más rápido.

Por tanto elegimos MongoDB como BBDD, pero como estamos usando azure se llama

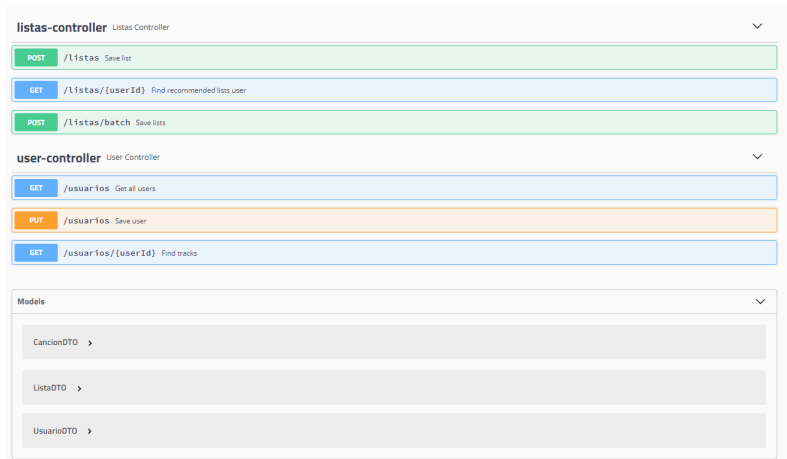


Fig. 3.1. API en swagger

CosmosDB, la única diferencia es que se establece una capa de personalización para que sea mas rápida y limitar el número de accesos en función a lo que se tenga contratado. También está preparada para trabajar con un número elevado de datos sin saturarse.

4. DESARROLLO DE LA APLICACIÓN

El primer paso para desarrollar nuestra aplicación es tener un IDE (Entorno de Desarrollo Integrado) instalado, en este caso usamos Eclipse.

Una vez que tenemos Eclipse instalado nos dirigimos a la web <https://start.spring.io/> para obtener un arquetipo de una aplicación usando el framework Spring Boot. Como observamos en la figura 4.1 elegimos Maven para el control de dependencias, escogemos java como lenguaje y por ultimo las dependencias que queremos incluir, MongoDB, Spring Web, Lombok y nos faltaría las dependencias de swagger pero esta web no las proporciona así que posteriormente las añadiremos.

Importamos en Eclipse como proyecto Maven y al compilar se descargarán todas las dependencias incluidas en el fichero pom.xml, es el momento de añadir las dependencias de swagger que vamos a necesitar.

Swagger es un software para diseñar, documentar y consumir servicios Rest. En la figura 4.2 podemos ver las dependencias necesarias para poder utilizarlo, la primera dependencia es para poder utilizar anotaciones de Swagger, y las dos siguientes son para que al desplegar la aplicación incorpore un recurso donde visualizar el documento Swagger de nuestro código y poder hacer llamadas a nuestros servicios, este documento lo vemos en la figura 3.1

Para configurar Swagger es necesario añadir la clase de configuración de la figura 4.3. Con la anotación `@EnableSwagger2` habilitamos la generación del dominio antes citado. En el método que tenemos configuramos de que paquete queremos que nos cree el API.

Creamos un repositorio en github (<https://github.com/jesusRienda/microservicio>) para poder guardar lo que tenemos hasta ahora. Es recomendable subir al repositorio cada poco tiempo y agrupando una funcionalidad para poder volver a un commit concreto y resolver errores.

El arquetipo contiene una clase App la cual hay que ejecutar para levantar el microservicio en local, para hacerlo al tener una dependencia de MongoDB tenemos que establecer la conexión con la BBDD.

Existen dos opciones: descargar mongo embebido y levantar una base de datos local o nuestro caso, crear en azure una instancia de CosmosDb (capa de personalización sobre Mongo) y configurar el microservicio para que apunte a esa base de datos. Para esta configuración creamos una clase java llamada `MongoDbConfig` y mediante la anotación `@Value` obtenemos de `application.properties` la conexión a cosmos y el nombre de la BBDD. Una vez hecho esto ya podemos levantar el microservicio. Esta clase de configuración la podemos ver en la figura 4.4

Lo siguiente es crear las tres capas que vamos a utilizar, la capa de persistence, donde

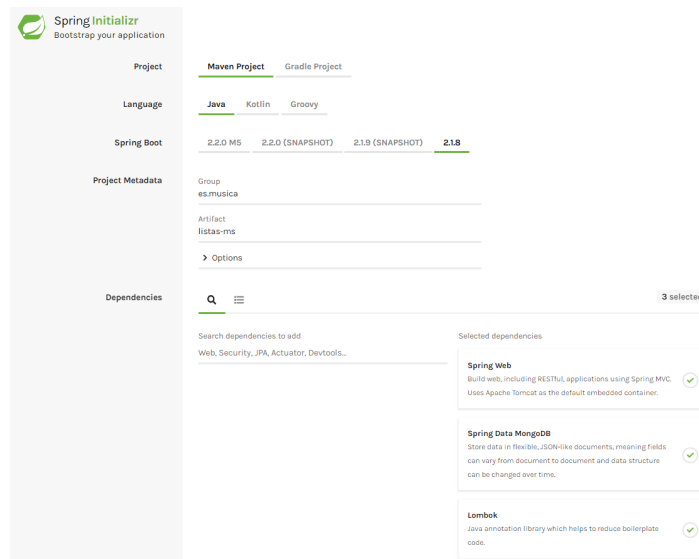


Fig. 4.1. Ejemplo configuración arquetipo

```
<dependency>
  <groupId>io.swagger.core.v3</groupId>
  <artifactId>swagger-annotations</artifactId>
  <version>2.0.7</version>
</dependency>
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.9.2</version>
</dependency>
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.9.2</version>
</dependency>
```

Fig. 4.2. Dependencias Swagger necesarias

```
package es.musica.listasms.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class SwaggerConfig {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.basePackage( "es.musica.listasms" ))
            .paths(PathSelectors.any())
            .build();
    }
}
```

Fig. 4.3. Clase configuración Swagger

```

package es.musica.listasms.persistence.config;

import java.net.UnknownHostException;

@Configuration
@Getter
@Setter
public class MongoDbConfig {
    @Value("${mongodb.datasource.db-name}")
    private String dbName;

    @Value("${mongodb.datasource.uri}")
    private String uri;

    @Bean
    public MongoDBFactory mongoDbFactory() {
        return new SimpleMongoDbFactory(new MongoClient(new MongoClientURI(uri)), dbName);
    }

    public @Bean(name = "mongoTemplate") MongoTemplate getMongoTemplate() throws UnknownHostException {
        MongoTemplate mongoTemplate = new MongoTemplate(mongoDbFactory());
        return mongoTemplate;
    }
}

```

Fig. 4.4. Clase de configuración MongoDB

se definen las queries, la capa de service donde se implementa la lógica de negocio y la capa controller donde se implementan los verbos http mediante los que se va a llamar al servicio. En la figura 4.5 se puede ver la estructura de capas que hemos utilizado.

Una buena práctica a la hora de programar en capas es que los objetos de la capa de persistencia no deben exponerse a la capa de presentación. Por ello hemos creado unos objetos Bean que son: ListaBean, UsuarioBean y CancionBean, estos objetos sirven para mapear el contenido de la BBDD, es decir en la base de datos guardamos un json el cual se transforma en una lista de estos objetos. Para las queries hemos utilizado el framework Spring Data, hemos creado una interfaz por colección en BBDD y dicha interfaz tiene que extender de MongoRepository, solo por ello tiene todos los métodos más usados como: dame todos los elementos, cuenta cuantos elementos hay, inserta un elemento, inserta muchos elementos, actualiza un elemento. etc. Además de ello si en la interfaz definimos un método teniendo en cuenta la sintaxis de Spring Data también funcionará. Por ejemplo en ListasDAO hemos creado el siguiente método:

```
List<ListaBean>findByTracksTrackUriIn(List<String>trackUris);
```

Si lo analizamos vemos que recibe una lista de String y devuelve una lista de ListaBean, el nombre del método define que es lo que va a devolver, find busca dentro del atributo Tracks del ListaBean en el atributo TrackUri que coincida con la lista que entra por parámetro.

Una vez tenemos los métodos necesarios en la capa de persistencia pasamos a la capa de servicio en la que vamos a crear una interfaz con los métodos que vamos a exponer a la capa de presentación y una implementación de esa interfaz. Nuevamente tenemos una interfaz para listas de reproducción y otra para usuarios. En la de usuarios es importante anotar la clase con @Service y los datos a los que vayamos a llamar con @Autowired para que se instancien.

En esta capa se llaman a los métodos necesarios de la capa anterior y además puedes llamar a métodos de un servicio diferente, por ejemplo si llamas al método de listas y le pasas un idUsuario tienes que llamar al servicio de usuarios para que te devuelva las

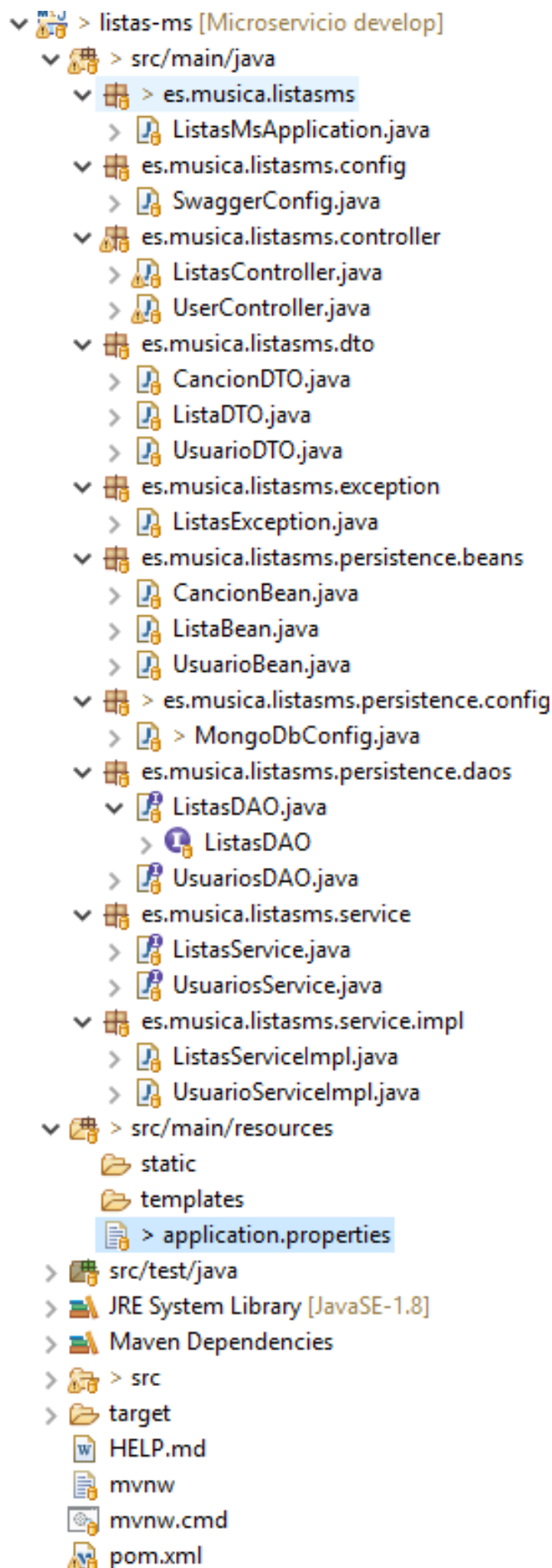


Fig. 4.5. Estructura microservicio

canciones de este usuario y así poder filtrar en las listas por esas canciones, además tiene que hacer el mapeo entre los objetos beans y los objetos DTO (Data Transfer Objet), que son los que se exponen en la salida.

La capa de presentación es la encargada de serializar un json a un objeto y en función del recurso, llamar a un service u otro. En ella se definen los recursos y los verbos de la petición.

5. MARCO REGULADOR

No existen leyes que se apliquen en el desarrollo de un microservicio, prácticamente todos los software que se utilizan son de código libre. Sin embargo si se ha de tratar con datos de clientes reales hay que tener en cuenta dos leyes:

- Ley Orgánica de Protección de Datos Personales y garantía de los derechos digitales [11]: En la ley Orgánica 3/2018, de 5 de diciembre, se establecen los principios de la protección datos personales: exactitud, confidencialidad, consentimiento. Ya que vamos almacenar datos de los usuarios, es necesario que cuando se cree un frontal que llame al servicio de insertar usuarios, antes se de consentimiento al uso de sus datos personales siempre bajo la ley. Además estos datos hay que almacenarlos cifrados.
- Ley General de Telecomunicaciones [12]: En la ley 9/2014, de 9 de mayo, se establece en el artículo 43 que debemos cifrar toda la información que viaje mediante redes de comunicación.

Por otra parte existen lo que se denomina buenas prácticas en el desarrollo, son consejos para que el código sea más limpio, legible, eficiente y mantenible. Estas buenas prácticas se pueden controlar a través de herramientas de calidad de código como Sonar.

6. ENTORNO SOCIO-ECONÓMICO

6.1. Presupuesto

En este punto vamos a calcular los costes del proyecto. Desde los costes de diseño e implementación hasta los recursos necesarios.

6.1.1. Mano de obra

El proyecto se ha desarrollado durante 8 meses con un total de 300 horas empleadas.

Coste/Hora(euros)	Horas	Total(euros)
15	300	4500

TABLA 6.1. COSTE PERSONAL.

Si este proyecto se desarrollara en una empresa, contaría con un arquitecto de sistemas que realice el diseño de la arquitectura y con un desarrollador para la implementación.

6.1.2. Servicios

Hemos necesitado recurrir a una nube de desarrollo donde almacenar nuestra base de datos y para ello hemos utilizado Microsoft Azure. A continuación se muestra el presupuesto que nos ofrece este proveedor para tener una BBDD en la nube con alta disponibilidad. Cuando el servicio afronte menos número de peticiones se reducirá el coste. Al contratar estos servicios, el proveedor nos ofrece soporte gratuito.

Tipo de servicio	Descripción	Coste estimado(euros)
Azure Cosmos DB	Escritura de una sola región - Europa Occidental; Pago por uso; 100 x 100 RU x 730 Horas; 50 GB de almacenamiento	534,67
Total mensual		534,67

TABLA 6.2. COSTES MICROSOFT AZURE.

Para trabajos futuros sería necesario contratar Azure Kubernetes Service (AKS) para poder gestionar los contenedores donde se desplegaría cada microservicio. Esto nos supondría un aumento de 87,6 Euros mensuales.

6.1.3. Coste total

Recopilando los presupuestos de mano de obra y servicios utilizados, el coste total de este proyecto sería el siguiente.

Concepto	Total(euros)
Mano de obra	4500
Servicios	1069,34
Total	5569,34

TABLA 6.3. COSTES TOTALES.

6.2. Impacto socio-económico

Uno de los objetivos de desarrollar aplicaciones con microservicios es que estas sean fácilmente mantenibles y que puedan evolucionar según las necesidades de la sociedad.

Una de las ventajas de implantar los microservicios en la nube es la mejor utilización de los recursos disponibles (energía, servidores) ya que permite aprovechar estos al máximo de una manera eficiente y además se centraliza la ubicación de las máquinas en granjas de servidores, las cuales intentan abastecerse con tecnologías renovables.

Anteriormente, cuando esperabas un número grande de peticiones era necesario adquirir un gran número de servidores para poder dar servicio al pico de trabajo, esto suponía que una vez pasado el pico tenías una serie de recursos sin utilizar, esto lo solucionan los microservicios en la nube.

Por otra parte, un inconveniente de esto es que para una aplicación pequeña que va a contar con mucho tráfico de datos, el coste es mucho mayor en la nube que teniendo un par de servidores.

7. CONCLUSIONES

Hemos conseguido cumplir el objetivo de diseñar una arquitectura capaz de recibir multitud de peticiones.

Para la base de datos elegimos una en la nube, aunque esta es la mejor opción, nos hemos dado cuenta que es muy caro para una pequeña aplicación ya que cobran por uso y almacenamiento. Sin embargo si que hemos podido ver que para el problema que teníamos, la base de datos elegida es ideal.

Implementar el microservicio ha sido una tarea de investigación ya que todas las tecnologías utilizadas son bastante novedosas y es complicado encontrar información al respecto. Pero gracias a ello hemos conseguido una aplicación limpia y fácilmente mantenible.

7.1. Trabajos futuros

A continuación explicaremos los posibles trabajos futuros que podrían implementarse para mejorar o complementar nuestra aplicación.

Aplicación Web

Se podría desarrollar una aplicación web para visualizar los datos de formas más amigable.

Nuevas funcionalidades

Añadir nuevas funcionalidades sobre los datos existentes, como por ejemplo un recomendador de listas.

Integración continua

Se podría añadir la aplicación a un circuito de integración continua para su utilización en una empresa. Pasaría por este circuito cada vez que se hiciera un cambio en una rama de git:

- Compilación con test del código.
- Validación de la calidad del código con una aplicación como por ejemplo Sonar.

Este tipo de circuitos se suelen realizar mediante la herramienta Jenkins

Dockerización

Por otra parte se podría dockerizar (contenerizar) la aplicación para poder integrarla en un contenedor y desplegar el contenedor en la nube.

Gestión de contenedores

Incluir un gestor de contenedores en Azure para gestionar el escalado y otras configuraciones de las aplicaciones que tengamos dockerizadas.

8. ANEXO I: DESPLIEGUE DE LA APLICACIÓN

Para el despliegue de la aplicación en local lo primero es compilar el código mediante maven. Para ello vamos a la ventana Run Configuration en Eclipse y creamos un nuevo Maven build como vemos en la figura 8.1. En base directory hay que poner lo que se muestra en la figura 8.1, en goals vamos a poner clean install para limpiar primero el proyecto y luego compilar, en user settings se puede dejar por defecto si todas las dependencias son de internet o configurar un setting personal donde se establezca a que repositorio ir a buscar las dependencias necesarias en maven runtime se puede utilizar el embebido en eclipse o una versión que nosotros descarguemos.

En la pestaña de JRE es importante seleccionar java 8 o superior y es necesario incluir la JDK y no la JRE como se ve en la figura 8.2

Una vez se tiene todo esto seleccionado se procede a dar a run, se compilara la aplicación desde cero y se pasaran las clases de test que existan, una vez esto pase se va a generar un jar de la aplicación compilada. Este jar se ubica en la carpeta target.

Para levantar la aplicación es necesario tener la base de datos a la que se va a conectar levantada.

De nuevo en la pantalla Run Configuration pero esta vez creamos una nueva configuración de Java Application como se muestra en la figura 8.3. En project se selecciona la raíz del proyecto y en main class seleccionamos nuestra clase ListasMsApplication.java que es la que contiene el main. Al dar a run se va a desplegar la aplicación.

Para acceder a ella se accede desde <http://localhost:8080> y se añade el recurso al que quieras acceder, si es un recurso get se puede acceder desde un navegador, sino hay que hacerlo mediante swagger o postman. Todos los recursos están creados en las clases controller.

Ademas de los recursos creados por nosotros, se crean dos recursos de swagger que son `/swagger-ui.html` y `/v2/api-docs`; el primero es la parte visual de swagger y la segunda es el fichero `.yaml` con el cual se genera la parte visual de swagger.

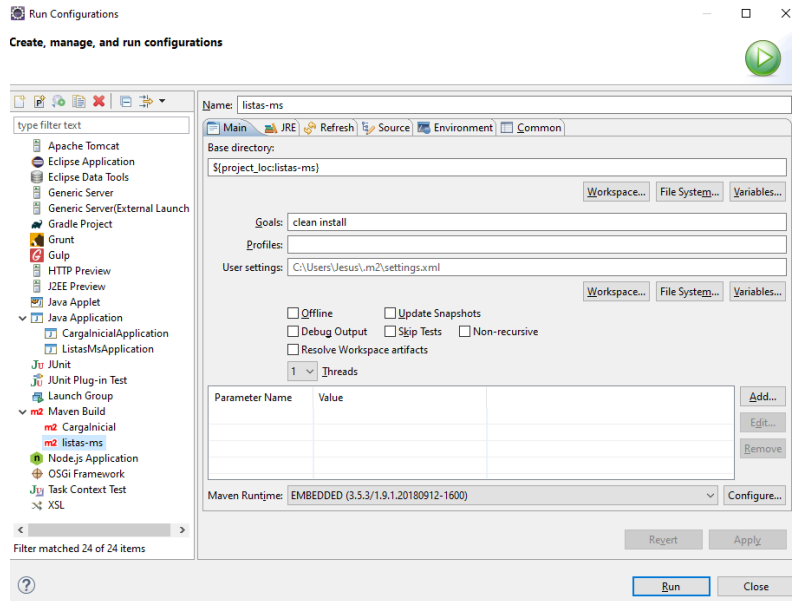


Fig. 8.1. Configuración compilación mediante Maven

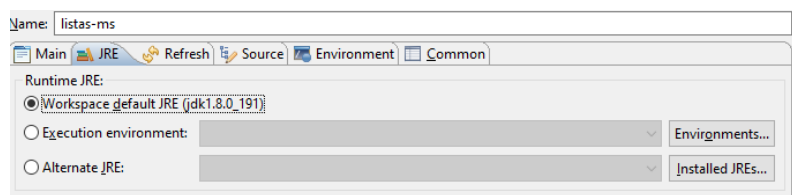


Fig. 8.2. Configuración JRE proyecto

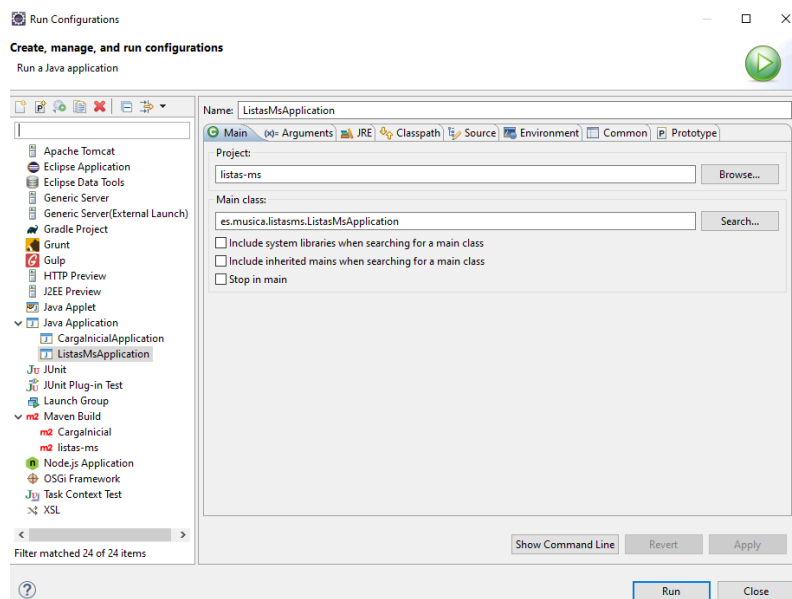


Fig. 8.3. Configuración Run despliegue

9. ANEXO II: DESCRIPCIÓN API MICROSERVICIO

Hemos desarrollado una serie de servicios en cada controlador, estos pueden verse en las figuras 9.1 y 9.2.

En el controlador de listas hemos creado un recurso `/listas` al que es posible acceder mediante el verbo Post y que su función es almacenar en BBDD una lista de reproducción nueva. El json de entrada necesario es el de la figura 9.3.

Este corresponde a una lista de reproducción con una única canción. Los campos que aparecen son los proporcionados por Spotify, simplemente se han renombrado para evitar el uso de barra baja.

Al llamar al servicio con el json creará un nuevo documento en BBDD.

En el recurso `/listas/{userId}` mediante el verbo Get, al pasarle el id de usuario y un número de canciones a coincidir, nos devuelve un listado en formato json con las listas de reproducción en las que el usuario tiene al menos el número antes indicado de canciones en común con la lista.

Por último en este controlador hemos añadido un recurso `/listas/batch` su función es la misma que la de `/listas` pero en la entrada recibe una lista de listas de reproducción y las da de alta en BBDD.

En el controlador de usuarios hemos creado el recurso `/usuarios` al que es posible acceder mediante el verbo Get y nos devuelve un json con todos los usuarios dados de alta en BBDD.

Hemos creado otro recurso en `/usuarios` esta vez con verbo Put para crear un usuario con las canciones que ha escuchado en BBDD. El json de entrada sería el de la figura 9.4.

Por último hemos creado un recurso `/usuarios/{userId}` con verbo Get el cual al pasarle el id de usuario nos devuelve ese usuario con las canciones que ha escuchado.

listas-controller		Listas Controller
POST	<code>/listas</code> Save list	
GET	<code>/listas/{userId}</code> Find recommended lists user	
POST	<code>/listas/batch</code> Save lists	

Fig. 9.1. Recursos disponibles dentro del controller Listas

user-controller User Controller	
GET	/usuarios Get all users
PUT	/usuarios Save user
GET	/usuarios/{userId} Find tracks

Fig. 9.2. Recursos disponibles dentro del controller Usuarios

```
{
  "name": "Throwbacks",
  "collaborative": "false",
  "pid": 0,
  "modifiedAt": 1493424000,
  "numTracks": 52,
  "numAlbums": 47,
  "numFollowers": 1,
  "tracks": [
    {
      "pos": 0,
      "artistName": "Missy Elliott",
      "trackUri": "spotify:track:0UaMYEvWZi0ZqiDOoHU3YI",
      "artistUri": "spotify:artist:2wIVse2owClT7golWT98tk",
      "trackName": "Lose Control (feat. Ciara & Fat Man Scoop)",
      "albumUri": "spotify:album:6vV5UrXcfyQDlWu4Qo2I9K",
      "durationMs": 226863,
      "albumName": "The Cookbook"
    }
  ],
  "numEdits": 6,
  "durationMs": 11532414,
  "numArtists": 37
}
```

Fig. 9.3. Json ejemplo lista reproducción

```
{
  "name": "Paco Sanchez Ruiz",
  "tracks": [
    {
      "pos": 0,
      "artistName": "Missy Elliott",
      "trackUri": "spotify:track:0UaMYEvWZi0ZqiDOoHU3YI",
      "artistUri": "spotify:artist:2wIVse2owClT7golWT98tk",
      "trackName": "Lose Control (feat. Ciara & Fat Man Scoop)",
      "albumUri": "spotify:album:6vV5UrXcfyQDlWu4Qo2I9K",
      "durationMs": 226863,
      "albumName": "The Cookbook"
    }
  ],
  "userId": "pSanchez"
}
```

Fig. 9.4. Json ejemplo usuario

BIBLIOGRAFÍA

- [1] S. E. Institute, “Software Architecture”, 2017. [En línea]. Disponible en: https://www.sei.cmu.edu/research-capabilities/all-work/display.cfm?customel_datapageid_4050=21328.
- [2] C. Alexander, S. Ishikawa y M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.
- [3] E. Gamma, *Patrones de diseño: elementos de software orientado a objetos reusable*, ép. Addison-Wesley professional computing series. Pearson Educación, 2002. [En línea]. Disponible en: https://books.google.es/books?id=gap_AAAACAAJ.
- [4] “APLICACIONES HÍBRIDAS: ¿QUÉ SON Y CÓMO USARLAS?”, [En línea]. Disponible en: <https://www.nextu.com/blog/aplicaciones-hibridas-que-son-y-como-usarlas/>.
- [5] G. Peiretti, “Strategy Pattern con Spring Boot”, 2017. [En línea]. Disponible en: <https://experto.dev/strategy-pattern-spring-boot/>.
- [6] E. F. Codd, “A Relational Model of Data for Large Shared Data Banks”, *Commun. ACM*, vol. 26, n.º 1, pp. 64-69, ene. de 1983. doi: 10.1145/357980.358007. [En línea]. Disponible en: <http://doi.acm.org/10.1145/357980.358007>.
- [7] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 1st. O’Reilly Media, 2015, p. 280.
- [8] L. Mauersberger, “A brief history of microservices”, *blog.leanix.net*, 2017. [En línea]. Disponible en: <https://blog.leanix.net/en/a-brief-history-of-microservices>.
- [9] N. Dragoni et al., “Microservices: Yesterday, Today, and Tomorrow”, en *Present and Ulterior Software Engineering*, M. Mazzara y B. Meyer, eds. Cham: Springer International Publishing, 2017, pp. 195-216. doi: 10.1007/978-3-319-67425-4_12. [En línea]. Disponible en: https://doi.org/10.1007/978-3-319-67425-4_12.
- [10] R. RV, *Spring Microservices*. Packt Publishing, 2016. [En línea]. Disponible en: <https://books.google.es/books?id=pwNwDQAAQBAJ>.
- [11] J. del Estado, “Ley Orgánica 3/2018, de 5 de diciembre, de Protección de Datos Personales y garantía de los derechos digitales.”, 2018. [En línea]. Disponible en: <https://www.boe.es/eli/es/lo/2018/12/05/3>.
- [12] ———, “Ley 9/2014, de 9 de mayo, General de Telecomunicaciones.”, 2014. [En línea]. Disponible en: <https://www.boe.es/eli/es/l/2014/05/09/9>.