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 Pelaez Moreno

Leganés, 2019

RESUMEN

Palabras clave:

DEDICATORIA

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 compañias 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 seria 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 especifico 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

1.3. Solución propuesta

Para la solución de nuestro problema vamos a necesitar productos de distintos proveedores, ya que cada pequeña parte del programa la gestiona un software diferente.

Uno de los puntos mas importantes a decidir seria que base de datos usar, en este caso

usaremos CosmosDB. Como pieza de software que se comunique con la bbdd, trate los datos y los devuelva crearemos un servicio web mediante la arquitectura de microservicios desarrollado en java con el framework Spring Boot. Como protocolo para la transferencia de datos usaremos REST apoyándonos en sus verbos GET, POST, PUT y DELETE.

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 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).

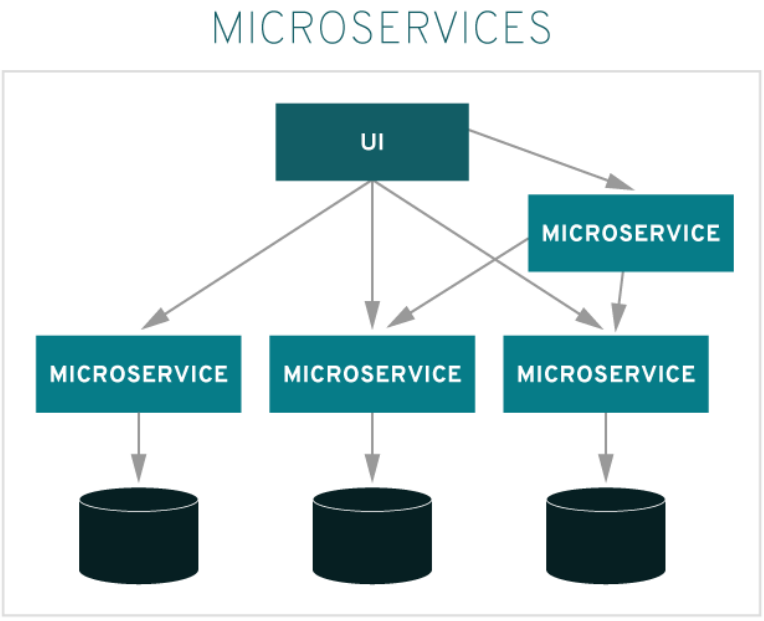

Fig. 2.1

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 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 abtracción 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 tambien existen los patrones arquitectonicos o arquetipos, los cuales tienen un nivel superior de abstraccion(explicar). Estos al igual que los patrones de diseño, solucionan problemas recurrentes de una forma reutilizable.

Los patrones de arquitectura mas usados son:

- Programación por capas: Se utiliza 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 mas 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 maquinas físicas, normalmente se utiliza 2 o 3 niveles. Si utilizáramos dos niveles tendríamos la capa de presentación en una maquina 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 requerimentos del negocio mediante la creación de servicios. Un servicio corresponde a un requerimento 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 requerimentos.

  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 sotfware 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.

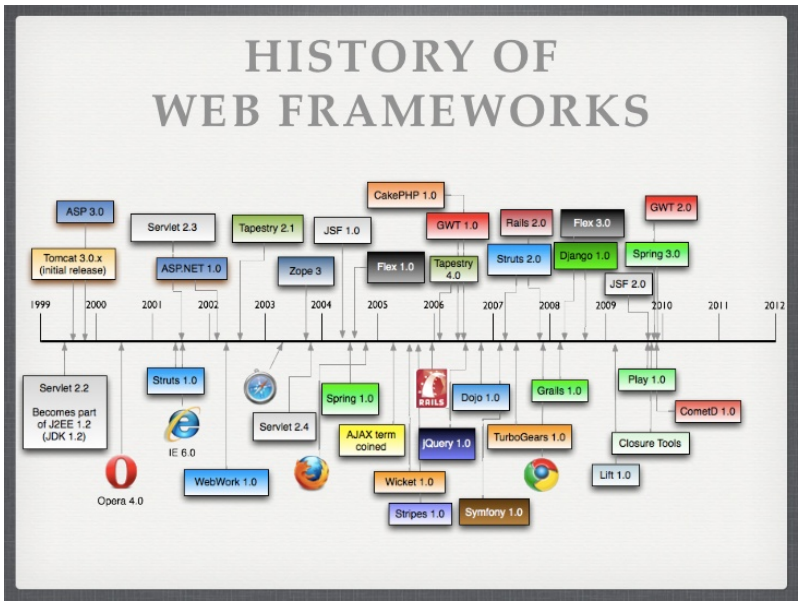

Fig. 2.2. Historia de los Web Frameworks

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 librerias 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 articulo 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 articulo [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 librerias 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 ultimas incorporaciones en la arquitectura. Pertenece a la compañía Oracle la cual lo utilizó internamente bajo el nombre "Java for Cloudz que finalmente se ha liberado para todo el publico.

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 ó 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 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”[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.

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.

3. JUSTIFICACIÓN DE LA SOLUCIÓN

La arquitectura de microservicios pretende dividir una aplicación compleja en pequeños servicios que solo realicen una función especifica y se comuniquen entre ellos para formar la aplicación final.

Cada microservico es totalmente independiente de desarrollar frente al conjunto, lo cual nos viene genial ya que nuestra idea es realizar una pequeña aplicación que aceda a una base de datos y en un futuro ampliar a varias aplicaciones o incluso un frontal. Los microservicios nos permiten desarrollar cada una en un lenguaje de programación diferente.

Al no disponer de una maquina donde desplegar la aplicación, es ideal que los microservicios lleven un servidor de despliegue(tomcat) embebido y así desplegar en la nube dentro de un contenedor de aplicaciones(Dockers).

Una vez desplegado en la nube decidimos que la comunicación sería 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 seria GET, POST, PUT y DELETE.

En cuanto a base de datos hemos elegido PostgreSQL ya que es open source y totalmente compatible con muchos lenguajes de programación, no solo con Java que es el caso de nuestra aplicación, sino que si en un futuro queremos creamos otro microservicio con python sería posible reutilizar la base de datos. Ademas también es capaz de responder a gran cantidad peticiones en un mismo instante y no bloquearse, esto es totalmente esencial ya que necesitamos alta disponibilidad.

3.1. SOA vs 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 3.1.

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 mo-

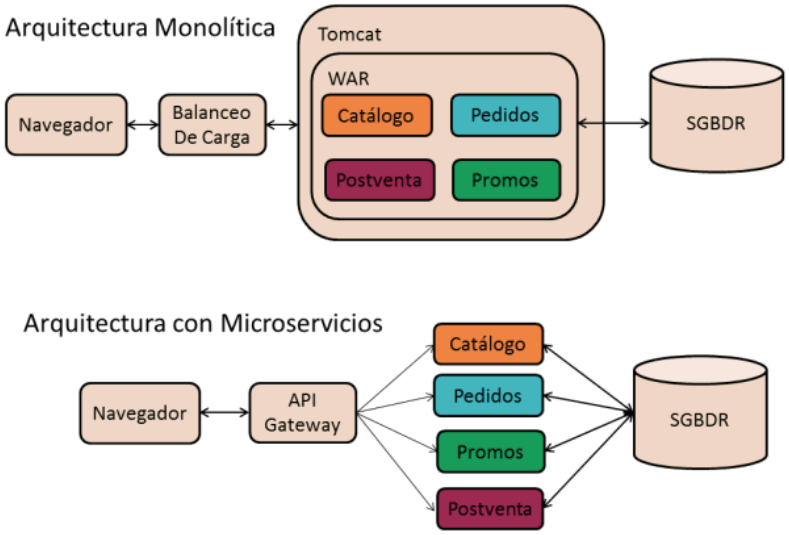

Fig. 3.1. Arquitectura Monolítica frente Microservicios

nolítica (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 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 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, desligues 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[9] y aun le queda mucho por mejorar y evolucionar.

Rajest RV en su libro "Spring Microservices"[10] nos muestra el gráfico 3.2 en el que se observa como es más rápido y ágil el desarrollo de aplicaciones con microservi-

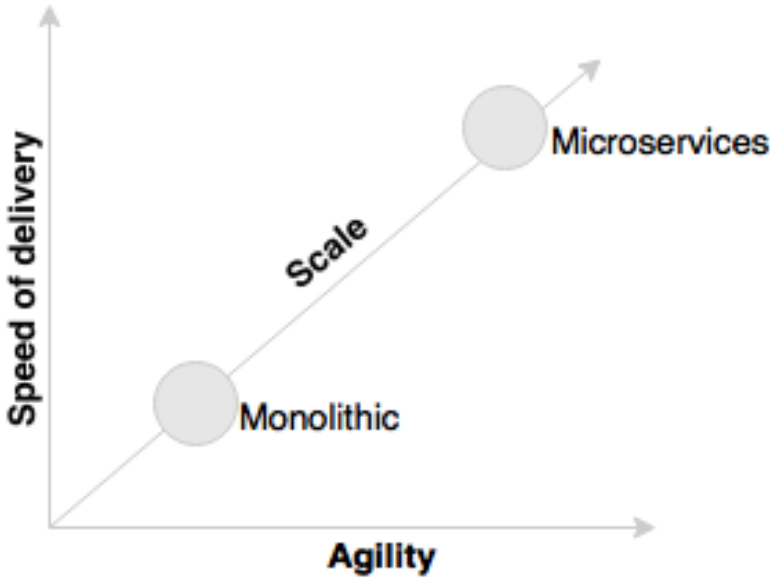

Fig. 3.2. Eficiencia microservicios frente SOA

cios 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 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.

3.1.1. 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 comple-

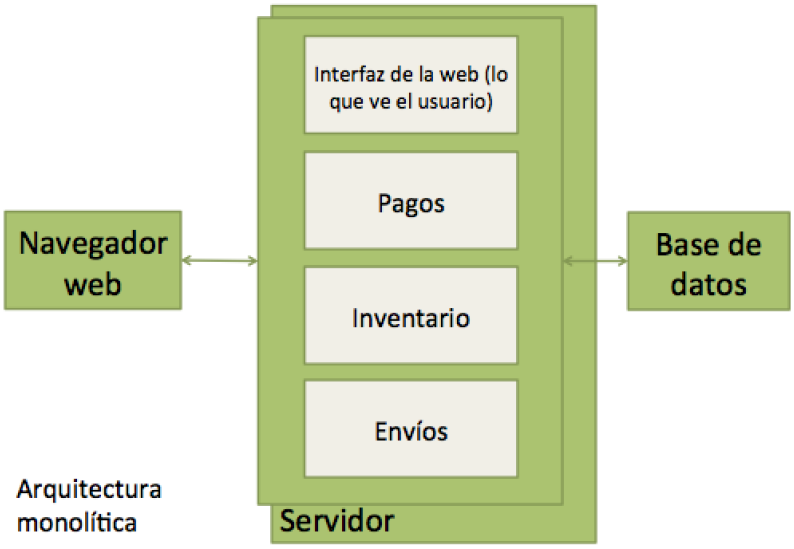

Fig. 3.3

ja, 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.

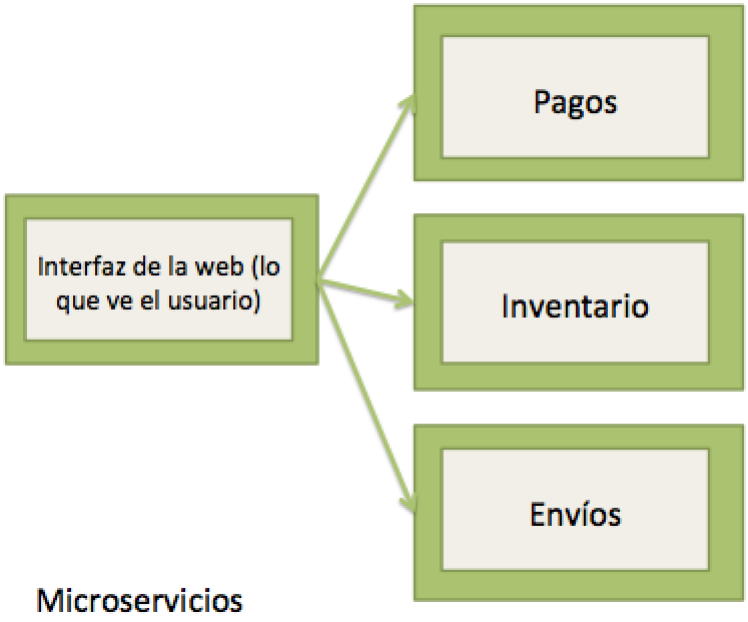
Pagos
Interfaz de la web (lo que ve el usuario)
Inventario
Envíos
Microservicios

Fig. 3.4

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 descargaran todas las dependencias incluidas en el fichero pom.xml, es el momento de añadir las dependencias de swagger que vamos a necesitar.

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.

Lo siguiente es crear las tres capas que vamos a utilizar, la capa de persistence, donde se definen las querys, 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 4.2 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 querys 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

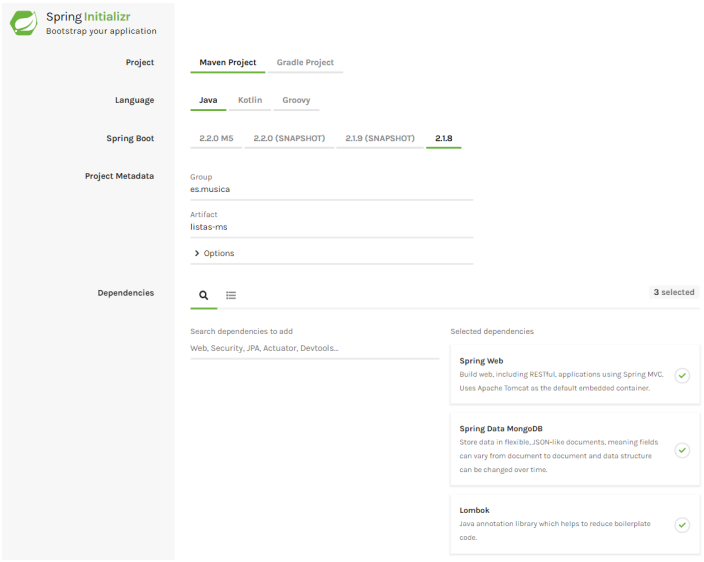

Fig. 4.1. Ejemplo configuración arquetipo

muchos elementos, actualiza un elemento. etc. Ademas 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 @Autowire 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 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.

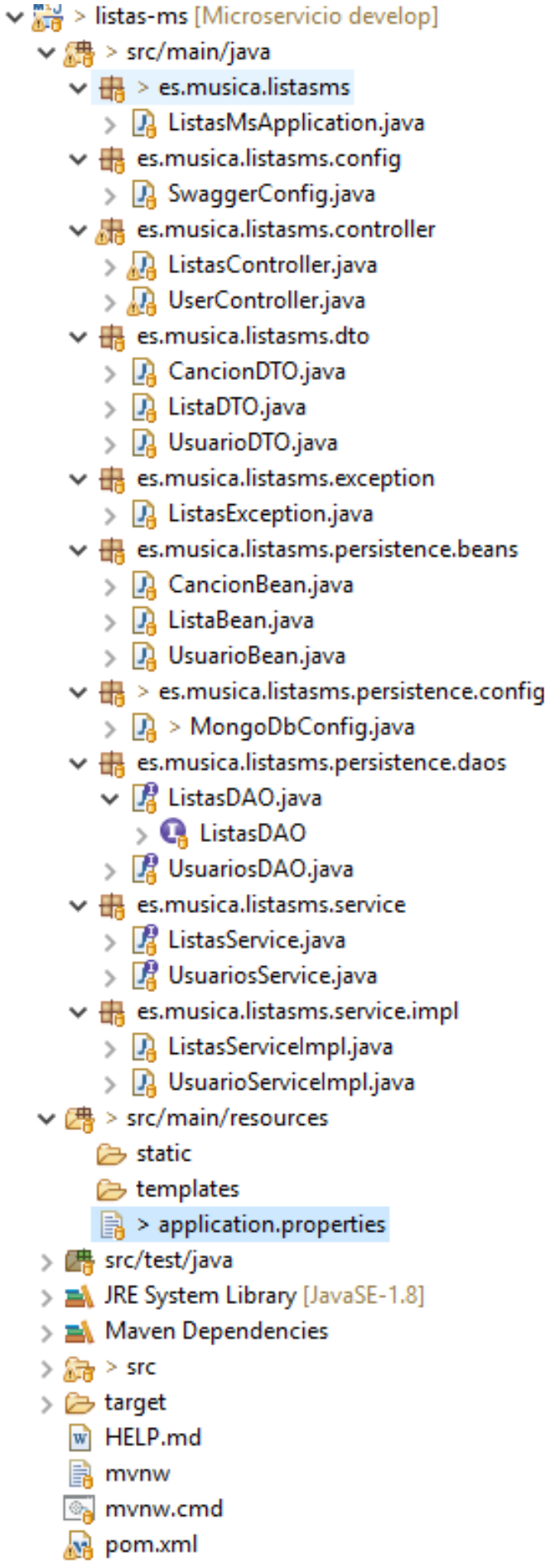

Fig. 4.2. Estructura microservicio

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 reutilizable*, 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.