Henry Gabriel Peralta Martinez - 201712289

Tarea 2

Justificación del Rediseño

El sistema de compras en línea fue inicialmente diseñado como una aplicación monolítica, un enfoque común en etapas tempranas del desarrollo por su simplicidad. Sin embargo, conforme el sistema fue creciendo en funcionalidades, usuarios y complejidad, este diseño comenzó a mostrar limitaciones significativas a nivel técnico y operativo.
En una arquitectura monolítica, todas las funcionalidades —como la gestión de usuarios, productos, pagos, pedidos, reseñas, devoluciones y más— se encuentran agrupadas en un solo bloque de código, que se ejecuta como una unidad. Esto implica que todos los componentes dependen entre sí, comparten la misma base de datos y deben desplegarse juntos, lo que acarrea varios problemas:

- Escalabilidad limitada: si una sola parte del sistema experimenta una alta demanda, se debe escalar toda la aplicación, consumiendo más recursos de los necesarios.

- Despliegue y mantenimiento complejos: cualquier cambio, incluso menor, requiere reconstruir y desplegar la aplicación entera, lo que puede causar interrupciones y aumenta el riesgo de errores.

- Acoplamiento entre módulos: el código está fuertemente interconectado; un fallo en un componente puede propagarse a otros, afectando la estabilidad general del sistema.

- Falta de flexibilidad tecnológica: al estar todo unificado, es difícil usar tecnologías distintas para diferentes necesidades.

- Baja productividad del equipo: a medida que el código crece, se vuelve más difícil entenderlo, probarlo y desarrollarlo en paralelo por diferentes equipos.

Para solventar estas limitaciones, se decidió rediseñar el sistema adoptando una arquitectura basada en microservicios. Esta se basa en dividir la aplicación en varios servicios pequeños, autónomos y bien definidos, donde cada uno se encarga de una única responsabilidad del dominio.

Entre los principales beneficios del rediseño destacan:

- Escalabilidad horizontal: cada microservicio puede escalarse de forma independiente según su carga, lo que optimiza el uso de recursos.

- Despliegue independiente: se pueden actualizar, reiniciar o reemplazar servicios sin afectar al resto del sistema.

- Alta resiliencia: un fallo en un servicio no detiene el funcionamiento de otros.

- Modularidad y mantenibilidad: los equipos pueden trabajar en paralelo sobre diferentes servicios, facilitando el desarrollo ágil y continuo.

- Mejor alineación con DevOps y CI/CD: cada servicio puede tener su propio flujo de integración y despliegue, favoreciendo la automatización.

Además, esta arquitectura permite el uso de herramientas modernas como contenedores Docker, orquestación con docker-compose, colas de mensajes, y API Gateways, lo que convierte al sistema en una solución más robusta, escalable y preparada para producción en entornos distribuidos.

Descripción de los Servicios

A partir del análisis funcional del sistema monolítico y el diagrama de casos de uso, se identificaron los dominios clave del negocio. Cada uno fue transformado en un microservicio independiente, aplicando el principio de separación de responsabilidades y evitando el acoplamiento entre servicios.

los principales microservicios del sistema de compras en línea:

- User Service: Este servicio se encarga de la gestión de usuarios, incluyendo el registro, inicio de sesión, autenticación con JWT y administración de roles. Es responsable de asegurar que cada solicitud que acceda al sistema esté autenticada y autorizada correctamente. Utiliza una base de datos PostgreSQL para almacenar la información de los usuarios.

- Product Service: Administra el catálogo de productos. Permite realizar operaciones CRUD sobre productos, incluyendo información como nombre, descripción, precio, disponibilidad y categoría. Para una mayor flexibilidad y rendimiento en consultas de catálogo, utiliza MongoDB como sistema de almacenamiento.

- Cart Service: Gestiona los carritos de compras individuales de los usuarios. Permite agregar productos, modificar cantidades y eliminar artículos del carrito. Debido a su naturaleza altamente volátil y necesidad de rapidez, utiliza Redis como base de datos en memoria para ofrecer tiempos de respuesta óptimos.

- Order Service: Se encarga de procesar los pedidos que realiza un usuario al finalizar su compra. Valida el stock, calcula totales, genera la orden y la almacena en una base de datos PostgreSQL. También interactúa con los servicios de pago y envío.

- Payment Service: Procesa los pagos realizados por los usuarios mediante integración con pasarelas externas. Administra cobros y solicitudes de reembolso en coordinación con el servicio de devoluciones. Por razones de seguridad, no almacena datos financieros sensibles localmente.

- Shipping Service: Organiza y realiza el seguimiento de los envíos. Gestiona la información relacionada con direcciones de entrega, transportistas y estados de envío. Puede integrarse con APIs de empresas de logística.

- Review Service: Permite a los usuarios registrar calificaciones y reseñas de los productos que han adquirido. También gestiona la visualización de las opiniones para otros compradores. Se utiliza MongoDB por su flexibilidad para almacenar estructuras variadas como comentarios, puntuaciones y metadatos.

- Return & Refund Service: Gestiona las solicitudes de devolución de productos y coordina los reembolsos. Este servicio se comunica asincrónicamente con otros servicios mediante RabbitMQ, lo cual permite manejar flujos de eventos como producto devuelto, reembolso aprobado y actualización de stock.

- Analytics Service: Procesa datos provenientes de diferentes eventos del sistema y genera estadísticas útiles para el negocio. Se utiliza Kafka para recibir y procesar eventos en tiempo real, permitiendo generar dashboards y reportes de ventas.

- Supplier Service: Se encarga de la administración de proveedores y de la reposición de inventario. Gestiona la información de proveedores, productos abastecidos y órdenes de compra. Se utiliza PostgreSQL y se comunica con el servicio de productos para actualizar el stock.

Comunicación entre Servicios

- Los servicios se comunican principalmente mediante HTTP REST para operaciones sincrónicas.

- Para tareas asincrónicas o desacopladas como devoluciones, notificaciones y eventos de negocio, se utilizan RabbitMQ cola de mensajes y Kafka eventos distribuidos.

- Un API Gateway centraliza las solicitudes externas, redirige a los servicios correspondientes, y se encarga de la autenticación, balanceo de carga y rutas.

Autonomía y Despliegue

Cada servicio cuenta con:

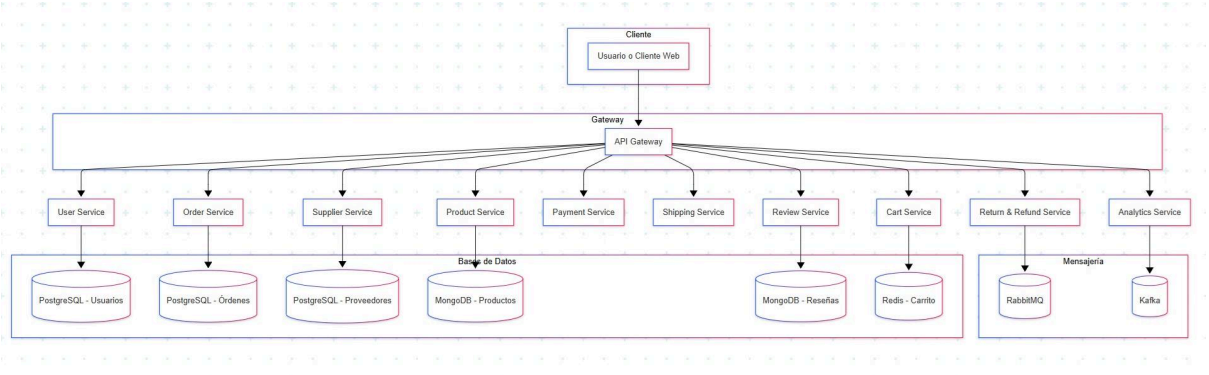
- Su propia base de datos.

- Su imagen Docker, con posibilidad de ejecutarse y escalar de forma independiente.

- Su endpoint y documentación de API para interacción con otros servicios o con el cliente.

Diagrama de Arquitectura

La arquitectura del sistema se organiza alrededor de un API Gateway, que actúa como punto de entrada centralizado para las solicitudes externas. Este gateway enruta las solicitudes a los diferentes microservicios según el dominio funcional al que pertenezcan. Los servicios se comunican entre sí utilizando tanto HTTP REST como mecanismos de mensajería como RabbitMQ y Kafka.
Cada microservicio es autónomo, tiene su propia base de datos y puede ser desplegado y escalado de forma independiente utilizando Docker. A continuación, se presenta el diagrama general de la arquitectura:



Lecciones Aprendidas y Desafíos Enfrentados

Lecciones Aprendidas

- Diseño modular y desacoplado: La transición a microservicios permitió comprender mejor la importancia de diseñar sistemas divididos en componentes bien definidos, facilitando el mantenimiento, escalabilidad y evolución de cada parte del sistema.

- Mensajería asincrónica como herramienta clave: El uso de RabbitMQ y Kafka demostró ser fundamental para implementar procesos que no requieren respuesta inmediata, reduciendo la dependencia entre servicios y mejorando la capacidad de respuesta general.

- Contenerización y despliegue consistente: Docker y docker-compose facilitaron el despliegue uniforme de todos los servicios, permitiendo trabajar en entornos aislados pero conectados, lo cual agiliza el desarrollo y las pruebas.

Desafíos Enfrentados

- Consistencia de datos entre servicios: Uno de los principales retos fue garantizar que los datos se mantuvieran coherentes al distribuir la lógica de negocio entre distintos servicios, especialmente en operaciones críticas como pagos y pedidos.

- Monitoreo y trazabilidad en sistemas distribuidos: En una arquitectura de microservicios, detectar errores y seguir el flujo completo de una operación se vuelve más complejo. Esto hizo evidente la necesidad de herramientas de monitoreo y

logging centralizado.

- Diseño de interfaces API claras y estables: Fue necesario dedicar tiempo al diseño de contratos entre servicios, asegurando que las APIs fueran intuitivas, reutilizables y resistentes a cambios.