uc3m | Universidad Carlos III de Madrid

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 Peláez Moreno

Leganés, 2019

DEDICATORIA

RESUMEN

En este trabajo fin de grado se ha realizado un estudio sobre las posibilidades existentes en el desarrollo de aplicaciones web, desde el diseño partiendo de cero hasta el uso de software de terceros como son los frameworks que facilitan el desarrollo y se centran 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 demostrar 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.

Abstract: This project makes a study of the different ways of developing a web application, from the design to the use of others software such as frameworks which make easier development and is focus in the functionality.

To get to the best solution to our architecture it is necessary to investigate each part of the architecture. First, we do some researches of the design patterns and their functionalities; then, the study of different types of architecture, after that the study of frameworks and finally about the database.

We have done a microservice based in the Spotify reproduction liss to demonstrate the project. To implement this project, the latest development technologies such as Spring Boot, Spring Data and Mondongo DB has been used. This microservice acts as an intermediary, filtering the reproduction lists in which you can find a song that the user has listen.

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

GLOSARIO

AKS: Corresponde a Azure Kubernetes Service. Es un gestor de aplicaciones en contenedores [1].

API: Corresponde a Application Programming Interface que en castellano lo traduciríamos como interfaz de programación de aplicaciones. Es un documento en el cual se define la entrada y la salida de cada recurso creado.

App: Corresponde a Application y en castellano lo utilizamos para referirnos a aplicación informática, web o móvil. Es un programa diseñado para que el usuario pueda realizar varios tipos de trabajos.

Azure: Es una nube de pago por uso, en ella se permite desplegar aplicaciones contenerizadas y ofrece servicios como el de bases de datos [2].

Back-end: Es la parte del desarrollo de software que corresponde al acceso y tratamiento de datos según la lógica de negocio.

Controller: Son las clases de java que pertenecen a la capa de presentación y en ellas se definen los recursos y los verbos disponibles.

CosmosDB: Es un servicio de base de datos multimodelo distribuido proporcionado por Azure.

DAO: Corresponde a Data Access Object. Es un patrón que separa la lógica de acceso a datos de la lógica de negocio.

Docker: Es una aplicación de código abierto para automatizar el despliegue de aplicaciones dentro de contenedores software [3].

Escalabilidad: Es la propiedad que tiene una aplicación para poder desplegarse en varias máquinas a la vez.

Framework: En castellano podría traducirse como marco de trabajo, aunque este término no se utiliza. Es una estructura definida para facilitar la solución a un determinado tipo de problema que facilita el desarrollo de la aplicación.

Front-end: Es la parte del desarrollo de software con la que interactúa el cliente y el Back-end.

GB: Es una unidad de medida de almacenamiento.

Getter, setter: Son los métodos necesarios en los objetos de Java para poder acceder desde otra clase a sus atributos privados. Getter sirve para que te devuelva el valor del atributo y setter para cambiar el valor.

Git: Es un software para el control de versiones.

Github: Es una plataforma de almacenamiento colaborativo para alojar proyectos

utilizando Git. Nuestro proyecto se encuentra alojado aquí [4].

Gradle: Es una herramienta gestionar dependencias en proyectos de desarrollo.

Groovy: Lenguaje de programación orientado a objetos basado en la plataforma de Java.

HTTP: Corresponde a Hypertext Transfer Protocol o es castellano Protocolo de transferencia de hipertexto. Es un protocolo de transmisión de información en la web.

JAR: Corresponde a Java Archive. Es un tipo de archivo en el que se incluye el código Java compilado.

Java: Es un lenguaje de programación tipado orientado a objetos para el desarrollo de aplicaciones Back-end.

Javascript: Es un lenguaje de programación no tipado orientado al desarrollo de aplicaciones Front-end.

Jenkins: Es un servidor de automatización para tareas como integración continua y despliegue [5].

JDK: Corresponde a Java Development Kit o es castellano Herramientas de desarrollo para Java. Es un software que contiene las herramientas necesarias para desarrollar y ejecutar programas en Java.

JRE: Corresponde a Java Runtime Environment o en castellano Entorno de Ejecución de Java. Es un conjunto de librerías de Java necesarias para la ejecución de aplicaciones. Está compuesto por la máquina virtual de Java.

JSON: Corresponde a JavaScript Object Notation o en castellano Notación de Objeto de JavaScript. Es un estándar de texto básico utilizado para el intercambio de información.

Lombok: Es una librería para Java que mediante anotaciones nos reduce el código repetitivo [6].

Maven: Es un instrumento que se utiliza para crear y gestionar proyectos en Java y sus dependencias.

MongoDB: Es una base de datos NoSQL orientada a documentos.

NoSQL: Corresponde a Not Only Structured Query Language o en castellano No Solo Lenguaje de Consulta Estructurada. Es el lenguaje informático utilizado para gestionar bases de datos sin estructura fija.

Nube: Es el nombre que se corresponde con el almacenamiento masivo de datos en servidores de internet.

RAML: Corresponde a Restful Api Modeling Language en castellano sería lenguaje para el modelado de APIs REST.

REST: Corresponde a Representational State Transfer en castellano se traduce como transferencia de estado representacional. Es el encargado de conectar varios sistemas

mediante el protocolo HTTP. Se apoya en los verbos **GET**, **PUT**, **POST** y **DELETE**.

RU: Corresponde a Request Units, en castellano sería unidades de solicitud. Es una unidad de medida creada por Azure para medir el número de peticiones y uso de almacenamiento que se consume por hora. Está se utiliza en CosmosDB.

SQL: Corresponde a Structured Query Language o en castellano Lenguaje de Consulta Estructurada. Es el lenguaje informático utilizado para gestionar bases de datos relacionales.

SOA: Corresponde a Service Oriented Architecture o en castellano Arquitectura Orientada a Servicios.

Sonar: Es un analizador de la calidad de código.

Swagger: Es un framework para documentación de APIs REST [7]. Se puede integrar en proyectos y autogenerar la documentación.

Typescript: Es un lenguaje de programación creado por Microsoft, basado en JavaScript y orientado a desarrollo de Front-end.

UI: Corresponde a User Interface o en castellano interfaz de usuario. Es el medio por el cual un usuario se comunica con una máquina o dispositivo.

WAR: Corresponde a Web Application Archive, en castellano significa archivo de aplicación web. Es un archivo que contiene dentro un archivo JAR, una configuración y unos archivos web.

XML: Corresponde a eXtensible Markup Language o en castellano lenguaje de marcado extensible.

YML: También llamado YAML, es un formato de serialización de datos. Sus siglas corresponden a Ain't Markup Language, en castellano sería que no es un lenguaje marcado.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

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, pero realmente ha surgido debido a la aparición de la nube donde todas estas empresas se están llevando sus softwares.

En la nube no se necesitan servidores físicos con el mantenimiento y la inversión que esto supone, sino que solo se paga por los recursos que se necesitan en cada momento. Esto ha sido una gran revolución para la informática, ya que 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, estos contienen información de cada una de las listas y de 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, ya que este no tiene por qué tener conocimientos técnicos sobre el lenguaje de gestión de la base de datos. Por tanto, tendremos que crear un programa que haga las consultas a la base de datos y devuelva los datos validados en un formato óptimo para mostrarlos por pantalla. Al invocarlo, nos devolverá 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 tratamientos necesarios sería filtrar las listas en función de las canciones que escuche cada usuario, si ha escuchado más de 3 canciones de una lista, deberíamos devolver dicha lista.

Este programa tendrá que tener una alta disponibilidad y escalarse cuando sea necesario para siempre tener unos tiempos de respuesta bajos, ya que existe la posibilidad de que gran cantidad de personas consuman 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 de dar servicio a nadie.

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

Implementar un microservicio que, al enviarle un usuario y un número de canciones a coincidir, devuelva las listas de reproducción que contengan, al menos, el número de canciones que coincidan con las escuchadas por el usuario.

1.3. Solución propuesta

Una vez analizadas las opciones posibles, adoptamos como solución a nuestro problema el diseño de una arquitectura de microservicios. Para ello implementamos un microservicio que se va a desarrollar con Java mediante el framework Spring Boot. La base de datos estará almacenada en Azure y será CosmosDB. Para llamar al microservicio creado utilizaremos Swagger. En los siguientes capítulos describiremos y discutiremos estas opciones.

1.4. Estructura trabajo

Este trabajo está estructurado de la siguiente forma:

- **Capítulo 1: Introducción.** En este capítulo se explica el problema al que nos enfrentamos, los objetivos que tiene que cumplir el trabajo y la solución propuesta a dicho problema.
- **Capítulo 2: Estado del arte.** En este capítulo se analiza el estado hoy en día del diseño de aplicaciones desde cero y en profundidad de microservicios.
- **Capítulo 3: Justificación de la solución.** En este capítulo se analiza que es lo que más conveniente para nuestros requerimientos según lo estudiado en los capítulos anteriores.
- **Capítulo 4: Desarrollo de la aplicación.** En este capítulo explicamos desde cero como ha sido desarrollado el microservicio, detallando los programas y configuraciones que se han utilizado durante el proceso.
- **Capítulo 5: Descripción API microservicio** En este capítulo se detallan los recursos y servicios que hemos creado en nuestra aplicación.
- **Capítulo 6: Despliegue de la aplicación** En este capítulo vamos a explicar mediante un tutorial como desplegar nuestra aplicación en local.
- **Capítulo 7: Marco regulador.** En este capítulo detallamos la regulación existente en el ámbito de los microservicios, mencionando las leyes a cumplir en casos específicos.
- **Capítulo 8: Entorno socio-económico.** En este capítulo recreamos el coste total que tendría el proyecto y el impacto que este supondría para la sociedad.

- **Capítulo 9: Conclusiones.** En este capítulo describimos las conclusiones sacadas a raíz de la realización del trabajo y las posibles tareas a realizar en un futuro.
- **Anexo: Project summary.** En este anexo realizamos un resumen en inglés, manteniendo la estructura del trabajo.

2. ESTADO DEL ARTE

En la década de los 60 surgió lo que hoy en día conocemos como arquitectura de software, esta fue tomando cada vez más interés hasta que en la década de 1980 se integró 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 de 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."[8].

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 sí. Por ejemplo, en la figura 2.1 podemos ver la estructura de un sistema basado en microservicios donde aparecen bases de datos, que serían un módulo donde habría que definir sus propiedades, tablas, colecciones de datos, etc.; para cada microservicio habría que definir un documento API en el que se indique la entrada y salida del módulo, el lenguaje en el que se va a desarrollar, etc.; por último tendríamos un módulo de presentación UI.

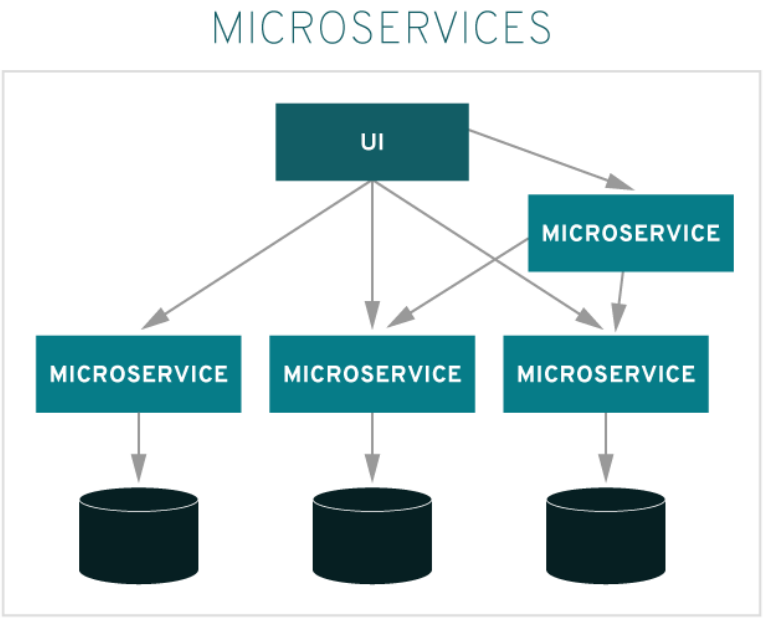

Fig. 2.1. Estructura arquitectura microservicios

2.1. Patrones de diseño

En el momento de diseñar 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" [9], 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 [10].

Los patrones definidos en el libro [10] se dividen en función del problema a resolver:

- Patrones creacionales: éstos 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 él.
    - Prototype: Un objeto se crea a partir de la clonación de otro objeto, es decir, se crea basándose en unas plantillas.

- Patrones estructurales: éstos nos facilitan la modelización de nuestro software, definiendo la forma en que las clases se relacionan entre sí. 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 que puedan evolucionar independientemente.
    - Composite: Sirve para crear objetos contenidos en un árbol 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, es decir, no especifican cómo escribir el código, pero sí cómo debe estar organizado y que elementos son importantes en la aplicación. 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: Utilizada para estructurar programas que pueden descomponerse en subtareas. 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 hoy en día.
- Cliente-servidor: Este patrón consta de dos partes: un servidor y múltiples clientes, el servidor será el que dé 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 tratado 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 simultánea, 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.

Siempre que se desarrolla un servicio web es necesario establecer un protocolo de comunicación con la aplicación, las dos más importantes son SOAP y REST. SOAP es un protocolo que establece que el intercambio de mensajes se va a realizar por medio de XML. REST es un protocolo basado en HTTP que para el intercambio de mensajería utiliza JSON o XML, aunque es más habitual usarlo con JSON ya que es más rápido.

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.

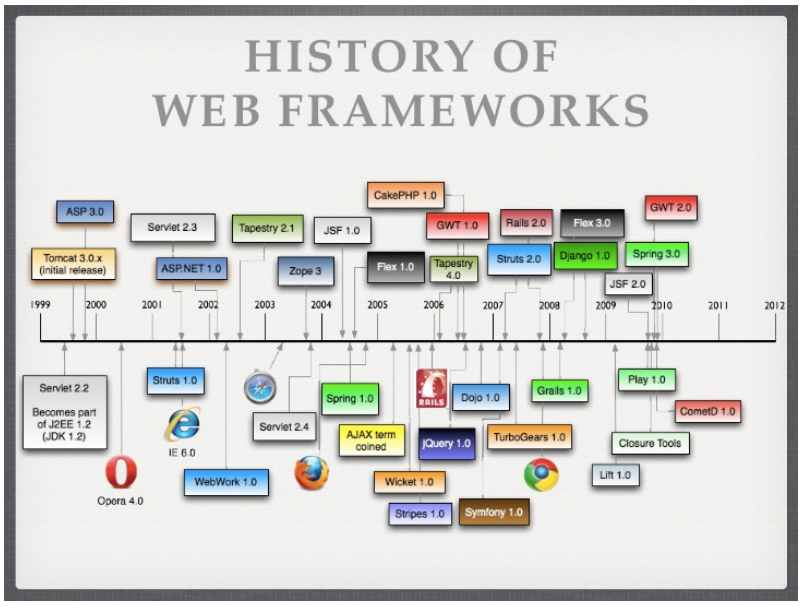

Fig. 2.2. Historia de los Web Frameworks

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.

Struts nació en 2001 y está basado en la arquitectura 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 más fácilmente. Éste es propiedad de Facebook.

Ionic surgió en 2013, siempre ha ido de la mano con Angular, desde AngularJs hasta las últimas versiones de este. Ionic está diseñado para desarrollar aplicaciones híbridas, según este artículo de Next-u [11] “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 frameworks que se basan en Java y ofrecen una arquitectura de microservicios, éstos han surgido de las grandes empresas del sector como Oracle, Spring, Eclipse. Entre ellos encontramos:

- Spring Boot: Es una capa de personalización sobre el famoso framework Spring, que ofrece un arquetipo de un microservicio básico, ampliable fácilmente con poco desarrollo. Por ejemplo, Gustavo Peiretti en el artículo [12], 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. Pertenece a la compañía Oracle que lo utilizó internamente bajo el nombre "Java for Cloud" y 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, éstas 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 sus 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” [13].

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 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, disparadores 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 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 ésta realice algún cambio en sus sistemas informáticos.
- No garantizan el buen funcionamiento si el sistema operativo donde se va a instalar no cumple 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 reformula 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, ésta 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”, [14]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 cada módulo, 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.

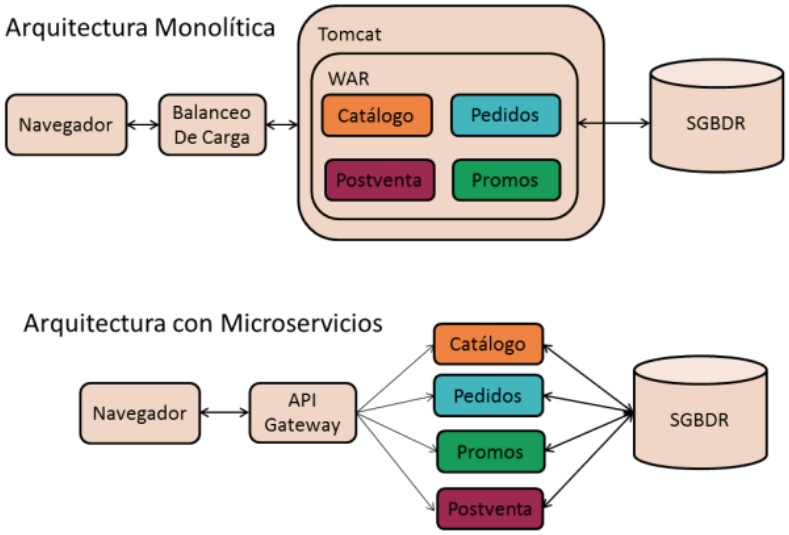

Fig. 2.3. Arquitectura monolítica frente microservicios

- **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.

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. Con ésta se desarrollaban todos los servicios necesarios y se empaquetaban en un archivo WAR para poder desplegarse en un servidor de aplicaciones, como por ejemplo 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 archivo WAR en varias máquinas o réplicas de esta. 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 redesplegar 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[15] se refirió a ello para describir la arquitectura que estaban usando grandes empresas como Netflix,

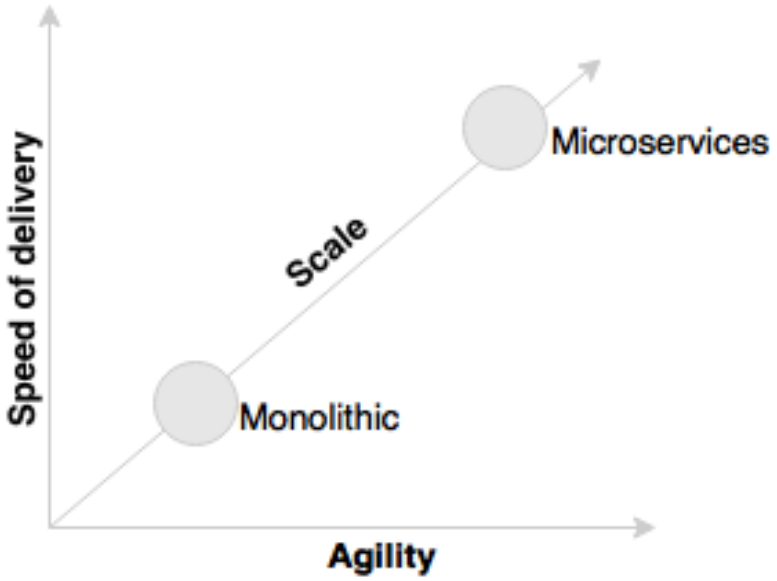

Fig. 2.4. Eficiencia microservicios frente SOA

Facebook, Amazon o PayPal.

Los microservicios gestionan la complejidad dividiéndose funcionalmente en un conjunto de servicios pequeños e independientes. Con esto se consigue que el equipo de 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 pueden o no tener una base de datos común.

El cambio más notable respeto 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 articulo [16] y aún le queda mucho por mejorar y evolucionar.

Rajest RV en su libro “Spring Microservices” [17] nos muestra el gráfico 2.4 en el que se observa cómo 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.".

Un aspecto que mejorar es la seguridad, ya que cuando se descompone una aplicación en cientos de microservicios aparecen dificultades en la depuración, monitoreo, auditoría y análisis de toda la aplicación. Los atacantes podrían aprovechar esta complejidad 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 el nombrarlo 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. 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 microservicios 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 Back-end y es independiente de los demás.

2.7. Herramientas de desarrollo

Una vez la arquitectura está montada, es decir una vez se han elegido todas sus propiedades y relaciones, entran en juego las herramientas de desarrollo. Éstas están enfocadas en un lenguaje de programación en particular.

El número de dependencias en los proyectos se ha disparado en los últimos tiempos, esto hace que cada vez sea más complicado mantener actualizadas dichas dependencias. Para facilitar la tarea de instalación y actualización de dependencias surgieron los gestores de dependencias.

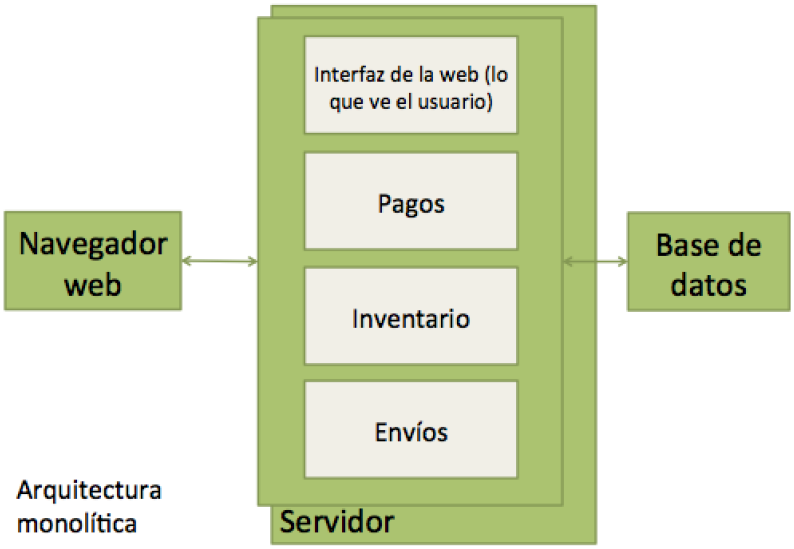

Fig. 2.5. Ejemplo de arquitectura monolitica (SOA)

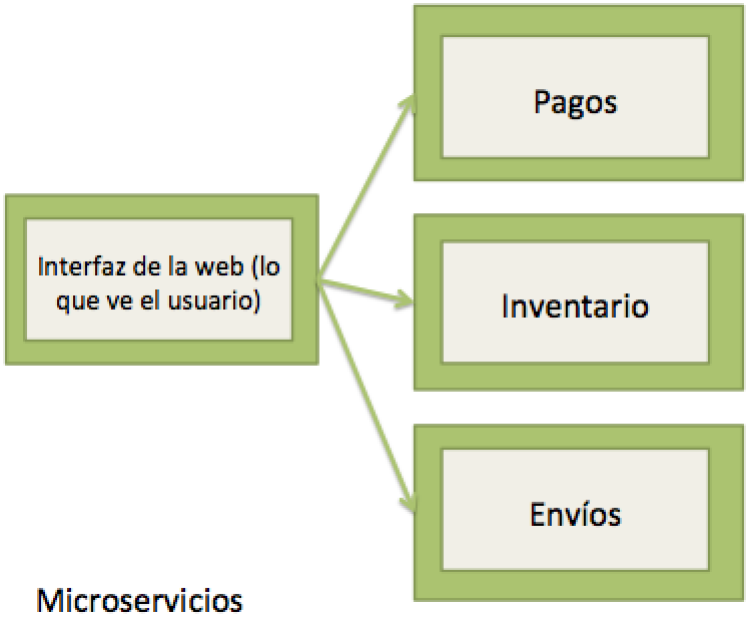

Fig. 2.6. Ejemplo arquitectura microservicios

Por otra parte, cuando se desarrolla siempre es necesario evitar el código duplicado para esta cuestión existen patrones de diseño como el Strategy, antes mencionado, que al ejecutar acciones en tiempo de ejecución, consigue reducir el código duplicado.

Existen una serie de herramientas orientadas a crear un documento API de un servicio web. Swagger es un ejemplo de ello, es un framework para la documentación de APIs REST y además es capaz de autogenerarse por medio de anotaciones en el código. Otra de las herramientas más utilizadas, aunque de reciente incorporación es RAML se trata de un lenguaje para el modelado de documentación.

2.7.1. Herramientas de desarrollo específicas de Java

Para la gestión de dependencias en proyectos Java hay dos gestores muy utilizados: Maven y Gradle. Maven se basa en ficheros de XML y tiene como objetivos la compilación del código y la generación del proyecto empaquetado en JAR. Gradle se basa en el lenguaje Groovy, comparte objetivos con Maven y tiene el objetivo de poder ser utilizado en lenguajes diferentes de Java.

Existe una librería creada para Java para la eliminación de código repetitivo en objetos y es Lombok. Ésta se basa en generar el código en tiempo de ejecución con el ahorro de código repetitivo que esto supone. Lombok no tiene alternativa, ya que no existe nada más parecido.

3. JUSTIFICACIÓN DE LA SOLUCIÓN PROPUESTA

Tenemos datos sobre listas de reproduccion y usuarios almacenados en unos ficheros de texto, para poder explotar estos datos es necesario almacenarlos en una base de datos, ya que la única forma de consultar los datos en el fichero sería de forma manual. El formato de estos datos es JSON y existen bases de datos capaces de explotar datos en este formato sin necesidad de realizar una conversión.

Como el cliente no va a tener un acceso directo a la base de datos, es necesario crear una interfaz que haga de intermediario entre la base de datos y el usuario.

Las acciones que va a realizar el cliente son:

- Consultar todos los datos de usuarios dados de alta.
- Consultar las canciones que ha escuchado un usuario.
- Dar de alta un usuario con las canciones que ha escuchado.
- Dar de alta nuevas listas de reproducción.
- Consultar las listas de reproduccion en las que un usuario tenga al menos un numero de canciones en común. Este número lo elegirá en cada ocasión el cliente.

El cliente necesitará en determinados momentos realizar consultas masivas al programa, por lo que es necesario que nuestra aplicación soporte ésto con unos tiempos de respuesta bajos.

En un futuro el cliente necesitará incluir nuevas consultas que vayan surgiendo, por lo tanto es necesario que el código sea fácil de actualizar.

Analizando el problema planteado obtenemos los siguientes requisitos a cumplir:

- Almacenar los datos en formato JSON en una base de datos.
- Crear un programa que nos permita realizar consultas a la base de datos.
- Filtrar los datos de las listas o de los usuarios antes de devolverlos.
- Crear un programa sencillo y fácilmente mantenible.
- Crear una interfaz que le facilite al usuario realizar sus consultas.
- Obtener unos tiempos de respuesta bajos.
- Tener una alta disponibilidad y la posibilidad de escalarse.

Hay tres tipos de arquitecturas candidatas que cumplen los primeros cinco requisitos. Una de ellas sería la arquitectura en capas, otra la arquitectura orientada a servicios (SOA) y por último la arquitectura de microservicios.

Con cualquiera de ellas podríamos crear un programa que filtrara las listas de reproducción en base de datos de una forma sencilla para que el usuario viera esta información a través de una interfaz, 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, se contratará en cada momento lo que se necesite y así se reducen gastos.

Lo ideal sería crear una estructura de tres capas para separar el acceso a los datos, la lógica de negocio y la presentación de los datos al usuario. La herramienta que mejor se adapta a esta estructura es Spring ya que se encarga de la configuración de cada capa evitándonos la creación de ficheros XML.

Para el desarrollo de la aplicación hemos elegido Spring Boot como framework de microservicios. Éste cuenta con una web donde poder descargar un arquetipo e incluye todas las ventajas de Spring para el desarrollo en capas que vamos a necesitar como hemos mencionado anteriormente. Además, es el framework más utilizado a día de hoy, por lo que es el que más soporte nos ofrece.

Para cualquier proyecto de desarrollo es necesario un software para la gestión de dependencias y así evitarnos tener que incluirlas de forma manual, para esta gestión elegimos Maven ya que es el más utilizado y totalmente dirigido para Java. Éste nos va a empaquetar la aplicación en un archivo JAR compilándola y añadiendo las dependencias que necesite.

Para conseguir un código limpio y mantenible podemos utilizar herramientas que eviten la repetición de código como Lombok que mediante anotaciones consigue generar en tiempo de ejecución los métodos getter, setter y los constructores.

Como queremos reducir la complejidad del código recurrimos a los Stream de Java 8 para que las interacciones y búsquedas en las listas de datos sean más eficientes y el código quede más limpio.

La comunicación con el microservicio va a ser mediante REST que está basado en el protocolo HTTP que es 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 y salida de él, en este caso vamos a utilizar anotaciones de Swagger para que este documento se autogenere, además nos ofrece la posibilidad de llamar a través de él, que es más visual, justo lo que el cliente necesita.

En la figura 3.1 vemos los métodos a los que podemos llamar y dentro de los métodos

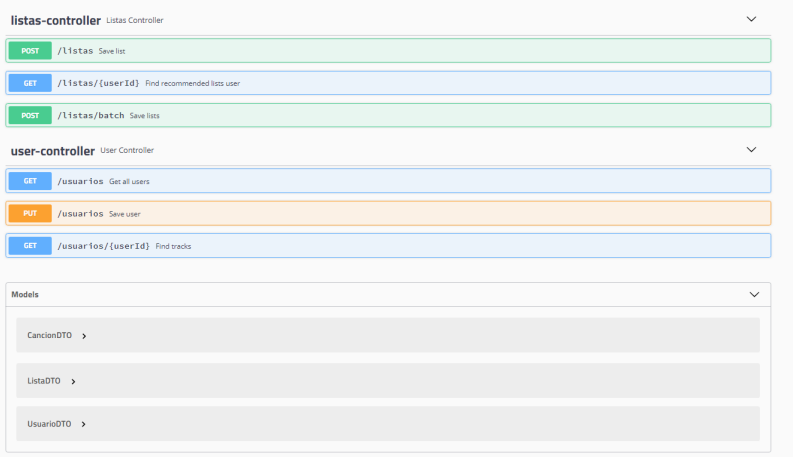

Fig. 3.1. API en Swagger

tenemos la entrada salida y una descripción de que hace cada uno.

Por otra parte, para la base de datos, como tenemos los datos en formato JSON, lo ideal es una base de datos 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 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 base de datos que al estar utilizando Azure se llama CosmosDB, la única diferencia es que se establece una capa de personalización para que sea más rápida, distribuyendo los datos entre varios servidores y limita 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.

Para que las consultas a base de datos queden más limpias utilizaremos Spring Data. Además, si en algún momento se cambiara a una base de datos SQL no habría que modificar las consultas ya que Spring Data se adapta a cualquier base de datos.

4. DESARROLLO DE LA APLICACIÓN

El primer paso para desarrollar nuestra aplicación es tener un IDE instalado, en este caso usamos Eclipse.

Una vez que tenemos Eclipse instalado, nos dirigimos a la web de start Spring [18] para obtener un arquetipo de 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 último las dependencias que queremos incluir, MongoDB, Spring Web, Lombok; nos faltarían las dependencias de Swagger, pero esta web no las proporciona así que posteriormente las añadiremos.

Importamos en Eclipse el arquetipo descargado como proyecto Maven y al compilarlo 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 se incorpore un recurso donde visualizar el documento Swagger de nuestro código y así 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 que se muestra en la figura 4.3. Con la anotación de la clase @EnableSwagger2 habilitamos la generación del dominio antes citado y además esta clase contiene un método donde configuramos de que paquete queremos que nos cree el API.

Creamos un repositorio en Github [4] para poder guardar lo que tenemos hasta ahora. Es recomendable ir subiendo 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 base de datos.

Existen dos opciones: descargar MongoDB embebido y levantar una base de datos local o nuestro caso, crear en Azure una instancia de CosmosDB y configurar el microservicio para que apunte a esa base de datos. Para esta configuración creamos una clase de Java llamada MongoDbConfig y mediante la anotación @Value, obtenemos de application.properties la conexión a CosmosDB y el nombre de la base de datos. Una vez hecho esto ya podemos levantar el microservicio. Esta clase de configuración la podemos ver en la figura 4.4

El siguiente paso consiste en crear las tres capas que vamos a utilizar; la capa de

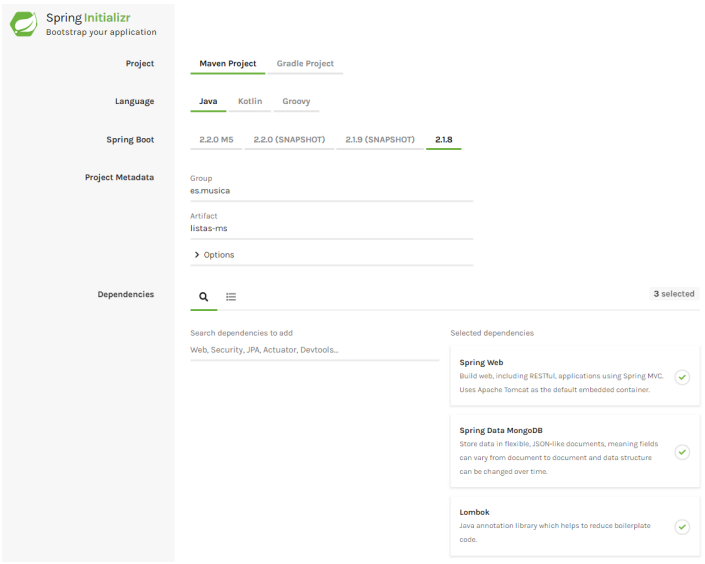

Fig. 4.1. Ejemplo configuración arquetipo

```xml
<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

```java
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

```java
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

persistencia, donde se definen las consultas a base de datos, la capa de servicio donde se implementa la lógica de negocio y la capa de presentación donde se implementan los verbos HTTP con 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, que sirven para mapear el contenido de la base de datos, es decir en la base de datos guardamos un JSON que se transforma en una lista de estos objetos. Para las consultas a base de datos hemos utilizado el framework Spring Data, hemos creado una interfaz por colección en base de datos y dicha interfaz tiene que extender de MongoRepository, solo por ello tiene todos los métodos más usados como, por ejemplo: 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, Spring data transforma el nombre del método en una consulta a base de datos. Con la palabra find indicamos que la operación que se va a realizar en base de datos es una búsqueda. ByTracks nos indica que la búsqueda se va a hacer en el objeto ListaBean en el campo tracks, que es una lista de CancionBean; a continuación, TrackUriIn va a buscar el atributo trackUri en los objetos CancionBean obtenidos y compara este campo con la lista que entra como parámetro. Por tanto, lo que hace el método es buscar todas las listas de reproducción que contienen alguna de las canciones que se pasan como 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 en la capa de presentación. Nuevamente tenemos una interfaz para listas de reproducción

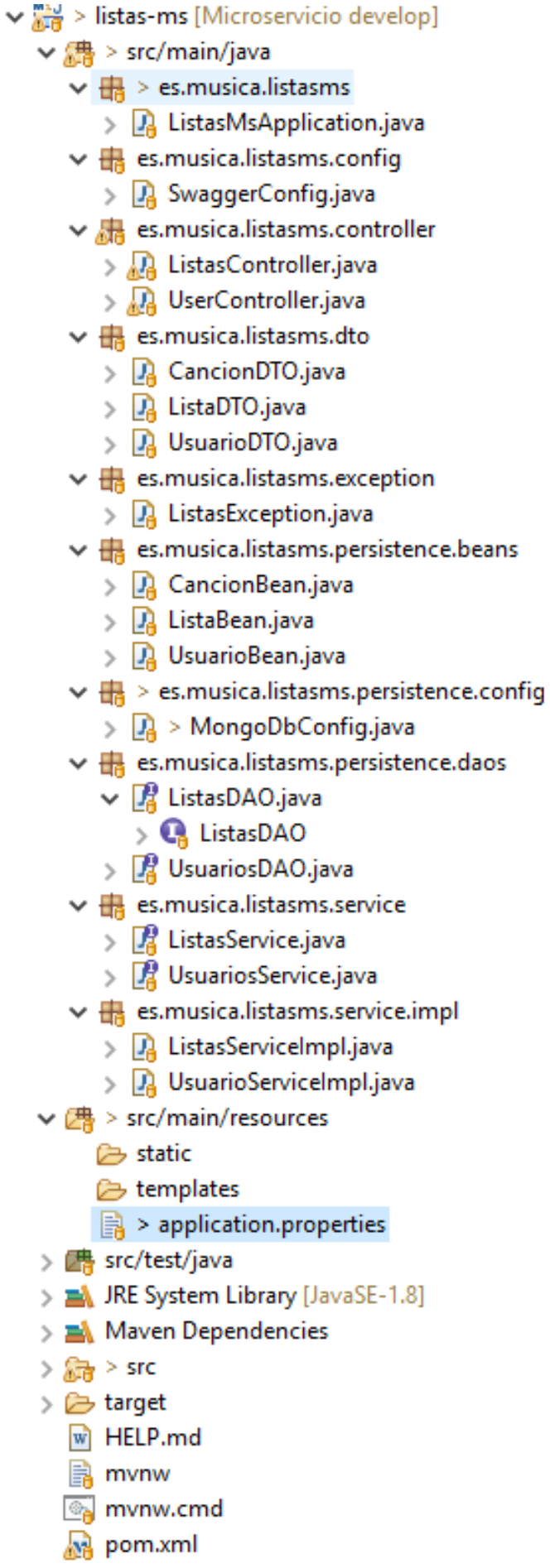

Fig. 4.5. Estructura microservicio

y otra para usuarios. En las implementaciones de las interfaces es importante anotar las clases con @Service y los DAOs a los que vayamos a llamar con @Autowire para que se instancien.

En esta capa se llama 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 canciones de éste y así poder filtrar en las listas por esas canciones, además tiene que hacer el mapeo entre los objetos beans y los objetos DTO 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 servicio u otro. En ella se definen los recursos y los verbos de la petición.

5. DESCRIPCIÓN DEL DOCUMENTO API DEL MICROSERVICIO

Hemos desarrollado una serie de servicios en cada controlador, estos pueden verse en las figuras 5.1 y 5.2.

En el controlador de listas hemos creado un recurso llamado /listas al que es posible acceder mediante el verbo POST y su función es almacenar en la base de datos una lista de reproducción nueva. El JSON de entrada necesario es el de la figura 5.3.

Este JSON corresponde a una lista de reproducción con una única canción. Los campos que aparecen son los proporcionados por Spotify y simplemente se han renombrado para evitar el uso de caracteres como el guion bajo.

Al llamar al servicio con el JSON se creará un nuevo documento en la base de datos.

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 llamado /listas/batch y 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 la base de datos.

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 base de datos.

Hemos creado otro recurso en /usuarios esta vez con el verbo PUT para crear un usuario con las canciones que ha escuchado en la base de datos. El JSON de entrada sería el de la figura 5.4.

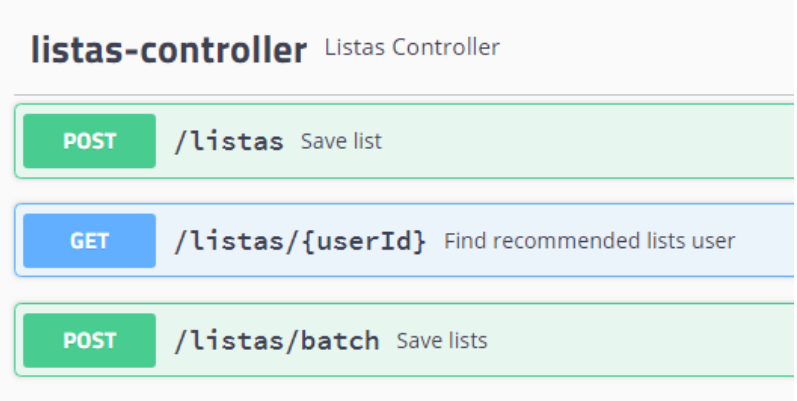

Fig. 5.1. Recursos disponibles dentro del controller Listas

Fig. 5.2. 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:2wIVse2owClT7go1WT98tk",
        "trackName": "Lose Control (feat. Ciara & Fat Man Scoop)",
        "albumUri": "spotify:album:6vV5UrXcfyQD1wu4Qo2I9K",
        "durationMs": 226863,
        "albumName": "The Cookbook"
    }
    ],
    "numEdits": 6,
    "durationMs": 11532414,
    "numArtists": 37
}
```


Fig. 5.3. JSON ejemplo lista reproducción


```
{
    "name": "Paco Sanchez Ruiz",
    "tracks": [
    {
        "pos": 0,
        "artistName": "Missy Elliott",
        "trackUri": "spotify:track:0UaMYEvWZi0ZqiDOoHU3YI",
        "artistUri": "spotify:artist:2wIVse2owClT7go1WT98tk",
        "trackName": "Lose Control (feat. Ciara & Fat Man Scoop)",
        "albumUri": "spotify:album:6vV5UrXcfyQD1wu4Qo2I9K",
        "durationMs": 226863,
        "albumName": "The Cookbook"
    }
    ],
    "userId": "pSanchez"
}
```


Fig. 5.4. JSON ejemplo usuario

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

6. DESPLIEGUE Y PRUEBA 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 6.1. En el campo base directory hay que poner lo que se muestra en la figura 6.1. En el campo goals vamos a poner clean install para limpiar primero el proyecto y luego compilarlo. En el campo user settings se puede dejar así configurado por defecto si todas las dependencias son de internet o configurar un fichero personal donde se establezca a que repositorio ir a buscar las dependencias necesarias. En el campo Maven runtime se puede utilizar el Maven incluido en eclipse o una versión que nosotros descarguemos.

En la pestaña de JRE es importante seleccionar Java 8 o superior ya que utilizamos funcionalidades que no son compatibles con versiones anteriores y es necesario incluir la JDK y no la JRE como se ve en la figura 6.2.

Una vez se tiene todo esto seleccionado se procede a dar al botón de Run, la aplicación se compilará desde cero y se ejecutaran las clases de test existentes. Una vez concluido lo anterior, en la carpeta target, se habrá generado un archivo JAR.

Para desplegar la aplicación es necesario tener activa la base de datos a la que se va a conectar.

De nuevo iremos a la pantalla de Run Configuration, esta vez creamos una nueva configuración de Java Application como se muestra en la figura 6.3. En el campo project se selecciona la raíz del proyecto y en el campo Main class, seleccionamos la ubicación de nuestra clase llamada ListasMsApplication.java, que es la que contiene el método main. Al dar al botón Run, se desplegará la aplicación.

Es posible acceder a la aplicación desde la siguiente dirección http://localhost:8080. Si queremos acceder a un recurso GET se puede hacer desde un navegador, si es otro tipo de recurso hay que hacerlo mediante Swagger o Postman. Todos los recursos están creados en las clases controller.

Además de los recursos creados por nosotros, se crean dos recursos de Swagger que son los llamados /swagger-ui.html (véase en la figura3.1) y /v2/api-docs; el primero es la parte visual de Swagger y el segundo es el fichero .yml con el que se genera la parte visual de Swagger.

Una vez desplegada la aplicación pasamos a probarla para comprobar que todo funciona como lo hemos implementado.

Tenemos que probar todos los recursos que hemos creado en el API. Probamos el recurso de crear usuarios, llamamos a él con el JSON de la figura 5.4, ahora debemos consultar en base de datos si se ha creado dicho usuario con todos los campos. Hemos

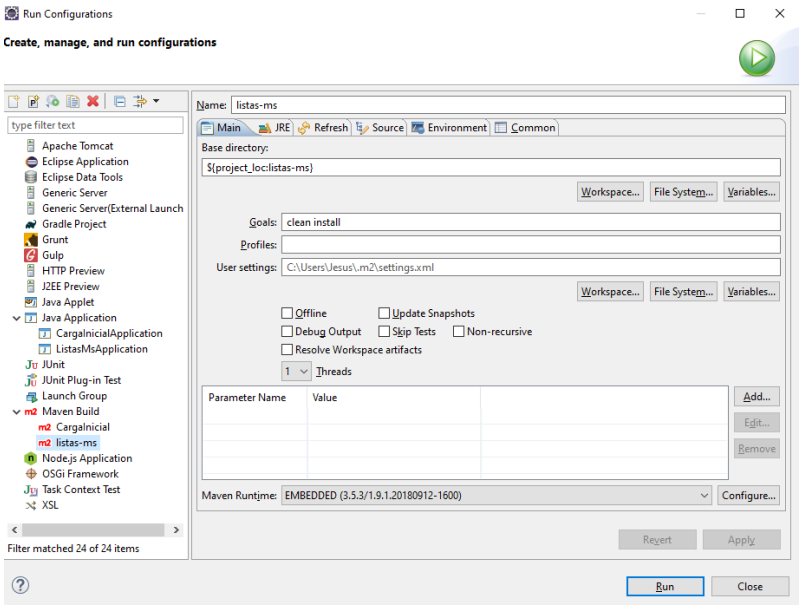

Fig. 6.1. Configuración compilación mediante Maven

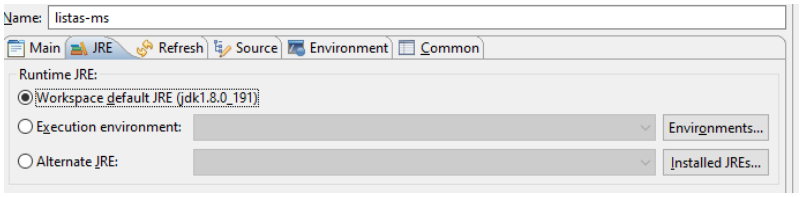

Fig. 6.2. Configuración JRE proyecto

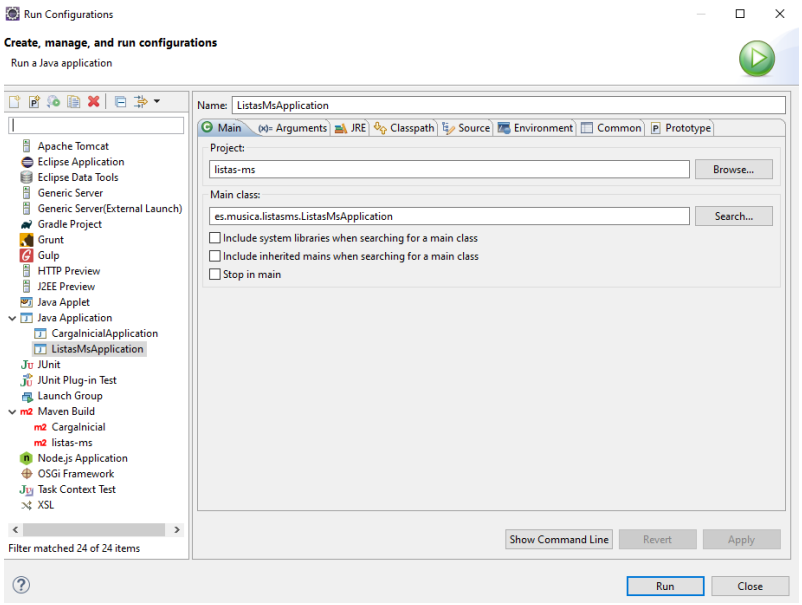

Fig. 6.3. Configuración Run despliegue

añadido una verificación para no insertar dos veces el mismo usuario, si ya existe nos devuelve un error indicándonoslo.

Para probar el servicio de devolver todos los usuarios, consultamos en base de datos todos los creados y al llamar al servicio nos debe devolver los mismos con todos sus datos.

Ahora llamamos al servicio de crear listas con el JSON de la figura 5.3 y comprobamos en base de datos que se ha creado. Este servicio tiene la misma validación que el de crear usuarios, debemos comprobarla.

Como el usuario que hemos creado y la lista tienen una canción en común, al llamar al servicio que filtra en listas por usuario si le decimos que tengan una canción en común nos devolverá la lista que acabamos de insertar.

Estas son las pruebas básicas que se han realizado, se puede probar una combinación de ellas para que los servicios devuelvan lo que deberían en cada caso.

7. MARCO REGULADOR

No existen leyes que se apliquen en el desarrollo de un microservicio, ya que prácticamente todos los softwares 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 [19]: En la ley Orgánica 3/2018, de 5 de diciembre, se establecen los principios de la protección de datos personales: exactitud, confidencialidad, consentimiento. Ya que vamos a 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.
- Ley General de Telecomunicaciones [20]: En la ley 9/2014, de 9 de mayo, se estable 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 que 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 por ejemplo Sonar.

8. ENTORNO SOCIO-ECONÓMICO

8.1. Presupuesto

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

8.1.1. 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 8.1. 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 de la aplicación.

8.1.2. Recursos físicos

A continuación pasamos a evaluar los gastos relacionados con los recursos amortizables. Para estimar el precio de cada recurso, se ha calculado una media entre los diferentes precios de cada proveedor.

La amortización corresponde a los meses de vida útil de cada recurso. Para poder calcular el precio final hemos realizado las siguientes operaciones; el proyecto se ha desarrollado a lo largo de 8 meses por lo que hemos dividido el precio del recurso entre la amortización en meses y lo multiplicamos por la duración del proyecto.

Los recursos utilizados y sus costes son los siguientes:

Recurso	Precio	Amortización (Meses)	Coste mensual (euros)	Total (euros)
Ordenador portátil HP Pavilion DV6	700	72	9,7	77,8
TeXstudio	0	12	0	0
Eclipse	0	12	0	0
NotePad ++	0	12	0	0
Total				77,8

TABLA 8.2. COSTE DE RECURSOS.

8.1.3. 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 base de datos 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 CosmosDB	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 8.3. COSTES MICROSOFT AZURE.

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

8.1.4. Coste total

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

Concepto	Total (euros)
Mano de obra	4500
Recursos	77,8
Servicios	1069,34
Total	5647,14

TABLA 8.4. COSTES TOTALES.

8.2. Impacto socio-económico

Uno de los objetivos de desarrollar aplicaciones con microservicios es que éstas 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 éstos al máximo de una manera eficiente y además se centraliza la ubicación de las maquinas en granjas de servidores que 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.

9. 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 ésta es la mejor opción, nos hemos dado cuenta de que tiene un coste muy elevado para una pequeña aplicación ya que cobran por uso y almacenamiento. Sin embargo, sí que hemos podido ver que para el problema al que nos enfrentamos, la base de datos elegida es la 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.

9.1. Trabajos futuros

A continuación explicamos 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. La aplicación 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

Contenerización

Por otra parte se podría contenerizar la aplicación para poder integrarla en un contenedor y desplegar éste 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 contenerizadas.

10. ANEXO: PROJECT SUMMARY

10.1. Introduction

Technology is based in tendencies. Thanks to Amazon, EBay, PayPal, Netflix... microservices are fashionables these days. However, this technology has born because of the cloud, where all these companies are taking their software.

The cloud doesn't need Physical Servers, so there is no need to maintain and invest in them. The cloud allows you to pay only for the resources you need in each moment. These is a great revolution because it's let you manage peaks of activity without losing services.

10.1.1. Problem Statement

We have real data from Spotify which contains the reproduction lists information. We also have users' data songs.

We need a data base where we could store all the parameters and to create a program that allows to query. This program assures a minimum downtime, which allows many people to query at the same time.

10.1.2. Objectives

To solve the problem stated before we aim at three main objectives:

1. To design the suitable software architecture.
2. To create a data base with the lists file data.
3. To implement a microservice.

10.1.3. Proposed solution

After analyzing the possible options, we adopted as a solution to our problem the design of an architecture of microservices. We are going to develop a microservice using Java with the Spring Boot framework. To call the microservice we would use Swagger.

The data base will be stored in Azure and it would be CosmosDB.

10.2. State of the art

The software architecture of a system or a collection of systems consists of all the important design decisions about the software structures and the interactions between those structures that comprise the systems.

The main carrier of system qualities such as efficiency, modifiability and security; is the architecture. The system is divided in software elements called modules. These modules can have both, internal and external properties.

Internal properties are the ones that define the language of the development and the implementation details such as objects, classes, modules, threats.

External properties are the contracts between modules and allow other modules to stablish dependencies and connexions between them.

There are architectural pattern, designs pattern and frameworks which offer a solution to the project. In the following sections we are going to explain the different available alternatives for each of the elements we need to solve the problem.

10.2.1. Design patterns

The design patterns are divided according the problem solution in:

- Creational patterns : these are used to simplify the new object creation. The most popular are Singleton and Builder.
- Structural patterns: It simplifies the software modelling, defining the way the classes relate between them. The most common patterns are Decorator and Proxy.
- Performance patterns: they manage algorithms, relation and responsibilities between objects. The most known pattern is Strategy.

10.2.2. Architectural Patterns

Architectural pattern defines the quality of the system. Like the design patterns, they give a solution to frequents problems in a reusable way.

The most common architectural patterns are:

- Layer programming: It is an architecture in which presentation, business logic, and data management functions are physically separated.
- Client-server: It has a server and multiple clients. The server will provide service to the different components from the client, and clients will request service from the server.

- Service-oriented architecture: provides functionality by creating services.
- Microservices' architecture: creates apps using micro services which communicates between them but running in an individual way.
- Model View Controller: It divides an interactive program into three parts: A model that directly manages the data, logic and rules of the application, the view that represents the information and the controller that who accepts input and converts it to commands for the model or view.

10.2.3. Frameworks

Frameworks are implemented software structures that a programmer can used to develop its own project. Frameworks follow not only the software patterns but also the architectural patterns. Frameworks normally include configuration files, libraries...

Most program languages have frameworks, but is important to choose the best option which better fits to the solution. As an example Angular would be a good framework to a front-end development. Another example is Spring which is oriented to de back-end development.

We can use the following frameworks to the microservices developments with Java:

- Spring Boot, that offers an archetype of a basic microservice and is easily extensible with a few development.
- Dropwizard, which is a framework for layer microservice architectures.
- Eclipse MicroProfile, that offers an archetype with the necessary dependencies.
- Helidon, which Oracle used named as "Java for Cloud" and now is for public used.

10.2.4. Database

A database is an organized collection of data which allows the query, modification and erase of the data. They were created to storage big quantities of information.

The first data bases which appear were the relational ones, in which the data are related. Then, non relational databases, the ones who have an identifier to relate a group of data with another.

The advantages of the relational databases are: their standards are well-defined, they are easy to program and there exists a lot of information about them. The disadvantages are: They grow to quickly and this makes the response time longer. Another disadvantage is that it is usually necessary to restructure database to make a change.

Regarding the non relational data base, their advantages are: they adapt without problems to the changes and the growths and there is no need to have huge servers. Their

disadvantages are: there is no guatantee to complete the information if the operation fails, it is necessary to have a large knowledge of the tool, there is no standard language and there is no graphic interface.

10.2.5. Microservices

The microservices are useful to develop an app splitting the service in smaller ones that run in an independent way and that communicate between them. Each microservice is small and belongs to a part of the app.

The advantages of the microservices are: They offer the possibility of deploying in an independent way the services, you can also develop with a minimum toolkit, they offer the possibility of using different program languages and there are easily scalable. The disadvantages of the microservices are: due to their distributed deployment, test are difficult. They also have a high memory consumption. It is complicated to integrate and to manage large quantities of services.

10.3. Justification of the adopted solution

The microservice architecture was selected because of the scalability and high availability.

To the development, it is choose Spring Boot, which is the framework which better fits to our project. This is because it is the most commonly used and older one. It also has the advantages of Sprint such as the functionality, which allows to connect with the data base, and the layers architecture.

Maven is the one selected to the dependencies management. It was selected because it is available for Java and it will package the app.

The use of Lombok is to refrain from the duplicity code. It also offer an implementation of the builder design pattern.

The communication with the microservice is going to be through REST, a simple and efficient protocol to make the different operations.

Every REST services needs an API document that describes the inputs and outputs. In this project Swagger is going to be used to auto-generate the API document, and Mondongo DB to store the data. Mondongo DB has no need to adapt the data because they store with the actual format.

To make the data base efficient, the data are going to be stored in the cloud and we will use Cosmos DB.

10.4. The app development

The first step is to install Eclipse. The second step is, by using Spring Boot, to navigate to the link [18]. It is necessary to choose Maven, for dependencies, Java as the language and the dependencies that we need to take (MongoDB, Spring Web and Lombok).

Swagger is the software to design, document and consume REST services. To configure Swagger it is necessary to add a configuration class with the @EnableSwagger2 notation to enable the domain generation.

The next step is to create a repository in GitHub [4] to host the development.

The archetype contains an App class to run the microservice in local. To make it possible, we need to stablish a connexion with the database because of the MongoDB dependency.

The following steps are to create an instance of CosmosDb and to configure the microservice to communicate with that database. First, create a class named MongoDbConfig and with the @Value notation. Secondly, we will have from application.properties the connexion to cosmos and the name of the data base. Once this is done, we can run the microservice.

The next step is to create the three layers that are going to be used: the persistence layer, where queries are defined, the service layer, where the business logic information is and the controller layer where the http verbs are implemented. With these verbs we will make the connexion with the service.

Once the methods have been created in the persistence layer, we create interfaces and their implementations with the methods that are going to be exhibit in the presentation layer, in the service layer. In the service layer you can call the methods from the persistence layer, and also from different services.

The presentation layer is the one which serialized a JSON in an object and depending on the resource, it will call one service or another. In this layer the statement verbs and the resources, are defined.

10.5. Regulatory framework

There is no regulation that applies to a microservice development. However, if real data are used it is mandatory to take into account the followings rules:

- General Data Protection Regulation [21].
- General Telecommunications Law [20].

10.6. Socio-economic environment

10.6.1. Budget

The project has been developed during eight weeks with 300 hours spent. This makes a cost of 4500 euros.

The amortizable resources we have needed are: a laptop HP Pavillion DV6, Notepad program, Eclipse program and TeXstudio program. This makes a cost of 77,8 euros.

The outsourced services, Azure Cosmos DB cost 1069,34 euros.

The overall project costs including manpower and outsourced services is 5647,14 euros.

10.6.2. The socio-economic impact

One of the goals of microservices is to develop apps which are easily maintainable and which allows the evolution with the society necessities.

One of the advantages of implementing microservices in the cloud is the best use of available resources (energy, servers) as it allows the maximum use of these in an efficient manner and also centralizes the location of machines in server farms that try to be supplied with renewable technologies.

Using the cloud allows to solve the problem that exists with physical servers (when there is a big amount of requests, is needed a great amount of servers that are not used when the service is given).

A disadvantage of microservices is that the cost is higher than using physical servers, for small apps.

10.7. Conclusions

We have achieved the goal of designing an architecture which is able to manage a great amount of requests.

The election of a cloud data base is the best option for our particular problem. However, due to the cost is for storing and for the use, the price raised when we design small apps.

Deploy a microservice is an research project due to its innovative technologies. It is also hard to find information about them. However, thanks to that, it is a clean application and it is easily maintainable.

10.7.1. Future works

There are the possible future works which can be undertaken to upgrade or complete the app:

- To develop a web app to display the data.
- To add new functionalities to the existing data.
- To docker the app to integrate it in a container and to deploy it in the cloud.
- To add a Docker manager in Azure to manage the grading and other configurations of the dockers' apps.

BIBLIOGRAFÍA

[1] (), [En línea]. Disponible en: https://azure.microsoft.com/es-es/services/kubernetes-service/ (Acceso: 05-09-2019).

[2] (). Microsoft Azure, [En línea]. Disponible en: https://azure.microsoft.com/ (Acceso: 01-09-2019).

[3] (), [En línea]. Disponible en: https://www.docker.com (Acceso: 10-08-2019).

[4] (), [En línea]. Disponible en: https://github.com/jesusRienda/microservicio (Acceso: 22-09-2019).

[5] (), [En línea]. Disponible en: https://jenkins.io/ (Acceso: 07-09-2019).

[6] (), [En línea]. Disponible en: https://projectlombok.org/ (Acceso: 01-06-2019).

[7] (), [En línea]. Disponible en: https://swagger.io/ (Acceso: 15-07-2019).

[8] 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 (Acceso: 20-05-2019).

[9] C. Alexander, S. Ishikawa y M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[10] E. Gamma, *Patrones de diseño: elementos de software orientado a objetos reutilizable*. Pearson Educación, 2002. [En línea]. Disponible en: https://books.google.es/books?id=gap_AAAACAAJ (Acceso: 23-05-2019).

[11] "Aplicaciones híbridas: ¿Qué son y cómo usarlas?", [En línea]. Disponible en: https://www.nextu.com/blog/aplicaciones-hibridas-que-son-y-como-usarlas/ (Acceso: 15-06-2019).

[12] G. Peiretti, "Strategy Pattern con Spring Boot", 2017. [En línea]. Disponible en: https://experto.dev/strategy-pattern-spring-boot/ (Acceso: 28-06-2019).

[13] 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 (Acceso: 28-06-2019).

[14] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 1st. O'Reilly Media, 2015, p. 280. (Acceso: 10-06-2019).

[15] J. Biggs, V. García y J. Salanova, *Building Intelligent Cloud Applications: Develop Scalable Models Using Serverless Architectures with Azure*. O'Reilly Media, 2019. [En línea]. Disponible en: https://books.google.es/books?id=NLWuDwAAQBAJ (Acceso: 25-05-2019).

[16] 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.

[17] R. RV, *Spring Microservices*. 2016. [En línea]. Disponible en: https://books.google.es/books?id=pwNwDQAAQBAJ (Acceso: 12-05-2019).

[18] (), [En línea]. Disponible en: https://start.spring.io/ (Acceso: 01-05-2019).

[19] 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 (Acceso: 28-08-2019).

[20] ——, “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 (Acceso: 28-08-2019).

[21] T. E. Parliament, “General Data Protection Regulation”, 2016. [En línea]. Disponible en: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679 (Acceso: 20-09-2019).