Pontificia Universidad
JAVERIANA
Bogotá

Integrantes

Karol Geraldine Ceballos Castro

Angela Catalina Llaña Arciniegas

Brayan Steven Carrillo Mora

Grupo 4

Taller2

Asignatura

Arquitectura de software

Profesor

Andres Sanchez Martin

Microservicios-Rest

Contenido

DEFINICIÓN, HISTORIA Y EVOLUCIÓN

MICROSERVICIOS

El inicio de los microservicios está estrechamente relacionada con la evolución de la arquitectura de software y las necesidades cambiantes de las empresas en términos de agilidad, escalabilidad y mantenibilidad de sus sistemas. Debido a que había problemas cuando se aplicaba la arquitectura monolítica ya que en la que la aplicación está contenida en un solo modelado de clase,posee sólo una sola lógica de negocio y una sola base de datos centralizada y todo código fuente de la aplicación se encuentra en un solo proyecto que se compila en un solo archivo ejecutable,aún esta arquitectura todavía sigue siendo algo común en las aplicaciones empresariales,la mayoría de las aplicaciones monolíticas se veían cuando aprendemos a programar como en programación orientada a objetos o aprendemos los conceptos de todo esto en el cual tenemos la interfaz gráfica, la lógica de negocio y la persistencia (capacidad de una aplicación para almacenar y recuperar datos de manera duradera).

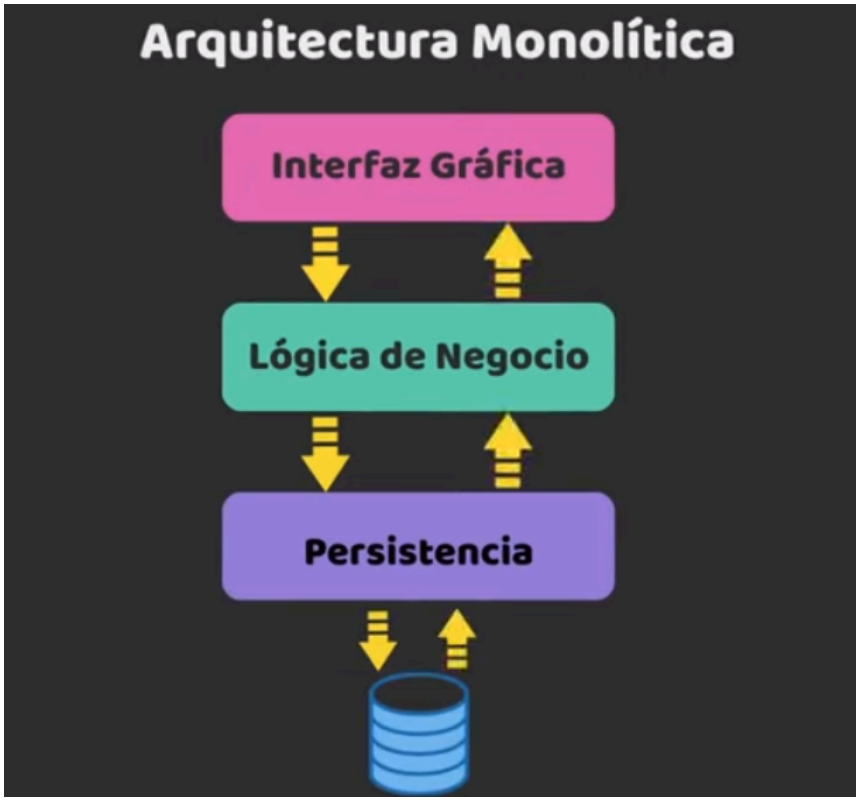


figura 1(Arquitectura monolítica)

Como se puede observar la arquitectura monolítica figura(1) todo está centralizado en un solo lugar y no está dividido en varios servicios diferentes.El problema al aplicar está arquitectura es que tiene gran complejidad al crecer a gran escala y una mayor probabilidad de fallos y cuello de botella.

Ahora antes de explicar que son microservicios,debemos tener en cuenta que La Arquitectura Orientada a Servicios (SOA) es un enfoque arquitectónico para el diseño y desarrollo de sistemas

de software que se popularizó en la década de 2000. Aunque los microservicios surgieron posteriormente, tienen sus raíces en los principios y prácticas basadas o establecidas por SOA.

SOA(Arquitectura orientada a servicio) es un enfoque arquitectónico que se centra en la creación de servicios independientes y reutilizables que se pueden compartir entre diferentes sistemas dentro de una organización. En SOA, los servicios suelen ser grandes y encapsulan una parte específica de la lógica empresarial. Estos servicios se comunican entre sí a través de interfaces bien definidas, como servicios web (por ejemplo, SOAP o REST).Los microservicios representan una evolución de la arquitectura de servicios orientada a objetos (SOA),A diferencia de SOA, donde los servicios suelen ser más grandes y acoplados donde tenemos ciertos compuestos que se forman con los servicios lo cual genera a su vez un desplegable del cual no es un monolito tan grande como una aplicación empresarial pero sigue siendo monolito porque depende de otros servicios, los microservicios reduce esa limitación en la que ya no queremos generar ese tipo de monolitos es decir queremos segmentar aún más,el servicio es tratado como un producto independiente:se desarrolla, se empaqueta, se despliega de forma independiente, facilitando la agilidad en el desarrollo y la rápida implementación de nuevas funcionalidades. Además, los microservicios son más escalables y tolerantes a fallos, ya que su tamaño reducido y su acoplamiento débil facilitan la distribución y la gestión de errores sin afectar al sistema en su conjunto,la comunicación debe ser asincrónica entre los distintos servicios.Entonces los microservicios es un servicio de la implementación de SOA pero con las características mencionadas anteriormente.

Ahora bien retomando lo anterior los microservicios son un enfoque para los sistemas distribuidos que promueven el uso de servicios detallados con sus propios ciclos de vida, que colaboran entre sí. Debido a que los microservicios se modelan principalmente en torno a dominios comerciales, evitan los problemas de las arquitecturas por niveles tradicionales

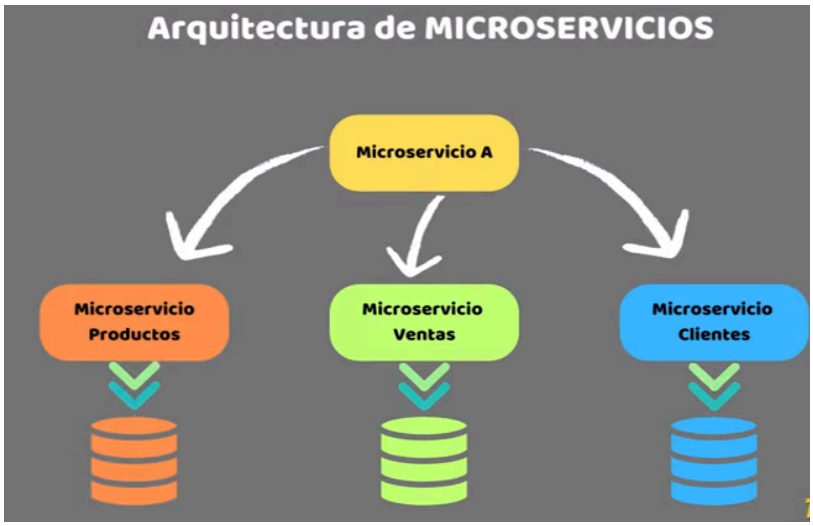


figura 2 (Arquitectura de microservicios)

con respecto a la figura(2) la aplicación que estemos desarrollando no solo esté centralizado en un solo lugar o sus partes en otro lugar sino todo lo que se haga en la aplicación se subdividen en servicios más pequeños y son autónomas, se ejecutan por sí mismos sin depender de otro servicio que esté en algún momento.

Cada microservicio tiene su propio modelado y su propia base de datos y lo que hace para comunicarse con otro servicios es utilizar APIs como intermediario.

Esta arquitectura se utiliza más en el desarrollo web en lo que es backend porque hace que el desarrollo sea mucho más ágil,la división de taras sean más sencillas, es decir un desarrollador se encarga de un microservicio, otro desarrollador se encarga de otro microservicio otro desarrollador de otro,luego se interconectan entre si todo eso y esto hace que sea escalable

lo que permite un microservicio es que si una aplicación deja de funcionar nos enfocamos en esa parte para solucionarlo y no deja de funcionar el resto de la aplicación ya que son servicios separados.

REST/JSON

La historia de REST y JSON está estrechamente vinculada al desarrollo de la web moderna y al cambio hacia una arquitectura centrada en recursos. En la década de 1990, con la expansión rápida de la World Wide Web, el Protocolo de Transferencia de Hipertexto (HTTP) se estableció como el protocolo fundamental para la comunicación en la web, permitiendo que los navegadores solicitan recursos de los servidores web.

Para facilitar la comunicación entre aplicaciones distribuidas en la web, surgió el Protocolo Simple de Acceso a Objetos (SOAP) como un estándar para el intercambio de mensajes, utilizando XML como formato predeterminado para los datos.

Sin embargo, en el año 2000, Roy Fielding propuso la arquitectura REST (Transferencia de Estado Representacional) como una alternativa a SOAP. REST se basa en los principios fundamentales de la web, como el uso de URI para identificar recursos y los métodos HTTP para operar en esos recursos. JSON, que es más ligero y fácil de usar en comparación con XML, comenzó a ganar popularidad como un formato de intercambio de datos.

A medida que la web evoluciona y la demanda de servicios web más simples y eficientes aumentaba, REST y JSON se convirtieron en la opción preferida para muchas aplicaciones y servicios de API. Su simplicidad, flexibilidad y adopción generalizada los han convertido en elementos esenciales de la infraestructura web moderna.

Con el tiempo, REST y JSON se han estandarizado, con la aparición de especificaciones y convenciones comunes como OpenAPI (anteriormente conocido como Swagger) para describir APIs REST y JSON Schema para validar la estructura de los datos JSON.

Entonces REST y JSON surgieron como una respuesta a la necesidad de una arquitectura ligera y fácil de usar para la comunicación entre sistemas distribuidos en la web, y su evolución ha sido fundamental en la construcción de la web moderna.

REST(Transferencia de Estado Representación).Es un estilo de arquitectura o lógica de restricciones y recomendaciones bajo la cual se puede construir un API.para diseñar sistemas de software distribuidos, es un conjunto de principios y convenciones que guían el diseño de aplicaciones web. Estos principios incluyen la separación clara entre clientes y servidores, la ausencia de estado en las interacciones (stateless), el uso de operaciones HTTP bien definidas (GET, POST, PUT, DELETE), la identificación de recursos mediante URLs, y una interfaz uniforme para las interacciones cliente-servidor.

RestFul Api funciona estrictamente a una arquitectura cliente servidor utilizando http como protocolo de comunicación,un cliente podemos ser nosotros utilizando y usando requerimientos desde una aplicación web o móvil y el servidor es el que recibirá nuestros requerimientos y ejecutará las acciones de acuerdo a nuestra petición,pero cuando realizamos un requerimiento ¿como sabe el servidor que acciones debe ejecutar? aqui es donde entra las especificaciones de REST, en REST cada procedimiento de nuestra API está compuesto por 3 partes importantes un verbo HTTP una dirección URI única y la información(datos ya sea json,xml,binario) necesaria que requiere el servidor para satisfacer el requerimiento incluyendo la información de autenticación del cliente.

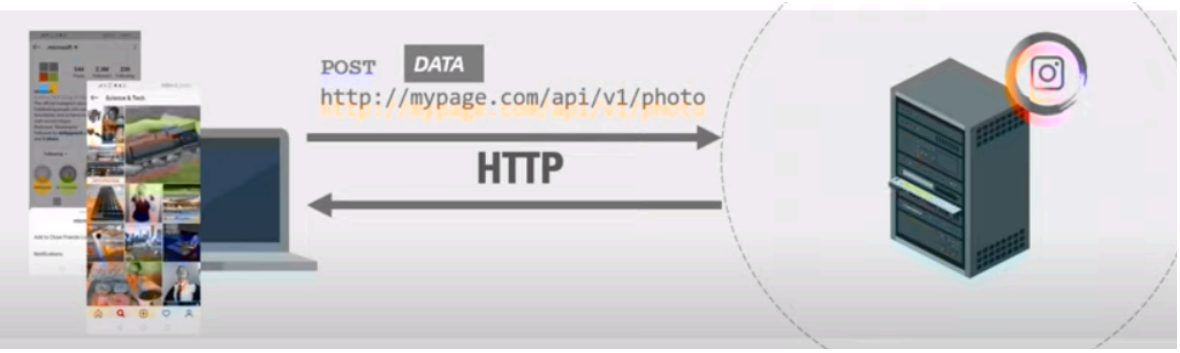


fig 3 (arquitectura rest Api)

los verbos HTTP son identificadores que determinan el objetivo de un requerimiento del cliente los verbos o operaciones más utilizados son (GET, POST, PUT/PATCH, DELETE),Cada uno cumple con un propósito específico: GET se una para los procedimientos que devuelven información al cliente,POST para que el cliente pueda crear recursos dentro del servidor ejemplo agregar un registro dentro de una base de datos,PUT/PATCH para que el cliente pueda editar recursos ya existentes dentro del servidor y finalmente DELET se utiliza para eliminar recursos del servidor.

La implementación de REST esque la implementación de estos procedimientos de servidor no le interesa el cliente ni las implementaciones dentro del cliente para realizar los requerimientos son como dos cajas negras intercambiando información entre sí es decir a ninguno le importa en qué lenguajes está programado el otro o que framework necesita el otro y eso se puede lograr debido a que en la lógica rest el servidor debe recibir toda la información necesaria para poder ejecutar cada requerimiento esto conlleva a definir un formato para intercambio de datos entre las dos partes este formato debe ser flexible como para nuestro caso JSON, XML, binario o texto plano. JSON es preferido por la comunidad.

Debido a que el servidor necesita que cada requerimiento tenga toda la información necesaria para poder ejecutar un procedimiento se dice que REST es stateless ya que no se necesita guardar

información o el estado de peticiones anteriores para poder satisfacer peticiones nuevas cada petición es independiente de otras.pero si admite las peticiones caché para guardar las peticiones hechas con anterioridad y de esta forma utilizarlas para poder satisfacer nuevas peticiones de los mismos recursos de manera rápida esto se utiliza en los requerimientos de tipo GET.

JSON, o JavaScript Object Notation, es un formato de intercambio de datos ligero y altamente legible por humanos. Surgió como una alternativa más eficiente y fácil de usar a otros formatos de intercambio de datos, como XML. Diseñado originalmente para ser utilizado con JavaScript, JSON se ha convertido en un estándar de facto en el desarrollo web debido a su simplicidad y versatilidad.

En el contexto de REST (Transferencia de Estado Representacional), JSON se utiliza ampliamente como formato de intercambio de datos entre clientes y servidores. Esto se debe a su capacidad para representar datos estructurados de manera clara y concisa, su facilidad de uso en entornos web y su compatibilidad con la mayoría de los lenguajes de programación modernos. Al utilizar JSON en aplicaciones RESTful, los desarrolladores pueden enviar y recibir datos de manera eficiente ya que envían y reciben datos entre clientes y servidores de manera eficiente para realizar operaciones como crear, leer, actualizar y eliminar recursos, lo que facilita la construcción de servicios web flexibles y escalables.

El servidor procesa la solicitud y envía una respuesta con la lista de tareas en formato JSON. La respuesta podría verse así:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {"id": 1, "task": "Completar informe", "completed": false},
    {"id": 2, "task": "Enviar correo electrónico", "completed": true}
]
```

fig 4 (respuesta del servidor al cliente)

Luego, el cliente puede enviar una nueva tarea al servidor para ser agregada a la lista. La solicitud POST puede verse así:

```
POST /tasks HTTP/1.1
Host: example.com
Content-Type: application/json

{"task": "Preparar presentación", "completed": false}
```

fig 5(solicitud del cliente)

El servidor procesa la solicitud y responde con un mensaje indicando que la tarea ha sido creada exitosamente, junto con los detalles de la tarea creada. La respuesta puede verse así:

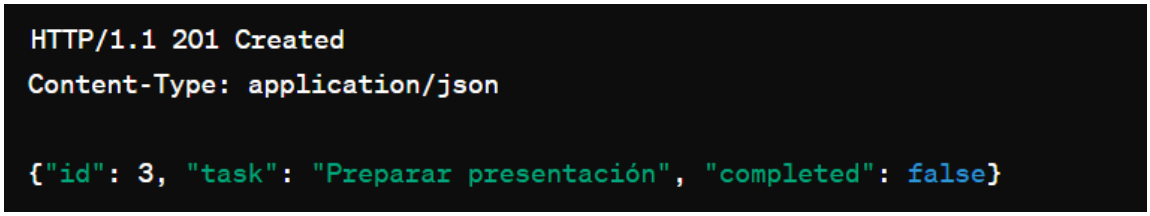


fig 6(respuesta de la tarea)

SPRING BOOT

La historia de Spring Boot se remonta a la evolución del framework Spring, concebido por Rod Johnson en 2002 como una alternativa liviana y flexible para el desarrollo de aplicaciones empresariales en Java. Spring Framework, con sus conceptos como la inversión de control y la programación orientada a aspectos, se convirtió rápidamente en uno de los más populares en el ecosistema Java.

Con el tiempo, Spring Framework se expandió para abarcar módulos como Spring MVC para desarrollo web, Spring Data para acceso a datos y Spring Security para la seguridad de aplicaciones. Sin embargo, su configuración compleja planteaba desafíos para los desarrolladores, lo que llevó a la necesidad de simplificar el proceso de desarrollo.

En 2014, surgió Spring Boot como un proyecto dentro de Spring con el objetivo de simplificar la creación de aplicaciones Spring con configuración mínima. Este marco de trabajo para el desarrollo de aplicaciones backend en Java proporciona un entorno completo para crear servicios web, APIs RESTful, lógica de negocio, acceso a bases de datos, seguridad, y mucho más. Basado en la convención sobre configuración, Spring Boot adopta configuraciones predeterminadas inteligentes que permiten a los desarrolladores centrarse en la lógica de negocio. Además, Spring Boot simplifica la configuración y el desarrollo de aplicaciones Java al proporcionar características como la autoconfiguración, la gestión de dependencias y la integración con servidores web embebidos, lo que se explicará más adelante en las características.

Java es la elección preferida para Spring debido a su amplia adopción en el desarrollo empresarial y su robustez como lenguaje de programación. La combinación de Spring y Java ofrece un entorno seguro y escalable para desarrollar una variedad de aplicaciones, desde microservicios hasta aplicaciones nativas de la nube. Además, Java proporciona características que son esenciales para implementar la programación reactiva, asincrónica y sin bloqueo, necesaria para aplicaciones modernas en la nube.

Al implementar Spring Boot, se abre la puerta a posibilidades para desarrollar aplicaciones escalables y adaptadas al entorno de la nube. Con su enfoque en la programación reactiva, que abarca la asincronía y la ausencia de bloqueos, Spring Boot se alinea perfectamente con los requisitos de las aplicaciones modernas en la nube.

La programación reactiva permite que las aplicaciones respondan de manera eficiente a un alto volumen de solicitudes, gestionando los recursos de manera óptima y manteniendo una experiencia de usuario fluida. Además, al adoptar esta metodología, las aplicaciones pueden aprovechar al máximo las características y servicios ofrecidos por los proveedores de la nube.

Spring Boot facilita el desarrollo de aplicaciones serverless, un modelo de ejecución de aplicaciones en la nube en el que el proveedor se encarga de manera dinámica de la asignación y gestión de recursos de la máquina. Esta arquitectura elimina la necesidad de gestionar la infraestructura subyacente, permitiendo a los desarrolladores enfocarse exclusivamente en la lógica de la aplicación.

Con la compatibilidad de Spring Boot con servicios como AWS Lambda y Google Cloud Functions, los desarrolladores pueden crear aplicaciones serverless de manera sencilla y aprovechar las capacidades de escalabilidad y flexibilidad que ofrecen estos servicios en la nube.

Características de Spring Boot:

1. Autoconfiguración: Spring Boot utiliza la detección automática para configurar tu aplicación en función de las dependencias agregadas y la infraestructura presente en el entorno de ejecución. Esto significa que Spring Boot puede configurar muchos aspectos de tu aplicación sin que necesites escribir una gran cantidad de código de configuración manualmente. Por ejemplo, puede configurar la base de datos, la seguridad, el servidor web, entre otros.

2. Inicio rápido: Spring Boot proporciona un entorno de desarrollo que permite iniciar rápidamente tu aplicación sin la necesidad de configurar un servidor de aplicaciones por separado. Esto se logra mediante el uso de servidores web embebidos, como Tomcat, Jetty o Undertow, que están integrados en la aplicación. Simplemente puedes ejecutar tu aplicación como una aplicación Java estándar y Spring Boot se encargará de iniciar el servidor web automáticamente Esto significa que puedes empaquetar tu aplicación como un archivo JAR ejecutable que incluye todo lo necesario para ejecutarse.

3. Gestión de dependencias: Spring Boot utiliza Maven o Gradle para gestionar las dependencias de tu aplicación de forma sencilla. Además, Spring Boot incluye un conjunto de "iniciadores de dependencias" que te permiten especificar las dependencias necesarias para tu aplicación. Spring Boot se encarga de agregarlas automáticamente al proyecto, junto con las versiones compatibles y las configuraciones necesarias.

Componentes que forman parte de la arquitectura de la aplicación:

Modelo entidad: presenta una entidad de negocio que será mapeada a una tabla en la base de datos.

Repositorio jpa:es una interfaz que extiende JpaRepository para acceder a la capa de

persistencia de JPA.

Capa de servicio:es un componente de servicio que encapsula la lógica de negocio

El controlador:es un controlador que maneja las solicitudes HTTP relacionadas con los productos y delega la lógica de negocio al servicio correspondiente.

ANGULAR/TYPESCRIPT

Cuando un cliente realiza una solicitud HTTP, el servidor responde normalmente con una página HTML, junto con otros recursos como imágenes, hojas de estilo CSS, entre otros. El cliente renderiza esta página en el navegador. A medida que el usuario interactúa con la página o hace clic en botones, desencadena nuevas solicitudes HTTP al servidor, que responde con otra página HTML y otros datos necesarios para su renderizado. Esta interacción continua entre el cliente y el servidor implica que con cada respuesta, el servidor debe generar una nueva página, lo que puede resultar en una carga de trabajo significativa para el servidor.

Este modelo, conocido como "múltiples páginas", puede presentar algunos inconvenientes, especialmente en términos de carga de trabajo para el servidor. En algunos casos, puede no ser deseable ya que puede aumentar los costos operativos, especialmente en entornos de nube donde los costos están directamente relacionados con los recursos asignados a los servidores.

Para abordar estos problemas y reducir la carga del servidor, así como mejorar la interactividad de la interfaz de usuario, es beneficioso adoptar un enfoque de "una sola página para múltiples páginas".

Angular, un framework desarrollado por Google y de código abierto, es una herramienta clave para implementar este enfoque.Su historia comienza en 2010, cuando dos ingenieros de Google, Misko Hevery y Adam Abrons, crearon una herramienta interna llamada "AngularJS" para mejorar el desarrollo de aplicaciones web en la empresa. AngularJS, lanzado en 2012 como una versión de código abierto, rápidamente ganó popularidad debido a su enfoque en la creación de aplicaciones web dinámicas y ricas en funcionalidades.

AngularJS introdujo el concepto de enlace de datos bidireccional, lo que significa que los cambios en el modelo de datos se reflejan automáticamente en la interfaz de usuario y viceversa, sin necesidad de manipulación manual del DOM (Modelo de Objetos del Documento). Esto simplificó en gran medida el desarrollo web y permitió a los desarrolladores crear aplicaciones más interactivas y receptivas.

Sin embargo, con el tiempo, AngularJS mostró algunas limitaciones en términos de rendimiento y escalabilidad, especialmente para aplicaciones más grandes y complejas. En respuesta a estos desafíos, el equipo de Angular en Google comenzó a trabajar en una nueva versión, conocida como Angular 2.

Angular 2, lanzado en 2016, fue una reescritura completa de AngularJS, con un enfoque en la modularidad, el rendimiento y la escalabilidad. Esta nueva versión adoptó TypeScript como su principal lenguaje de programación, lo que proporcionó una serie de ventajas, como la detección temprana de errores y una mejor capacidad para trabajar con grandes equipos de desarrollo.

Desde entonces, Angular ha seguido evolucionando con regularidad, con lanzamientos periódicos de nuevas versiones que introducen características y mejoras significativas. Angular se ha convertido en una opción popular para el desarrollo de aplicaciones web y móviles, utilizado por empresas grandes y pequeñas en una amplia variedad de sectores.

Angular se utiliza para crear la interfaz de usuario de aplicaciones web y móviles, permitiendo la creación de vistas dinámicas y ricas en funcionalidades. Facilita la gestión de la lógica de presentación y la interacción del usuario, así como la comunicación con el backend a través de solicitudes HTTP para obtener o enviar datos.

Angular es una herramienta que facilita la creación de aplicaciones de una sola página (SPA) que pueden proporcionar múltiples vistas sin tener que cargar páginas completas cada vez mejora de la eficiencia y optimización de los costos operativos puede aplicarse al enfoque de una sola página para múltiples páginas en general, independientemente de la tecnología específica utilizada para implementarlo.Además, proporciona funcionalidades como enlace de datos bidireccional, inyección de dependencias y un sistema de enrutamiento, lo que ayuda a los desarrolladores a construir aplicaciones complejas de manera más eficiente.

Angular está basado en TypeScript, un lenguaje que ofrece mejoras significativas sobre JavaScript. TypeScript es más estricto en términos de tipado, lo que significa que permite definir tipos de datos y restringe los tipos de valores que pueden asignarse a variables y parámetros. Además, TypeScript está orientado a objetos, permite la devolución de valores de las funciones y proporciona características modernas que no están presentes en JavaScript.

Dado que los navegadores no son compatibles con TypeScript de manera nativa, se utiliza un compilador de TypeScript para traducir el código a JavaScript, que es el lenguaje que se ejecuta en el navegador.

Adoptar Angular y el enfoque de una sola página para múltiples páginas puede ayudar a reducir la carga del servidor, mejorar la eficiencia y la experiencia del usuario, y optimizar los costos operativos, especialmente en entornos de nube donde se prioriza la asignación eficiente de recursos.

Cuando creamos el proyecto vamos a tener una sola página intex. html y componentes,a medida de que el usuario vaya navegando por los distintos componentes angular va creando,actualizando o destruyendo ciertos componentes.Los componentes son bloques fundamentales de construcción de la interfaz de usuario de una aplicación.

Se utiliza el patrón mvc donde el modelo en Angular representa los datos y la lógica relacionada con los datos de la aplicación. Puede ser una clase TypeScript que define la estructura y el comportamiento de los datos, o puede ser un servicio que interactúa con el backend para obtener o manipular los datos.

La vista es la parte de la interfaz de usuario que los usuarios ven y con la que interactúan. En Angular, las vistas están representadas por componentes. Los componentes son clases TypeScript que contienen la lógica de presentación y el HTML asociado. Cada componente está vinculado a una vista específica y se encarga de manejar las interacciones del usuario en esa vista.

El controlador puede estar representado por el componente mismo, ya que los componentes encapsulan tanto la lógica de presentación como el comportamiento del controlador. Sin embargo, para la lógica de negocio más compleja o para la lógica que se necesita compartir entre varios componentes, se pueden utilizar servicios. Los servicios en Angular son clases TypeScript que encapsulan la lógica reutilizable y pueden ser inyectados en varios componentes para su uso.

BASES DE DATOS SQL

El lenguaje de consulta estructurado SQL (Structured Query Language) se ha convertido en un estándar de la industria para interactuar con sistemas de gestión de bases de datos relacionales (RDBMS). Surgió a partir del modelo relacional propuesto por Edgar F. Codd en la década de 1970, el cual introdujo la idea de representar los datos en forma de tablas con filas y columnas. Este enfoque permitió una gestión más eficiente y flexible de los datos en comparación con los modelos anteriores.

IBM System R fue el primer sistema de gestión de bases de datos relacionales basado en el modelo relacional, desarrollado a principios de la década de 1970. Sin embargo, fue Oracle Corporation quien lanzó el primer producto comercialmente exitoso basado en SQL en 1979, lo que marcó el comienzo de la adopción generalizada del lenguaje.

Con el tiempo, SQL se convirtió en un estándar de facto en la industria de bases de datos, y organizaciones como ANSI (American National Standards Institute) e ISO (International Organization for Standardization) formalizaron el lenguaje a través de la creación de estándares SQL Esto garantiza la interoperabilidad entre diferentes sistemas de gestión de bases de datos.

SQL es esencial para llevar a cabo operaciones clave en la administración de datos, incluyendo la Creación, Lectura, Actualización y Eliminación de datos, conocidas como operaciones CRUD. Ofrece un conjunto robusto de comandos para ejecutar estas acciones de manera eficiente y coherente en cualquier sistema de base de datos, además de la capacidad de realizar consultas (queries) para obtener información específica de una base de datos.

A lo largo de los años, SQL ha experimentado actualizaciones y mejoras para adaptarse a las

necesidades cambiantes de la industria y abordar nuevos desafíos en el manejo de datos. Esto ha dado lugar a diferentes variantes de SQL, cada una con sus propias características y extensiones específicas. Sigue siendo ampliamente utilizado en una variedad de aplicaciones, desde sistemas empresariales hasta aplicaciones web y móviles. Los sistemas de gestión de bases de datos más populares que admiten SQL incluyen PostgreSQL, SQLite, MySQL y Microsoft SQL Server, entre otros. Estos sistemas ofrecen a los usuarios una amplia gama de herramientas y funcionalidades para administrar y manipular datos de manera eficiente y segura.

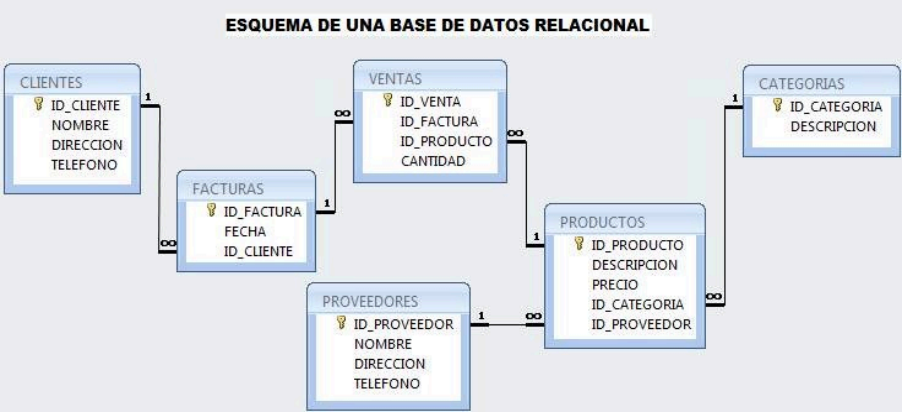


fig 7(esquema bases de datos)

RELACIÓN

En un entorno de microservicios, la combinación de Angular con TypeScript en el frontend y Spring Boot con Java en los microservicios proporciona una arquitectura sólida y flexible para desarrollar aplicaciones distribuidas y escalables.

Angular con TypeScript se destaca por su capacidad para crear interfaces de usuario interactivas y adaptables. Su amplia gama de características, como enlace de datos bidireccional, inyección de dependencias y soporte para componentes, facilita el desarrollo de experiencias de usuario modernas y dinámicas. Además, Angular ofrece herramientas robustas para la gestión del estado de la aplicación, enrutamiento y manejo de eventos, lo que permite construir interfaces de usuario complejas de manera eficiente.

Por otro lado, Spring Boot con Java proporciona un marco de trabajo robusto y maduro para el desarrollo de microservicios. Spring Boot simplifica la configuración y el desarrollo de aplicaciones Java, permitiendo a los desarrolladores centrarse en la lógica de negocio en lugar de en la infraestructura. Además, Spring Boot ofrece características como inyección de dependencias, controladores RESTful, seguridad, y acceso a bases de datos que son fundamentales para el desarrollo de microservicios.

La comunicación entre el frontend y los microservicios se lleva a cabo a través de API RESTful, lo que permite una interacción eficiente y desacoplada entre los diferentes componentes de la

aplicación. Angular utiliza el módulo HttpClientModule para realizar solicitudes HTTP a los microservicios, mientras que Spring Boot proporciona facilidades para crear y exponer API RESTful utilizando anotaciones como @RestController y @RequestMapping.

Cada microservicio tiene su propia base de datos MySQL, lo que garantiza que los datos estén encapsulados y sean gestionados de manera independiente por cada servicio. Esto promueve la modularidad y la escalabilidad de la aplicación, ya que cada microservicio puede escalar horizontalmente según sus necesidades de capacidad de almacenamiento y rendimiento de base de datos.

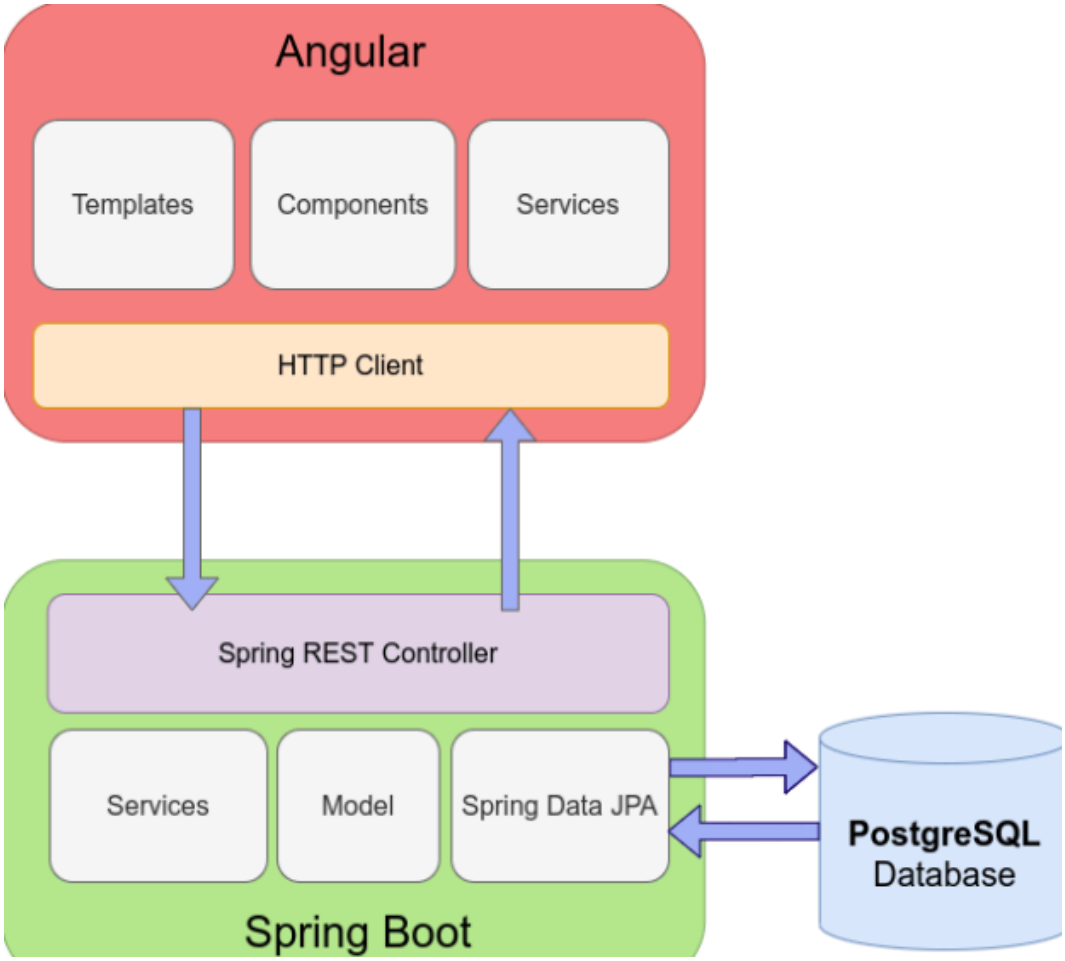


figura 8 (*arquitectura de Aplicación Web aplicando Angular-SpringBoot-MySQL*)

Como se observa en la figura (8) en el lado del cliente, Angular se encarga de la interfaz de usuario, donde los templates representan la estructura visual y los componentes controlan la lógica de la aplicación. Los services manejan la comunicación con el servidor a través de HTTP requests utilizando el HttpClient de Angular.

En el lado del servidor, Spring Boot actúa como el backend de la aplicación. Aquí, los servicios

de Spring gestionan la lógica de negocio, mientras que los modelos representan la estructura de datos de la aplicación. Spring Data JPA se utiliza para interactuar con la base de datos SQL proporcionando una capa de abstracción para realizar operaciones CRUD de manera eficiente.

Cuando se realiza una solicitud desde Angular, el HttpClient envía la solicitud al controlador REST de Spring Boot,el intercambio de datos se realizará en formato JSON . El controlador procesa la solicitud, utilizando los servicios de Spring para llevar a cabo las operaciones necesarias en los datos, utilizando Spring Data JPA para interactuar con la base de datos SQL

en el JPA proporcionan un mecanismo para manipular la base de datos a través de funciones de repositorio sin necesidad de escribir consultas SQL manualmente. Esto se logra utilizando convenciones de nombres y métodos que permiten realizar operaciones comunes como guardar, actualizar, eliminar y recuperar datos de la base de datos de una manera más fácil y segura. Los métodos más comunes son save(), findById(), findAll(), deleteById().

Una vez que se completan las operaciones en el servidor, los datos se envían de vuelta al cliente Angular a través del controlador REST, completando así el ciclo de interacción entre el frontend y el backend de la aplicación.

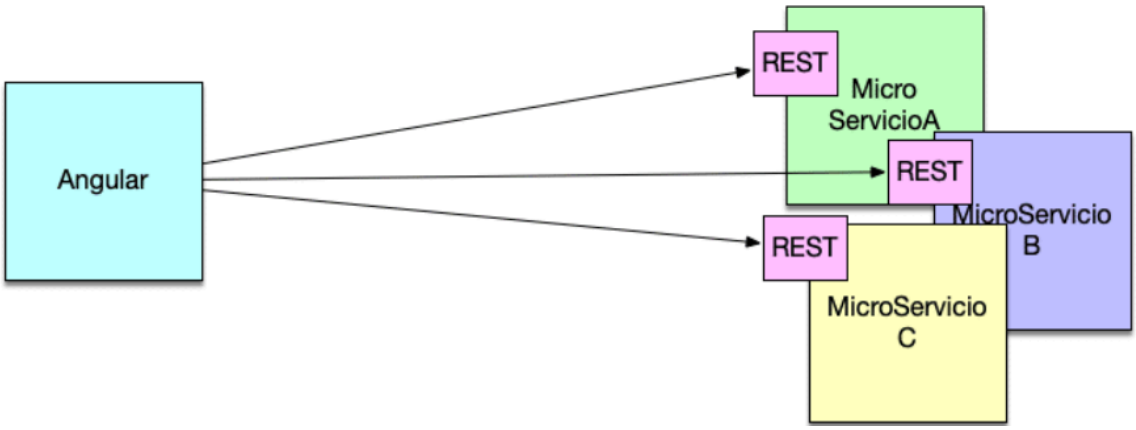


figura 9 (conexión a microservicios)

Como se puede observar en la figura (9) el front end angular puede conectarse a varios microservicios donde cada microservicio tiene su propia lógica de negocio en spring boot y su propia base de datos mediante el consumo de las APIs RESTful proporcionadas por cada microservicio desde la aplicación Angular.

SITUACIONES Y/O PROBLEMAS DONDE SE PUEDEN APLICAR

A continuación se mostrará algunos casos en el que se utilizan estos microservicios rest:

Desarrollo de una aplicación web empresarial:

En el caso de una empresa que necesita una aplicación web para gestionar sus procesos internos, como gestión de proyectos, recursos humanos, seguimiento de ventas, etc. Esta aplicación debe tener un front-end dinámico y atractivo para que los usuarios puedan interactuar de manera efectiva. Debe integrarse con servicios externos para acceder a datos de terceros, como API de pagos, servicios de mapas, etc. La lógica de negocio es compleja y puede requerir procesamiento de datos intensivo y operaciones avanzadas. La información generada y manipulada por la aplicación debe almacenarse de forma estructurada para su posterior consulta y análisis.

Sistema de gestión de inventario:

En este caso, se necesita un sistema de gestión de inventario para una empresa que vende productos físicos. La interfaz de usuario, desarrollada con Angular, permitirá a los usuarios realizar tareas como agregar nuevos productos, actualizar existencias, generar informes, etc. La lógica de negocio, implementada en Spring Boot, se encargará de procesar estas acciones, realizar cálculos de inventario, generar alertas de stock bajo, etc. Los datos sobre productos y existencias se almacenarán en una base de datos MySQL, lo que permitirá un acceso eficiente y estructurado a la información de inventario.

Sistema de gestión de usuarios y autenticación:

Este sistema se enfoca en la gestión de usuarios y sus credenciales de acceso. La interfaz de usuario, desarrollada con Angular, permitirá a los usuarios registrarse, iniciar sesión, actualizar su perfil, etc. La lógica de autenticación y autorización, implementada en Spring Boot, se encargará de gestionar los procesos de autenticación, la asignación de roles y permisos, la gestión de sesiones, etc. La información de usuarios se almacenará en una base de datos MySQL de forma segura y estructurada, lo que garantizará la integridad y confidencialidad de los datos de los usuarios.

Otro caso es el que vamos a explicar más adelante en la documentación sobre una aplicación sobre la gestión de una librería.

ESTILOS Y PATRONES

VENTAJAS Y DESVENTAJAS

Arquitectura/tecnología	VENTAJAS	DESVENTAJAS
Microservicios	Escalabilidad:permite escalar partes especifica de manera independiente facilitando la gestión de carga y el rendimiento Flexibilidad: diferentes tecnologías,bases de datos y lenguaje de programación para cada uno permitiendo elegir la mejor herramienta para cada tarea. Despliegue independiente: Cada microservicio puede ser desarrollado, probado y desplegado de forma independiente, lo que acelera el ciclo de desarrollo y facilita la implementación Mejora la colaboración: Al dividir una aplicación en microservicios, diferentes equipos pueden trabajar en paralelo en diferentes partes de la aplicación, lo que fomenta la colaboración y el trabajo en equipo. Resistencia a fallos: Dado que cada microservicio es independiente, un fallo en uno no necesariamente afecta a los demás. Esto aumenta la tolerancia a fallos y la disponibilidad del sistema en su conjunto. Reutilización de código: Los microservicios pueden ser diseñados para ser reutilizables en diferentes partes de la aplicación o incluso en otras aplicaciones, lo que promueve la eficiencia y reduce el tiempo de desarrollo.	● Complejidad que puede llegar a alcanzar. ● Se pueden producir fallos de comunicación entre microservicios. ● Requiere mucha más infraestructura. ● A mayor infraestructura mayor es el costo.

Spring boot/java	Facilidad de uso: Spring Boot proporciona una forma sencilla de instalar y configurar proyectos, reduciendo la cantidad de código repetitivo y simplificando el inicio. Configuración automática : Spring Boot utiliza valores predeterminados inteligentes para configurar su aplicación, lo que elimina la necesidad de configuración manual y facilita el inicio rápido. Funciones listas para producción : Spring Boot incluye una serie de funciones listas para producción, como seguridad, acceso a datos y una interfaz de usuario basada en web, lo que le permite concentrarse en crear su aplicación principal. Compatibilidad con microservicios : Spring Boot facilita la creación y ejecución de microservicios, proporcionando una forma sencilla de crear e implementar API RESTful. Servidor integrable : Spring Boot incluye un servidor integrado, lo que facilita la ejecución de su aplicación sin la necesidad de un servidor de aplicaciones separado. Capacidad de prueba : Spring Boot proporciona una serie de funciones de prueba, lo que facilita la escritura y ejecución de pruebas para su aplicación. Flexibilidad : Spring Boot es altamente personalizable y se puede ampliar para admitir una amplia variedad de casos de uso, lo que le permite crear la aplicación que necesita.	Complejidad : aunque Spring Boot proporciona muchas funciones listas para usar, el marco puede ser complejo de entender y usar para los desarrolladores que son nuevos en Spring o no están familiarizados con sus conceptos. Personalización limitada : es posible que las configuraciones predeterminadas de Spring Boot no siempre se ajusten a las necesidades de cada proyecto, y personalizar el marco puede resultar difícil. Rendimiento : Spring Boot puede consumir muchos recursos, especialmente en comparación con otros marcos diseñados para aplicaciones más livianas. Gestión de dependencias : Spring Boot puede tener una gran cantidad de dependencias, lo que puede dificultar la gestión y el mantenimiento de la aplicación con el tiempo. Escalabilidad limitada : Spring Boot está optimizado para crear aplicaciones de tamaño pequeño a mediano y puede no ser la mejor opción para aplicaciones de gran escala que requieren mucha personalización. Curva de aprendizaje pronunciada : Spring Boot requiere una inversión significativa en tiempo y esfuerzo para aprender y comprender, especialmente para los desarrolladores que son nuevos en el marco.

		Desafíos de integración : la integración de Spring Boot con otras tecnologías y herramientas puede ser un desafío y puede requerir un esfuerzo significativo, particularmente para los desarrolladores que son nuevos en el marco.
REST/JSON	Simplicidad y uniformidad: REST sigue principios claros y simples, lo que facilita la comprensión y la implementación. Utiliza métodos HTTP estándar (GET, POST, PUT, DELETE) y convenciones de URI, lo que hace que sea fácil para los desarrolladores entender y utilizar. Visibilidad y transparencia: Al usar URI (Uniform Resource Identifier) para identificar recursos y métodos HTTP para operaciones, REST proporciona una interfaz clara y visible que hace que sea fácil de entender y depurar. Interoperabilidad: Debido a su simplicidad y uso de estándares web abiertos como HTTP, URI y JSON, las APIs REST son altamente interoperables y pueden ser utilizadas por una amplia variedad de plataformas y dispositivos. Facilidad de prueba y depuración:La naturaleza stateless de REST facilita la prueba y la depuración de las API, ya que cada solicitud es independiente y no requiere ningún estado del servidor. Cacheabilidad: REST aprovecha el sistema de caché de HTTP, lo que permite a los clientes almacenar en caché las respuestas de las solicitudes, reduciendo así la carga en el servidor y mejorando la velocidad de la aplicación.	Cambio de mentalidad y reciclaje del equipo: Adoptar el enfoque de desarrollo basado en API REST requiere un cambio de mentalidad por parte de los equipos de trabajo. Es necesario abandonar la idea de un servidor centralizado y aprender a trabajar con múltiples servidores independientes. Este cambio puede implicar un período de adaptación y reciclaje del equipo. Complejidad en la gestión del estado: La falta de estado en las APIs REST puede complicar la gestión de la aplicación, ya que se necesita implementar una infraestructura adicional para mantener el estado de la sesión del usuario y gestionar la coherencia de la aplicación en general. Esto puede aumentar la complejidad del desarrollo y requerir un esfuerzo adicional en la implementación y mantenimiento. Posibles mayores tiempos de desarrollo inicial: Al principio, el desarrollo de una API REST puede llevar más tiempo debido a la necesidad de establecer toda la infraestructura y la lógica de la API. Esto puede resultar en un aumento en los tiempos de

	Seguridad: REST permite la implementación de seguridad a través de estándares como HTTPS y OAuth, lo que garantiza la autenticación y la protección de los datos durante la transferencia. Documentación automática: Debido a su estructura uniforme y predecible, las APIs REST pueden ser fácilmente documentadas de forma automática, lo que facilita su comprensión y uso por parte de otros desarrolladores.	desarrollo iniciales antes de que los beneficios a largo plazo puedan ser realizados. Posible rigidez en el desarrollo: La separación entre el backend y los frontends en un entorno basado en REST puede conducir a situaciones de desincronización, donde los equipos trabajan a diferentes ritmos o tienen diferentes prioridades. Esto puede causar rigidez en el desarrollo y dificultades para mantener la coherencia entre las partes del sistema. Mayor necesidad de conocimientos: El desarrollo con API REST puede requerir un conjunto más amplio de conocimientos, ya que los desarrolladores necesitan entender no solo sus lenguajes de programación y bases de datos, sino también el protocolo HTTP y los principios de diseño de API REST. Esto puede implicar un mayor tiempo de aprendizaje y una curva de aprendizaje más pronunciada para algunos equipos.
Angular/typescript	Tipado estático con TypeScript:TypeScript ofrece tipado estático que permite detectar errores en tiempo de compilación, lo que facilita la detección temprana de errores y mejora la robustez y mantenibilidad del código. Arquitectura basada en componentes: Angular utiliza una arquitectura basada en componentes que facilita la reutilización del código y la modularidad del proyecto. Los componentes son elementos autocontenidos	Curva de aprendizaje inicial: Angular tiene una curva de aprendizaje inicial más pronunciada en comparación con otros marcos JavaScript debido a su complejidad y a la necesidad de aprender TypeScript. Esto puede requerir más tiempo y esfuerzo para que los nuevos desarrolladores se familiaricen con la plataforma.

	que encapsulan tanto la lógica como la presentación, lo que facilita el desarrollo y la depuración. Inyección de dependencias:Angular proporciona un sistema de inyección de dependencias integrado que facilita la gestión de las dependencias entre los componentes de la aplicación. Esto promueve la modularidad y la reutilización del código, así como facilita la prueba unitaria. Soporte de bibliotecas y herramientas: Angular cuenta con una amplia variedad de bibliotecas y herramientas desarrolladas por la comunidad y respaldadas por Google. Esto incluye bibliotecas como Angular Material para diseño de interfaz de usuario, Angular CLI para automatización de tareas y Angular Universal para renderización del lado del servidor. Rendimiento: Angular ofrece un buen rendimiento gracias a su optimización de rendimiento incorporada, como el uso de la detección de cambios unidireccional y la optimización del árbol de cambio. Además, la compilación AOT (Ahead-of-Time) mejora el rendimiento de la aplicación al reducir el tamaño del paquete y acelerar la carga inicial. Desarrollo de aplicaciones de una sola página (SPA)	Tamaño de la aplicación: Las aplicaciones desarrolladas con Angular tienden a tener un tamaño de bundle inicial más grande en comparación con otros marcos JavaScript. Esto puede afectar negativamente el tiempo de carga inicial de la aplicación, especialmente en conexiones lentas o dispositivos móviles. Complejidad en proyectos pequeños: Angular puede resultar excesivo para proyectos pequeños o simples debido a su estructura y características complejas. Para aplicaciones simples, un marco más ligero como React puede ser más adecuado y fácil de usar. Dependencia de Angular: Utilizar Angular implica una dependencia directa del marco, lo que puede limitar la flexibilidad y la portabilidad de la aplicación. Cambiar a otro marco o migrar a una versión más reciente de Angular puede requerir un esfuerzo significativo y puede afectar la estabilidad de la aplicación. Documentación y recursos: Aunque Angular cuenta con una documentación oficial completa y una amplia comunidad de desarrolladores, algunos usuarios pueden encontrar que la documentación es compleja o difícil de entender en comparación con otros marcos JavaScript. Esto puede dificultar la resolución de problemas y la adopción de nuevas características para algunos desarrolladores.

MySQL	MySQL software es Open Source Velocidad al realizar las operaciones, lo que le hace uno de los gestores con mejor rendimiento. Bajo costo en requerimientos para la elaboración de bases de datos, ya que debido a su bajo consumo puede ser ejecutado en una máquina con escasos recursos sin ningún problema. Facilidad de configuración e instalación. Soporta gran variedad de Sistemas Operativos. Baja probabilidad de corromper datos, incluso si los errores no se producen en el propio gestor, sino en el sistema en el que está. Su conectividad, velocidad, y seguridad hacen de MySQL Server altamente apropiado para acceder bases de datos en Internet Es encriptada y confiable:las contraseñas están en MySQL El software MySQL usa la licencia GPL	No maneja de manera tan eficiente una base de datos con un tamaño muy grande. varias de las utilidades de MySQL no están documentadas No es del todo intuitivo en comparación de otros programas La Función de Conversión CAST no soporta la conversación a tiempo REAL o BIGINT

PRINCIPIOS SOLID

¿Cómo se aplican los principios SOLID en los microservicios?

- **S - Responsabilidad única:** Los microservicios deben ser especialistas o responsables de una única funcionalidad que puede estar relacionada con el punto de vista del negocio, por ejemplo, un microservicio responsable de la información personal del cliente; o desde el punto de vista técnico, por ejemplo, consumir o producir eventos en un sistema de mensajería. Cada microservicio debe tener una tarea específica y no debe ser responsable de múltiples aspectos o funcionalidades. Aplicar este principio ayuda a mantener la

claridad en la responsabilidad de cada microservicio y facilita la gestión y escalabilidad del sistema.

- **O - Abierto/Cerrado:** Todas las nuevas responsabilidades relacionadas con el mismo elemento deberían agregarse en un nuevo microservicio en lugar de ampliar la responsabilidad de uno ya existente. Por ejemplo, desde un punto de vista de negocio, si se desea diseñar un microservicio para la información de la cuenta bancaria del cliente, no se debe extender el microservicio existente de información personal del cliente. Cada microservicio debe tener una responsabilidad claramente definida y no debe asumir tareas adicionales más allá de su ámbito inicial. Esto ayuda a mantener la modularidad y la cohesión en la arquitectura de microservicios, lo que facilita su mantenimiento y evolución a largo plazo.
- **L - Sustitución de Liskov:** Una nueva versión de un microservicio siempre debería poder sustituir a una versión anterior sin romper nada. Una sustitución nunca debería romper ningún sistema que actualice sus versiones de microservicios usados.
- **I - Segregación de la interfaz:** Un microservicio no debe exponer métodos que no estén directamente relacionados. Por ejemplo, si se tiene un microservicio que ofrece múltiples funciones que no están directamente relacionadas y solo se utiliza el 20% de la funcionalidades que ofrece este microservicio, entonces es probable que en realidad no se esté utilizando un verdadero microservicio. Es necesario dividir el microservicio en funcionalidades más específicas y pequeñas.
- **D - Inversión de dependencia:** Un microservicio no debe llamar directamente a otro microservicio. En cambio, se puede utilizar un módulo de descubrimiento de servicios para localizar la instancia de microservicio a llamar o a través de llamados a interfaces.

ATRIBUTOS DE CALIDAD

Atributos de calidad asociados con los microservicios:

- **Escalabilidad:** Los microservicios permiten escalar individualmente los distintos componentes de una aplicación, lo que implica la capacidad de asignar recursos adicionales a aquellos microservicios que requieran un rendimiento superior, y ajustar la capacidad de manera acordes a las necesidades del negocio.
- **Modularidad:** Los microservicios permiten dividir una aplicación en componentes que pueden ser desarrollados, desplegados y escalados de forma independiente según sea necesario. Esto facilita la evolución y mantenimiento del sistema a largo plazo.
- **Funcionalidad:** Los microservicios pueden ser diseñados para cumplir funciones específicas de manera precisa y completa, lo que facilita la adaptación de funcionalidades del sistema a las necesidades del negocio.
- **Confiabilidad:** Pueden proporcionar confiabilidad al ser diseñados para operar de manera independiente, lo que reduce el riesgo de fallos en cascada y permite la recuperación rápida ante fallos en componentes individuales.
- **Eficiencia de desempeño:** Los microservicios pueden mejorar la eficiencia del desempeño al permitir el escalado horizontal y vertical de componentes específicos según la demanda, lo que optimiza el uso de recursos y mejora los tiempos de respuesta.

- **Compatibilidad:** La independencia entre microservicios puede facilitar la compatibilidad al permitir la integración con sistemas existentes y la adopción de tecnologías emergentes sin afectar al sistema.
- **Reusabilidad:** Los microservicios fomentan la reusabilidad al separar la lógica de negocio en servicios independientes que pueden ser utilizados por diferentes partes del sistema.
- **Portabilidad:** La encapsulación e independencia entre microservicios puede mejorar la portabilidad al facilitar la migración y el despliegue en diferentes entornos de ejecución sin modificaciones significativas.

CASOS DE ESTUDIO

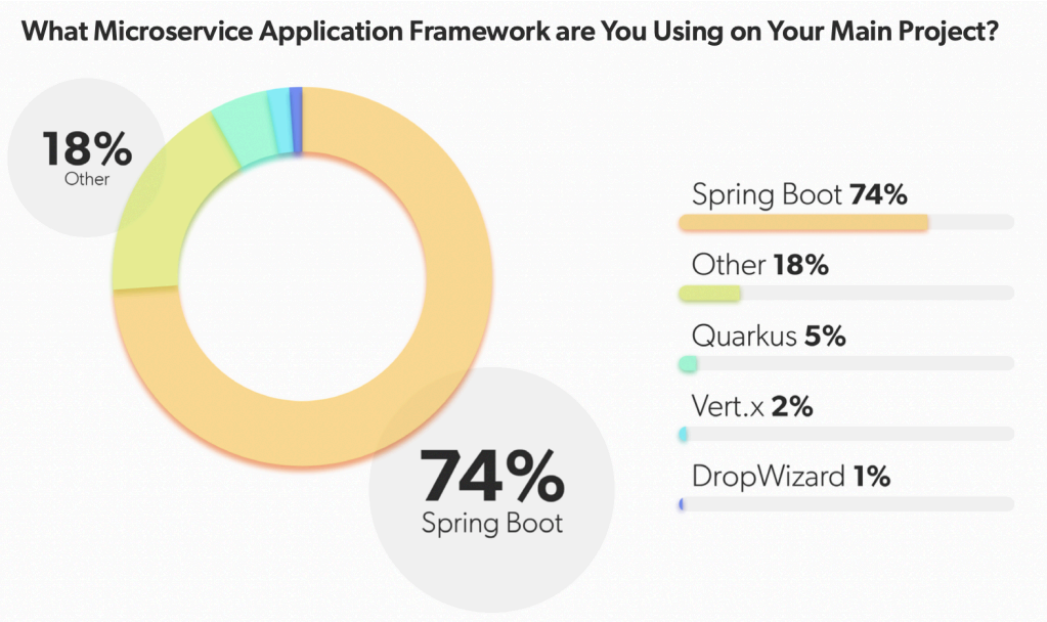
- **Netflix:** Desde hace unos años, esta plataforma ha migrado hacia una arquitectura basada en microservicios, coincidiendo con su crecimiento significativo. Diariamente, gestiona aproximadamente mil millones de llamadas a sus diversos servicios y es capaz de adaptarse a más de 800 tipos de dispositivos por medio de su API de streaming de video, la cual ofrece un servicio más estable. Por cada solicitud que se le hace, esta realiza 5 solicitudes a diferentes servidores para no perder la continuidad de transmisión.
- **Ebay:** Una empresa reconocida por su visión de vanguardia, ha sido pionera en la adopción de tecnologías innovadoras como Docker, además de los microservicios. Su aplicación principal está compuesta por múltiples servicios independientes, cada uno encargado de ejecutar la lógica específica de las diversas áreas funcionales ofrecidas a los clientes.
- **Spotify:** La arquitectura de Spotify se basa completamente en microservicios. Fue una de las primeras empresas en defender los beneficios de las arquitecturas de microservicios. Utiliza una arquitectura de microservicios para impulsar su servicio de streaming de música. Esta arquitectura se compone de cientos de servicios responsables de diferentes aspectos de la aplicación. Estos servicios se desarrollan, despliegan y escalan de manera independiente, lo que permite que Spotify pueda iterar y lanzar nuevas funciones rápidamente.
  Los microservicios le permiten a Spotify gestionar un gran volumen de tráfico y varios equipos trabajando simultáneamente en diferentes servicios. Estos han sido fundamentales para el éxito de Spotify como servicio de música por medio de streaming.
  La arquitectura de software basada en microservicios de Spotify depende en gran medida de los servicios. El backend de Spotify requiere de cientos de servicios, la mayoría muy pequeños y sencillos, pero que deben servirse en tiempo de ejecución sin demora.
  Para atender a estos cientos de servicios, el equipo técnico de Spotify desplegó una arquitectura basada en microservicios, que sirve para atender una sola necesidad de servicio a la vez, ya sea la recuperación de canciones, el intercambio de recomendaciones, la búsqueda, o simplemente la verificación de usuarios (tanto para el modelo gratis como el de pago). https://www.techaheadcorp.com/blog/decoding-software-architecture-of-spotify-how-microservices-empowers-spotify/

DETALLES DE LA IMPLEMENTACIÓN

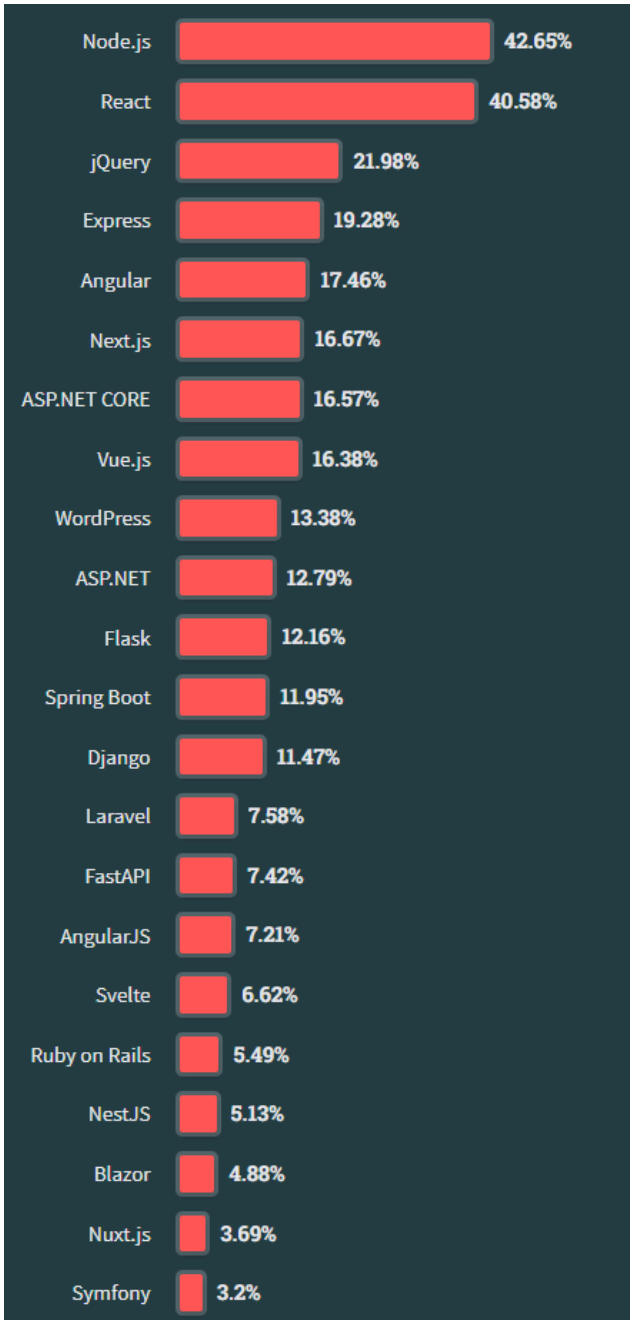
Para este taller, se llevo a cabo el desarrollo de la solución con el stack planteado, en el siguiente repositorio de GitHub se puede encontrar los archivos fuentes del proyecto:
Raaiinn/Stack-Microservices-MAS: MAJ -> MySQL, Angular, SpringBoot(Java) (github.com)

ANÁLISIS DE MERCADO

- **Spring Boot/Java:**
  - Según el Informe de Productividad del Desarrollador de Java 2022, el 32% de los de los desarrolladores utilizan los microservicios como arquitectura principal y el framework para el desarrollo de microservicios más utilizado es Spring Boot con un porcentaje del 74%:



  - Según la página “‘Glassdoor´”, el salario base de un desarrollador Spring Boot en Estados Unidos se encuentra entre los $64.000 y $120.000 USD al año.
- **Angular/TypeScript:**
  - Según la Encuesta para Desarrolladores hecha por Stack Overflow de 2023, entre los frameworks web y tecnologías más utilizadas, Angular ocupó el 5to puesto con un porcentaje del 17.28%. Los frameworks y tecnologías que superaron a Angular fueron Node.js, React, jQuery y Express.



    - Según la página "Glassdoor", el rango salarial base para un desarrollador de Angular en los Estados Unidos, se encuentra entre $69.000 y $129.000 USD al año.
- **MySQL:** Algunos datos según el reporte hecho por Gitnux sobre MySQ:
    - Es uno de los sistemas de gestión de base de datos más populares y utilizados del mundo. Tiene una cuota de mercado del 32.36%, convirtiéndolo en el segundo sistema más popular después de Microsoft SQL Server.
    - MySQL es utilizado por el 79.6% de todos los sitios web que utilizan un sistema de gestión de bases de datos conocido.
    - El 53% de las empresas de la industria TI y servicios utilizan MySQL como

solución de gestión de bases de datos.

- Es utilizado por más del 25% de los sitios web que funcionan con tecnologías Java.
- El salario promedio de un desarrollador de MySQL en los Estados Unidos es de $87.000 USD al año.