Benemérita Universidad Autónoma de Puebla
Facultad de Ciencias de la Computación

Propuesta de una arquitectura escalable basada en microservicios y análisis de datos para el desarrollo de aplicaciones de toma de decisiones empresariales.

Tesis presentada para obtener el grado de:
Licenciatura en Ingeniería en Tecnologías de la Información

Presenta:
Hugo Alexis Chiquito Onofre

Asesor:
Dr. Abraham Sánchez López

Marzo 2024

Dedicatoria

A mi madre, Sonia Onofre, cuya fuerza, amor, y sabiduría han sido mi guía en cada paso del camino. Esta obra es el fruto de tus sacrificios y tu inquebrantable fe en mí.

A mi padre Hugo Chiquito, por su esfuerzo y su ejemplo de perseverancia, enseñándome que los sueños se construyen sobre la base del trabajo.

A mi hermana Alyson Chiquito, mi fuente de inspiración diaria, recordándome siempre la importancia de seguir adelante, sin importar los obstáculos.

A Dios, quien ha sido mi luz en los momentos oscuros y mi fortaleza en los momentos de debilidad.

Y en memoria de mi abuela, Bernardina Heredia de Onofre cuyo amor y creencia en mis capacidades me alentaron a alcanzar grandes metas.

Agradecimientos

En este momento tan importante en mi vida, cuando concluyo una etapa fundamental dentro de mi formación académica, me siento profundamente agradecido con todas las personas que han sido parte de este viaje. Desde aquellos primeros días en la escuela primaria hasta este instante, cada maestro, amigo y alma generosa que me extendió su mano en momentos de adversidad, quiero decirles, gracias.

Me gustaría dar un agradecimiento especial al Dr. Abraham Sánchez, cuya guía y apoyo han sido cruciales no solo en la elaboración de esta tesis, sino a lo largo de toda la carreara.

A mis amigos, fieles compañeros, gracias por su apoyo incondicional y los buenos momentos que pasamos juntos, el tenerlos hizo mi vida universitaria más alegre y sencilla.

A mi familia, mi refugio y mi fuerza, gracias por su amor y apoyo constante. Desde que tomé la decisión de estudiar en otro estado, han estado allí para mí, asegurándose de que nunca me falté nada y brindándome la motivación para seguir.

Finalmente, a todos quienes han formado parte de esta travesía, en momentos grandes y pequeños, su presencia ha sido un regalo invaluable. Este logro no solo es un testimonio de mi esfuerzo, sino también del gran apoyo que he recibido. Gracias por creer en mí, por impulsarme a seguir adelante y por estar ahí en cada paso del camino.

"El éxito no es definitivo, el fracaso no es fatal: lo que cuenta es el valor para continuar." - Winston Churchill

Índice general

Índice de figuras

1. Introducción

Los sistemas informáticos han tomado gran relevancia dentro del contexto empresarial en los últimos años debido a la gran ventaja que estos suponen dentro de la toma de decisiones. Esto ha traído consigo nuevos desafíos para las arquitecturas actuales, pues día tras día dentro de la industria el número de usuarios, así como la información que estos producen va en aumento. En consecuencia, las arquitecturas han tenido que evolucionar para hacer frente a una mayor carga de trabajo.

Es aquí donde la arquitectura de microservicios toma valor, pues al trabajar con microservicios se permite diseñar sistemas de información capaces de adaptarse a diferentes cargas de trabajo, escalando según sea necesario, ya sea dentro de un servidor propio o en la nube.

Gracias a esto, los sistemas son capaces de manipular grandes cantidades de datos. Es aquí donde el análisis de datos entra en juego, pues un conjunto de datos por sí solo no brinda ningún beneficio a la industria, estos datos necesitan ser debidamente preparados y consumidos para ofrecer información relevante.

Dentro de este trabajo de tesis se propone una arquitectura escalable con un enfoque en el desarrollo de aplicaciones que apoyen a la recolección y análisis de datos. La propuesta se basa en la combinación de diferentes tecnologías, arquitecturas y patrones de diseño, esto con el objetivo de ofrecer una infraestructura flexible y ágil que permita a las empresas reaccionar eficazmente ante las crecientes demandas.

A lo largo del documento, se exploran los fundamentos teóricos para la arquitectura propuesta y se detallan las tecnologías que fueron aplicadas. De igual manera se brinda un ejemplo de implementación de la arquitectura propuesta.

1.1. Objetivos

1.1.1 Objetivo general

Proponer, desarrollar y evaluar una arquitectura de software altamente escalable para sistemas que apoyan a la toma de decisiones empresariales utilizando análisis de datos.

1.1.2 Objetivo específico

1. Recolectar y analizar información sobre las arquitecturas de software.
2. Identificar los factores clave de una arquitectura escalable para un sistema de toma de decisiones.
3. Diseñar una arquitectura que cumpla con los factores clave identificados, considerando la gestión de datos y procesamiento en tiempo real.
4. Implementar un prototipo funcional de la arquitectura propuesta.
5. Evaluar el rendimiento y efectividad del prototipo en términos de escalabilidad, analizando factores como tiempo de respuesta y uso de recursos.
6. Proponer mejoras y recomendaciones.

1.2. Estructura de la tesis

Capítulo 1. Introducción de la tesis, se define brevemente el objetivo principal de este trabajo de tesis, así como su alcance y la problemática que se busca resolver.

Capítulo 2. Se hace un análisis sobre las arquitecturas ya existentes y su evolución a lo largo de los años.

Capítulo 3. Se profundiza sobre los conceptos necesarios para entender más sobre la arquitectura propuesta, ofreciendo definiciones sobre los elementos y principios utilizados dentro de la arquitectura de software.

Capítulo 4. Se detalla la propuesta de la arquitectura siguiendo como base los fundamentos establecidos en el capítulo 3. Dentro de este capítulo se detalla el por qué y que tecnologías, conceptos o herramientas se utilizan para el desarrollo.

Capítulo 5. Dentro de este capítulo se ofrece un ejemplo de implementación sobre la arquitectura propuesta, en él se profundiza en mayor manera sobre como implementar las herramientas, componentes y sus relaciones.

Capítulo 6. Se da una conclusión sobre el trabajo de tesis, analizando las ventajas y los resultados obtenidos.

Capítulo 7. Se listan todas las fuentes consultadas para la elaboración de este documento.

2. Estado del arte

2.1. Introducción a la arquitectura de software

La arquitectura de software es un área dentro de la ingeniería de software que a lo largo de los años ha ido evolucionando, adaptándose a las nuevas tendencias dentro del desarrollo de software.

Coloquialmente, el término 'arquitectura', hace referencia a la disciplina encargada de la planificación y el diseño de distintas construcciones. De manera similar, la arquitectura de software se encarga de guiar el proceso de planificación y diseño del desarrollo de software a través de principios y patrones que rigen el desarrollo, asegurando de esta forma una estructura sólida, eficiente y adaptable.[35]

Al igual que dentro de la arquitectura 'común' existen diferentes procesos, planos y diseños para cada tipo de estructura, dentro de la arquitectura de software existen una gran variedad de arquitecturas y elementos, cada una adaptada a situaciones o problemas diferentes.

Es aquí donde entra la importancia de conocer los diferentes tipos existentes, así como sus características, elementos, ventajas y mejores casos de implementación. Al conocer todos estos detalles podemos elegir, adaptar e inclusive crear nuestra propia arquitectura de software.[3]

Dicho esto, a continuación, se hará un recorrido sobre la evolución que ha tenido la arquitectura de software a lo largo de los años, destacando el problema al que buscaba dar solución y un resumen sobre sus elementos y características principales.

2.2. Evolución de la arquitectura de software

2.2.1 Origen y primeros elementos de la arquitectura de software

En un inicio, el desarrollo de software carecía de guías o principios para la construcción de aplicaciones, las mismas tecnologías con las que se trabajaban no eran tan complejas o variadas como lo son hoy en día, en aquellos tiempos las aplicaciones se ejecutaban sobre enormes computadoras que rara vez tenían la posibilidad de interconectarse con otras. Como resultado, las aplicaciones de software que se desarrollaban no eran complejas y eran construidas de manera "centralizada", es decir, todos los recursos que necesitaba la aplicación para funcionar estaban en un solo lugar, una sola unidad. [18]

Dentro de este contexto es donde surgen las primeras arquitecturas, siendo una de las primeras la arquitectura monolítica.

Arquitecturas monolíticas

La arquitectura monolítica tiene como características principales integrar y desplegar todos los elementos de una aplicación dentro de una sola unidad de código, haciendo que no dependa de aplicaciones externas para funcionar. [2]

Este enfoque proporciona un desarrollo rápido y sencillo al desarrollar e implementar aplicaciones pequeñas y medianas. Al mantener todos los elementos dentro de una sola unidad, la relación que hay entre ellos, así como las pruebas y el mantenimiento son más fáciles de gestionar. [2]

Sin embargo, la escalabilidad de esta arquitectura es limitada, con el paso del tiempo la aplicación tendría problemas al adaptarse a una mayor carga de trabajo, así como a las nuevas tecnologías. Al tratarse de una sola unidad conformada por un conjunto de elementos, el más mínimo cambio que se haga dentro de alguno de sus elementos puede ocasionar terribles problemas, si un fallo ocurre en alguno de ellos el funcionamiento del sistema completo se vería afectado. [18]

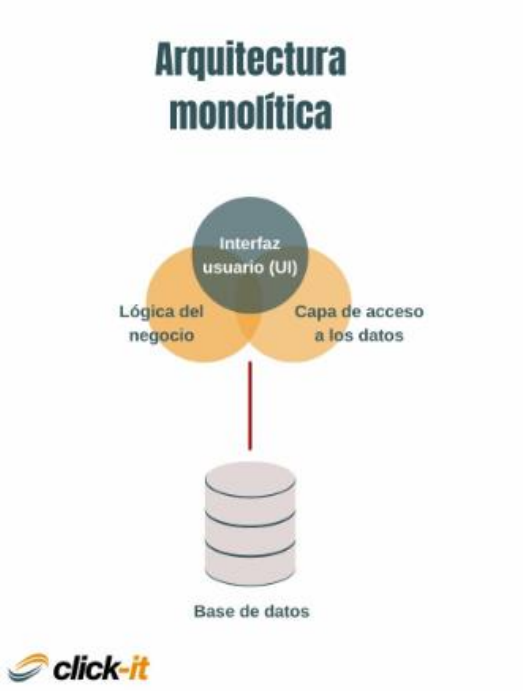


Figura 2-1 Arquitectura monolítica, recuperado de [5]

2.2.2 Transición hacia arquitectura modulares

El concepto de programación modular fue presentado por primera vez en el Simposio Nacional del año 1968 por el ingeniero Larry L. Constantine. Este nuevo concepto supuso una revolución para el área del desarrollo de software pues proponía romper y separar los clásicos sistemas monolíticos en diferentes módulos, cada uno de estos con una funcionalidad específica. [18]

La transición hacia arquitecturas modulares surgió como una respuesta a las limitaciones de los sistemas monolíticos frente a la necesidad de manejar sistemas más grandes y con requisitos más complejos. Esta necesidad de adaptabilidad y escalabilidad impulsó el desarrollo de paradigmas como la programación estructurada y posteriormente, la programación orientada a objetos, marcando un hito en la evolución del diseño de software hacia sistemas más flexibles y mantenibles. [18]

Arquitecturas basadas en capas

Su característica principal es la de distribuir las responsabilidades entre "capas" que pueden ser definidas según sea necesario en la implementación. Dividir la aplicación en distintos segmentos por responsabilidades ofrece diversas ventajas, siendo las más relevantes la modularidad, reutilización, escalabilidad y flexibilidad. [4]

Al ofrecer una fuerte flexibilidad y libertad esto puede conllevar a algunas desventajas como una alta complejidad del proyecto que si no se maneja o planifica correctamente podría conllevar a un consumo de recursos excesivo tanto de tiempo y esfuerzo de desarrollo como de rendimiento y financiero.

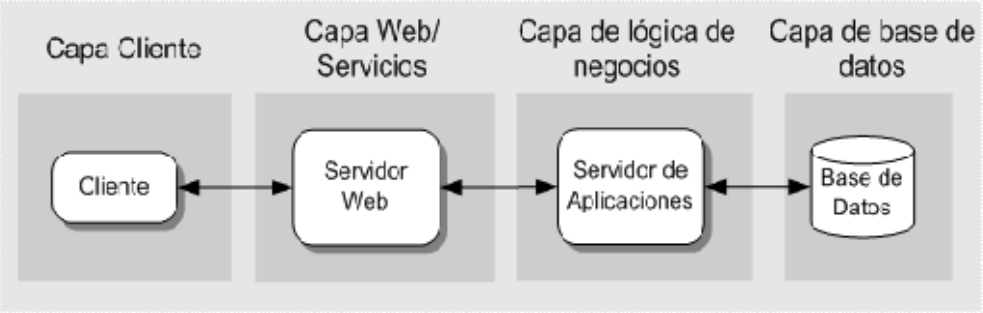


Figura 2-2 Arquitectura basada en capas, recuperado de [35https://geeks.ms/jkpelaez/2009/05/30/arquitectura-basada-en-capas/]

Arquitecturas orientadas a servicios (SOA)

A finales de la década de los 90 e inicios de los 2000 con la llegada del internet, las computadoras obtuvieron la capacidad de compartir recursos y comunicarse entre ellas. Dentro de este contexto, la arquitectura orientada a servicios (SOA) surge como una solución para el desarrollo de aplicaciones capaces de compartir y consumir recursos de otras aplicaciones. [18]

Esta nueva arquitectura trajo consigo nuevos conceptos y tecnologías, como servicios web, cuya función principal radica en habilitar la comunicación entre diferentes aplicaciones y sistemas sin importar las tecnologías empleadas en el

desarrollo de estas. Además, al tener como característica nueva la comunicación entre aplicaciones, promovió el uso de XML. [29]

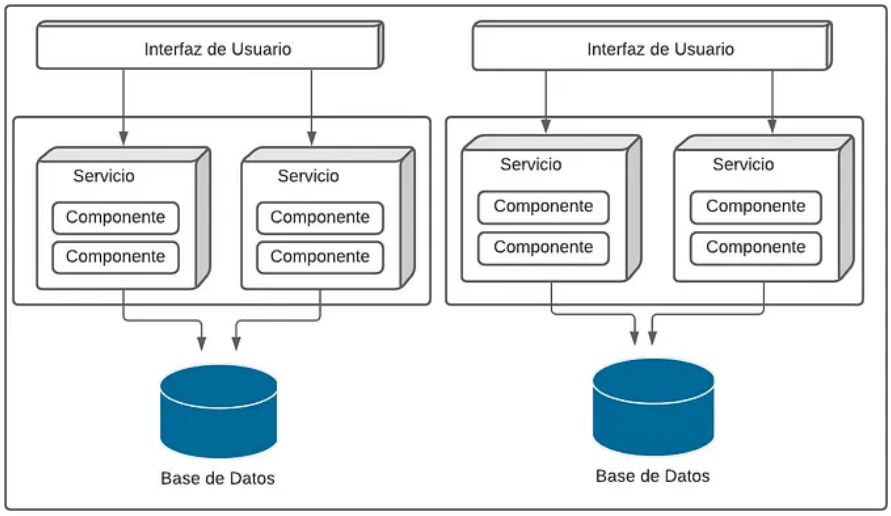


Figura 2-3 Ejemplo arquitectura basada en servicios, recuperado de [36]

Arquitecturas basadas en eventos

Esta arquitectura como su nombre lo indica, tiene como característica principal realizar determinadas acciones en base a eventos. Los eventos registran sucesos o cambios importantes en el sistema, estos pueden provenir del usuario o de fuentes externas (sensores u otros sistemas). [28]

Para ejecutar estas acciones, la arquitectura o el sistema debe contar con al menos un productor (publicador) y un consumidor(suscriptor). El primero se encarga de detectar el evento y transmitirlo de manera asíncrona hacia el consumidor.

Al transmitir el mensaje entre estos dos componentes se utiliza un administrador de middleware, existen varias opciones, pero para este proyecto vamos a destacar Kafka.

Dentro de esta arquitectura existen diferentes formas en las que puede construirse, la primera es basándose en un modelo de publicación/suscripción o en un modelo de flujo de eventos. [28]

En el modelo de publicación y suscripción no se utiliza un registro de eventos, simplemente cuando un evento ocurre se les notifica a todos los componentes que estén “suscritos” a dicho evento.

Por otra parte, el modelo de flujo de eventos utiliza un registro de eventos para funcionar, de este registro de eventos surgen dos diferentes formas de utilizarlo. La primera consiste proceso sencillo, donde básicamente cuando ocurre un evento, este desencadena de forma inmediata una acción dentro del consumidor. La segunda conlleva un procesamiento complejo que requiere que el consumidor procese los eventos en busca de algún patrón.

La ventaja principal de esta arquitectura está en la capacidad que tiene para tomar decisiones de forma prácticamente instantánea.

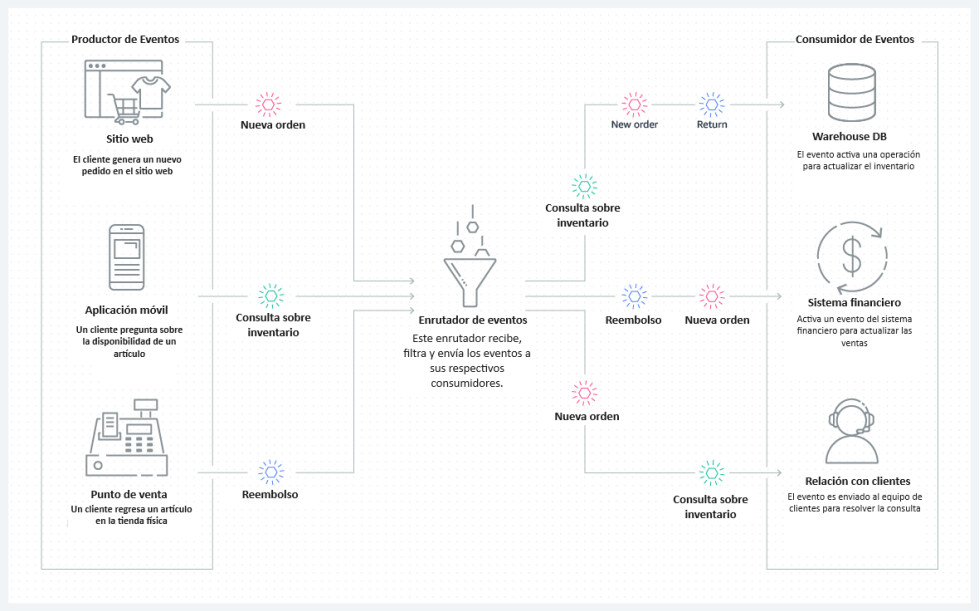


Figura 2-4 Ejemplo arquitectura basada en eventos, recuperada y traducida de [37]

Arquitecturas basadas en microservicios

Años más tarde con la llegada de nuevas tendencias como el computo en la nube, Docker, DevOps, etc. La arquitectura SOA se enfrentó a nuevos desafíos que no pudo encarar eficazmente, siendo estos principalmente la necesidad de una mayor agilidad y escalabilidad. [18]

SOA estaba destinada principalmente en compartir servicios a gran escala, mientras que la arquitectura basada en microservicios su principal enfoque radica en separar en pequeños servicios independientes la aplicación, esto con el objetivo de brindar a cada uno de estos servicios la capacidad de escalar y trabajar independientemente comunicándose vía Http o haciendo uso de un middleware.

Con la llegade de los microservicios la arquitectura basada en eventos tomo fuerza adaptándose a este nuevo concepto junto con la llegada de nuevas tecnologías como Kafka, RabbitMQ entre otras.

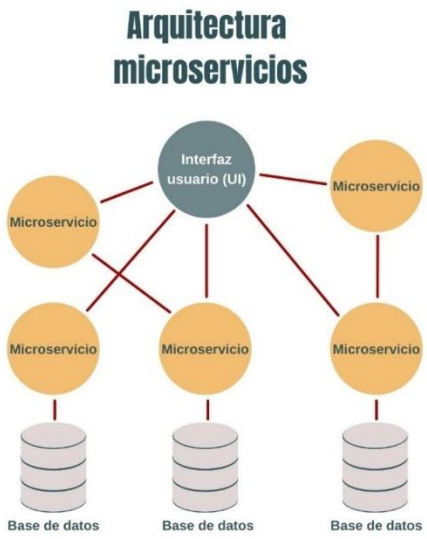


Figura 2-5 Arquitectura de microservicios, recuperada de [5]

Arquitecturas sin servidor (Serverless)

Esta arquitectura fue popularizada en el año 2014 por Amazon con el lanzamiento de AWS Lambda.

Este nuevo enfoque se puede ver como la continuación de las arquitecturas basadas en microservicios, en este enfoque la granularidad es más fina, teniendo como componente principal funciones. Además, con este enfoque los desarrolladores pueden olvidarse de la configuración y limitaciones de los servidores para centrarse única y exclusivamente en la funcionalidad. El resto de las tareas relacionadas al servidor quedan a cargo del proveedor de servicios FaaS.

El uso de este tipo de arquitecturas ayuda en la reducción de costos tanto en infraestructura como en la mano de obra, simplifica el proceso de escalado de las aplicaciones. Sin embargo, la implementación de este tipo de arquitectura puede suponer un desafío por el gran esfuerzo inicial para adaptar las estructuras a esta nueva modalidad, por otra parte, al depender completamente del proveedor de servicios FaaS limita los cambios, lenguajes y tecnologías. [17]

Figura 2-6 Funcionamiento de arquitectura Serverless, recuperado y traducido de [37]

2.3. Síntesis y transición hacia la propuesta de la arquitectura

A lo largo de este capítulo, se han explorado las principales arquitecturas que han surgido a lo largo de la historia del desarrollo del software, tras analizar la evolución de estas, se puede notar que cada nueva arquitectura busca mejorar la escalabilidad y la flexibilidad, así como la importancia que tiene adaptarse a las nuevas tecnologías y las cambiantes demandas del mercado.

A partir de estos conceptos, esta tesis propone una arquitectura escalable basada en microservicios que brinde una infraestructura capaz de soportar el procesamiento intensivo de datos, así como simplificar y maximizar la eficiencia y fluidez del análisis de datos. Dentro de la arquitectura propuesta, se busca implementar los principios fundamentales de las arquitecturas exploradas, siendo estos: escalabilidad, modularidad, autonomía y agilidad. Buscando así ofrecer una arquitectura que cumpla con los objetivos estratégicos de las empresas y que a su vez sirva de apoyo para la siguiente generación de arquitecturas.

En el siguiente capítulo se detalla la arquitectura propuesta, adentrándose en conceptos, definiendo los elementos de la arquitectura, la interacción entre ellos, su contribución e impacto, así como las tecnologías utilizadas para el desarrollo.

3. Fundamentos de diseño y arquitectura

3.1. Definición de la arquitectura de software

Para reforzar el concepto de arquitectura de software, a continuación, se analizarán diversas definiciones de arquitectura de software aportadas por autores importantes dentro del campo de la ingeniería de software. Haciendo énfasis en identificar los elementos comunes e indispensables entre las diferentes perspectivas de los autores. Esto con el objetivo de demostrar que la arquitectura propuesta se encuentra basada en principios ampliamente reconocidos dentro de la ingeniería de software.

Definición 1: Bass, Clements, y Kazman

"La arquitectura de software se define como las estructuras necesarias para entender el sistema. Definiendo sus elementos, las relaciones entre ellos y las propiedades de ambos. La arquitectura de software es el diseño de alto nivel de un sistema en su contexto, un marco para entender los componentes y la relación entre ellos" (Bass, Clements, & Kazman, 2022).[1]

Definición 2: Martin Fowler

"La arquitectura de software se refiere a las decisiones importantes que se toman con respecto a la organización de un sistema de software, la selección de los elementos estructurales y sus interfaces, así como su comportamiento dentro de la colaboración de esos elementos" (Fowler, 2003)[2].

[1] Bass, L., Clements, P., & Kazman, R. (2022). Software Architecture in Practice (4a ed.). Addison Wesley.

[2] Fowler, M. (2017). Patterns of Enterprise Application Architecture.

En resumen, se puede definir a la arquitectura de software como la estructura que resulta del proceso de definir y seleccionar un conjunto de elementos junto con sus interacciones, con el objetivo de satisfaces eficientemente los requisitos funcionales y no funcionales.

3.2. Elementos de la arquitectura de software

En vista de la definición proporcionada, a continuación, se detallan los principales elementos de una arquitectura de software, ofreciendo una descripción, así como ejemplos para cada uno.

Cabe destacar que, hasta la fecha, los autores no han llegado a un consenso sobre cuáles son los elementos de una arquitectura de software, para algunos autores ciertos elementos pueden diferir con sus definiciones, por ejemplo, para algunos un 'estilo arquitectónico', no es lo mismo que un 'patrón. Esta variedad de definiciones o discrepancia entre autores se debe a diferentes factores que vale la pena tomar en cuenta para futuros trabajos, siendo estos principalmente:

- Avances tecnológicos.
- Diversidad de sistemas.
- Variedad de contextos empresariales.

Tomando en cuenta esto, la estructura de elementos que se ofrece más adelante busca mantener los elementos principales en los que los autores (al menos, la mayoría) están de acuerdo. [6]

Componentes

Define los elementos funcionales que componen el sistema. Dependiendo el estilo arquitectónico elegido pueden varias desde finos microservicios hasta grandes monolitos. Es de suma importancia definir el nivel de granularidad y tipos de

componentes con los que se trabajara, adaptándose lo mejor posible a los requisitos del sistema.

Conectores

Se refiere a los mecanismos que se utilizan en la interacción entre los componentes de un sistema. Estos pueden ser asíncronos y síncronos según sea necesario. Ejemplos de estos pueden ser API's, servicios web, protocolos de comunicación de bajo nivel como TCP/IP, etc. El papel principal de los conectores se encuentra en definir y facilitar el flujo de la comunicación. La elección de estos suele estar fuertemente ligada con la elección de tipo de módulos, así como los estilos de arquitectura que se planean utilizar.

Almacenamiento de datos

Los sistemas de almacenamiento de datos tienen como objetivo principal facilitar la manipulación de los datos dentro del sistema, para esto pueden utilizarse diversas tecnologías o tipos de sistemas de almacenamientos, ya sea bases de datos SQL o NoSQL, así como sistemas externos como contenedores en la nube, sistemas de archivos, etc. De igual forma la elección de que sistemas de almacenamiento utilizar y cómo hacerlo depende de gran medida de los requisitos y el estilo de arquitectura que se planea utilizar, por ejemplo, en una arquitectura monolítica lo usual sería mantener una sola base de datos que sea manipulada por toda la aplicación junto con un sistema de almacenamiento de archivos. Por otra parte, en una arquitectura basada en microservicios se esperaría contar con varias bases de datos, así como sistemas de almacenamiento especializados, ya sea en nube o de igual forma localmente.

La correcta elección del modelo y estructuras de almacenamiento de datos es crucial para el rendimiento, escalabilidad y la capacidad de análisis de los sistemas.

Interfaces de usuario

Son las encargadas de exponer los puntos de entrada y salida de datos hacia los usuarios. Aunque para el desarrollo de esta tesis no se entrara en gran detalle sobre las interfaces, se debe estacar que es importante ofrecer un correcto diseño para garantizar la entrada y salida correcta de datos. Dentro de las posibles opciones para crear la interfaz de usuario se tiene la interfaz gráfica de usuario (mayormente conocida como GUI por sus siglas en ingles), interfaz de línea de comandos (CLI), una de las primeras interfaces desarrolladas y hasta API's o puntos de acceso diseñados para ser utilizados por otros sistemas.

Middleware

Conjunto de servicios que actúan como intermediarios o puentes entre diferentes componentes de la arquitectura y que a su vez ofrece funcionalidades más complejas, como la gestión de transacción de datos, seguridad, mensajería, gestión de base de datos, brokers, etc.

En resumen, aunque el middleware puede sonar muy similar a un conector, este tiene objetico principal abstraer la complejidad y proporcionar una base para el desarrollo del software.

3.3. Patrones de diseño, patrones de arquitectura y estilos arquitectónicos

Para el desarrollo de una arquitectura robusta que cumpla y se adapte a diversos principios como escalabilidad, flexibilidad, mantenibilidad, etc. Es necesario aplicar una serie de medidas que han sido probadas y creadas para la solución de problemas específicos dentro del desarrollo del software.

Patrones de diseño

Comencemos definiendo el concepto de patrones de diseño, los patrones de diseño son soluciones habituales a problemas comunes dentro del desarrollo de software, están probados, son reutilizables y personalizables. Su enfoque principal se encuentra en definir a nivel de clase la interacción entre los objetos. [24]

Se dividen en 3 categorías en base a su propósito, siendo estas: patrones creacionales (Ejemplo: Singleton), estructurales (Ejemplo: Adapter) y de comportamiento (Ejemplo: Strategy)[23].

Estilo de arquitectura

Los estilos de arquitectura o también conocidos como estilos arquitectónicos definen la organización del sistema en orden de favorecer los principios deseados como escalabilidad o reusabilidad. Ejemplos de esto sería el estilo de arquitectura basado en microservicios, en el que la organización tiene como foco principal promover y mejorar la capacidad de adaptabilidad y escalabilidad del sistema haciendo uso de componentes autónomos, reusables, independientes y de granularidad fina.

Patrones de arquitectura

Representan buenas prácticas y proporcionan un conjunto de directrices que ayudan a solucionar problemas a nivel de sistema. Forman parte del estilo de arquitectura y a menudo es necesario su uso para una correcta implementación.

Por ejemplo, dentro del estilo de arquitectura de microservicios, es necesario o recomendable implementar el patrón de arquitectura Gateway para resolver el problema enrutamiento de peticiones desde el exterior hacia cada microservicio. [11]

4. Propuesta de la arquitectura

4.1. Introducción

En el entorno del desarrollo de sistemas empresariales, específicamente los enfocados a la recolección y análisis de información como sistemas financieros o comerciales, el manejo eficiente de grandes volúmenes de información es un elemento crucial. Estos sistemas también deben ser capaces de adaptarse a diferentes niveles de carga y ajustarse eficientemente a las nuevas necesidades del flujo empresarial. La correcta implementación de estos mecanismos brinda una importante ventaja a las empresas para tomar decisiones y hacer frente a nuevos desafíos. Dentro de este capítulo se explora a mayor detalle una propuesta de arquitectura de software orientada a resolver los desafíos ya mencionados ofreciendo una solución altamente escalable, flexible y eficiente para sistemas empresariales de alta demanda.

La propuesta se basa principalmente en la arquitectura de microservicios y la arquitectura basada en eventos, tomando las características de crear componentes de granularidad fina separando responsabilidades y el manejo de eventos haciendo uso de tecnologías como Kafka. Todo esto apoyado de una serie de elementos como patrones de diseño y estilos de arquitectura que mejoran significativamente la capacidad de los sistemas para abordar los desafíos ya planteados.

4.2. Principios de diseño de la arquitectura propuesta

Antes de entrar en mayor detalle en la propuesta, es necesario hablar sobre los principios de diseño sobre los que la arquitectura se basa y como estos nos ayudan a lograr los objetivos del proyecto.

Modularidad

La modularidad es un concepto clave dentro de la arquitectura que se propone, este principio brinda a la arquitectura la característica de descomponerse en pequeños componentes independientes, flexibles y fáciles de gestionar. [14]

Dentro de la arquitectura este concepto fue implementado al seguir el estilo arquitectónico de microservicios, cada microservicio representa un módulo que sigue correctamente el concepto de responsabilidad única.

En cuanto a los beneficios que se obtienen al seguir este concepto se encuentran principalmente una mayor flexibilidad al desplegar el sistema, una escalabilidad focalizada, una mayor resiliencia y un mantenimiento más eficiente y simple.

Escalabilidad

Podemos entender la escalabilidad como la capacidad de ampliación y adaptación de un sistema para satisfacer diferentes tipos de carga, a favor de optimizar el rendimiento.

La escalabilidad es una de las características fundamentales dentro de la arquitectura, debido a que esta nos permite mejorar el rendimiento del sistema aumentando los recursos como añadir más servidores o instancias para trabajar de forma distribuida (escalar horizontalmente) así como aumentar los recursos del servidor (escalar verticalmente). [31]

Dentro de la propuesta esta característica se implementa al escalar de manera horizontal añadiendo más instancias del microservicio según sea necesario, así como verticalmente mejorando los recursos y las capacidades de las instancias ya implementadas. Para lograr una escalabilidad más eficiente, también se implementaron patrones de diseño y de arquitectura como CQRS, cuyo objetivo principal es el dividir la responsabilidad del sistema entre la actualización de los datos del sistema y la consulta.

Resiliencia

La resiliencia hace referencia a la capacidad que tiene el sistema para recuperarse de fallos. Este concepto es implementado dentro de la arquitectura al trabajar con microservicios que se comunican por eventos, al trabajar con este enfoque, los componentes del sistema adquieren la posibilidad de fallar sin afectar al sistema en general gracias a la separación de responsabilidades y al desacoplamiento que tienen cada uno de ellos. [15]

Separación de responsabilidades

Como su nombre lo indica, este principio tiene como objetivo el organizar y estructurar tanto los componentes como la estructura interna de estos de manera que cada uno de estos se relacione o se enfoque en una función o requisito específico. Dentro de la arquitectura propuesta esto se logra al implementar patrones y estilos como el dividir el funcionamiento de los componentes en diferentes capas, cada una de estas con responsabilidades y funciones únicas.

Al implementar este principio se reduce de manera considerable el acoplamiento entre componentes, así como potenciar otros principios, en especial los anteriormente mencionados.

Encapsulamiento

El encapsulamiento hace referencia a la capacidad que tiene el sistema de ocultar la lógica interna al usuario. Dentro de la propuesta este principio fue implementado gracias a la implementación de interfaces para la comunicación tanto con otros servicios como con diferentes usuarios.

La ventaja que este principio aporta es la de poder ocultar información sensible minimizando así el riesgo de posibles fallas provocadas dentro del sistema.

4.3. Aplicación de patrones arquitecturales.

4.3.1 Estilos de arquitectura aplicados

Estilo de microservicios

Este estilo tiene como concepto principal la implementación de microservicios, los microservicios son pequeños módulos autónomos que normalmente suelen delimitarse en base a la división natural de la empresa, por ejemplo, basándose en funcionalidades de algún proceso de negocio. [11]

Este estilo proporciona numerosas ventajas, siendo estas las siguientes:

- Ofrece una fuerte flexibilidad al poder trabajar con diferentes lenguajes y tecnologías en cada uno de los microservicios.
- Al ser autónomos pueden desplegarse de forma independiente sin afectar al resto del sistema.
- La comunicación entre microservicios puede ser manejada de varias formas, desde el uso de API's hasta la implementación de brokers u otros medios de comunicación.
- Proporciona una mayor seguridad al delegar una única responsabilidad a cada uno de los microservicios, permitiendo de esta forma manipular los datos de una forma más sencilla y segura.

Además del uso de microservicios, también es necesario implementar una puerta de enlace para que funcione como punto de entrada y controlar el flujo de información, así como un medio de comunicación y gestión para los microservicios.

Dicho esto, dentro de la propuesta se adoptó este estilo debido a las ventajas anteriormente mencionadas, para lograr el objetivo de una arquitectura altamente escalable y que tenga un buen desarrollo y evolución dentro del contexto empresarial, el estilo de microservicios nos proporciona una mejora considerable en la capacidad de escalamiento del sistema, así como la manipulación y mantenimiento, permitiendo que

al hacer una modificación dentro del sistema este pueda seguir operando sin afectar a otros sistemas o usuarios. [11]

En resumen, este estilo arquitectónico nos brinda los conceptos clave para el diseño e implementación de los componentes de nuestra arquitectura, de él se define que los componentes están enfocados en ser altamente independientes y escalables. Además de la implementación de una puerta de enlace como punto de acceso y el uso de un middleware para la comunicación entre estos.

Estilo basado en eventos

Este estilo de arquitectura tal cual como su nombre lo indica se basa en eventos, para lograr esto se define que se requiere tener al menos un componente productor y un componente consumidor. El componente productor es el encargado de generar el flujo de información basándose en alguna regla o evento en particular, por otra parte, el componente consumidor se encarga de recibir la información y realizar ciertas acciones en base a esta.

Como características relevantes sobre los componentes se puede destacar que ninguno de los componentes tiene información específica sobre quien creó o envió cierta información, esto ayuda dentro de la arquitectura para crear un sistema que encapsule de manera eficiente el funcionamiento de este.

Dentro de este estilo se tienen dos modelos diferentes, pub/sub y streaming de datos. Para esta propuesta se utilizó el modelo de streaming, este proporciona la capacidad para que los eventos puedan ser consumidos por múltiples clientes controlando el flujo de eventos. Con este modelo el sistema puede llevar un registro de eventos y, de ser necesario, recuperar información perdida usando estos.

De este estilo se extrajo el concepto de tener componentes publicadores y consumidores, en combinación con el estilo de microservicios se obtuvo que la

arquitectura tendría como componentes principales microservicios con dos posibles roles, consumir o producir datos.

Este estilo fue seleccionado principalmente para mejorar el flujo de datos dentro del sistema, al poder compartir como evento ciertas acciones como podría ser una venta o un registro de un producto, el microservicio encargado de crearlo lo publica para posteriormente se consumido por uno o más microservicios, como por ejemplo ser consumidor por un microservicio encargado de llevar el control de ventas, otro de inventario y otro encargado de hacer análisis de datos con esta información. Además de mejorar el flujo de información, también se mejora la escalabilidad al distribuir las tareas de producir y consumir datos, así como dotar al sistema de poder reaccionar a acciones en tiempo real.

Estilo de contenedores

Este estilo está altamente relacionado con el estilo basado en microservicios, básicamente se podría considerar como un complemento al estilo de microservicios.

El estilo de componentes da solución a los principales desafíos que se tienen al trabajar con microservicios, siendo estos principalmente el manejo de instancias de microservicios.[9]

Al implementar este estilo junto con sus principios, se obtiene la capacidad de implementar un orquestador, el orquestador es el encargado de desplegar las instancias, por ejemplo, si este está configurado para mantener 3 instancias de cierto microservicio, este hará todo lo necesario para mantener en todo momento 3 instancias ejecutándose.

Dentro de la arquitectura este estilo se implementa al hacer uso de contenedores y un orquestador, esto haciendo uso de tecnologías como Docker y kubernetes. Con este estilo se solucionan los problemas de escalabilidad y facilita la gestión de instancias de microservicios.

4.3.2 Patrones de arquitectura aplicados

Patrón de puerta de enlace (Gateway)

Este patrón propone la implementación de un 'Gateway' para mejorar la interacción entre el cliente y los microservicios haciendo de intermediario entre estos dos. Su implementación ayuda a mantener el principio de encapsulación manteniendo ocultos los detalles internos de la arquitectura de microservicios compartiendo en un solo punto solo los puntos de acceso que se deseen. [22]

Dentro de la arquitectura este patrón se implementa al utilizar un gateway como punto de acceso hacia los microservicios. Este desempeña las tareas de recibir y redirigir las peticiones REST hacia los respectivos microservicios.

Patrón de disyuntor (Circuit Breaker)

Circuit Breaker fue agregado con el objetivo de mejorar la resiliencia y estabilidad del sistema. Esto se consigue buscando siempre evitar ciclos repetitivos al hacer peticiones que dan error. Con este patrón, el sistema adquiere la capacidad de detectar cuando hay problemas con algún modulo, cuando detecta un problema suspende las peticiones hacia ese modulo en específico hasta que el sistema detecte que se encuentra disponible de nuevo.[21]

CQRS

El patrón CQRS fue implementado para optimizar el rendimiento, la escalabilidad y la seguridad de la arquitectura. Este patrón propone dividir las responsabilidades de comandos y consultas.

Este patrón surgió de la necesidad de mejorar el rendimiento del modelo de datos de los sistemas, típicamente en las arquitecturas tradicionales estas dos operaciones solían manejarse en un solo lugar, esto era eficiente, rápido y sencillo cuando las arquitecturas y los sistemas no eran tan complejos. Hoy en día los sistemas suelen tener diferentes requisitos para realizar estas dos operaciones, por un lado, se

tiene que, al realizar un comando de actualización en los datos, estos suelen necesitar un proceso de validación y lógica de negocio adicional, mientras que, por otro lado, las consultas suelen variar tanto en los datos recuperados como en la cantidad de solicitudes. Además, el número de consultas y comandos suele variar, como resultado los sistemas con enfoques tradicionales donde ambas acciones están unidas tienen problemas para adaptarse a diferentes niveles de carga. Pues de necesitar más instancias o recursos para las acciones de consultas, obligatoriamente también se tendría que escalar las de comandos. [20]

En la propuesta este patrón se implementa al separar las responsabilidades de lectura y escritura, dando como resultado dos microservicios en donde originalmente solo se tendría uno, además este patrón se combina con el enfoque de eventos para mantener la consistencia de datos entre ambos microservicios a través de eventos.

4.3.3 Patrones de diseño

Strategy

La implementación de este patrón trae consigo ventajas como facilitar el mantenimiento o la implementación de nuevas estrategias y en nuestro caso específico, eventos. [34]

4.4. Tecnologías utilizadas

Java Spring Boot Framework

Popular framework que destaca en el desarrollo de microservicios principalmente por la función de inyección de dependencias, facilitando de esta manera el desarrollo de aplicaciones modulares. Además, este framework ofrece soporte a funciones como el manejo de datos con repositorios, nube, seguridad, gestión de recursos, validaciones, excepciones y muchas más. Como se puede observar este framework ofrece un campo de trabajo completo para la elaboración de aplicaciones modulares, además al integrarse con spring boot se facilita aún más el proceso al ofrecer este configuraciones más sencillas o inclusive automáticas. [27]

MySQL

MySQL es un sistema de gestión de base datos relaciones, realmente dentro de esta propuesta no se profundizo mucho sobre la selección de gestores de bases de datos específicos, se recomienda seleccionar la base de datos que mejor se adecue al proyecto en cuestión. Al trabajar con el marco de microservicios, cada uno de estos puede tener bases de datos diferentes, e inclusive ser no relacionales, lo único que se debe realizar es adaptar los campos e implementar las validaciones necesarias para garantizar la consistencia de datos.

Docker

Es una tecnología de organización que permite la creación y manipulación de contendores de Linux. El uso de contenedores es importante dentro de la arquitectura, al hacer uso de contendores se simplifica el despliegue y la gestión de dependencias.

en la implementación de sistemas que necesiten ser muy escalables y flexibles, siendo este el caso de nuestra propuesta. [26]

Kafka

Apache Kafka es la plataforma elegida para la transmisión de datos entre microservicios, esta plataforma además de poder publicar y procesar los eventos, también brinda la posibilidad de administrar estos flujos y enviarlos a diferentes usuarios o en nuestro caso componentes a la vez. Además de tener una excelente sinergia con contenedores y por lo tanto con kubernetes. [25]

4.5. Estructura y elementos de la arquitectura propuesta

4.5.1 Diagrama de la arquitectura

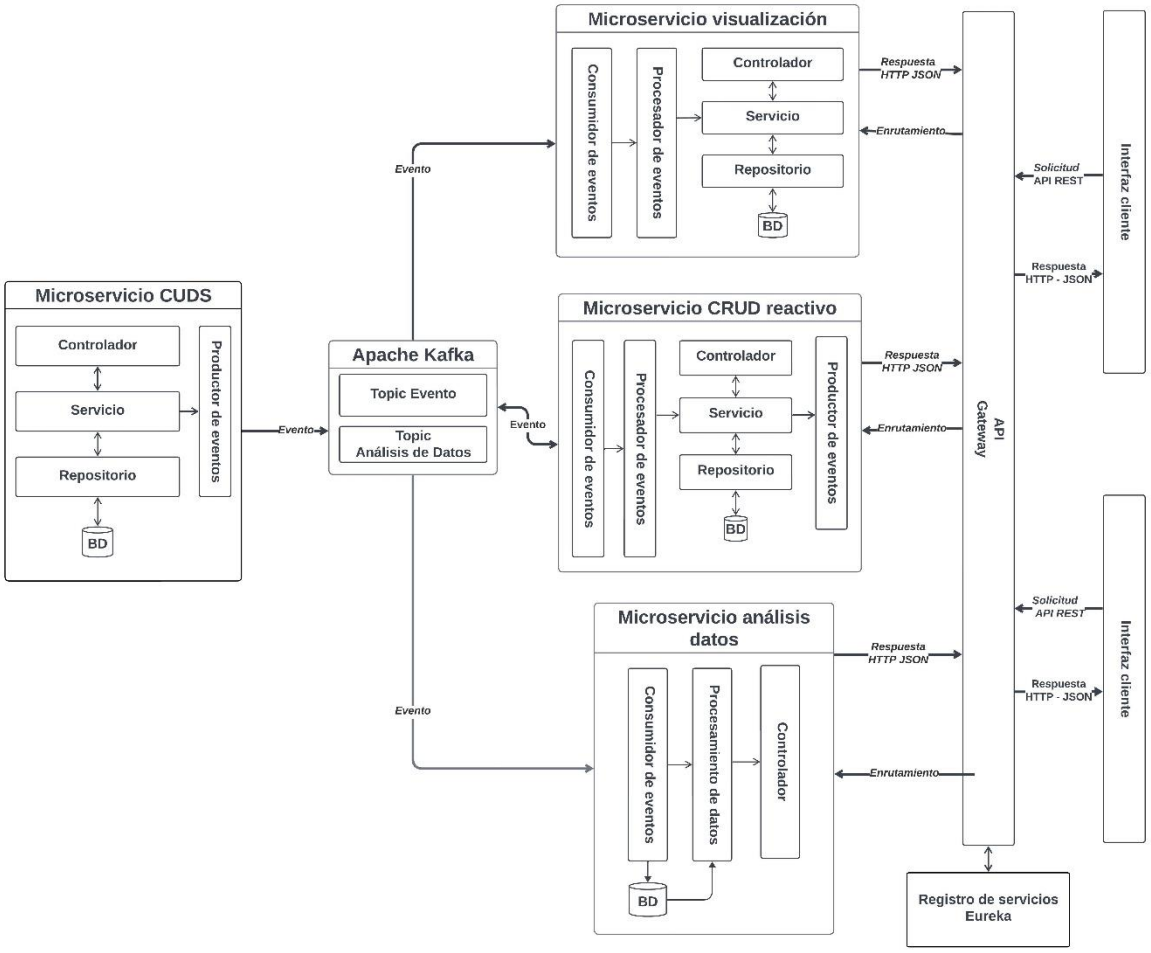


Figura 4-1 Diagrama general de la arquitectura

4.5.2 Elementos de la arquitectura propuesta

4.5.2.1 Componentes

La arquitectura que se propone se constituye de 4 componentes esenciales que son:

- Microservicio CUDS
- Microservicio de visualización
- Microservicio CRUD reactivo
- Microservicio de análisis de datos

Estos componentes fueron pensados para ser utilizados como una base o modelo para el desarrollo de múltiples instancias que puedan ser adaptadas a necesidades y contextos de negocio específicos, más adelante en el apartado de implementación se dará un ejemplo de esto.

Cada uno de los componentes fueron creados a partir de la combinación de los estilos de arquitectura, patrones de arquitectura y patrones de diseño que fueron discutidos en el apartado anterior. Aunque cada uno tiene un enfoque y responsabilidades diferentes, estos comparten algunos elementos clave que sería bueno discutir antes de entrar en más detalle.

En primer lugar, todos los componentes de microservicios comparten una estructura similar, todos utilizan 3 entidades o módulos para el manejo de las peticiones siendo estos los siguientes:

Controlador

Esta entidad es la encargada de recibir las peticiones, en general, las peticiones provienen del gateway que recibe estas a través las interfaces o de otros microservicios. Estas peticiones siguen el estilo REST, por lo que, para ser recibidas correctamente, estas

deben seguir una estructura bien definida para el diseño de la URI, así como el correcto uso de los métodos HTTP (PUT POST DELETE CREATE) para la manipulación de los datos.

Dentro de esta entidad o módulo se realizan ciertas verificaciones antes de ser procesadas hacia las entidades de servicios. Una vez que el servicio completa el procesamiento el controlador devuelve una respuesta que contiene el código de respuesta HTTP.

Servicio

Dentro de esta entidad se encuentra la mayor parte de la lógica de negocio, este módulo es el encargado de procesar la información y llamar al o las entidades de repositorio para almacenar la información, también se encarga de llamar a otros elementos como los productores de evento.

Repositorio

Este módulo es el encargado de persistir y gestionar el almacenamiento de datos, para esto dentro de la propuesta se recomienda el uso de herramientas para agilizar el procesamiento de datos, para bases de datos relacionales se recomienda el uso de JPA y para bases de datos no relacionales o sistemas de almacenamiento externos el uso de sus correspondientes herramientas especializadas que normalmente son ofrecidas para los diferentes frameworks.

Dicho esto, otra característica que comparten los componentes es el uso de clases DTO para el intercambio de información entre los diferentes módulos y entidades. DTO nos brinda flexibilidad a la hora de definir qué datos serán enviados entre los diferentes módulos y componentes, logrando de esta manera ahorrar recursos al no enviar información no relevante para la acción, así como definir respuestas para las diferentes peticiones de los usuarios más especializadas, aumentando así la eficiencia de la arquitectura.

Para el manejo de errores todos estos componentes cuentan con un manejador de excepciones global, este se encarga de atrapar todas las excepciones que ocurren y realizar una acción en respuesta de esto, en su mayoría se busca deshacer el movimiento para posteriormente regresar una respuesta que contenga el código de error, el nombre de la excepción e información sobre qué fue lo que falló y posibles soluciones. Este comportamiento puede varias al definir las excepciones que se desean manipular y la forma en que son gestionadas.

En base a todo lo anterior, a continuación, se explica en mayor detalle cada uno de los componentes que se proponen, haciendo mayor énfasis en sus responsabilidades y características únicas.

Microservicio C.U.D.S.

El acrónimo C.U.D.S., fue creado específicamente para esta propuesta de arquitectura, este proviene del famoso acrónico CRUD (Create, Read, Update, Delete), con la diferencia de que, en este caso, se elimina la sigla R (Read) y se agrega una 'S' (Share).

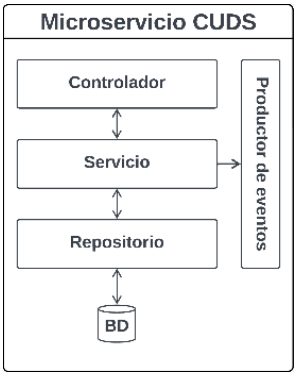


Figura 4-2 Estructura del componente microservicio CUDS

Como las siglas lo indican, este componente es el encargado de crear, actualizar, eliminar y compartir las entidades del sistema. Este comportamiento surge gracias a la implementación del patrón CQRS, que como anteriormente se mencionó, propone la separación de responsabilidades en dos, uno encargado de la actualización de datos y otro única y específicamente para la visualización.

La estructura de este componente se compone de 4 módulos, cada uno de estos con una función específica.

En primer lugar, el controlador recibe las peticiones REST, se hace una verificación rápida sobre los datos recibidos y posteriormente se pasa el mando al o los

servicios. Dentro del módulo de servicio los datos son procesados y en caso de ser necesario algún movimiento de persistencia de datos, se invoca al módulo de repositorio para realizar la operación, posterior a esto se verifica si la operación fue realizada con éxito y se crea una entidad en donde se almacena el movimiento y los datos para posteriormente ser enviada al módulo de eventos.

Finalmente, el módulo de eventos crea y pública un evento con la información proporcionada y espera una respuesta por parte del broker, si se confirma que el evento fue publicado exitosamente se envía una respuesta notificando que la operación fue realizada con éxito. En caso de que ocurra algún error a lo largo del proceso, este será atrapado por la clase de excepciones y enviará una respuesta personalizada notificando el error.

Microservicio de visualización

Este componente tiene como función principal la manipulación de las operaciones de lectura, está fuertemente relacionado con el resto de los componentes dado a que este microservicio tiene la capacidad de consumir los eventos publicados por el resto de los microservicios.

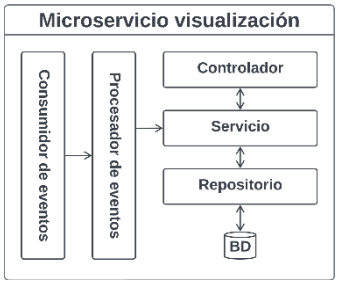


Figura 4-3 Estructura del componente microservicio visualización

De manera similar este componente cuenta con 4 módulos o capas, pero un funcionamiento un tanto diferente. Para comenzar, este componente hace uso del patrón observer y la manipulación de los eventos con el broker para mantener su información actualizada en base al resto de componentes. Cuando se recibe un evento, este microservicio hace uso de una clase encargada de consumir y recuperar la información que fue enviada como la fecha, el orden, tipo de acción y los datos. Una vez que la información fue recibida exitosamente por la entidad consumidor, la información es enviada hacia el módulo de procesamiento, dentro de este módulo se sugiere implementar el patrón de diseño Command para poder manipular los diferentes

tipos de solicitudes, ya que puede variar desde la entidad sobre la que se tiene que actuar, así como el tipo de acción. Tras procesar exitosamente el evento se almacena el resultado del movimiento, permitiendo de esta manera mantener el componente actualizado en tiempo real.

En caso de que algún error ocurriera a lo largo del proceso de consumo o procesamiento de eventos, se tienen diferentes escenarios, en primer lugar, el componente intentará resolver el problema reintentando y encolando un número limitado de veces el evento. En caso de que el error persista el componente almacenara en un listado especial el evento que no pudo ser procesado y notificará para poder ser intervenido y auxiliado por un desarrollador.

Microservicio CRUD reactivo

Este componente podría considerarse como una unión de los dos componentes anteriormente mencionados, es decir sus responsabilidades principales consisten en manipular los datos y a su vez tiene la capacidad de consumir y publicar eventos. En primera instancia puede parecer que es mejor utilizar varios componentes de este tipo, pero al igual que utilizar este tipo de componente tiene sus ventajas como tener una mejor consistencia en los datos, también tiene sus desventajas al no ser implementado de forma correcta. Por ejemplo, en escenarios donde no se espera una gran disparidad entre el número de solicitudes entre las peticiones de lectura y actualización de datos, este componente sería la mejor elección, pero en caso de que el número de peticiones tenga una enorme disparidad, este podría no ser la mejor opción por el simple hecho de que sería complicado escalar y por lo tanto adaptarse a las diferentes cargas de trabajo.

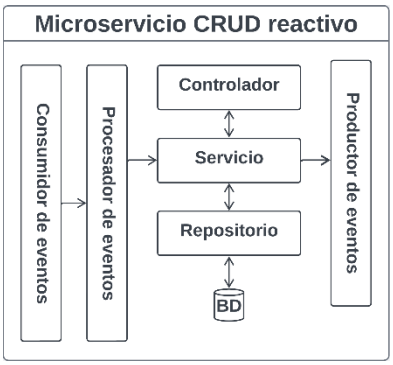


Figura 4-4 Estructura microservicio CRUD reactivo

Al implementar el patrón CQRS y obtener dos componentes donde originalmente teníamos uno, nos permite controlar aún más la escalabilidad de nuestro sistema. Por lo tanto, es importante tener en cuenta estos escenarios al elegir que componente se utilizará.

Dicho esto, este componente no funciona de manera muy distinta a los que ya hemos explicado, como se mencionó en un inicio también se compone de los 3 módulos base y los utiliza de igual manera, solo que en este caso al pasar los datos y procesarlos en el módulo de servicios, este microservicio puede generar eventos y publicarlos. Otra de sus características es que tiene la posibilidad de recibir información no solo de las peticiones, sino que también de otros componentes, lo que da como resultado un componente robusto, escalable y eficiente.

Microservicio análisis de datos

Este componente tiene como tarea principal el recopilar, procesar, almacenar y generar información a través del análisis de datos.

Para lograr esto de igual forma necesita consumir los datos de los eventos que sean de interés, para posteriormente procesarlos y almacenarlos los repositorios adecuados. Para este microservicio se propone utilizar Python, esto debido a la gran popularidad y herramientas que este tiene para trabajar con datos. Además, al implementar un microservicio que no utiliza las herramientas y lenguaje que hemos propuesto hasta el momento, demostramos la gran flexibilidad que se tiene al trabajar con microservicios.

En cuanto a las técnicas y análisis de datos, se tiene una gran variedad de herramientas para elaborar estos análisis y devolver estadísticas sobre la información recuperada, así como gráficos que ayuden al entendimiento de esta.

4.5.2.2 Conectores

Para la comunicación entre los componentes, se propone utilizar en combinación la comunicación asíncrona y síncrona.

Al utilizar de base un estilo basado en eventos, nos vemos en la necesidad de integrar mecanismos asíncronos para el manejo de solicitudes, esto con el propósito de que nuestros componentes puedan seguir funcionando a la par de que procesan un evento. Pero, por otra parte, algunos de los movimientos pueden poner en riesgo la consistencia de datos en todo nuestro sistema, por lo cual para esta propuesta se sugiere el uso de mecanismos síncronos para poder esperar una confirmación antes de hacer permanentes los cambios en la persistencia de datos. Esto claro tiene un fuerte impacto pues el usuario tendría que esperar a que la solicitud se complete antes de poder continuar, pero a cambio se puede garantizar una mayor consistencia dentro del sistema.

En base a lo anterior, se puede decir que uno de los mayores desafíos que tiene esta propuesta, es el de poder garantizar la integridad y consistencia de la información entre los diferentes componentes.

Para lograr esto, se implementaron una serie de medidas, patrones y herramientas para ayudar a reducir las posibilidades de una mala actualización.

Entre los más destacados se tiene la implementación de colas de reintento, circuit breaker y un control estricto sobre el offset y las particiones del broker Kafka. Mas adelante en la implementación se dará más detalles sobre cómo lograr esto.

En cuanto al medio que se utiliza para la comunicación se tienen dos posibilidades, una es el intercambio de peticiones a través de peticiones HTTP REST y otra haciendo uso del broker Kafka para el manejo de eventos.

4.5.2.3 Interfaces

En cuanto a la interfaz, se propone integrar un API Gateway en conjunto con APIs REST en cada microservicio. Este enfoque facilita la interacción entre los usuarios y el sistema, así como entre los componentes internos al ofrecer un punto de entrada unificado, estandarizado y centralizado

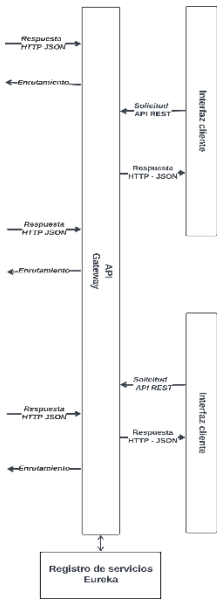


Figura 4-5 Interacción entre gateway e interfaces

Para conseguir esto el gateway hace uso de Eureka Server para el descubrimiento y registro de microservicios. De esta forma se consigue mejorar la escalabilidad y mantenibilidad del sistema al facilitar la gestión de las URI, así como de los recursos.

4.5.2.4 Almacenamiento de datos

Respecto a la selección de que tipos de almacenamiento utilizar dentro de la arquitectura, nuestro enfoque modular basado en microservicios nos sugiere que cada microservicio debe poseer su propia base de datos, la elección sobre qué tipo utilizar entre las relacionales y las no relacionales depende en gran medida sobre el tipo de operaciones que realizará. En esta propuesta se sugiere que para los componentes basados en el microservicio CUDS, se implementen con bases de datos relacionales, esto debido a su enfoque centrado en actualizar y mantener los datos, al hacer uso de una base de datos relacional se promueve una mayor integridad de los datos, así como un mejor rendimiento al hacer actualizaciones que involucren múltiples entidades. Por otra parte, si el componente está basado en el microservicio de visualización, una base de datos no relacional podría ser de gran ayuda.

En cuanto al almacenamiento de archivos de gran tamaño, se sugiere la implementación de sistemas de archivos externos, estos pueden ser sistemas locales, así como en nube o servidores remotos. Para mantener y recuperar estos archivos también se sugiere almacenar la referencia sobre donde fueron almacenados dentro de la base de datos. De este modo al tener estos grandes archivos en un sistema externo

e independiente, estos pueden ser recuperados por múltiples componentes haciendo uso de la referencia.

Finalmente, en cuanto a que tecnologías utilizar para cada uno de estos casos, se sugiere por regla general utilizar las herramientas que cada sistema de almacenamiento provee a los frameworks o lenguajes, en nuestro caso al estar haciendo uso del framework spring boot, para el manejo de bases de datos relacionales se hace uso de JPA en conjunto de otras herramientas como Hibernate, Lombok, SQL, etc. Para datos no relacionales se utiliza la herramienta proporcionada directamente por la base de datos, como por ejemplo Spring Data MongoDB y en cuanto a sistemas de almacenamiento externos, estos también proveen librerías especializadas para la manipulación de recursos como Azure con Azure Blob Storage.

4.5.2.5 Middleware

Kakfa Broker

Como ya se definió anteriormente, apache kafka es una plataforma de streaming de eventos que facilita la comunicación entre componentes, en el contexto de nuestra propuesta, kafka esta encargado de distribuir y gestionar los eventos hacia y entre los microservicios.

Spring Gateway

En cuanto a spring gateway, este componente tiene como responsabilidad la gestión del enrutamiento hacia las instancias de los microservicios, dentro de este se aplican patrones y principios como el balanceador de carga, que promueven y mejoran la eficacia del sistema al permitir distribuir de forma correcta la carga entre los diferentes microservicios y sus instancias. Además, proporciona un punto único de entrada para los usuarios, reduciendo de esta forma la complejidad al consumir diversos microservicios y encapsulando el funcionamiento.

Eureka Server

Eureka Server a pesar de no interactuar directamente con los microservicios, este es de suma importancia para el funcionamiento general del sistema, ya que, es el encargado de llevar un seguimiento y registro sobre las instancias disponibles, así como de los microservicios, información que es utilizada principalmente por el gateway, para conocer todas las instancias y redirigir las peticiones de forma exitosa. Además, su implementación en conjunto de gateway, permite implementar microservicios u otros componentes que no sean o utilicen el mismo ecosistema de desarrollo, lo cual es una gran ventaja, pues en lugar de preocuparse por la compatibilidad entre sistema, puedes enfocarte en usar la mejor herramienta, framework o lenguaje y simplemente conectarlo a través de estos componentes middleware.

5. Implementación de la arquitectura propuesta.

5.1. Introducción

Dentro de este capítulo se ofrece un ejemplo de implementación sobre la arquitectura propuesta, esto con el objetivo de demostrar y reforzar los conceptos que se han ido explicando a lo largo de este documento sobre la arquitectura propuesta.

En base en un contexto de ejemplo, a lo largo del capítulo se guía el proceso de la creación del sistema en base a nuestra arquitectura, haciendo especial énfasis en aquellas partes donde se implementa una tecnología, concepto, patrón o estilo para resolver un problema.

5.2. Descripción del proyecto de ejemplo

El ejemplo de implementación se centra en un sistema simplificado que controla y gestiona la creación de productos y de ventas. Esta elección se basa en que estos procesos enfrentan de manera cotidiana diversos problemas, como el manejo de grandes volúmenes de información y la alta y variable demanda que estos deben de manejar. Problemas que nuestra propuesta de arquitectura busca dar solución y, además, la necesidad de garantizar una alta consistencia entre estos componentes, la disponibilidad, así como la independencia de estos, se alinean con los principios de diseño que hemos detallado a lo largo de este trabajo.

De manera deliberada se ha dejado de lado otros procedimientos que pueden ser de interés, como el manejo de inventario. La razón de esto radica en la necesidad de demostrar de manera centrada como la arquitectura apoya en los momentos críticos del sistema, por tal motivo se optó por elegir aquellos procesos que se apegan más a nuestra arquitectura.

Dicho esto, los objetivos que se buscan alcanzar dentro de este ejemplo de implementación son los siguientes:

- **Gestión eficiente de productos**: El sistema debe ser capaz de realizar las operaciones de creación, eliminación y actualización garantizando la consistencia de datos en todo el sistema.
- **Registro preciso de ventas:** Dentro de las funcionalidades del sistema, este debe permitir el registro detallado de las ventas realizadas, conteniendo datos como el producto, cantidad, precio, cantidad, entre otros.
- **Escalabilidad y rendimiento:** Siendo uno de nuestros objetivos principales, el sistema debe tener la capacidad de manejar eficientemente distintos niveles de carga, buscando en todo momento el mejor rendimiento.
- **Facilitar el análisis de datos:** El flujo de datos dentro de la aplicación debe de facilitar el consumo y procesamiento, permitiendo de esta forma obtener información importante sobre los procesos de venta.

5.3. Desarrollo del sistema

El sistema que se explica a continuación se compone de 4 microservicios, cada uno de estos refleja una implementación directa de los componentes que explicamos en el capítulo anterior. Estos microservicios se comunican mediante apache Kafka, en esta implementación, nuestro broker Kafka contará con 3 topics diferentes, así como 3 grupos.

Como se puede observar, se tiene también el registro de servicios eureka y api gateway, siendo estos los encargados de conectar los microservicios y gestionar el flujo de solicitudes.

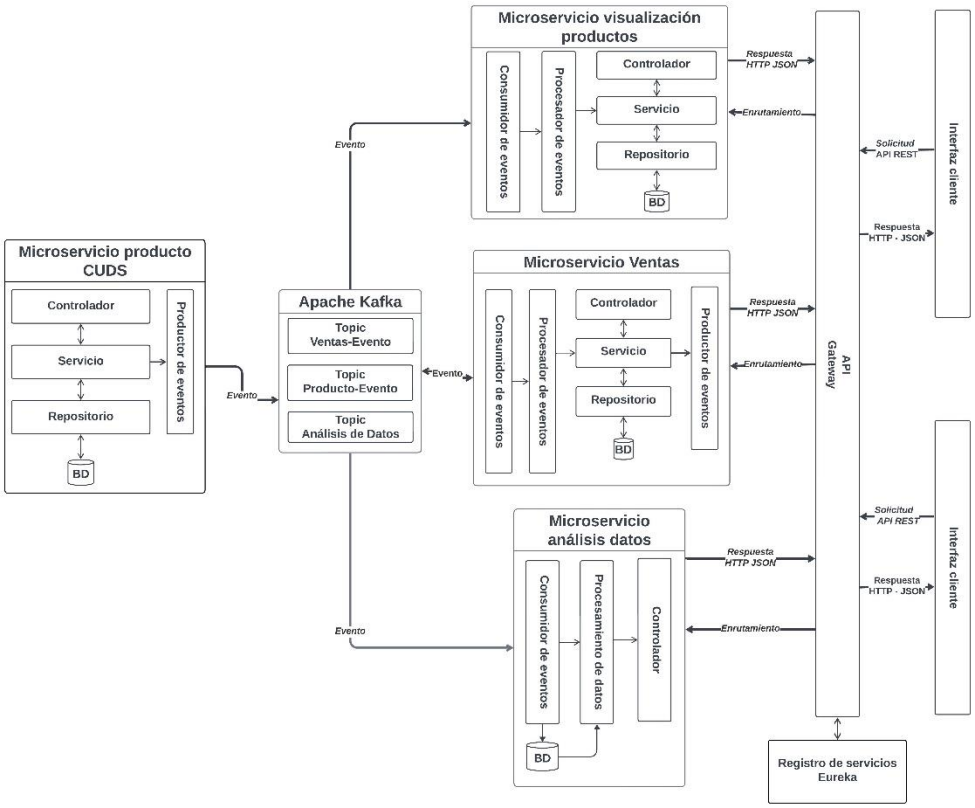


Figura 5-1 Diagrama de implementación de la propuesta de arquitectura sobre el contexto de gestión de ventas y productos

5.3.1 Eureka Server

Iniciemos el desarrollo implementando el registro de servicios Eureka, este componente (middleware) es de suma importancia para el sistema, pues como su nombre lo indica este brinda al sistema la capacidad de detectar microservicios, habilitando así su comunicación con otros componentes, como gateway.

Ya que se está trabajando con spring boot, el primer paso es crear nuestra aplicación e integrar las dependencias necesarias para su uso.

Para fines prácticos este proceso solo se mostrará en esta ocasión, pues lo único que puede variar entre estos son las dependencias por utilizar, pero estas serán explicadas para cada uno de los componentes.

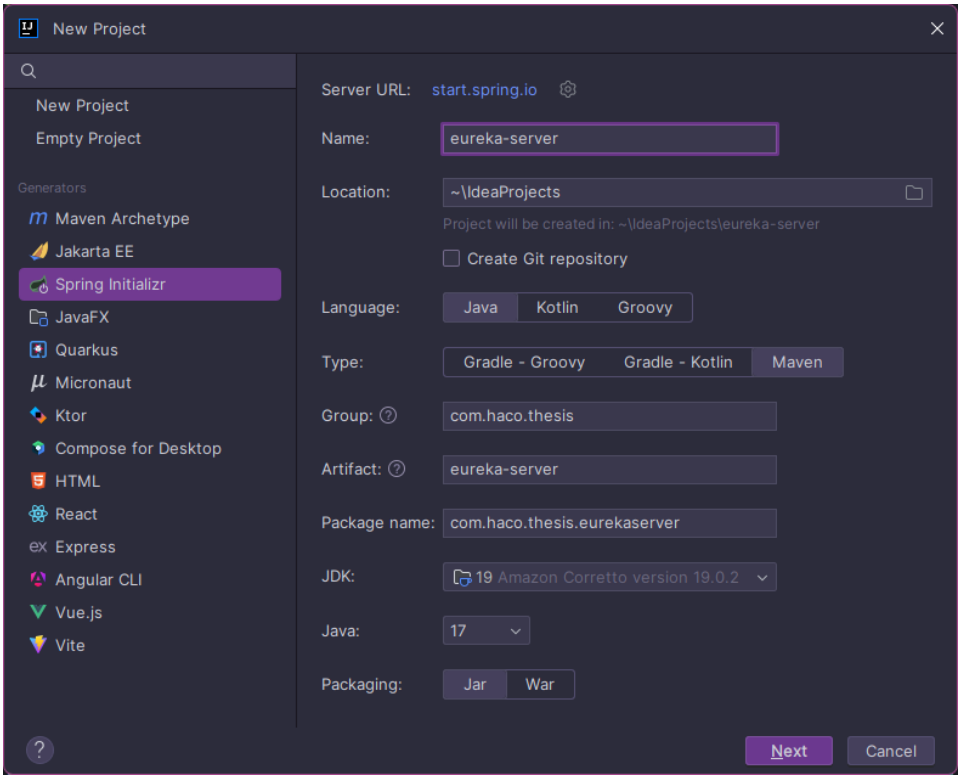


Figura 5-2 Creación de modulo spring Eureka Server

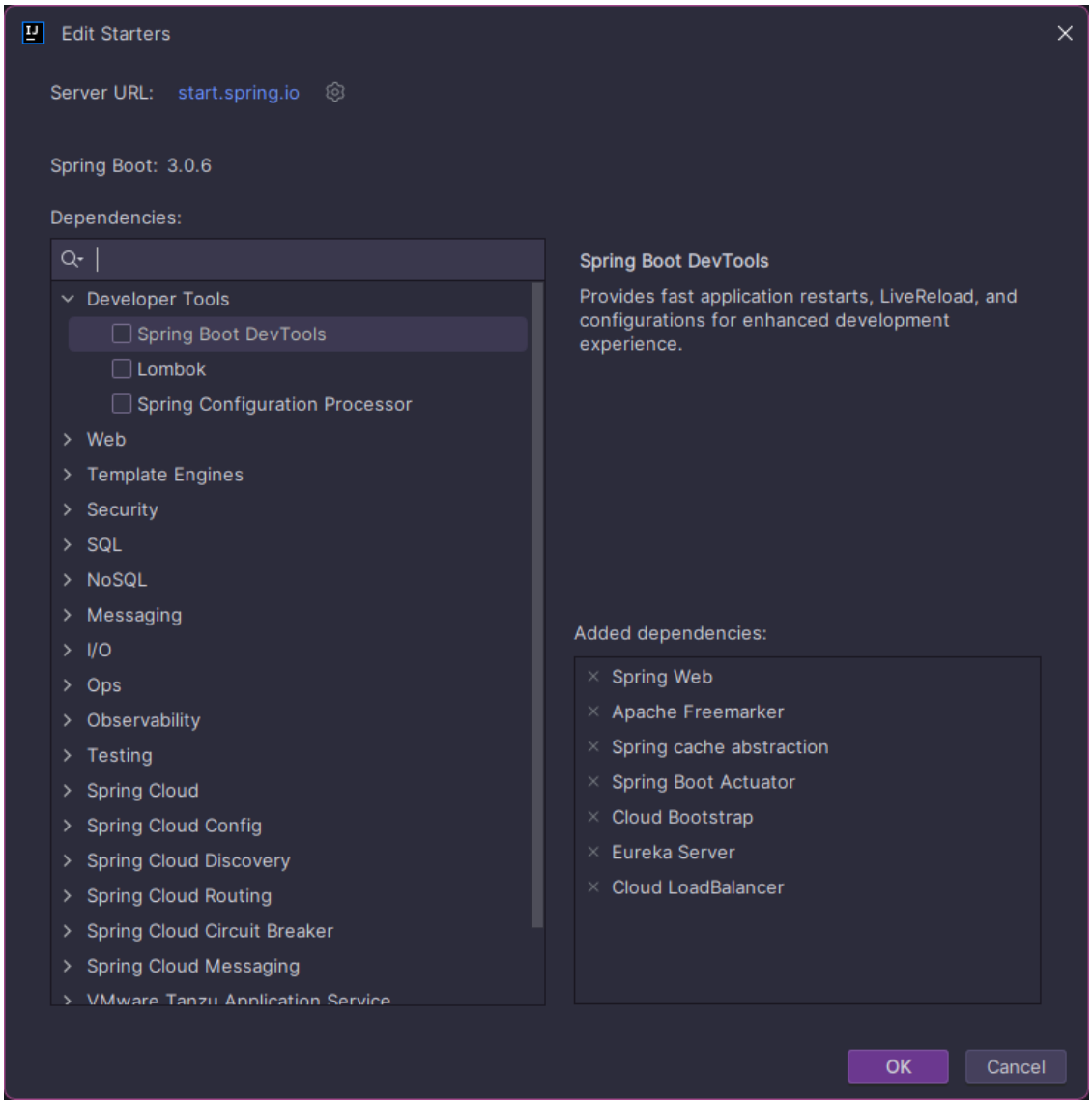


Figura 5-3 Dependencias necesarias para el funcionamiento de Eureka Server

Una vez creado el proyecto con las dependencia necesarias, es necesario configurar nuestro servidor eureka de la siguiente manera:

Figura 5-4 Configuración básica del servidor eureka

Todos los módulos spring cuentan con un archivo llamado application.properties o application, este archivo nos permite asignar valores para configurar aspectos de nuestro sistema, en el caso del servidor eureka, se le asigna un nombre y un puerto, la penúltima línea deshabilita la opción de que el cliente eureka se registe a sí mismo y la última deshabilita que el resto de los componentes puedan descubrir quienes están registrados.

Finalmente habilitamos el servidor eureka editando el archivo principal y agregando la etiqueta correspondiente de la siguiente forma:

EurekaServerApplication.java

```
1  package com.haco.eureka.server;
2
3  import ...
6
   Hugo Alexis Chiquito Onofre
7  @EnableEurekaServer
8  @SpringBootApplication
9  public class EurekaServerApplication {
10
       Hugo Alexis Chiquito Onofre
11     public static void main(String[] args) { SpringApplication.run(EurekaServerApplication.class, args); }
14
15 }
```

Figura 5-5 Habilitando servidor eureka

Completados estos pasos, ejecutamos el servidor y si navegamos al puerto que le fue asignado en un navegador obtendremos la siguiente ventana.

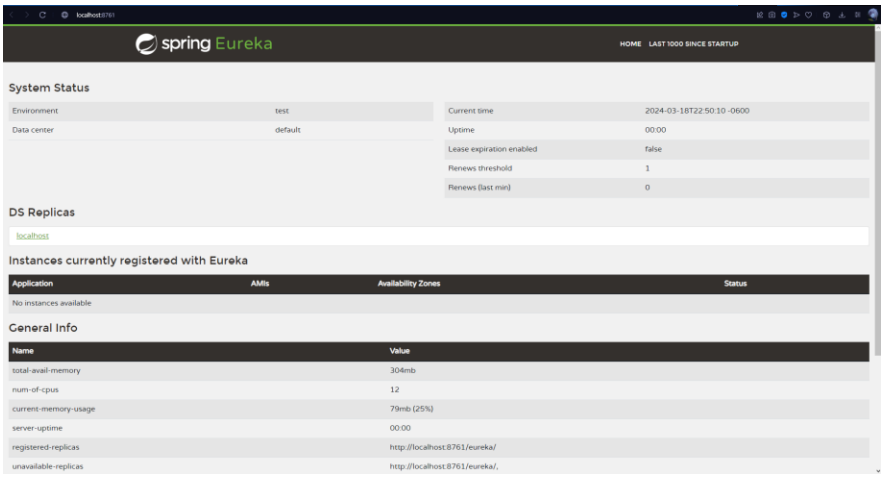


Figura 5-6 Panel del servidor Eureka

Este panel es de suma importancia, en él se pueden obtener detalles como las instancias de los microservicios que se están ejecutando, sus puertos y más información relevante.

Con el servidor eureka desarrollado, ahora podemos continuar con el desarrollo del resto de componentes.

5.3.2 API Gateway

De manera similar a eureka, se creó un nuevo módulo de spring con las siguientes dependencias:

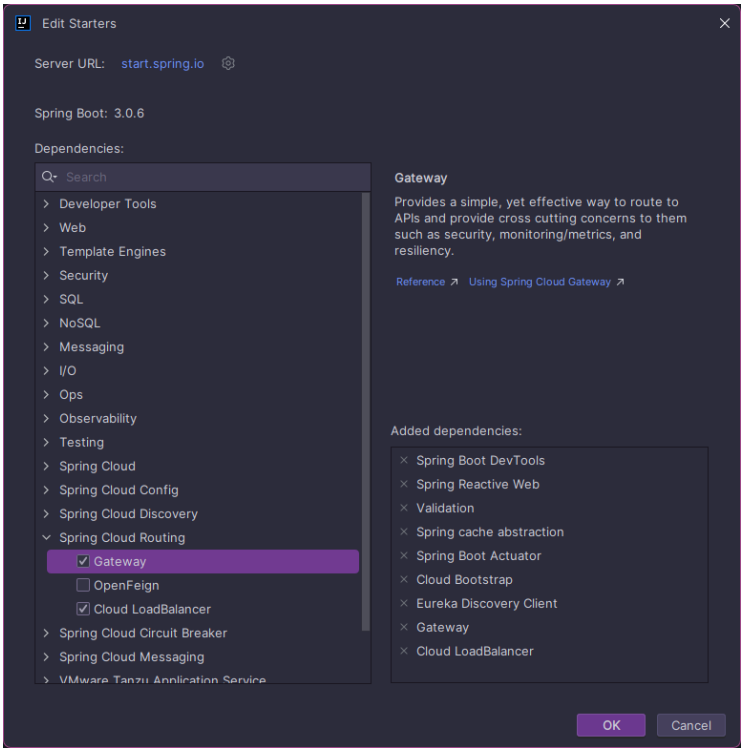


Figura 5-7 Dependencias ocupadas en el componente gateway

Dentro de estas dependencias se puede destacar las siguientes:

Eureka Discovery Client: Esta dependencia brinda al componente la capacidad de ser detectado por el servidor eureka.

Para lograr esto, se debe de activar añadiendo la etiqueta **@EnableDiscoveryClient** en el archivo principal del componente, este proceso se repite en el resto de los componentes, por lo que es importante entenderlo.

Figura 5-8 Habilitando cliente servidor eureka

Gateway: brinda al componente la capacidad de gestionar las peticiones, redireccionándolas a los microservicios o componentes necesarios, así como monitoreando su estado.

Para lograr esto, es necesario modificar el archivo de configuraciones de la siguiente manera:

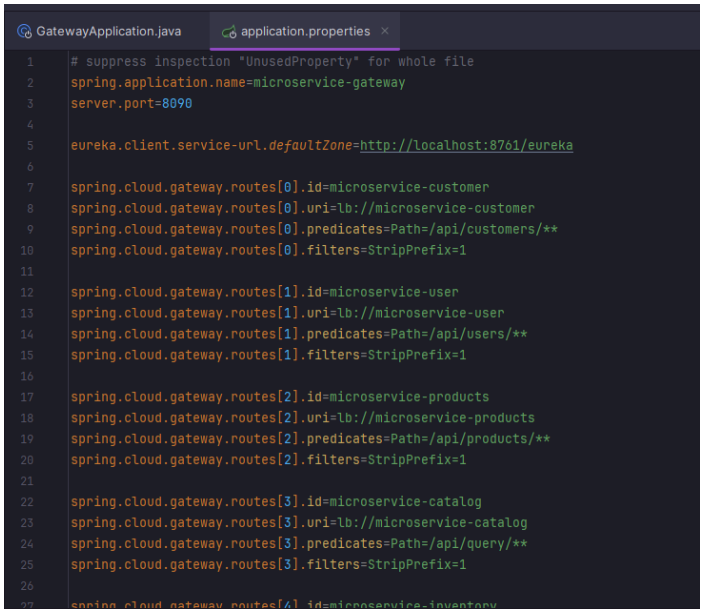


Figura 5-9 Configuración de gateway

Esta configuración, consiste, de igual forma en asignar un nombre y un puerto al componente, el puerto que se defina en la configuración es el que será utilizado a lo largo de la aplicación para acceder a los recursos de este, por lo cual es importante recordarlo.

El resto de configuración corresponde a la asignación de rutas a través de la url, para hacerlo es necesario indicar cuatro campos, el primero corresponde al nombre del microservicio al que se quiere redirigir la petición, el resto hace referencia al uso de LoadBalancer para balancear la carga entre las instancias de un mismo componente y que url esta asignada a este componente.

5.3.3 Apache Kafka

La configuración de Kafka puede parecer algo muy impresionante al inicio, pues ofrece una gran cantidad de configuraciones para poder adaptarse a distintos escenarios, con lo cual esta configuración siempre debe pensarse y adaptarse en base a los requisitos de la aplicación.

Para este ejemplo práctico se optó por un clúster de Kafka compuesto por tres nodos (kafka1, kafka2, kafka3) y un zookeper.

```
zoo1:
  image: confluentinc/cp-zookeeper:7.3.2
  hostname: zoo1
  container_name: zoo1
  ports:
    - "2181:2181"
  environment:
    ZOOKEEPER_CLIENT_PORT: 2181
    ZOOKEEPER_SERVER_ID: 1
    ZOOKEEPER_SERVERS: zoo1:2888:3888
```

Figura 5-10 Nodo zookeper

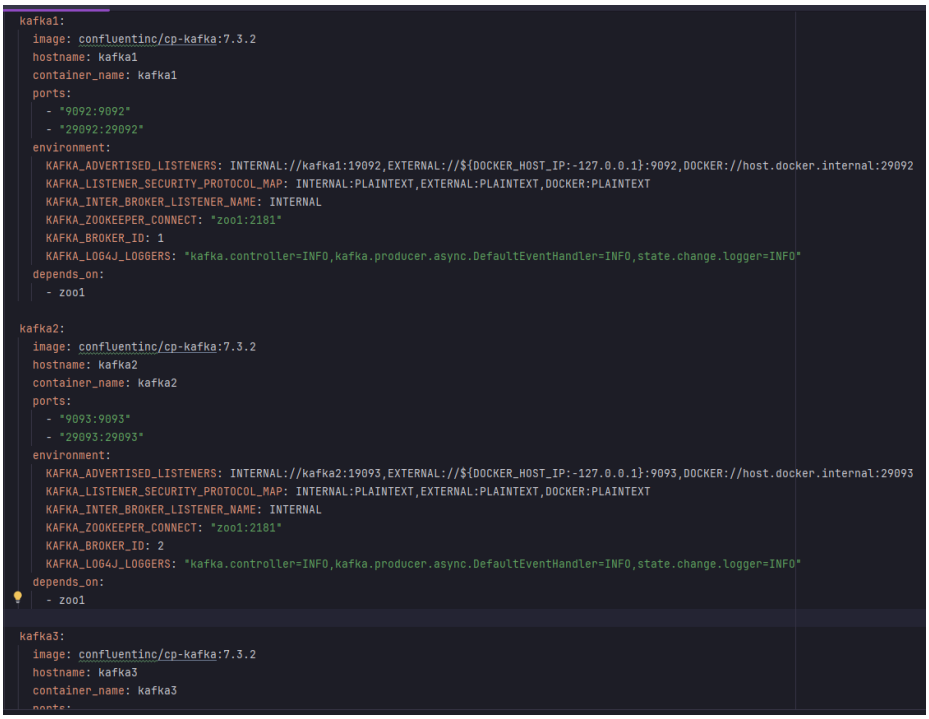


Figura 5-11 Nodos kafka

El nodo de zookeper es el encargado de coordinar los nodos Kafka, dentro de su configuración se específica la imagen a utilizar, puertos y otras configuraciones del ambiente, como los puertos para la conexión con el cliente y los servidores.

Por otra parte, los nodos Kafka tienen una configuración similar entre ellos, también tienen una imagen para su ejecución, y cada uno de estos tiene sus puertos definidos.

Esta configuración podría considerarse como un estándar y proporciona un buen ambiente de pruebas para el ejemplo que estamos desarrollando.

Tras explicar el funcionamiento y la configuración de los componentes intermediarios de la arquitectura y de nuestro ejemplo, a continuación, se detallará el proceso de creación de los componentes estrella del sistema.

5.3.4 Microservicio productos CUDS

Este microservicio es el encargado de manipular y gestionar todos los cambios de la entidad producto y publicar estos eventos en apache Kafka para su consumo. Para lograr esto, es necesario tener una serie de elementos para su correcto funcionamiento.

De igual forma este primer microservicio será explicado detalladamente para brindar un panorama general sobre la configuración de un microservicio, pero más adelante para el resto de los componentes se omitirán aquellos detalles, como la creación y manipulación de repositorios, servicios y controladores. Centrándonos únicamente en los elementos característicos o clave de cada componente.

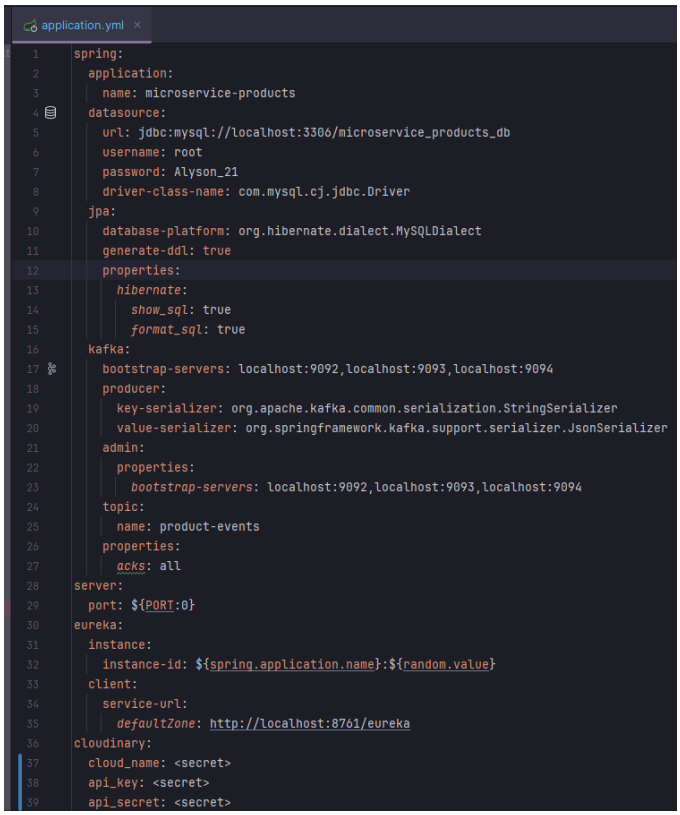


Figura 5-12 Configuración microservicio productos cuds

Dentro de la configuración de este microservicio, se pueden apreciar nuevos campos, en primer lugar, se tiene la configuración de la base de datos, dentro de esta se especifican todos los campos necesarios para poder realizar una conexión a la base de datos.

Para este microservicio que tiene como objetivo gestionar los cambios de escritura de una entidad, se optó por una base de datos relacional SQL. Esto debido a que según el patrón CQRS, el patrón CUDS que funge como el encargado de gestionar la información, debe enfocarse en mantener la consistencia de datos lo más posible, cosa que las bases de datos relacionales manejan de excelente forma.

Posterior a la configuración de la base de datos, se tiene la configuración de JPA, esta herramienta es de gran utilidad al realizar consultas a la base de datos, no es obligatoria, pero si es altamente recomendable incluirla en microservicios que trabajen con bases de datos SQL.

La configuración de Kafka es importante de analizar, en este caso al tratarse de un componente que va a producir eventos (recordemos que estamos aplicando el estilo de arquitectura basado en eventos), la configuración que se asigna está orientada a la producción de eventos, dentro de las configuraciones se asignan los puertos de nuestro clúster, el método de serialización y serialización que se ocupara, y el topic sobre el cual se publicara la información.

Finalmente, al último se configura un sistema de almacenamiento en nube llamado cloudinary, este nos pide información sensible que nos brinda el sistema de almacenamiento al crear una cuenta.

Este sistema de almacenamiento fue implementado para mostrar la versatilidad que los microservicios tienen al manipular recursos e información y para demostrar como la implementación de un sistema de archivos para archivos pesados como multimedia, agiliza y reduce la carga de nuestro sistema de manera importante.

Tras ver la configuración general del microservicio, comencemos analizando las entidades que este maneja.

```java
@Entity
@Table(name = "products")
@Getter
@Setter
@ToString
@RequiredArgsConstructor
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String code;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String description;

    @Column(nullable = false)
    private Double price;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "product",
            cascade = CascadeType.ALL, orphanRemoval = true)
    @ToString.Exclude
    @EqualsAndHashCode.Exclude
    private List<ImageProduct> imageUrlList;

    @Column(name = "creation_date", nullable = false, updatable = false)
    private LocalDate creationDate;

    @Version
    private Long version;

    Hugo Alexis Chiquito Onofre
    @PrePersist
    public void prePersist() { this.creationDate = LocalDate.now(); }
```

Figura 5-13 Entidad producto

```java
22 usages  Hugo Alexis Chiquito Onofre
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "products_images")
public class ImageProduct {
    @Id
    @GeneratedValue
    private Long id;
    @Column(nullable = false)
    private String url;
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "product_id")
    private Product product;
    @Column(nullable = false, name = "hash_code")
    private String hashCode;
    Hugo Alexis Chiquito Onofre
    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (!(obj instanceof ImageProduct imageProduct)) {
            return false;
        }
        return this.hashCode != null && this.hashCode.equals(imageProduct.getHashCode());
    }
}
```

Figura 5-14 Entidad ProductImage

Como se puede apreciar, este microservicio cuenta con dos entidades principales, cada una de estas utiliza herramientas como lombok y jpa, para facilitar la creación de métodos de manipulación y relación entre entidades.

Lo que se puede destacar es la relación que se tiene entre un producto y el listado de fotos. Estas entidades están exclusivamente destinadas para ser manipuladas por el repositorio.

Para crear un repositorio solo es necesario crear una interfaz y hacer que extienda de **JPARepository** y asignarle los campos necesarios, siendo estos la entidad que se almacenara en la base de datos y el tipo de dato que almacena como ID.

```
8 usages  Hugo Alexis +1 *
public interface ProductRepository extends JpaRepository<Product, Long> {
    1 usage  Hugo Alexis
    @EntityGraph("Product.withoutImages")
    Optional<Product> findProductById(Long id);
}
```

Figura 5-15 Repositorio product

Dentro de este repositorio es posible generar consultas personalizadas, pero en general con solo extender de JPARepository ya se obtienen los comandos básicos como la creación, actualización y eliminación de entidades.

Para poder continuar con el desarrollo, antes es necesario configurar el productor de eventos de kafka, esto se consigue de la siguiente manera:

En primera instancia es necesario configurar el topic sobre el que se va a trabajar, dentro de la configuración se asigna el nombre que ya habíamos asignado en nuestro archivo de configuración y además se especifica el número de particionas y replicas que este utilizara, estas configuraciones son de gran importancia, puesto que al tener más replicas y particiones asignadas, podemos mejorar la resiliencia y la cohesión de los microservicios implicados.

```
Hugo Alexis Chiquito Onofre
@Configuration
public class KafkaTopicConfig {

    @Value("product-events")
    private String topicName;

    // spring bean kafka topic
    Hugo Alexis Chiquito Onofre
    @Bean
    public NewTopic topic() {
        return TopicBuilder
                .name(topicName)
                .partitions( partitionCount: 3)
                .replicas( replicaCount: 3)
                .build();
    }

}
```

Figura 5-16 Configuración del topic kafka

Para la producción de eventos, también es necesario crear una clase encargada de la manipulación de estos. Dentro de esta clase se utiliza la configuración del topic anteriormente proporcionada, acompañada de otras configuraciones.

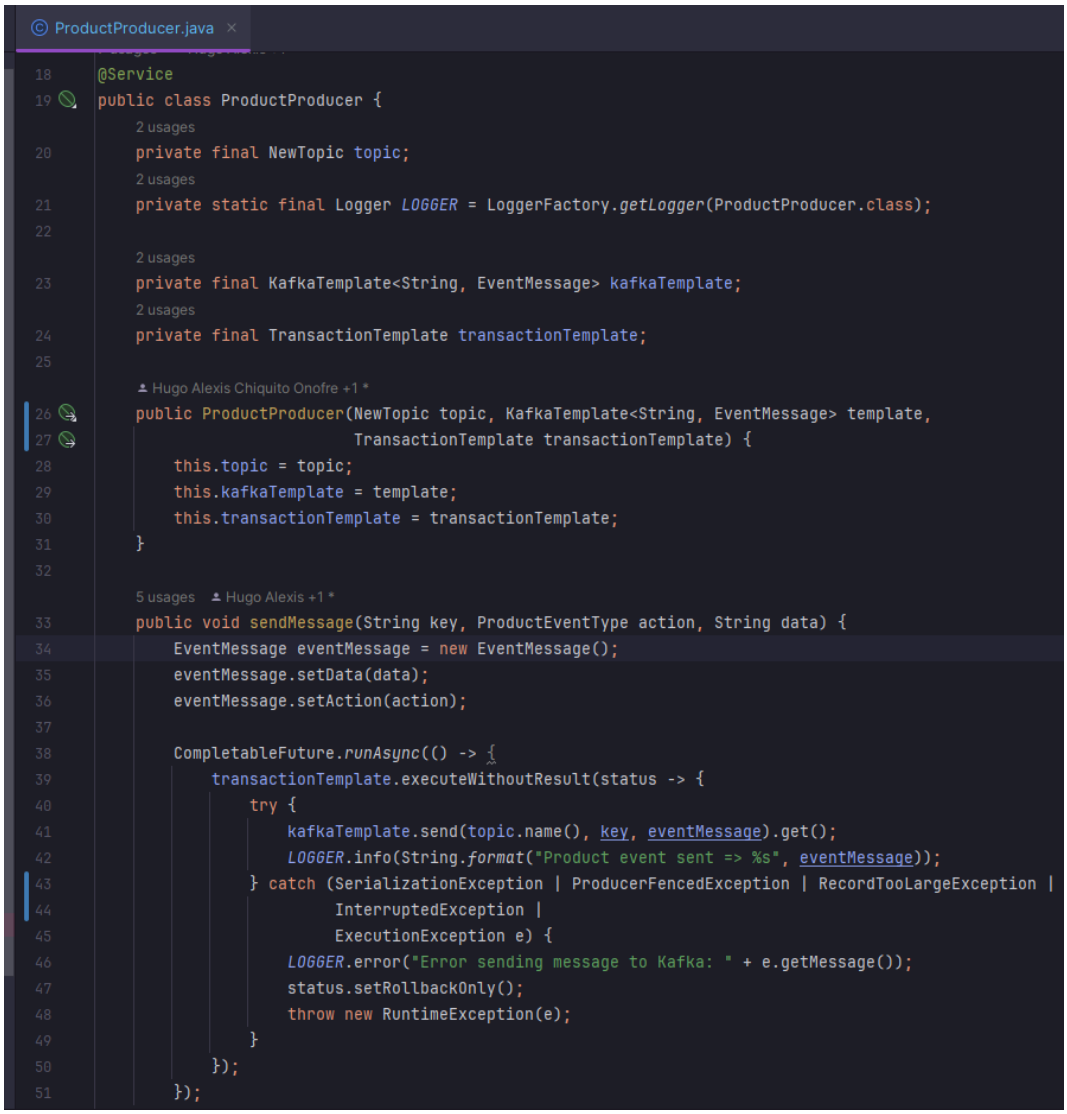


Figura 5-17 Clase encargada de publicar eventos

Dentro de la función send message se utiliza un objeto personalizado para él envió de información llamado EventMessage.

```
18 usages  Hugo Alexis +1
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class EventMessage implements Serializable {
    private ProductEventType action;
    private String data;
}
```

Figura 5-18 Clase Event Message

Esta clase tiene como atributos un String encargado de almacenar el JSON resultante de las solicitudes y un enum **ProductEventType**.

Dentro de este enum podemos listar las diferentes acciones u eventos que queremos mandar hacia nuestro broker kafka, facilitando de esta manera el intercambio de información y disminuyendo la posibilidad de errores al implementar el consumo y producción de eventos.

```
25 usages  Hugo Alexis *
public enum ProductEventType {
    3 usages
    CREATE_PRODUCT,
    2 usages
    UPDATE_PRODUCT,
    2 usages
    DELETE_PRODUCT,
    2 usages
    ADD_IMAGE,
    1 usage
    REMOVE_IMAGE,
    1 usage
    CREATE_SALE,
}
```

Figura 5-19 enum ProductEventType

Con todo esto configurado, ahora el microservicio está listo para producir eventos, como dato extra pero sumamente importante, la configuración que se ofrece da al microservicio la capacidad de producir los mensajes incluso si el broker kafka no se encuentra disponible, en cuanto este detecte que está disponible, reanuda él envió de eventos.

Antes de entrar en el módulo de servicio, hay varios aspectos que debemos abarcar.

En primer lugar, para el manejo de los datos entre diferentes módulos o capas es recomendable utilizar entidades DTO, ya que con estas podemos agilizar y seleccionar que información es enviada, reduciendo así él envió de información redundante o innecesaria, de igual forma al mostrar un resultado son bastante útiles para poder agregar información sobre la operación.

Figura 5-20 Clase ProductDTO

Esta clase DTO está destinada para transmitir información entre nuestro controlador y el servicio, por lo que cuenta con campos de **validación** como **@NotBlank**, **@NotNull**, etc.

En segundo lugar, está la necesidad de contar con un controlador de excepciones global. Esta clase es la encargada de gestionar todos las excepciones que puedan producirse dentro del microservicio.

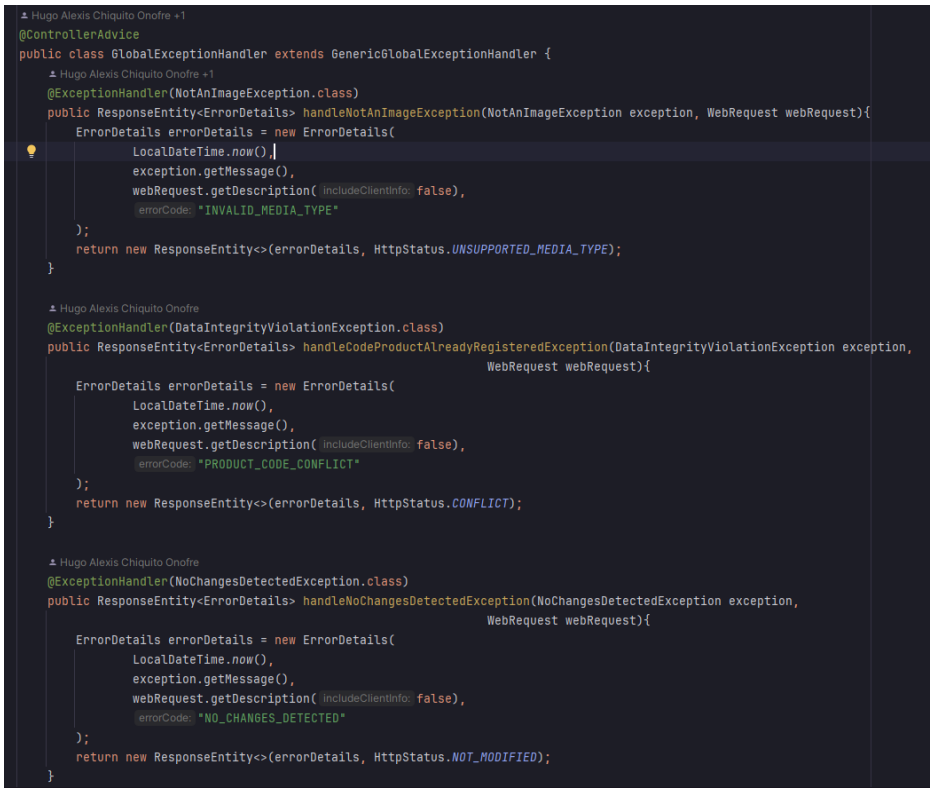


Figura 5-21 Controlador global de excepciones

Este controlador puede atrapar todo tipo de excepciones, desde las ya implementadas hasta personalizadas. Al detectar una lanzamiento de excepción este controlador es capaz de generar una respuesta al cliente, en donde puede enviar datos como un código de error, descripción, mensaje y aún más importante, un código de error HTTP adecuado para cada tipo de error, facilitando de esta forma la detección y manejo de errores ya sea dentro o fuera del componente.

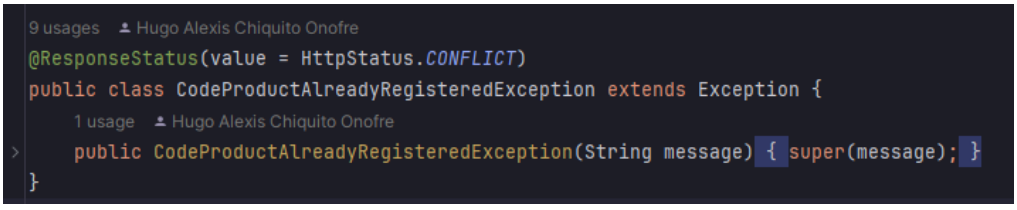


Figura 5-22 Ejemplo de excepción personalizada

Por último, algo a considerar antes de entrar en la lógica de negocio, es el uso de mappers para la transformación de la información entre diferentes clases.

Esto puede hacerse de dos formas, manual asignando campo por campo a la entidad nueva o semiautomática haciendo uso de herramientas como mapstruct.

En este ejemplo se hace uso de mapstruct para automatizar la transformación de objetos. Como se puede ver en la siguiente imagen, su ejecución es bastante simple, pero permite ajustar con el uso de etiquetas especiales en caso de ser necesario.

```
10 usages  1 implementation  Hugo Alexis +1
@Mapper
public interface ProductMapper {
    3 usages
    ProductMapper INSTANCE = Mappers.getMapper(ProductMapper.class);

    3 usages  1 implementation  Hugo Alexis
    ProductDTO mapProductToDto(Product product);
    1 usage  1 implementation  Hugo Alexis
    Product mapProductDTOToEntity(ProductDTO productDTO);
    3 usages  1 implementation  Hugo Alexis
    ImageProductDTO mapImageProductToDTO(ImageProduct entity);
    1 usage  1 implementation  Hugo Alexis
    ImageProduct mapImageProductToEntity(ImageProductDTO entityDTO);

    1 usage  1 implementation  Hugo Alexis Chiquito Onofre
    @Mapping(target = "imageUrlList", ignore = true)
    void updateProductFromDTO(ProductDTO productDTO, @MappingTarget Product product);
}
```

Figura 5-23 ProductMapper

ProductMapper.java | ProductMapperImpl.java

```
1   package com.haco.microservice.products.mapper;
2
3   import ...
10
    no usages
11  @Generated(
12      value = "org.mapstruct.ap.MappingProcessor",
13      date = "2024-03-18T19:01:01-0600",
14      comments = "version: 1.5.3.Final, compiler: javac, environment: Java 19.0.2 (Amazon.com Inc.)"
15  )
16  public class ProductMapperImpl implements ProductMapper {
17
        3 usages
18      @Override
19      public ProductDTO mapProductToDto(Product product) {
20          if ( product == null ) {
21              return null;
22          }
23
24          ProductDTO productDTO = new ProductDTO();
25
26          productDTO.setId( product.getId() );
27          productDTO.setCode( product.getCode() );
28          productDTO.setName( product.getName() );
29          productDTO.setDescription( product.getDescription() );
30          productDTO.setPrice( product.getPrice() );
31          productDTO.setImageUrlList( imageProductListToImageProductDTOList( product.getImageUrlList() ) );
32
33          return productDTO;
34      }
```

Figura 5-24 Código generado por Mapstruct para el mapeo de entidades

Con base a los componentes anteriores, ahora es momento de analizar la implementación de la lógica de negocio dentro del microservicio.

Para esto es necesario contar con la cantidad adecuada de clases dedicadas a este objetivo, asegurando que cada una de estas no se encuentre sobrecargada y se separen las responsabilidades de forma eficiente.

En nuestro caso de ejemplo, este microservicio tiene dos entidades a su cargo, una que almacena los datos del producto y otra encargada de almacenar el de las imágenes.

Por lo tanto, para poder manipular correctamente estas dos entidades, es necesario contar con dos servicios, uno dedicado a la manipulación del repositorio producto y otro para la manipulación de las imágenes en el sistema de almacenamiento externo.

```
5 usages  1 implementation  Hugo Alexis Chiquito Onofre *
public interface ProductService {
    1 usage  1 implementation  Hugo Alexis Chiquito Onofre
    ProductDTO create(ProductDTO productDTO) throws CodeProductAlreadyRegisteredException;
    1 usage  1 implementation  Hugo Alexis Chiquito Onofre
    ProductDTO update(ProductDTO productDTO) throws CodeProductAlreadyRegisteredException, NoChangesDetectedException;
    1 usage  1 implementation  Hugo Alexis Chiquito Onofre
    void deleteById(Long id);
}
```

Figura 5-25 Interfaz del servicio para el procesamiento de productos.

```
5 usages  1 implementation  Hugo Alexis Chiquito Onofre +1
public interface ImageService {
    1 usage  1 implementation  Hugo Alexis Chiquito Onofre
    ImageProcessingResult processImageFiles(MultipartFile[] files, Long idProduct);
    1 usage  1 implementation  Hugo Alexis
    void deleteImage(Long idProduct, Long idImage);
}
```

Figura 5-26 Interfaz del servicio para el procesamiento de imágenes.

Comencemos hablando de manera general sobre la implementación del servicio producto. Como se puede apreciar en la siguiente imagen, es en esta clase donde incorporamos todos los elementos que se han ido explicado hasta este momento.

Cada uno de estos elementos es esencial para el correcto funcionamiento del sistema. Aquí es donde spring boot brilla, pues con solo declarar las entidades que necesitamos en el constructor, spring boot se encarga de inyectar las dependencias necesarias para su funcionamiento.

Para la lógica de negocio, se separa en funciones basándose en la acción que se desea realizar. Cada una de estas acciones cuentan con la etiqueta **@Trasanctional**, esta etiqueta nos permite manipular la transacciones a la base de datos, proporcionando confianza y coherencia en los datos.

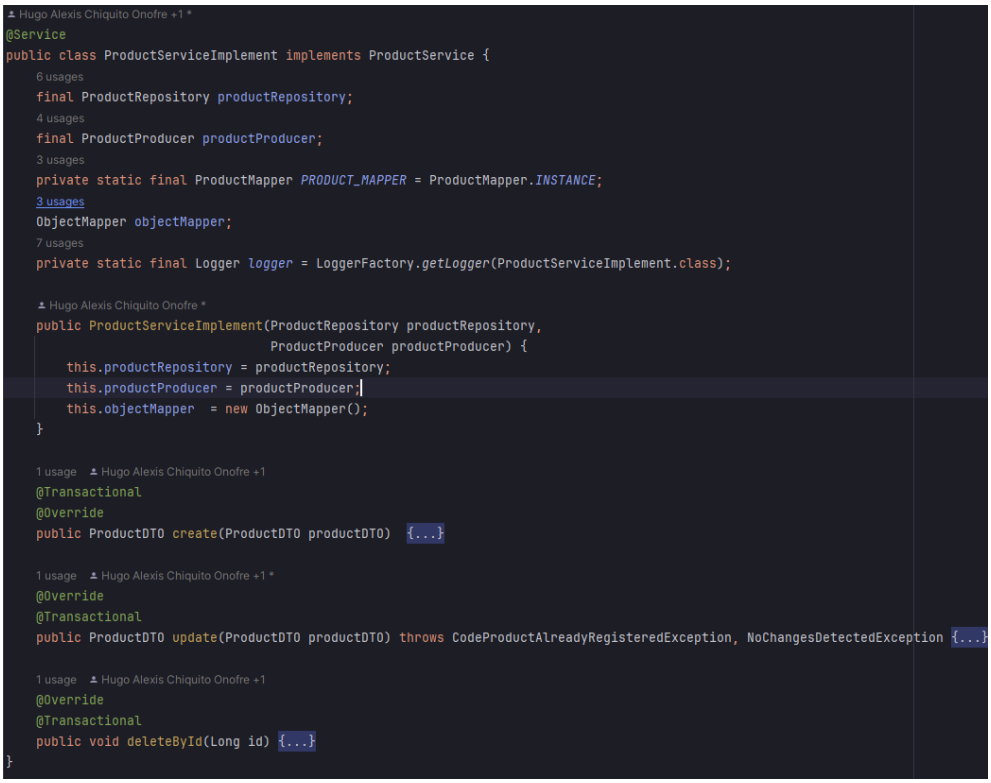


Figura 5-27 Implementación de la interfaz servicio producto.

Ya que la clase servicio fue analizada de manera general, ahora se entrará en detalle en una de las funciones para analizar cómo se maneja el flujo de información y la comunicación con el resto de los componentes.

La función seleccionada fue la encargada de actualizar un producto, esto debido a que es la que presenta un mayor desafió a la hora de actualizar la información y, por tanto, es donde se puede apreciar de mejor manera las verificaciones y el manejo de eventos.

Esta función recibe como parámetro un objeto **ProductDTO**, este objeto contiene todos los datos actualizados de la entidad, así como su ID.

En primera instancia se verifica haciendo uso del **repositorio producto** que el producto en cuestión ya se encuentre registrado en la base de datos, en caso de que no se encuentre registrado se lanzará una excepción personalizada llamada **ResourceNotFoundException** que será atrapada por nuestro **controlador global de excepciones** y enviará un mensaje de error notificando este problema al usuario.

En caso de que exista el producto, se continúa verificando si el producto que se quiere actualizar contiene información diferente a la que ya se tiene registrada, en caso de no ser así se lanza la excepción personalizada.

Luego de pasar las verificaciones necesarias, el objeto ProductDTO es **transformado** a un objeto Product utilizando el objeto **ProductMapper**.

Posterior a esto, dentro de un bloque try – catch, se utiliza el objeto repositorio para actualizar el producto, en caso de que ninguna excepción sea registrada, se llama a la instancia de **ProductProducer** que, recordemos, es la encargada de enviar los eventos hacia el **broker kafka**. A través de esta instancia, se llama la función *sendMessage* para enviar la siguiente información: una llave, el tipo de evento y el objeto transformado a formato String.

En esta función se usó el id del producto como llave por una simple razón, esta llave permite al **broker kafka** determinar a qué partición enviara este evento. Esto es de suma importancia, ya que al ser enviadas a una partición en especifica en base a su id, permite al sistema llevar un control estricto sobre el orden en que estas solicitudes están siendo procesadas, evitando de esta manera incongruencias en los datos y, por lo tanto, en el resto de los microservicios.

Por último, dentro de las **excepciones** que se pueden atrapar en este proceso, se encuentra que el producto ya estuviera previamente registrado y que el objeto haya sido modificado por otra operación. Esta última utiliza el **versionado** de la entidad para poder verificarlo. Su correcta implementación es vital para garantizar la coherencia entre los datos dentro de nuestro sistema y arquitectura en general.

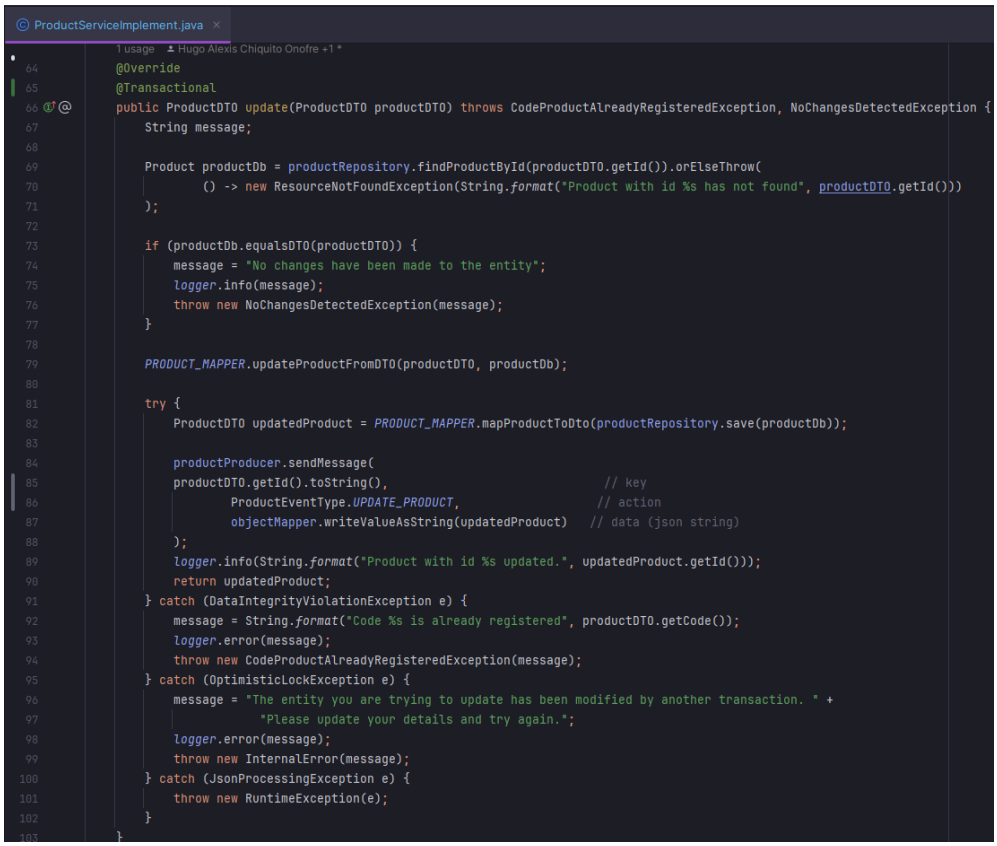


Figura 5-28 Función para actualizar un producto

Continuando con el siguiente servicio, se analiza ahora el servicio de procesamiento de imágenes.

De igual manera, este microservicio tiene todas las dependencias necesarias, solo que, en este se puede resaltar el uso de la instancia Cloudinary, esta instancia es la que permite el manejo de imágenes en el sistema de almacenamiento externo.

Para este servicio, se analiza la función *processImageFiles* por las mismas razones que actualizar producto fue seleccionada anteriormente.

En primera instancia, esta función y este servicio utiliza un nuevo tipo de clase DTO llamada **ImageEventDTO**, esta tiene la responsabilidad de almacenar la información que será enviada hacia el **broker kafka**.

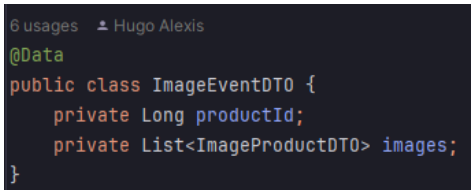


Figura 5-29 Clase DTO dedicada al almacenamiento de las imágenes que son enviadas al broker kafka.

De igual manera se comienza haciendo validaciones y asignando los valores necesarios para hacer funcionar la relación entre imágenes y producto.

Luego de esto, haciendo uso del arreglo de archivos recibido en la función, se recorre una a una para validar cosas como que la imagen sea realmente una imagen y del formato adecuado, así como que la imagen no se encuentre registrada haciendo uso de un hash y el id del producto para su identificación.

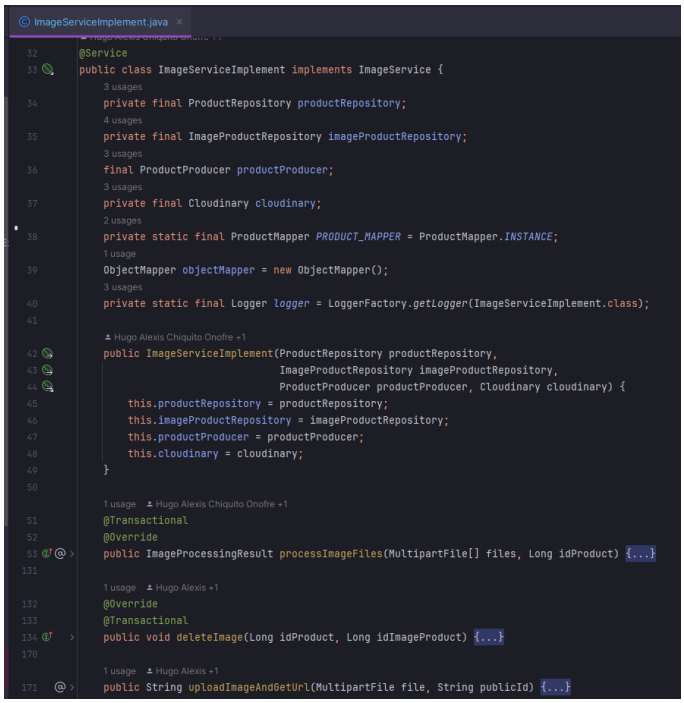
Luego de pasar por todas estas comprobaciones, se envía la imagen al servidor externo y se almacena la url pública junto con otros datos importantes para su identificación en la base de datos utilizando el **repositorio de imágenes**.

Tras finalizar el recorrido de imágenes, haciendo uso de una clase llamada ImageProcessingResult que actúa como una clase DTO, aquellas que fueron procesadas exitosamente estarán almacenadas en un arreglo de imágenes y las que tuvieron problemas, el error que tuvieron junto con su ID serán almacenadas en otro arreglo.

```
8 usages  Hugo Alexis Chiquito Onofre
@Getter
@Setter
@AllArgsConstructor
public class ImageProcessingResult {
    1 usage  Hugo Alexis Chiquito Onofre
    public ImageProcessingResult() {
        this.errors = new ArrayList<>();
        this.images = new ArrayList<>();
    }
    private String status;
    private String message;
    @JsonInclude(JsonInclude.Include.NON_EMPTY)
    private Long productId;
    @JsonInclude(JsonInclude.Include.NON_EMPTY)
    private List<ImageProductDTO> images;
    @JsonInclude(JsonInclude.Include.NON_EMPTY)
    private List<String> errors;
}
```

Figura 5-30 Clase DTO para él envió del resultado del procesamiento de imágenes

Finalmente, aquellas imágenes que fueron procesadas correctamente serán enviadas a través del evento haciendo uso de las clases DTO especiales para esta tarea y se enviará el resultado hacia el **controlador**.



```
ImageServiceImplement.java
@Service
public class ImageServiceImplement implements ImageService {
    3 usages
    private final ProductRepository productRepository;
    4 usages
    private final ImageProductRepository imageProductRepository;
    3 usages
    final ProductProducer productProducer;
    3 usages
    private final Cloudinary cloudinary;
    2 usages
    private static final ProductMapper PRODUCT_MAPPER = ProductMapper.INSTANCE;
    1 usage
    ObjectMapper objectMapper = new ObjectMapper();
    3 usages
    private static final Logger logger = LoggerFactory.getLogger(ImageServiceImplement.class);

    Hugo Alexis Chiquito Onofre +1
    public ImageServiceImplement(ProductRepository productRepository,
                                 ImageProductRepository imageProductRepository,
                                 ProductProducer productProducer, Cloudinary cloudinary) {
        this.productRepository = productRepository;
        this.imageProductRepository = imageProductRepository;
        this.productProducer = productProducer;
        this.cloudinary = cloudinary;
    }

    1 usage  Hugo Alexis Chiquito Onofre +1
    @Transactional
    @Override
    public ImageProcessingResult processImageFiles(MultipartFile[] files, Long idProduct) {...}

    1 usage  Hugo Alexis +1
    @Override
    @Transactional
    public void deleteImage(Long idProduct, Long idImageProduct) {...}

    1 usage  Hugo Alexis +1
    public String uploadImageAndGetUrl(MultipartFile file, String publicId) {...}
```

Figura 5-31 Vista genera del servicio de imágenes.

```java
public ImageProcessingResult processImageFiles(MultipartFile[] files, Long idProduct) {

    ImageEventDTO imageEventDTO = new ImageEventDTO();
    ImageProcessingResult imageProcessingResult = new ImageProcessingResult();
    String message;
    String status;

    Product productDb = productRepository.findById(idProduct).orElseThrow(
            () -> new ResourceNotFoundException(
                    String.format("Product with id %s has not found", idProduct)
            )
    );

    imageProcessingResult.setProductId(idProduct);

    for (MultipartFile file : files) {
        ImageProduct imageProduct = new ImageProduct();
        ImageProduct imageProductSaved;

        try {
            if (!isImage(file)) {
                imageProcessingResult.getErrors().add(String.format(
                        "%s is not an image, only jpg, jpeg and png files are allowed",
                        file.getOriginalFilename()
                ));
                logger.error("Unsupported media file type");
                continue;
            }

            imageProduct.setProduct(productDb);
            imageProduct.setHashCode(DigestUtils.md5Hex(file.getInputStream()));

            if (productDb.getImageUrlList().contains(imageProduct) ||
                    imageProcessingResult.getImages().contains(
                            PRODUCT_MAPPER.mapImageProductToDTO(imageProduct))) {
                imageProcessingResult.getErrors().add(String.format(
                        "'%s' is already uploaded.", file.getOriginalFilename()
                ));
                logger.warn("Duplicated file");
                continue;
            }

            imageProduct.setUrl(uploadImageAndGetUrl(file, imageProduct.getHashCode()));
            imageProductSaved = imageProductRepository.save(imageProduct);
            imageProcessingResult.getImages().add(PRODUCT_MAPPER.mapImageProductToDTO(imageProductSaved));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    boolean imagesAddedBool = !imageProcessingResult.getImages().isEmpty();
    boolean errorsBool = !imageProcessingResult.getErrors().isEmpty();

    message = imagesAddedBool ?
            (errorsBool ? "Some images were processed successfully, but others encountered errors. " +
                    "Check the details."
            : "All images were processed successfully without errors.")
            : "All images encountered errors during processing.";

    status = imagesAddedBool ?
            (errorsBool ? "success_with_errors" : "success") : "failure";

    imageProcessingResult.setStatus(status);
    imageProcessingResult.setMessage(message);
    logger.info(String.format("Status image processing: %s", status));

    if (!imageProcessingResult.getStatus().equals("failure")) {
        imageEventDTO.setProductId(idProduct);
        imageEventDTO.setImages(imageProcessingResult.getImages());
        try {
            productProducer.sendMessage(
                    productDb.getId().toString(),   // key
                    ProductEventType.ADD_IMAGE,    // action
                    objectMapper.writeValueAsString(imageEventDTO)  // data (json string)
            );
        } catch (JsonProcessingException e) {...}
    }
    return imageProcessingResult;
```

Figura 5-32 Función encargada del procesamiento de imágenes

Ahora que hemos explorado a detalle el funcionamiento de los componentes esenciales del microservicio, es momento de analizar el funcionamiento del componente final, el controlador.

El controlador cuenta con dos etiquetas importantes para su funcionamiento, **@RestController** y **@RequestMapping** estas dos son las que habilita al microservicio la capacidad de recibir peticiones REST, y este componente es al que el gateway redirecciona las peticiones.

El controlador debe declarar las instancias de los servicios que esta por utilizar, en este ejemplo, se llaman los servicios anteriormente llamados.

Luego de esto se generan los *endpoints* o *puntos de entrada*, para esto es vital etiquetar la función con **@PostMapping, @PutMapping, @DeleteMapping** o **@GetMapping** según sea necesario para la operación.

```java
@RestController
@RequestMapping("products")
public class ProductController {
    4 usages
    final ProductService productService;
    3 usages
    final ImageService imageService;

    Hugo Alexis Chiquito Onofre
    public ProductController(ProductService productService, ImageService imageService) {...}

    Hugo Alexis Chiquito Onofre *
    @Operation(summary = "Create product", description = "Create a new product")
    @ApiResponse(responseCode = "201", description = "Created")
    @PostMapping
    public ResponseEntity<?> createProduct(@Valid @RequestBody ProductDTO productDTO)
            throws CodeProductAlreadyRegisteredException {...}

    Hugo Alexis Chiquito Onofre
    @PutMapping("/{idProduct}")
    @Operation(summary = "Update product", description = "Update product data")
    @ApiResponse(responseCode = "200", description = "Updated")
    public ResponseEntity<?> updateProduct( @PathVariable Long idProduct, @Valid @RequestBody ProductDTO productDTO)
            throws CodeProductAlreadyRegisteredException, NoChangesDetectedException {...}

    Hugo Alexis Chiquito Onofre
    @DeleteMapping("/{idProduct}")
    @Operation(summary = "Delete product", description = "Delete a  product by ID")
    @ApiResponse(responseCode = "204", description = "Deleted successfully, there is not content to return")
    public ResponseEntity<?> deleteProduct(@PathVariable Long idProduct) {...}

    /* IMAGES ENDPOINTS  */
    Hugo Alexis Chiquito Onofre
    @PostMapping("/{idProduct}/images")
    @Operation(summary = "Upload product images", description = "Upload images of a product using ID and a list of files")
    @ApiResponse(responseCode = "200", description = "All the images were uploaded correctly")
    @ApiResponse(responseCode = "400", description = "Something wrong, check details")
    public ResponseEntity<?> uploadProductImages(@PathVariable Long idProduct, @RequestParam MultipartFile[] files)  {...}
```

Figura 5-33 Controlador del microservicio producto CUDS

Estas funciones retornan una respuesta de tipo **ResponseEntity**, la cual es encargada de almacenar y retornar el código de respuesta, así como los datos.

Internamente la lógica que estas manejan no suele o se recomienda ser muy complicada, básicamente se encargan de llamar los servicios necesarios y devolver la respuesta adecuada para cada operación.

```
/* IMAGES ENDPOINTS  */
Hugo Alexis Chiquito Onofre
@PostMapping("/{idProduct}/images")
@Operation(summary = "Upload product images", description = "Upload images of a product using ID and a list of files")
@ApiResponse(responseCode = "200", description = "All the images were uploaded correctly")
@ApiResponse(responseCode = "400", description = "Something wrong, check details")
public ResponseEntity<?> uploadProductImages(@PathVariable Long idProduct, @RequestParam MultipartFile[] files)  {
    ImageProcessingResult imageProcessingResult = imageService.processImageFiles(files, idProduct);
    return imageProcessingResult.getStatus().equals("failure") ?
            ResponseEntity.status(HttpStatus.BAD_REQUEST).body(imageProcessingResult) :
            ResponseEntity.ok(imageProcessingResult);
}
```

Figura 5-34 Función punto de entrada para el procesamiento de imágenes.

En resumen, se han examinado los elementos principales del microservicio y e se ha verificado la aplicación de los principios y patrones, así como las problemáticas que estos resuelven.

Es importante recordar que es microservicio no cuenta con endpoints de tipo GET, debido a que estamos siguiendo el patrón CQRS, el cual nos pone como objetivo principal la separación de responsabilidades entre la lectura de datos y la edición de estos.

5.3.5 Microservicio visualización productos

El microservicio de visualización de productos basa su funcionamiento en el patrón CQRS, el estilo de arquitectura basada en eventos y microservicios.

Anteriormente ya se ha explicado la implementación del componente encargado de la escritura, actualización y publicación de los datos, ahora es turno de

explicar la implementación del componente encargado de consumir y visualizar dichos datos.

Para esto, comencemos definiendo la estructura del microservicio analizando la siguiente figura.

La estructura del microservicio de visualización contiene bastantes elementos similares al microservicio anterior y en general con cualquier otro microservicio.

Este microservicio cuenta con un controlador, encargado de recibir las peticiones y enviar respuestas, un controlador global de excepciones, sus respectivos servicios para el manejo de solicitudes que en este caso corresponden a la visualización, clases DTO para la transferencia de información, modelos para el almacenamiento de estos datos junto con sus respectivas clases mapper para la transformación de datos.

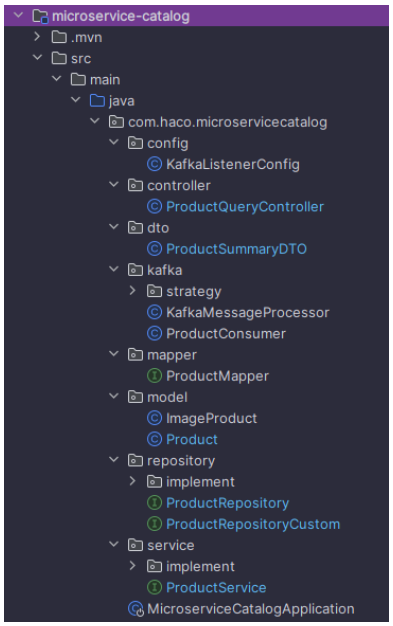


Figura 5-35 Estructura del microservicio de visualización

Claramente dentro de estos la lógica se adapta los requisitos y el funcionamiento de este microservicio, pero su funcionamiento e implementación es relativamente similar, por lo tanto, no se entrará en detalle sobre la explicación de cada uno de estos, en cambio se hará especial énfasis en aquellas partes del microservicio en donde se puede apreciar el consumo de información, así como la forma de visualizar esta información a través de los endpoints.

Dicho esto, comencemos analizando los modelos, aunque es cierto que comparte similitudes, estos se manejan un poco diferentes debido a que, para este tipo de microservicio se propuso utilizar una base de datos no relacional, lo cual trae consigo cambios en la estructura del modelo.

Para este ejemplo se optó por utilizar MongoDB, la configuración de esta base de datos es bastante sencilla, como se puede ver en la siguiente figura.

```
data:
  mongodb:
    uri: mongodb+srv://<user>:<password>@productquerydb.7okb2rb.mongodb.net/?retryWrites=true&w=majority&appName=ProductQueryDB
    database: product_query_db
```

Figura 5-36 Configuración base de datos MongoDB

Para manipular los modelos en una base de datos MongoDB, tenemos que etiquetar la clase con la etiqueta **@Document**, con esta podremos generar los registros necesarios en nuestra base de datos sin problema, por otra parte, los campos son similares a los del otro microservicio debido a que estará recibiendo en gran medida todos ellos.

Otra diferencia importante que se puede visualizar es la desaparición de la relación entre la clase Product y la clase ImageProduct, esto ocurre debido a que en lugar de genera dos entidades diferentes en la base de datos, en este caso al tratarse de una base de datos no relacional y optimizada para leer grandes cantidades de información, se optó por almacenar directamente estos campos dentro de el mismo documento.

```java
@Getter
@Setter
@ToString
@RequiredArgsConstructor
@Document(collection = "products")
public class Product {
    @Id
    private Long id;
    private String code;
    private String name;
    private String description;
    private Double price;
    private List<ImageProduct> images;
    @Version
    private Long version;
}
```

Figura 5-37 Modelo documento del producto

```java
1 usage  Hugo Alexis
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class ImageProduct {
    private String id;
    private String url;
}
```

Figura 5-38 Clase de apoyo para el almacenamiento para las imágenes

De igual forma, para la manipulación de estos datos es necesario crear un repositorio. La creación del repositorio funciona de manera similar, solo que, en lugar de extender de JPARepository, para repositorios MongoDB se extiende de MongoRepository y se le pasan los mismo valores, el modelo y el tipo de variable de su ID.

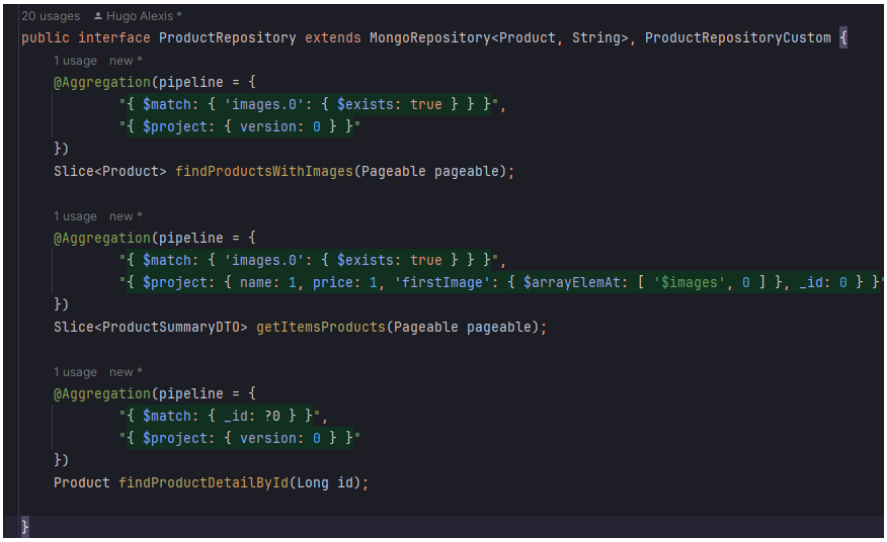


Figura 5-39 Repositorio productos MongoDB

Dentro de este repositorio se cuentan con tres consultas personalizadas, esto debido a que es recomendable crear consultas y clases DTO para procesar la información que es necesaria. De igual forma se recomienda trabajar con páginas al recuperar la información para reducir la carga al consumir esta información. Implementando estas medidas garantizamos un mejor rendimiento y manejo de los datos.

Continuando con el flujo del manejo de la información, este microservicio a diferencia del anterior consigue o recopila la información a través de los eventos publicados por otros microservicios, en este caso, del microservicio productos cuds.

Para lograr esto, el microservicio cuenta con un módulo especial para la recepción de eventos que vamos a analizar a continuación.

Partamos analizando la configuración para el consumidor kafka, la clase ***KafkaListenerConfig*** en primer lugar, se configura un contenedor del consumidor kafka, en él se define el un bean de tipo *ConcurrentKafkaListenerContainerFactory* que se encarga de configurar las instancias encargadas de consumir los mensajes.

```java
@Configuration
@Slf4j
public class KafkaListenerConfig {

    Hugo Alexis
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerContainerFactory(
            ConsumerFactory<String, String> consumerFactory) {

        ConcurrentKafkaListenerContainerFactory<String, String> factory = new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory);
        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);
        factory.setCommonErrorHandler(errorHandler());
        return factory;
    }
```

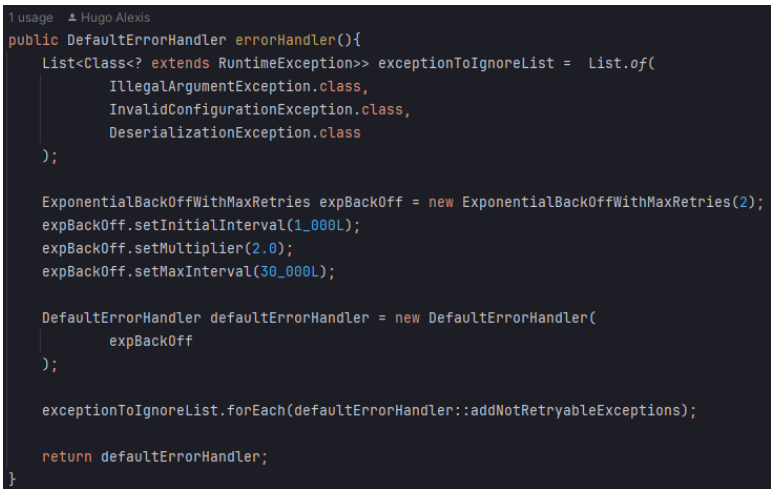
Figura 5-40 Configuración del contenedor de escucha kafka

Dentro de esta configuración, se específica que la confirmación del consumo de mensajes se hará de forma inmediata y manual y también se especifica que el manejo de errores tendrá un procesamiento customizado que es detallado más adelante.

Para especificar la configuración personalizada del manejo de errores, se crea un método en el que se instancia un *DefaultErrorHandler* que es el que se encargará de manejar los errores que ocurran durante el consumo o la escucha de mensajes y eventos.

Dentro de esta instancia se especifica una lista de excepciones las cuales, de llegar a detectarse, los eventos que las provocaron no serán procesados o reintentados.

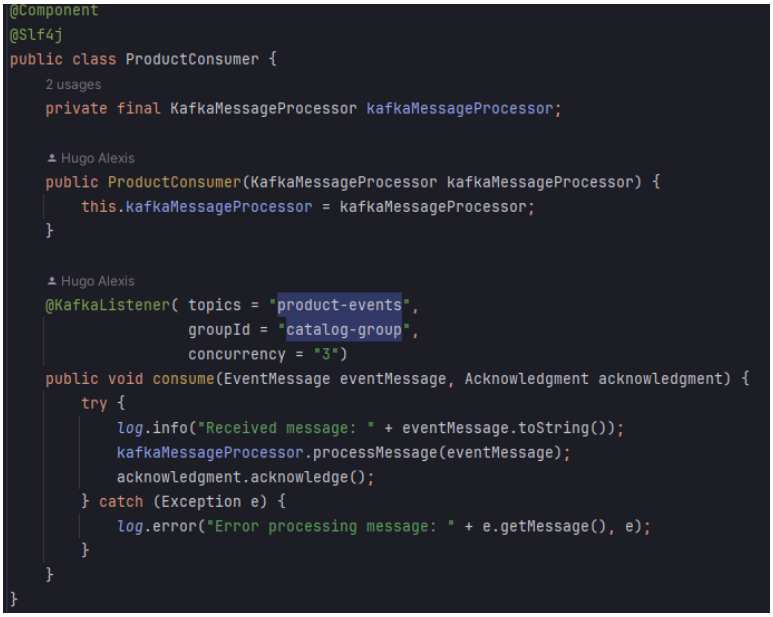
De igual manera, configura el mecanismo de reintentos para el manejo de errores, proponiendo un intervalo de 1000 milisegundos con un multiplicador de 2.0 y un intervalo de máximo 30000 milisegundos. Básicamente que, por cada reintento, el tiempo de espera se duplicara hasta llegar al límite de 30 segundos.



```
1 usage  Hugo Alexis
public DefaultErrorHandler errorHandler(){
    List<Class<? extends RuntimeException>> exceptionToIgnoreList =  List.of(
            IllegalArgumentException.class,
            InvalidConfigurationException.class,
            DeserializationException.class
    );

    ExponentialBackOffWithMaxRetries expBackOff = new ExponentialBackOffWithMaxRetries(2);
    expBackOff.setInitialInterval(1_000L);
    expBackOff.setMultiplier(2.0);
    expBackOff.setMaxInterval(30_000L);

    DefaultErrorHandler defaultErrorHandler = new DefaultErrorHandler(
            expBackOff
    );

    exceptionToIgnoreList.forEach(defaultErrorHandler::addNotRetryableExceptions);

    return defaultErrorHandler;
}
```

Figura 5-41 Configuración para el manejo de errores en el consumidor kafka.

Ya que hemos explicado la configuración del contenedor de escucha, es momento de analizar el consumidor, las configuraciones adicionales y como aplica las configuraciones previas.



```
@Component
@Slf4j
public class ProductConsumer {
    2 usages
    private final KafkaMessageProcessor kafkaMessageProcessor;

    Hugo Alexis
    public ProductConsumer(KafkaMessageProcessor kafkaMessageProcessor) {
        this.kafkaMessageProcessor = kafkaMessageProcessor;
    }

    Hugo Alexis
    @KafkaListener( topics = "product-events",
                    groupId = "catalog-group",
                    concurrency = "3")
    public void consume(EventMessage eventMessage, Acknowledgment acknowledgment) {
        try {
            log.info("Received message: " + eventMessage.toString());
            kafkaMessageProcessor.processMessage(eventMessage);
            acknowledgment.acknowledge();
        } catch (Exception e) {
            log.error("Error processing message: " + e.getMessage(), e);
        }
    }
}
```

Figura 5-42 Consumidor de eventos kafka.

De manera un tanto similar al productor, se asigna el tópico al cual se desea consumir y al tratarse de un consumidor, este necesita de un *groupId,* con este el broker kafka podrá llevar el control de que mensajes ha consumido y el estado en el que el microservicio se encuentra.

En cuando a la función de consumo es bastante simple, haciendo uso de la misma clase que se usó para él envió de información, se recibe la información y es enviada a un procesador de eventos, si este mensaje es procesado correctamente se actualiza el offset dentro del broker kafka.

Para el consumo de mensajes se propuso la implementación del patrón Strategy, esto debido a que por naturaleza este módulo de procesamiento de eventos estará recibiendo diferentes tipos de eventos y por lo tanto necesita cambiar su comportamiento en base a estos eventos.

Para implementar el patrón Strategy fue necesario implementar los siguientes elementos:

1. Una interfaz Strategy, la cual sirve de base para las estrategias.

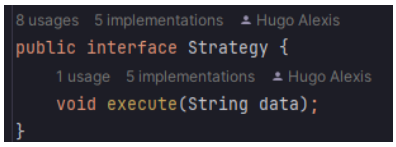


Figura 5-43 Interfaz Strategy

2. Una clase *KafkaMessageProcesor* que funge como la clase contexto manteniendo una referencia a cada una de las estrategias concretas.

Figura 5-44 Clase KafkaMessageProcesor o clase contexto del patrón Strategy

3. Estrategias concretas. Para cada uno de los tipos de evento es necesario implementa la estrategia concreta.

```java
public class CreateProductStrategy implements Strategy {
    3 usages
    private final ProductRepository productRepository;
    7 usages
    private static final Logger LOGGER = LoggerFactory.getLogger(CreateProductStrategy.class);
    1 usage
    private final ProductMapper PRODUCT_MAPPER = ProductMapper.INSTANCE;
    1 usage  Hugo Alexis
    public CreateProductStrategy(ProductRepository productRepository) { this.productRepository = productRepository; }

    1 usage  Hugo Alexis
    @Override
    @Transactional
    public void execute(String data) {
        ObjectMapper objectMapper = new ObjectMapper();
        ProductDTO productDTO;

        try {
            productDTO = objectMapper.readValue(data, ProductDTO.class);
        } catch (JsonProcessingException e) {
            LOGGER.error("Error deserializing product data from JSON: {}", e.getMessage());
            LOGGER.debug("JSON input causing error: {}", sanitizeJson(data));
            return;
        }

        try {
            if (productRepository.findById(productDTO.getId().toString()).isPresent()) {
                LOGGER.info("Product with unique identifier {} already exists. No action taken.", productDTO.getId());
                return;
            }

            productRepository.save(PRODUCT_MAPPER.mapProductDTOToDocument(productDTO));
            LOGGER.info("New product with id {} has been created", productDTO.getId());
        } catch (ValidationException e) {
            LOGGER.error("Error creating product with name '{}': {}", productDTO.getName(), e.getMessage());
        } catch (OptimisticLockingFailureException e) {
            LOGGER.error("Conflict detected when processing update for Product ID {}: version mismatch. " +
                    "Data may have been modified by another transaction.", productDTO.getId(), e);
        } catch (Exception e) {
            LOGGER.error("Unexpected error creating product with ID {}: {}", productDTO.getId(), e.getMessage(), e);
```

Figura 5-45 Clase de estrategia concreta

Con la correcta implementación del patrón Strategy, obtenemos un manejo de eventos mucho más robusto, confiable, eficiente y de fácil mantenimiento o modificación, ya que de ser necesario agregar un nuevo evento, es tan fácil como crear otra estrategia concreta y añadirla su referencia de la clase contexto (*KafkaMessageProcesor*).

En resumen, se ha explorado la configuración clave para que el microservicio de visualización pueda funcionar correctamente. Se ha discutido sobre la implementación y configuración del consumidor Kafka, así como el manejo de errores personalizado y recomendado.

De igual manera se detalló el cómo ajustar tanto los repositorios como los modelos para el consumo de datos desde una base de datos no relacional con MongoDB. Por lo que ahora se explicará otro tipo de microservicio, en el que se continuará con el enfoque centrado en discutir aquellos elementos relevantes y distintivos del microservicio.

5.3.6 Microservicio de venta (crud reactivo)

Tal y como se mencionó en el capítulo cuatro, propuesta de la arquitectura, este microservicio puede tomarse como una combinación de los dos primeros, es decir, en él no se aplica de forma directa la separación de responsabilidades en cuanto al manejo de datos, pero adquiere la posibilidad de consumir y producir eventos.

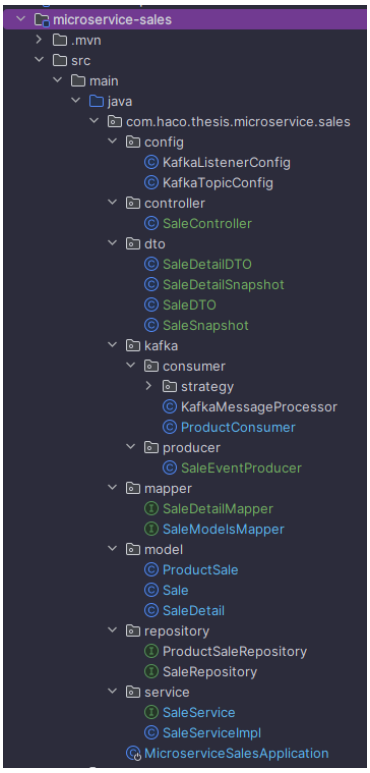


Figura 5-46 Estructura del microservicio de ventas (crud reactivo)

Para lograr esto como se puede esperar, comparte componentes o similitudes con los dos anteriores microservicios analizados.

Como se puede apreciar en la figura, realmente no se encuentran presentes nuevos módulos, incluso la logica para consumir y producir eventos es similar.

Se puede resaltar el uso de nuevas clases y modelos que fueron diseñados siguiendo nuestro principio de solo manipular la información necesaria, por lo que por ejemplo a diferencia del Product de los otros microservicios, este solo almacena la información relevante para la venta como el nombre y precio.

La clase SaleSnapshot se encarga de crear un replicado de venta que sirva de historial almacenado el precio y el nombre sin importar que ese se actualice mas adelante.

En cuanto a por que este microservicio tiene su propia base de datos con la entidad de modelos, se debe a que de esta forma aprovechando el flujo de evetos, podemos mantener nuestro microservicio mucho mas independiente evitando consultas directas hacia el microservicio de visualización. Promoviendo de esta forma la independencia, escalabilidad y confianza de nuestro microservicio.

A cotinuación se muestran fragmentos de código que puede ser de interes, sin embargo no se espera entrar en detalle en la mayoría de estos.

```yaml
kafka:
  bootstrap-servers: localhost:9092,localhost:9093,localhost:9094
  consumer:
    bootstrap-servers: localhost:9092,localhost:9093,localhost:9094
    auto-offset-reset: earliest
    enable-auto-commit: false
    group-id: sales-group
    key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
    value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer
    properties:
      spring.json.trusted.packages: "*"
  producer:
    key-serializer: org.apache.kafka.common.serialization.StringSerializer
    value-serializer: org.springframework.kafka.support.serializer.JsonSerializer
  admin:
    properties:
      bootstrap-servers: localhost:9092,localhost:9093,localhost:9094
  topic:
    consumer:
      name: product-events
    producer:
      name: sale-events
```

Figura 5-47 Configuración kafka para consumir y producir eventos.

```java
@Getter
@Setter
@NoArgsConstructor
public class SaleDTO {
    @JsonProperty("sale_details")
    @NotNull
    private List<SaleDetailDTO> saleDetails
    @NotNull
    private BigDecimal total;
    private Date createdDate;
}
```

Figura 5-48 Clase DTO para el procesamiento de datos de venta.

```java
4 usages  new *
@Getter
@Setter
@NoArgsConstructor
public class SaleDetailDTO {
    @JsonProperty("product_id")
    @NotNull(message = "id must not be empty or null")
    @Positive(message = "id must be greater than 0")
    private Long productId;
    @NotNull(message = "quantity must not be empty or null")
    @Positive(message = "quantity must be greater than 0")
    private BigDecimal quantity;
    @NotNull(message = "quantity must not be empty or null")
    @Positive(message = "quantity must be greater than 0")
    private BigDecimal amount;
}
```

Figura 5-49 Clase DTO para el manejo de datos de ventas.

```java
@Getter
@Setter
@NoArgsConstructor
public class SaleSnapshot {
    private Long id;
    @JsonProperty("sale_details")
    private List<SaleDetailSnapshot> saleDetails;
    private BigDecimal total;
    private Date createdDate;
}
```

Figura 5-50 Clase DTO para el almacenamiento de ventas como historial y envió como evento

```java
@Getter
@Setter
@NoArgsConstructor
public class SaleDetailSnapshot {
    private String code;
    private String name;
    private BigDecimal price;
    private BigDecimal quantity;
    private BigDecimal amount;
}
```

Figura 5-51 Clase DTO con el detalle de veta para el almacenamiento de historial y envió como evento.

```java
@Data
@Entity
@Table(name = "products")
public class ProductSale {
    @Id
    private Long id;
    @Column(nullable = false, unique = true)
    private String code;
    @Column(nullable = false)
    private String name;
    @Column(nullable = false)
    private BigDecimal price;
}
```

Figura 5-52 Modelo del producto adaptado para el microservicio de ventas.

```java
@Data
@Entity
@Table(name = "sales")
public class Sale {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @OneToMany(mappedBy = "sale", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<SaleDetail> saleDetails;
    @NotNull
    private BigDecimal total;
    @Column(name = "created_date")
    @Temporal(TemporalType.TIMESTAMP)
    Date createdDate;
}
```

Figura 5-53 Modelo de ventas

```java
17 usages  Hugo Alexis *
@Entity
@Getter
@Setter
@NoArgsConstructor
@Table(name = "sale_details")
public class SaleDetail {
    @Id
    @GeneratedValue
    private Long id;
    @ManyToOne
    @JoinColumn(name = "sale_id")
    private Sale sale;
    @ManyToOne
    private ProductSale product;
    @NotNull
    private BigDecimal quantity;
    @NotNull
    private BigDecimal amount;
}
```

Figura 5-54 Modelo de ventas detalle

```java
1 Hugo Alexis *
public SaleServiceImpl(SaleRepository saleRepository, ProductSaleRepository productSaleRepositor
    this.saleRepository = saleRepository;
    this.productSaleRepository = productSaleRepository;
    this.saleEventProducer = saleEventProducer;
}

1 usage  new *
@Override
@Transactional
public SaleSnapshot create(SaleDTO saleDTO) {

    Sale sale = new Sale();
    List<SaleDetail> saleDetails = new ArrayList<>();

    sale.setTotal(saleDTO.getTotal());
    sale.setCreatedDate(new Date());

    Set<Long> productIds = saleDTO.getSaleDetails().stream() Stream<SaleDetailDTO>
            .map(SaleDetailDTO::getProductId) Stream<Long>
            .collect(Collectors.toSet());
    logger.info(productIds.toString());
    Map<Long, ProductSale> productsMap = productSaleRepository.findAllById(productIds) List<Produ
            .stream() Stream<ProductSale>
            .collect(Collectors.toMap(ProductSale::getId, Function.identity()));
    logger.info(productsMap.toString());

    for (SaleDetailDTO detailDTO: saleDTO.getSaleDetails()) {
        SaleDetail saleDetail = new SaleDetail();

        saleDetail.setSale(sale);
        saleDetail.setQuantity(detailDTO.getQuantity());
        saleDetail.setAmount(detailDTO.getAmount());

        if (detailDTO.getProductId() != null) {
            ProductSale productSale = productsMap.get(detailDTO.getProductId());
            if (productSale == null) {
                throw new ResourceNotFoundException("Product with ID " +
                        detailDTO.getProductId() + " not found.");
            }
            saleDetail.setProduct(productSale);
        }
        saleDetails.add(saleDetail);
    }
    sale.setSaleDetails(saleDetails);
    logger.info(sale.toString());


    SaleSnapshot saleSnapshot = SALE_MAPPER.mapSaleToSaleSnapshot(saleRepository.save(sale));

    try {
        saleEventProducer.sendMessage(
                saleSnapshot.getId().toString(),
                ProductEventType.CREATE_SALE,
                objectMapper.writeValueAsString(saleSnapshot));
    } catch (JsonProcessingException e) {
        throw new RuntimeException(e);
    }
    return saleSnapshot;
}
```

Figura 5-55 Implementación de la función crear venta dentro del servicio ventas.

5.3.7 Microservicio análisis de datos

Finalmente, en el último componente tenemos el microservicio encargado de analizar los datos generados. En este ejemplo de implementación este componente estará encargado de recibir los datos provenientes del microservicio de ventas, para posteriormente generar información analizando los datos y regresando estadísticas e inclusive gráficos.

Para conseguir este funcionamiento, es necesario preparar algunas configuraciones. Recordemos que, para este componente, se propuso utilizar Python como leguaje principal debido a las ventajas que ofrece su ecosistema para analizar datos de forma rápida y eficiente. Por tanto, algunas configuraciones son distintas a las que hemos visto previamente y por ello son detalladas a continuación.

Comencemos con las configuraciones para habilitar la comunicación con el resto de los componentes, en primera instancia es necesario configurar el cliente eureka, su configuración no es muy diferente de lo que se hacía en java, sigue necesitando de los mismo valores para poder registrarse y funcionar correctamente.

```
1  from py_eureka_client import eureka_client
2
3
   2 usages
4  def register_with_eureka(app_name, eureka_server="http://localhost:8761/eureka", port=5000, instance_host="127.0.0.1"):
5      eureka_client.init(eureka_server=eureka_server,
6                         app_name=app_name,
7                         instance_port=port,
8                         instance_host=instance_host)
9
```

Figura 5-56 Configuración del cliente Eureka en Python

Otra configuración importante es la conexión con la base de datos, para este ejemplo de implementación se decidió utilizar una base de datos relacional, pero también es posible implementar otros tipos de almacenamiento más robustos como almacenes de datos.

En cuanto a la configuración, se utilizaron librerías como SQLAlchemy para facilitar la manipulación de los datos, los campos que requieren nuevamente no son muy diferentes a los que se solicitan en java.

```python
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()


2 usages
def init_db(app):
    app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql+pymysql://root:Alyson_21@localhost:3306/microservice_data_sales_db'
    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
    db.init_app(app)

    with app.app_context():
        db.create_all()
```

Figura 5-57 Configuración de la base de datos en Python

Por otra parte, los modelos empleados para la manipulación de los datos no difieren sobre que campos o datos son almacenados, en general solo fueron adaptados a la sintaxis empleada dentro del ecosistema de Python.

```python
from datetime import datetime, timezone
from database_config import db



class Sale(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    total = db.Column(db.Numeric(10, 2))
    created_date = db.Column(db.DateTime, default=datetime.now(timezone.utc))
    sale_details = db.relationship('SaleDetail', backref='sale', lazy=True)



class SaleDetail(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    code = db.Column(db.String(120), nullable=False)
    name = db.Column(db.String(120), nullable=False)
    price = db.Column(db.Numeric(10, 2))
    quantity = db.Column(db.Numeric(10, 2))
    amount = db.Column(db.Numeric(10, 2))
    sale_id = db.Column(db.Integer, db.ForeignKey('sale.id'), nullable=False)

```

Ilustración 1 Modelos empleados en el componente de análisis de datos Python.

Por último, se configura e implementa el consumidor de eventos kafka, a diferencia de java, en esta propuesta se implementa de una forma más directa.

```python
2 usages
def consume_messages(app):
    client = KafkaClient(hosts='localhost:9092,localhost:9093,localhost:9094')
    topic = client.topics[b'sale-events']
    consumer = topic.get_simple_consumer()

    with app.app_context():
        for message in consumer:
            print("Message received:", message.value)
            if message is not None:
                try:
                    data = json.loads(message.value.decode('utf-8'))
                    print(data)

                    action = data.get('action')
                    if action == 'CREATE_SALE':
                        sale_data_str = data.get('data')
                        sale_data = json.loads(sale_data_str)

                        sale_id = sale_data.get('id')
                        sale = Sale.query.get(sale_id)

                        if sale is None:
                            sale = Sale(
                                id=sale_id,
                                total=sale_data.get('total'),
                                created_date=datetime.fromtimestamp(int(sale_data.get('createdDate')) / 1000,
                                                                    tz=timezone.utc)
                            )
                            db.session.add(sale)

                            sale_details = sale_data.get('sale_details')
                            for detail_data in sale_details:
                                sale_detail = SaleDetail(
                                    code=detail_data.get('code'),
                                    name=detail_data.get('name'),
                                    price=detail_data.get('price'),
                                    quantity=detail_data.get('quantity'),
                                    amount=detail_data.get('amount'),
                                    sale=sale
                                )
                                db.session.add(sale_detail)

                            db.session.commit()
                            print(f"Sale was successfully registered in the system with id: {sale_id}")

                except SQLAlchemyError as e:
                    print(f"Something went wrong processing kafka message: {e}")
                    db.session.rollback()
```

Figura 5-58 Implementación del consumidor kafka en Python.

Para que funcione adecuadamente, es necesario implementar un hilo para que este se ejecute en segundo plano y pueda escuchar los eventos y hacer las actualizaciones en el sistema.

Dentro del archivo principal de la aplicación, se crearon funciones para implementar las configuraciones y módulos que fueron detallados anteriormente. Cabe destacar que es necesario prestar mucha atención al orden y el momento en que estos son llamados e implementados. Pues puede afectar de formas inesperadas la ejecución.

```python
from flask import Flask
from database_config import init_db
from routes import register_routes
from eureka_config import register_with_eureka
import threading
from kafka_consumer import consume_messages
import logging

logging.basicConfig(level=logging.DEBUG)


1 usage
def create_app():
    app = Flask(__name__)
    init_db(app)
    register_with_eureka(app_name="sales-data-microservice")
    register_routes(app)
    return app


1 usage
def start_consumer(app):
    with app.app_context():
        consume_messages(app)


if __name__ == '__main__':
    print("Starting app instance")
    app_instance = create_app()
    consumer_thread = threading.Thread(target=start_consumer, args=(app_instance,))
    consumer_thread.start()
    app_instance.run(debug=True, host='0.0.0.0')
```

Figura 5-59 Puesta en marcha del microservicio de análisis de datos.

Por último, se implementa algo similar a un controlador de Java en Python. Para este ejemplo, esta entidad se encarga de recibir las solicitudes y de realizar el análisis solicitado para posteriormente enviarlo.

En cuanto a las herramientas que fueron utilizadas para realizar estos análisis, se puede destacar matplotlib y pandas, siendo estas una de las razones principales para utilizar Python para este microservicio y no Java Spring Boot.

```python
from flask import Flask, Response, jsonify
import pandas as pd
from sqlalchemy.exc import SQLAlchemyError
from database_config import db
from models import Sale, SaleDetail
from flask import jsonify
import matplotlib.pyplot as plt
import io
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas


2 usages
def register_routes(app):
    @app.route('/analysis/sales_by_product', methods=['GET'])
    def get_sales_by_product():
        try:
            details_df = pd.read_sql(db.session.query(SaleDetail).statement, db.engine)
            sales_df = pd.read_sql(db.session.query(Sale).statement, db.engine)

            df_merged = pd.merge(sales_df, details_df, left_on='id', right_on='sale_id')
            total_sales_product = df_merged.groupby('name')['amount'].sum().reset_index()

            sales_product_json = total_sales_product.to_json(orient='records')

            return Response(sales_product_json, mimetype='application/json')
        except SQLAlchemyError as e:
            app.logger.error(f"Error when performing sales analysis by product: {e}")
            return jsonify({"error": "Error processing request"}), 500

    @app.route('/analysis/sales_percentage_by_product', methods=['GET'])
    def get_sales_percentage_by_product():
        try:
            # Read sale details and calculate total sales by product
            details_df = pd.read_sql(db.session.query(SaleDetail).statement, db.engine)
            sales_by_product = details_df.groupby('name')['amount'].sum()

            # Calculate the sales percentage per product relative to the total
            total_sales = sales_by_product.sum()
            sales_percentage = (sales_by_product / total_sales) * 100

            # Sort the percentages from highest to lowest
            sales_percentage_sorted = sales_percentage.sort_values(ascending=False)

            # Create a pie chart
            fig, ax = plt.subplots(figsize=(10, 6))  # Adjust the figure size as needed here
            wedges, texts, autotexts = ax.pie(sales_percentage_sorted, autopct='%1.1f%%', startangle=90,
                                              counterclock=False, colors=plt.cm.tab20.colors)

            # Improve visualization
            ax.set_title("Sales Percentage by Product")

            # Create the legend on the side with the product names and their respective colors
            legend = ax.legend(wedges, sales_percentage_sorted.index, title="Products", loc="center left",
                               bbox_to_anchor=(1, 0.5))

            plt.setp(autotexts, size=8, weight="bold")
            ax.set_ylabel('')  # Remove the y-axis label

            # Layout adjustments to accommodate the legend outside the graph
            fig.tight_layout(rect=[0, 0, 0.75, 1])  # Adjust the rectangle to leave space for the legend

            # Convert the figure to PNG in bytes and send it as a response
            output = io.BytesIO()
            FigureCanvas(fig).print_png(output)
            return Response(output.getvalue(), mimetype='image/png')
        except SQLAlchemyError as e:
            app.logger.error(f"Error performing sales percentage by product analysis: {e}")
            return jsonify({"error": "Error processing request"}), 500
```

Figura 5-60 Modulo encargado de recibir y realizar el análisis de datos.

5.4. Resultados y evaluación

En esta sección se presenta un resumen de los resultados obtenidos y la evaluación del sistema de ejemplo implementado, para esto el análisis estará enfocado en el flujo de operaciones que ocurren al crear un producto, realizare una venta y el impacto que estas operaciones tienen con el resto de los componentes. Además, se mostrará un ejemplo sobre como el microservicio de análisis de datos utiliza este flujo y los datos creados para procesar y extraer información de valor.

Antes de entrar en profundidad sobre el manejo de operaciones y el flujo de eventos y datos entre los microservicios, a continuación, se ofrecen capturas que detallan la implementación de nuestros componentes.

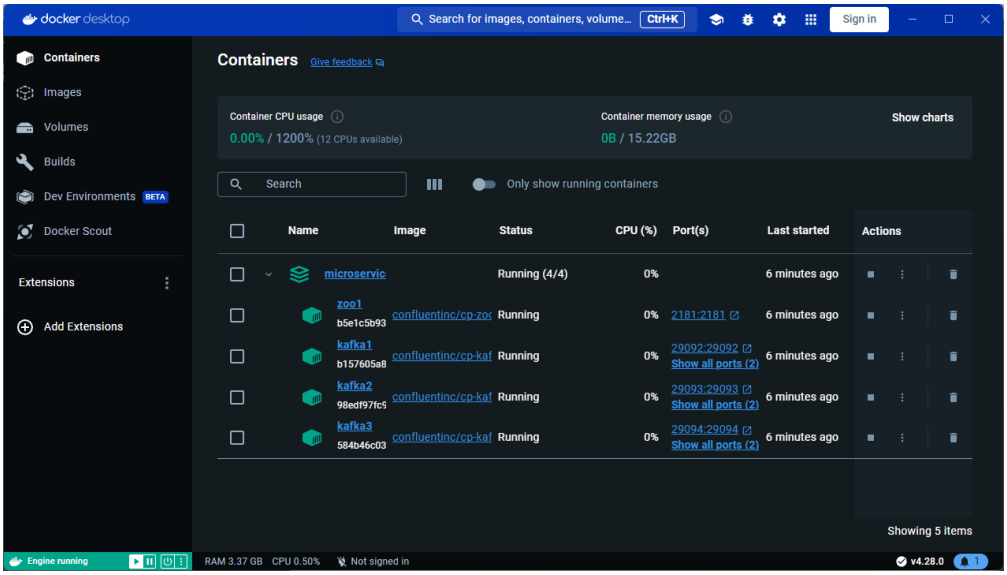


Figura 5-61 Apache kafka broker ejecutándose en contenedores Docker.

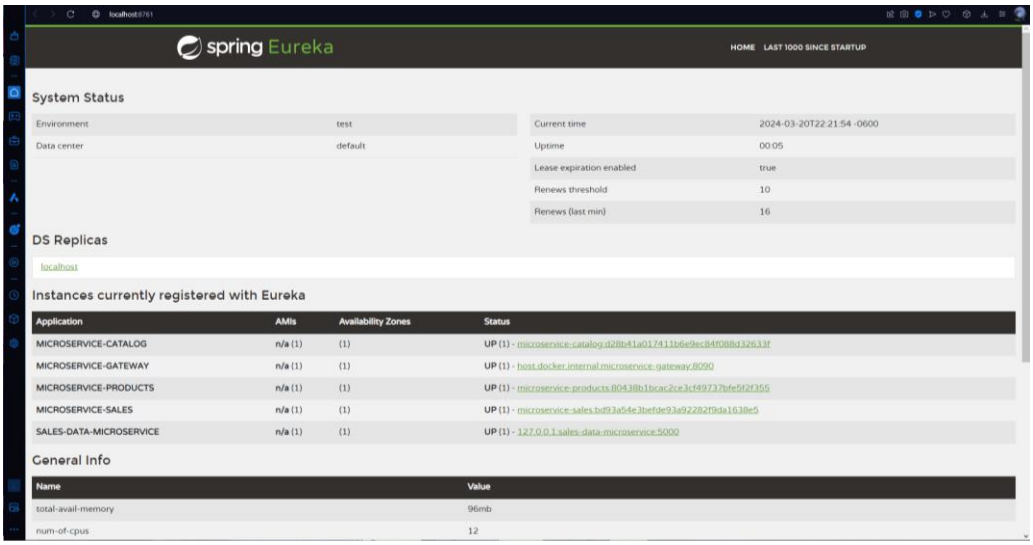


Figura 5-62 Microservicios, servidor eureka y gateway ejecutándose correctamente.

Una vez detallado esto, podemos comenzar a detallar nuestros procesos.

El proceso de creación del producto comienza con la creación de un producto, para realizar esto se hace uso de la herramienta Postman para realizar peticiones hacia nuestra API.

Para realizar una petición exitosa, es necesario utilizar la dirección del recurso correcta y enviar los datos necesarios para procesar correctamente la solicitud. En este caso se envía esta información de ejemplo con el formato correcto.

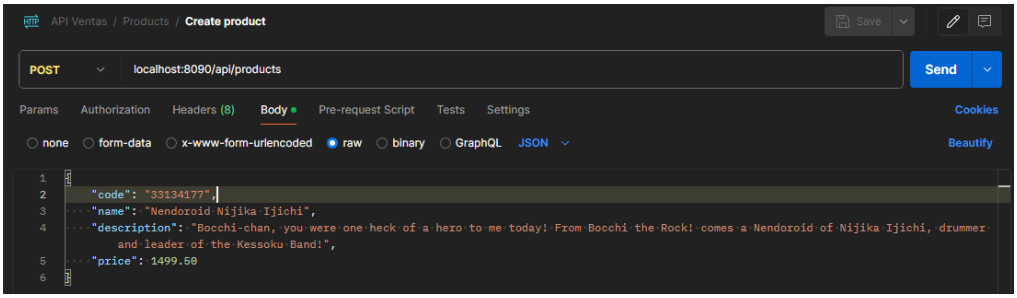


Figura 5-63 Petición para la creación de un producto (formato correcto).

Al enviar la solicitud ocurrirán muchas cosas dentro de nuestro sistema, en primer lugar, el microservicio producto cuds verificará la información enviada y de pasar todas las validaciones, devolverá la entidad creada junto con su ID y el código de estatus 201. De igual manera también generará y compartirá el evento con la información de la operación, en este caso la creación de un producto y los datos del producto que hemos creado.

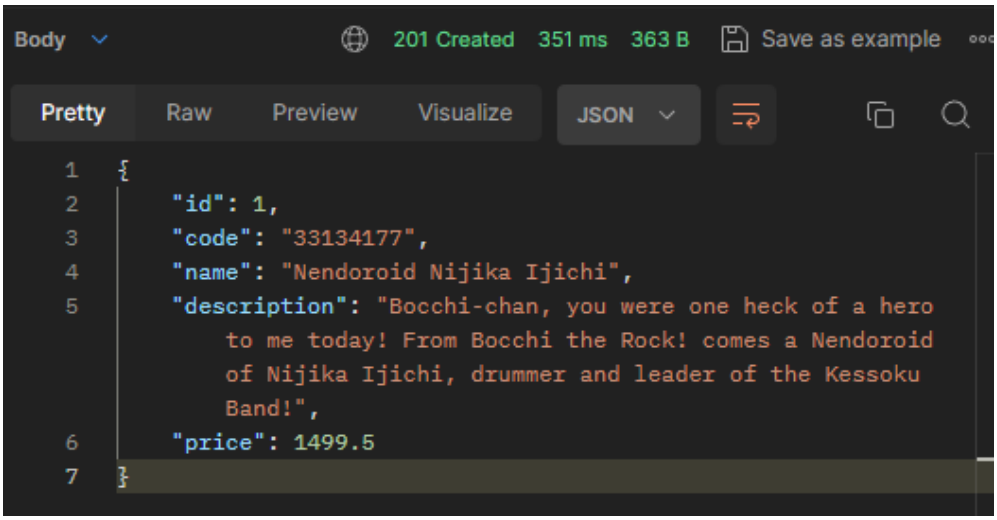


Figura 5-64 Respuesta a la petición de crear producto exitosa.

Al generase el evento después de una ejecución correcta, el resto de los componentes que están consumiendo el topic de eventos en el que fue creado, recibirán este evento y realizaran la actualización en sus propias bases de datos.

Figura 5-65 Lectura y creación del evento crear producto en microservicio ventas.

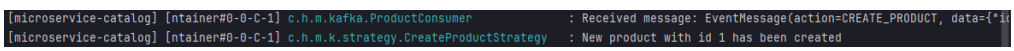


Figura 5-66 Consumo y creación del evento crear producto en microservicio de visualización.

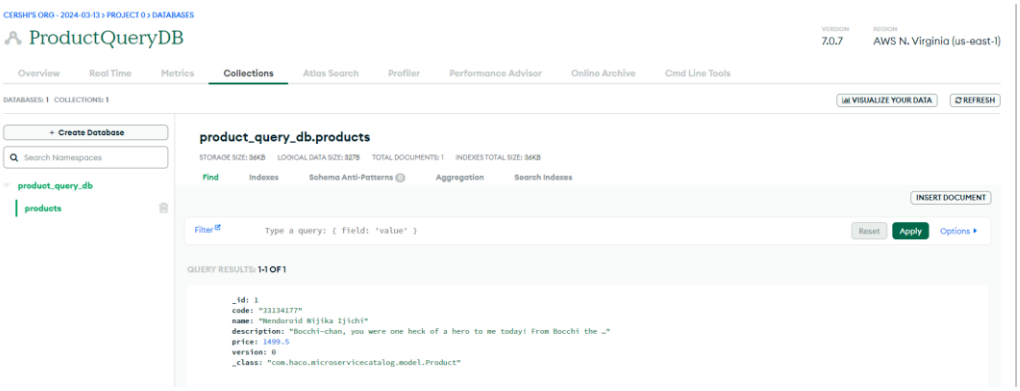


Figura 5-67 Verificación de creación del producto en la base de datos MongoDB del microservicio de visualización

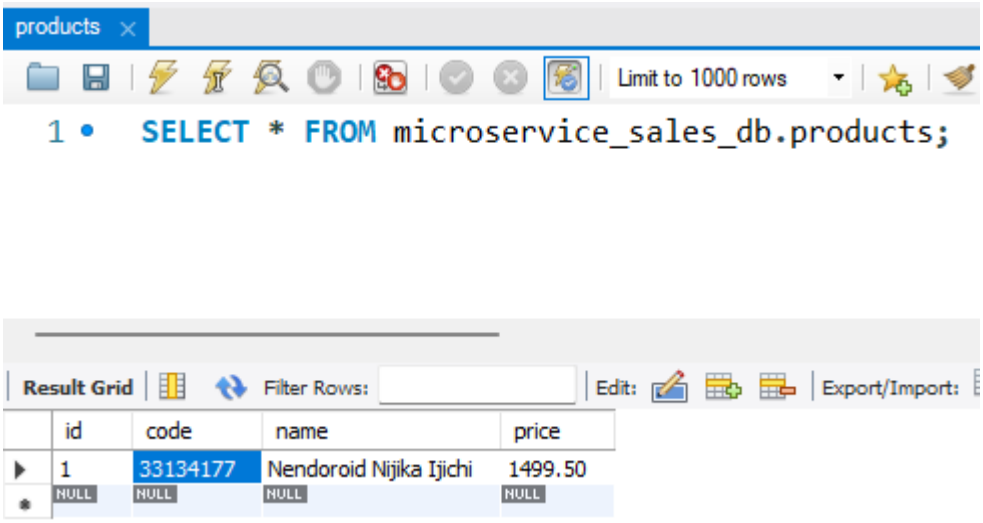


Figura 5-68 Verificación de creación del producto en la base de datos del microservicio de ventas.

Tras insertar y ser consumido el producto de forma exitosa por los componentes, ahora se analizará el funcionamiento al insertar imágenes sobre el producto que hemos creado. Para esto realizamos la petición con los archivos necesarios de la siguiente manera.

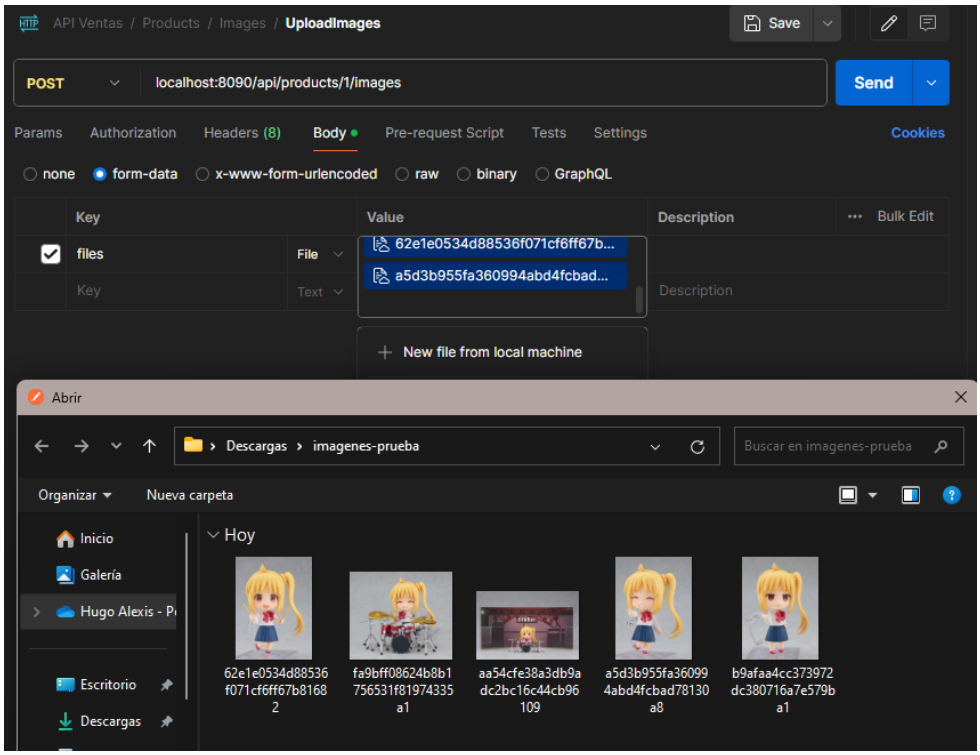


Figura 5-69 imágenes a procesar y petición para el procesamiento de imágenes.

Al enviar la petición el ID del producto y las imágenes necesarias, el sistema hará las verificaciones y posteriormente subirá estás imágenes al sistema de archivos externo para posteriormente recuperar las URL de las imágenes procesadas y enviar la respuesta, así como crear y publicar el evento de subida de imágenes.

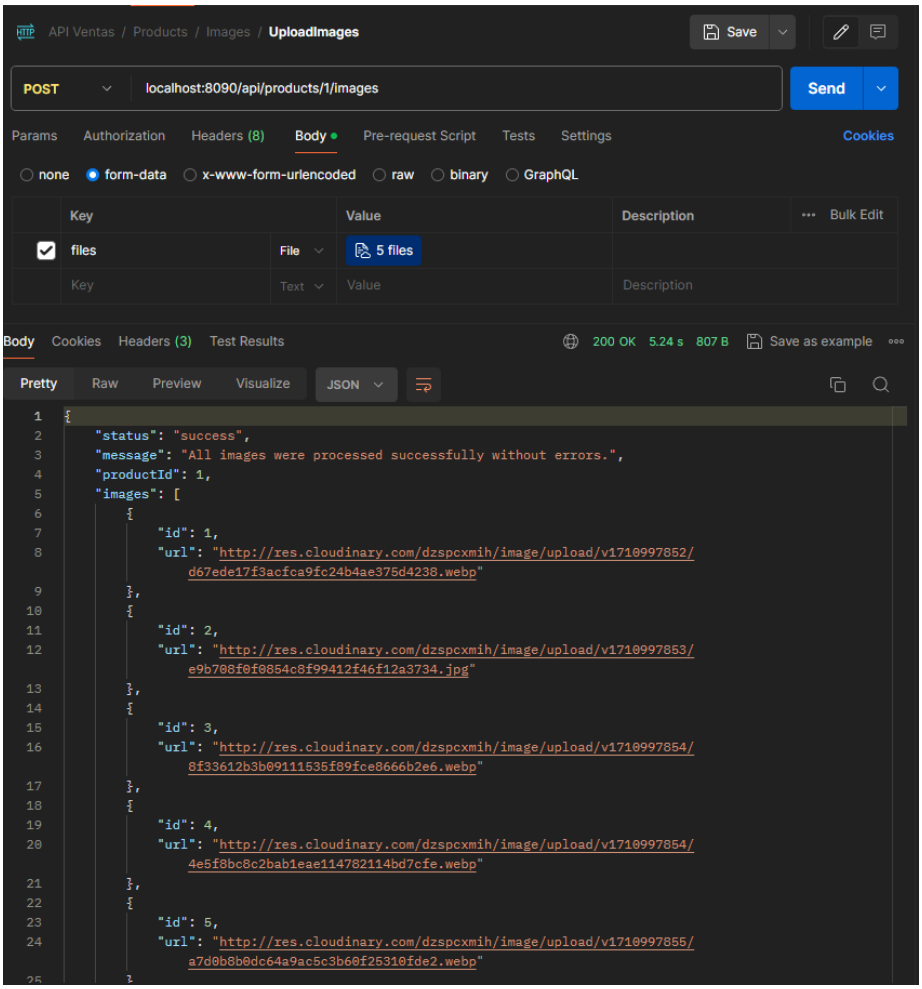


Figura 5-70 Subida de imágenes exitosa.

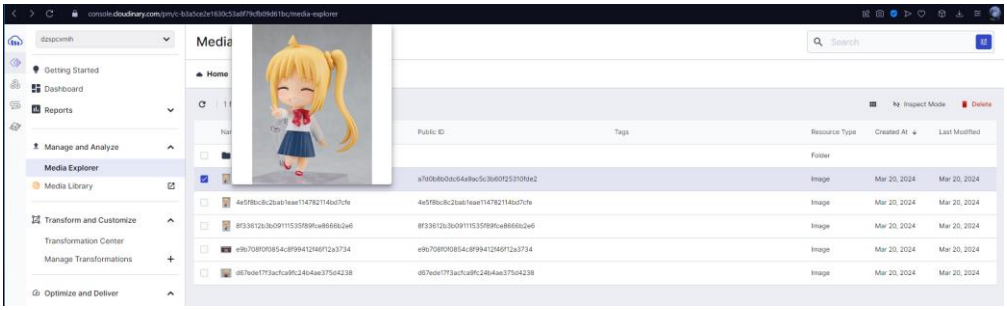


Figura 5-71 Verificación en el sistema externo de almacenamiento sobre la subida exitosa de las imágenes.

Una vez completado el producto, podemos visualizar la forma en que estos fueron almacenados por el resto de los microservicios. Recordemos que el microservicio de visualización almacena una copia con todos los datos que fueron ingresados incluidas las direcciones de las imágenes.

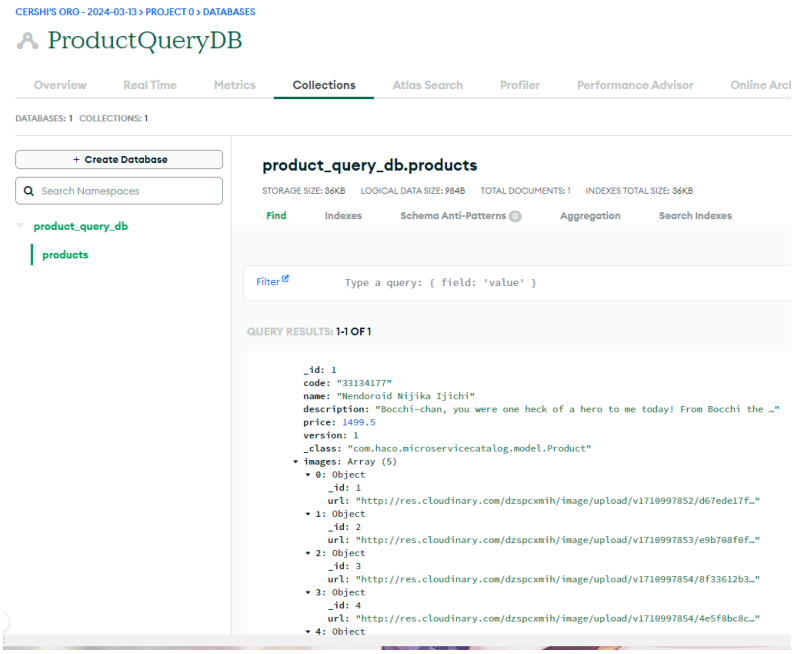


Figura 5-72 Verificación de la actualización en la base de datos de visualización.

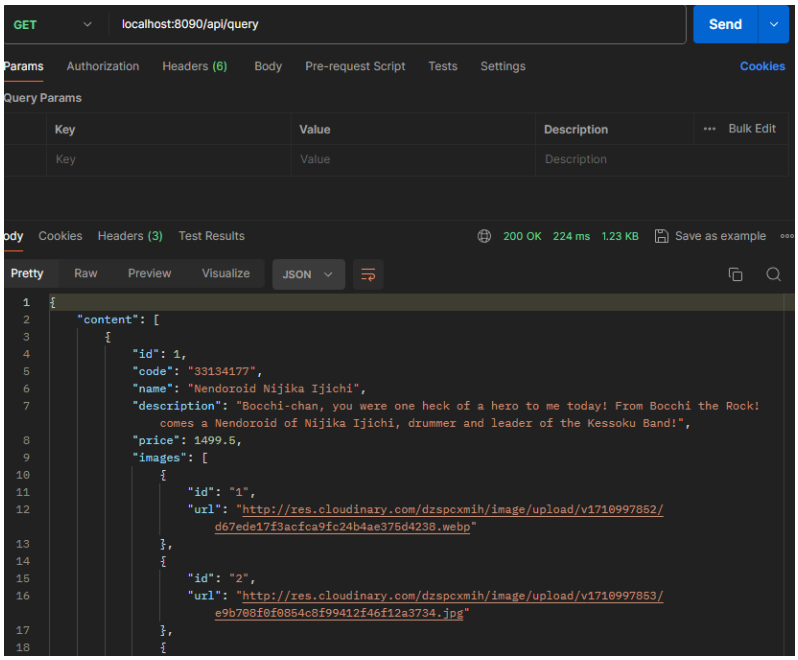


Figura 5-73 Verificación a través de una solicitud al microservicio de visualización.

Tras insertar algunos datos más de prueba, ahora se pone aprueba a la visualización personalizada.

Comenzando por la visualización de los productos resumida, en donde solo se regresa el nombre, precio y la primera imagen utilizando paginado.

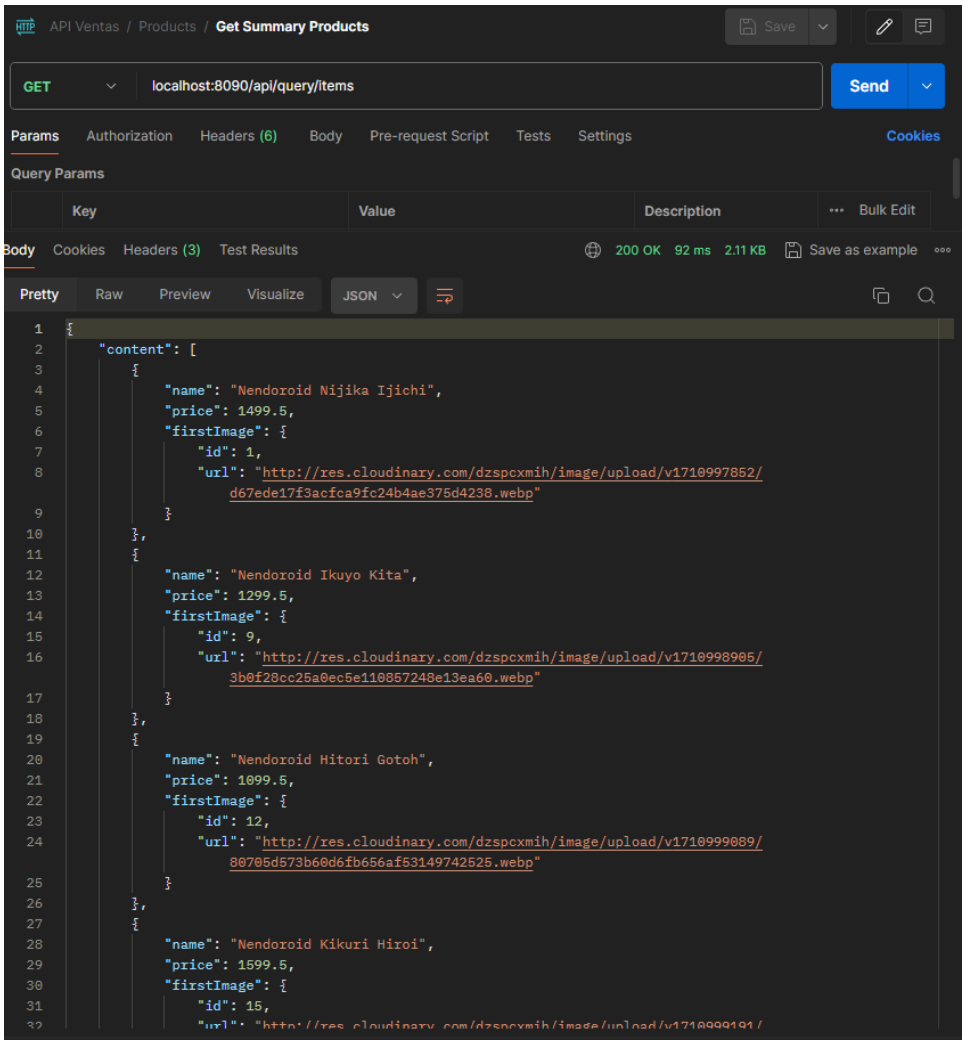


Figura 5-74 Visualización personalizada a través del microservicio de visualización.

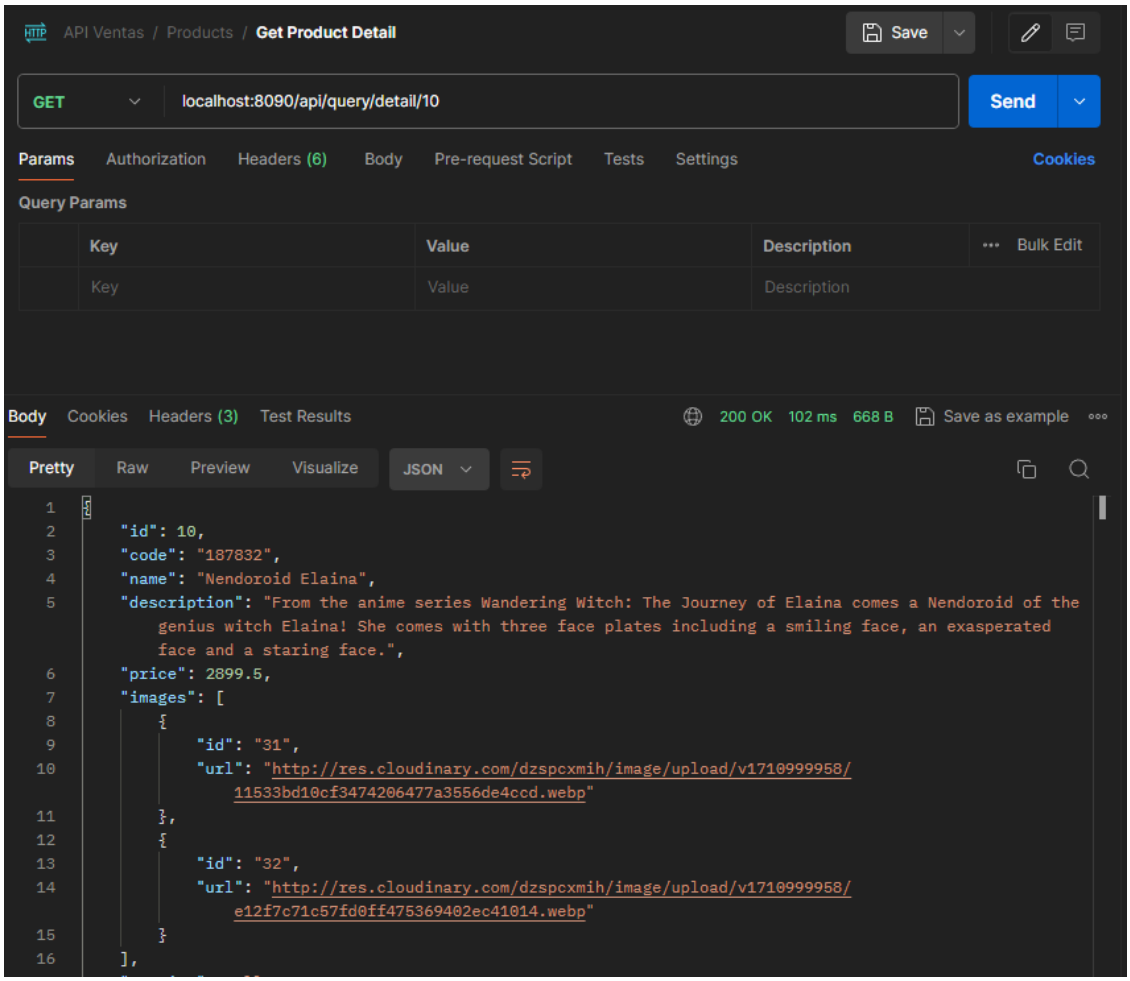


Figura 5-75 Visualización en detalle utilizando el ID del producto.

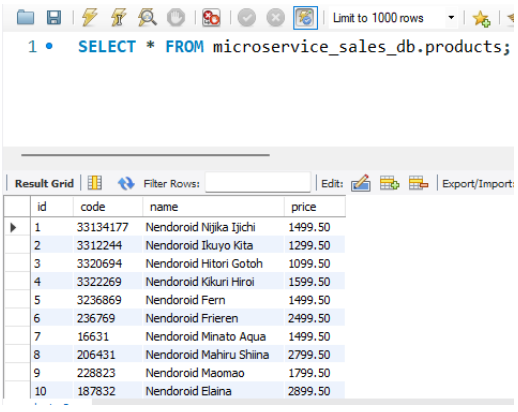


Figura 5-76 Productos en el microservicio de ventas.

Luego de analizar el funcionamiento de los componentes de visualización y gestión de datos de productos, es momento de entrar al microservicio de ventas.

Como se mencionó anteriormente, a continuación, se pondrá a prueba el funcionamiento al crear una venta.

Para crear una venta es necesario enviar la solicitud con la URL adecuada y los datos necesarios, como el listado con el ID del producto, el precio al que se vendió, la cantidad, el monto total del producto y un total que englobe todos los productos vendidos.

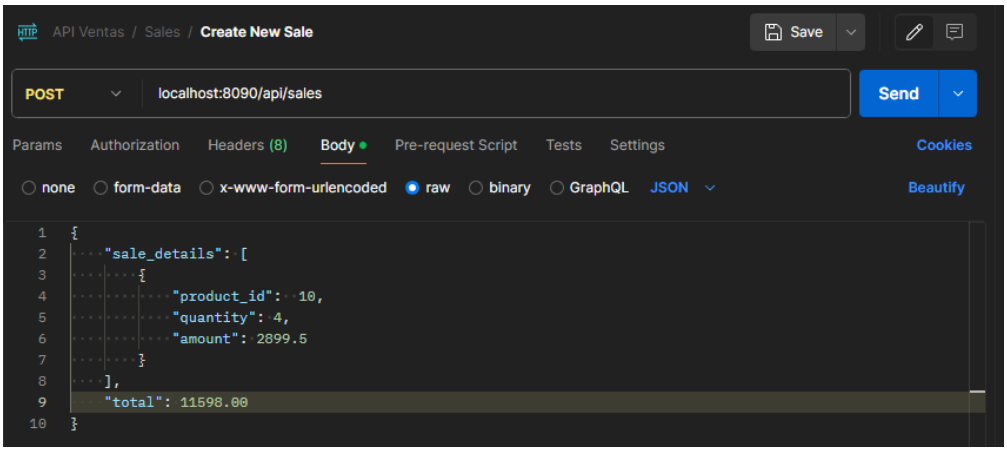


Figura 5-77 Solicitud para realizar venta.

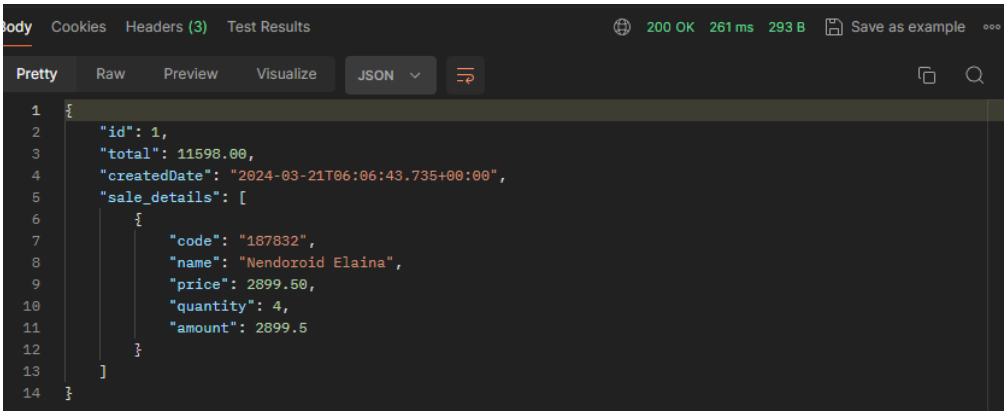


Figura 5-78 Respuesta del servicio a la solicitud correcta.

Como se puede apreciar en la imagen anterior, al enviar la solicitud el sistema validará los campos para luego registrar la venta, posterior a esto el sistema publicará un evento de tipo venta creada.

Este evento es consumido finalmente por nuestro microservicio de análisis de datos en Python, en donde será procesado y almacenado para poder trabajar con el más adelante al realizar peticiones de análisis a este sistema.

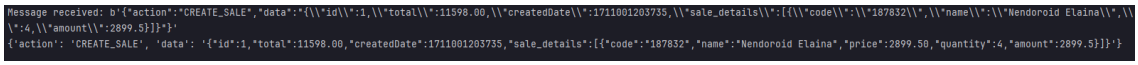


Figura 5-79 Notificación en el microservicio de análisis de datos sobre el evento recibido y procesado.

Una vez que el microservicio de análisis de datos cuenta con información, e posible comenzar a generar consultas de análisis de datos.

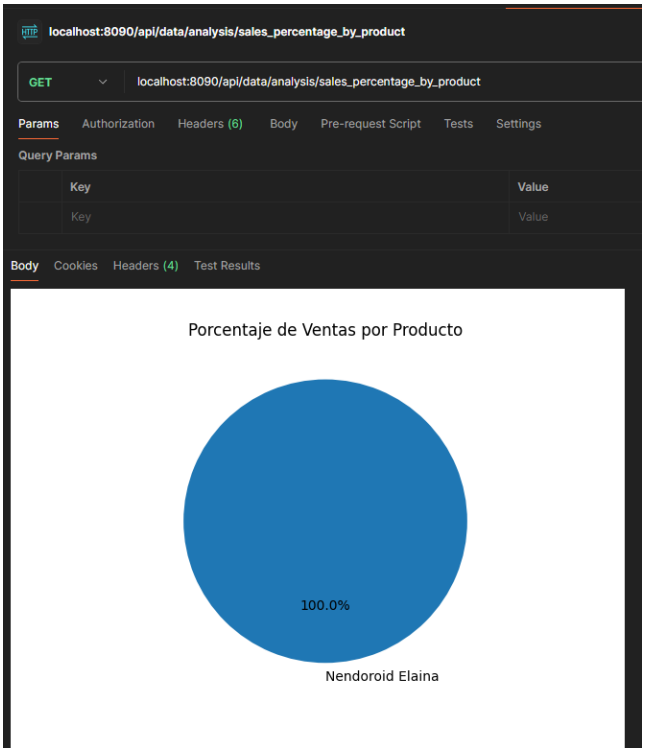


Figura 5-80 Consulta de análisis de datos para obtener gráfica sobre el porcentaje de venta por productos.

En este caso, al solo tener una venta registrada el grafico completo corresponde solo a ese producto.

Al agregar más ventas, nuestro microservicio de análisis de datos recibe, almacena y procesa la información de forma inmediata, logrando de esta manera obtener este gráfico tras un par de inserciones.

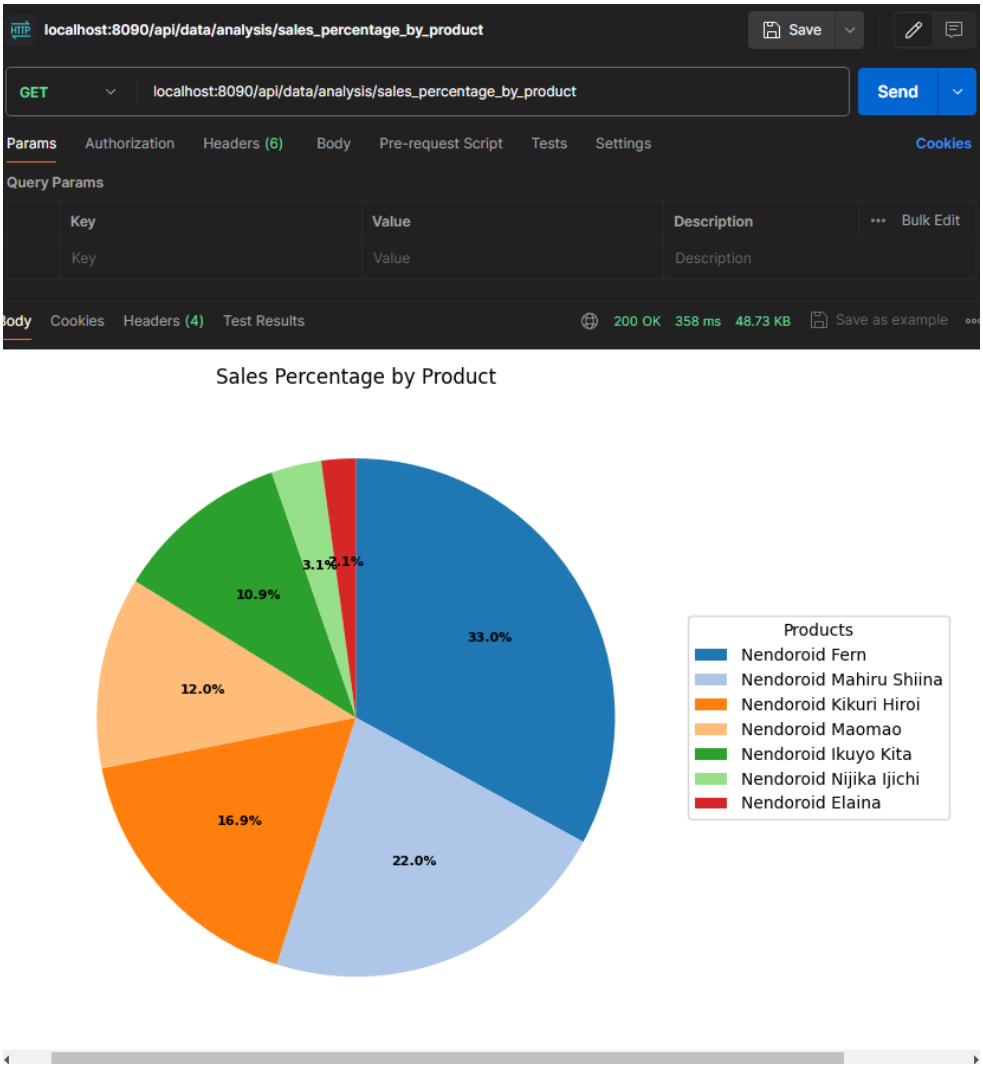


Figura 5-81 Respuesta a solicitud de análisis de productos.

6. Conclusiones

A lo largo del desarrollo de este trabajo de tesis se ha recorrido un camino extenso, en el que en primera instancia se analizaron las arquitecturas y como estas han ido evolucionando y adaptándose para cumplir con los requisitos de negocio e incluir nuevas tecnologías, también se exploraron los fundamentos de las arquitecturas de software, partiendo desde su definición hasta sus elementos y conceptos relacionados como patrones de diseño, de arquitectura y estilos de arquitectura.

La arquitectura propuesta fue diseñada con el objetivo de ofrecer una fuerte flexibilidad en conjunto de una escalabilidad eficiente y una comunicación entre microservicios rápida y eficiente. Todo esto con el objetivo de crear aplicaciones que puedan adaptarse al contexto actual, en el que analizar los datos lo más rápido posible es algo esencial.

Por otra parte, la implementación de ejemplo presentada en el capítulo final fue realizada para servir como prueba concreta sobre como la combinación de conceptos, métodos, tecnologías y diseños fueron aplicados alineándose a los principios que de la propuesta.

En conclusión, este trabajo de tesis busca subrayar la importancia de entender la evolución de las arquitecturas, así como sus fundamentos para poder diseñar y brindar soluciones que respondan eficazmente a las necesidades actuales del campo empresarial. En este caso específico, la arquitectura basada en microservicios que se propuso se destaca como una opción viable para abordar retos en sistemas que tengan necesidades altas en escalabilidad y análisis de datos.

Espero que este trabajo funcione como base para futuras investigaciones y desarrollo dentro del campo de la arquitectura de software para la creación de sistemas sostenibles, altamente escalables, eficaces y preparados para adaptarse al futuro.

7. Bibliografía

[1] Varanasi, B., & Bartkov, M. (2021). Spring REST: Building java microservices and cloud applications (2a ed.). APress.

[2] Arquitectura de Monolítico. (s/f). Reactiveprogramming.io. Recuperado el 21 de diciembre de 2023, de https://reactiveprogramming.io/blog/es/estilos-arquitectonicos/monolitico

[3] Arquitectura de software: ¿Qué es y qué tipos hay? (s/f). Gluo. Recuperado el 21 de diciembre de 2023, de https://www.gluo.mx/blog/arquitectura-de-software-que-es-y-que-tipos-hay

[4] Arquitectura en Capas. (s/f). Reactiveprogramming.io. Recuperado el 21 de diciembre de 2023, de https://reactiveprogramming.io/blog/es/estilos-arquitectonicos/capas

[5] Arquitectura monolítica vs arquitectura de microservicios: ¿cuál debo elegir? (s/f). Click-it.es. Recuperado el 21 de diciembre de 2023, de https://click-it.es/arquitectura-monolitica-vs-arquitectura-de-microservicios-cual-debo-elegir/

[6] Bass, L., Clements, P., & Kazman, R. (2022). Software architecture in practice (4a ed.). Addison Wesley.

[7] Command. (s/f). Refactoring.guru. Recuperado el 1 de enero de 2024, de https://refactoring.guru/es/design-patterns/command

[8] Datadog. (2021, junio 28). Recuperado el 5 de enero de 2024, de Serverless architecture: What it is & how it works. Datadog. https://www.datadoghq.com/knowledge-center/serverless-architecture/

[9] Diseño de la arquitectura de aplicaciones basadas en contenedores y microservicios. (s/f). Microsoft.com. Recuperado el 28 de enero de 2024, de https://learn.microsoft.com/es-es/dotnet/architecture/microservices/architect-microservice-container-applications/

[10] Estilo de arquitectura basada en eventos. (s/f). Microsoft.com. Recuperado el 28 de enero de 2024, de https://learn.microsoft.com/es-es/azure/architecture/guide/architecture-styles/event-driven

[11] Estilo de arquitectura de microservicios. (s/f-a). Microsoft.com. Recuperado el 26 de enero de 2024, de

https://learn.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices

[12] Estilo de arquitectura de microservicios. (s/f-b). Microsoft.com. Recuperado el 28 de enero de 2024, de https://learn.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices

[13] Fowler, M. (2017). Patterns of Enterprise Application Architecture. Createspace Independent Publishing Platform.

[14] IBM Documentation. (2023a, marzo 24). Recuperado el 15 de enero de 2024, de Ibm.com. https://www.ibm.com/docs/es/aix/7.3?topic=processes-modularity

[15] IBM Documentation. (2023b, abril 11). Recuperado el 15 de enero de 2024 Ibm.com. https://www.ibm.com/docs/es/i/7.5?topic=availability-application-resilience

[16] IBM Documentation. (2023c, octubre 10). Recuperado el 15 de enero de 2024 Ibm.com. https://www.ibm.com/docs/es/i/7.5?topic=security-separation-duties

[17] IT Solutions de BETWEEN. (s/f). Serverless: qué es y qué ventajas tiene. Between.tech. Recuperado el 13 de febrero de 2024, de https://impulsate.between.tech/serverless-que-es-ventajas

[18] Malandrino, P.-J. (2023, noviembre 28). Evolution of software architecture: From monoliths to microservices and beyond. Dzone.com; DZone. https://dzone.com/articles/evolution-of-software-architecture-from-monoliths

[19] Oleksii. (2021, octubre 22). Recuperado el 15 de enero de 2024, Monolithic architecture. Advantages and disadvantages. Datamify. https://datamify.com/architecture/how-to-understand-monolithic-architecture/

[20] Patrón CQRS. (s/f). Microsoft.com. Recuperado el 29 de febrero de 2024, de https://learn.microsoft.com/es-es/azure/architecture/patterns/cqrs

[21] Patrón de disyuntor. (s/f). Microsoft.com. Recuperado el 29 de febrero de 2024, de https://learn.microsoft.com/es-es/azure/architecture/patterns/circuit-breaker

[22] Patrón Gateway Routing. (s/f). Microsoft.com. Recuperado el 29 de febrero de 2024, de https://learn.microsoft.com/es-es/azure/architecture/patterns/gateway-routing

[23] Patrones de comportamiento. (s/f). Refactoring.guru. Recuperado el 17 de marzo de 2024, de https://refactoring.guru/es/design-patterns/behavioral-patterns

[24] Patrones de diseño. (s/f). Refactoring.guru. Recuperado el 22 de febrero de 2024, de https://refactoring.guru/es/design-patterns

[25] ¿Qué es Apache Kafka? (s/f). Redhat.com. Recuperado el 1 de marzo de 2024, de https://www.redhat.com/es/topics/integration/what-is-apache-kafka

[26] ¿Qué es Docker y cómo funciona? (s/f). Redhat.com. Recuperado el 1 de marzo de 2024, de https://www.redhat.com/es/topics/containers/what-is-docker

[27] ¿Qué es Java Spring Boot? (s/f). Ibm.com. Recuperado el 1 de marzo de 2024, de https://www.ibm.com/es-es/topics/java-spring-boot

[28] ¿Qué es la arquitectura basada en eventos? (s/f). Redhat.com. Recuperado el 21 de febrero de 2024, de https://www.redhat.com/es/topics/integration/what-is-event-driven-architecture

[29] ¿Qué es la arquitectura orientada a los servicios (SOA)? (s/f). Redhat.com. Recuperado el 21 de febrero de 2024, de https://www.redhat.com/es/topics/cloud-native-apps/what-is-service-oriented-architecture

[30] Qué es la arquitectura sin servidor. (s/f). Google Cloud. Recuperado el 17 de marzo de 2024, de https://cloud.google.com/discover/what-is-serverless-architecture?hl=es

[31] ¿Qué es la escalabilidad? (s/f). Microsoft.com. Recuperado el 26 de febrero de 2024, de https://learn.microsoft.com/es-es/biztalk/core/what-is-scalability

[32] ¿Qué es una arquitectura de aplicaciones? (s/f). Redhat.com. Recuperado el 21 de enero de 2024, de https://www.redhat.com/es/topics/cloud-native-apps/what-is-an-application-architecture

[33] Rick-Anderson. (s/f). Crear objetos de transferencia de datos (DTO). Microsoft.com. Recuperado el 7 de marzo de 2024, de https://learn.microsoft.com/es-es/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-5

[34] Strategy. (s/f). Refactoring.guru. Recuperado el 15 de marzo de 2024, de https://refactoring.guru/es/design-patterns/strategy

[35] Jkpelaez, A. (2009, 30 mayo). Arquitectura basada en capas. Recuperado el 12 de marzo de 2024, – Blog de Juan Peláez en Geeks.ms. https://geeks.ms/jkpelaez/2009/05/30/arquitectura-basada-en-capas/

[36] Poblete, H. (2022, 7 enero). Recuperado el 20 de enero de 2024, Arquitectura basada en servicios — El sistema distribuido más simple. Medium. https://hectorpoblete.medium.com/arquitectura-basada-en-servicios-el-sistema-distribuido-m%C3%A1s-simple-626eb333f279

[37] ¿Qué es la EDA? - Explicación sobre la arquitectura basada en eventos Recuperado el 20 de enero de 2024, - AWS. (s. f.). Amazon Web Services, Inc. https://aws.amazon.com/es/what-is/eda/

[38] Zapata, M. (2019, diciembre 26). Recuperado el 17 de enero de 2024, ¿Qué es arquitectura de software? Manuel Zapata. https://manuelzapata.co/que-es-arquitectura-de-software/