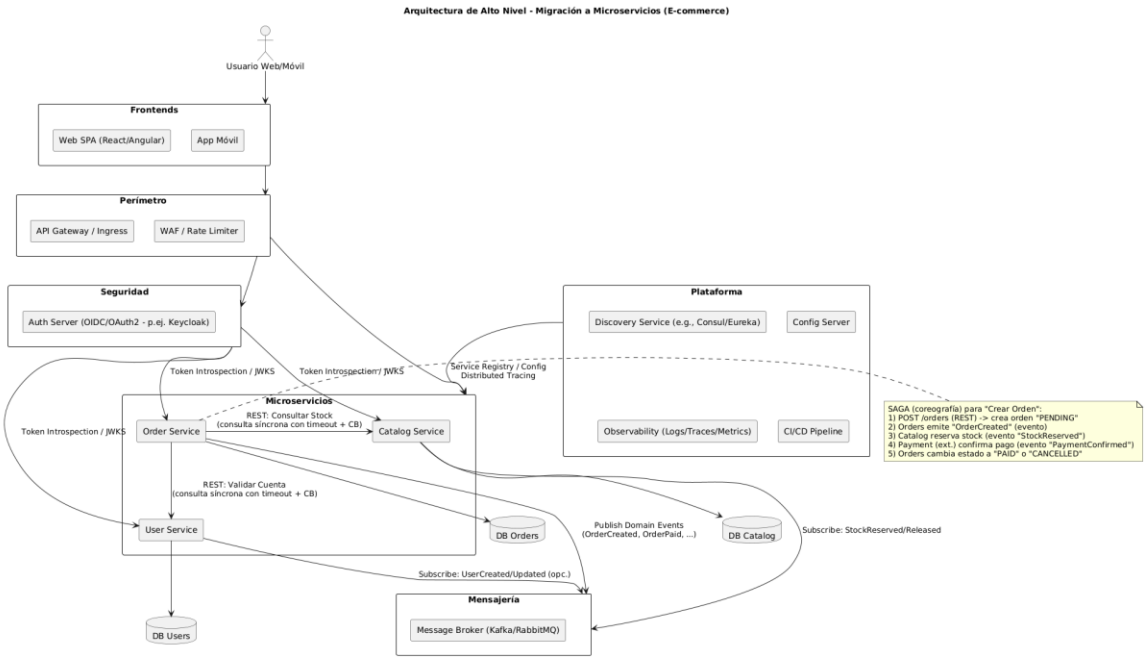
Decisión de Arquitectura – Migración de Monolito Java a Microservicios

1. Contexto y Objetivo

La empresa posee una aplicación monolítica en Java que concentra su catálogo de productos, cuentas de usuario y órdenes. Se propone una arquitectura de alto nivel basada en microservicios para mejorar escalabilidad, desacoplamiento y velocidad de despliegue, preservando la continuidad operativa con una estrategia de modernización progresiva.

1.2 Diagrama de Arquitectura



arquitectura_ecommerce_microservicios.puml

2. Identificación de Servicios

• Catalog Service: Gestiona productos, categorías, inventario y precios. Expone APIs para consulta de catálogo y operaciones de reserva y/o liberación de stock.

• User Service: Administra cuentas, perfiles y direcciones. Integra autenticación/autorización vía OIDC/OAuth2. Proceso de seguridad que utiliza los protocolos OAuth 2.0 para la autorización y OpenID Connect (OIDC) para la autenticación,

permitiendo que un usuario conceda a una aplicación acceso limitado a sus recursos sin compartir sus credenciales. Este proceso implica que el servidor de autorización verifica la identidad del usuario y le otorga un token de acceso para que la aplicación pueda interactuar con los servicios protegidos en su nombre

• Order Service: Orquesta el ciclo de vida de la orden (creación, validación, reserva de stock, pago, facturación) mediante eventos de dominio y SAGA. Los eventos de dominio son notificaciones sobre algo significativo que ya ocurrió en un sistema, encapsulan estos hechos inmutables y son útiles para desacoplar componentes y activar lógica de negocio adicional. El patrón SAGA es una forma de orquestar y coordinar procesos distribuidos complejos utilizando estos eventos, permitiendo la gestión de transacciones distribuidas, la compensación de fallos y el mantenimiento de la consistencia del sistema mediante coreografía o orquestación.

3. Patrones de Comunicación – Caso de uso: Crear una nueva orden

Entrada del cliente → API Gateway → Order Service (REST síncrono).

Dentro del dominio: preferencia por comunicación asíncrona vía broker de mensajes para pasos de larga duración (reserva de stock, confirmación de pago), con eventos de dominio (coreografía SAGA).

Justificación: la creación de la orden no debe bloquear por integraciones lentas o no disponibles; se mejora resiliencia y desacoplamiento. Para consultas breves de consistencia inmediata (p.ej., validar cuenta/stock puntual) se permiten llamadas REST con timeout, reintento y Circuit Breaker.

Trade-offs: asíncrono añade complejidad (eventual consistencia, idependencia, trazabilidad) que se mitiga con outbox, esquemas de eventos versionados, y trazas distribuidas.

4. Estrategia de Datos

Patrón: Base de datos por servicio.

• Catalog Service → DB Catalog (relacional).

• User Service → DB Users (relacional).

• Order Service → DB Orders (relacional).

Motivación: autonomía por servicio, aislamiento de cambios y escalado independiente. Consistencia entre dominios: eventual, lograda mediante eventos; Querys de lectura compuestas a través de vistas materializadas o proyecciones (CQRS) Segregación de la Responsabilidad de Comandos y Consultas es un patrón de diseño de software que separa los modelos y la lógica de lectura (consultas) de los de escritura (comandos) en una aplicación

5. Componentes Transversales

Como arquitecta de software, en la nueva arquitectura basada en microservicios gestionaría la autenticación y autorización bajo un modelo centralizado, estandarizado y desacoplado. Con los Siguientes componentes necesarios

• API Gateway / Ingress: routing, rate limiting, SSL/TLS, validación de JWT, canary/blue-green.

• Servidor de Autorización (OIDC/OAuth2, p.ej., Keycloak): emisión de tokens (JWT), flujos Authorization Code/Client Credentials.

• Service Discovery: registro/detección de instancias (Eureka/Consul) y balanceo.

• Config Server: configuración centralizada por entorno.

• Observabilidad: logs centralizados (ELK/EFK), métricas (Prometheus), trazas (OpenTelemetry + Jaeger/Tempo), tableros (Grafana).

• Resiliencia: Circuit Breaker, Bulkhead, Timeouts, Retries con backoff, Rate limiter.

• Seguridad: políticas de secretos (Vault/KMS), SBOM, escaneo SCA/SAST/DAST, firma de contenedores.

• Entorno: contenedores (Docker) + orquestación (Kubernetes), autoscaling horizontal, HPA, PodDisruptionBudgets.

• Mensajería: Kafka/RabbitMQ + patrón Outbox para entrega exacta-al-menos-una y orden dentro de partición.

6. Bounded Contexts (DDD) → Límites de Microservicios

• Catálogo (Productos/Inventario) → Catalog Service.

• Cuentas/Identidad → User Service.

• Órdenes → Order Service.

7. Preguntas de contexto

7.2 Atributos de Calidad (Requisitos No Funcionales):

Son propiedades del sistema como rendimiento, disponibilidad, seguridad, mantenibilidad, etc. Ejemplo de conflicto: rendimiento vs. seguridad. Cifrar y validar más capas puede aumentar latencia; se equilibra con caché, TLS acelerado y offloading en el gateway. Otro conflicto común: consistencia fuerte vs. disponibilidad en presencia de particiones.

7.3 Teorema CAP:

En presencia de una partición de red, un sistema distribuido debe elegir entre Consistencia o Disponibilidad. Las bases CP priorizan consistencia (p.ej., consenso), mientras las AP priorizan disponibilidad y aceptan consistencia eventual. La elección de base de datos y

topología debe alinearse con los requisitos de negocio y latencia inter-zonas. El teorema de CAP (Consistencia, Disponibilidad, Tolerancia a particiones) influye de forma directa en la selección de la base de datos para un sistemadistribuido, porque define que atributos de caidad se pueden garantizar cuando ocurran fallos de red (particiones)

7.4 Patrón de Resiliencia - Circuit Breaker:

Coloca un interruptor entre consumidor y proveedor; al detectar fallos repetidos abre el circuito y evita llamadas, previniendo fallos en cascada y saturación de recursos en arquitecturas de microservicios, evitando que:

- Se bloqueen recursos de los consumidores (threads, conexiones, CPU).
- Se degrade todo el sistema por un solo punto de fallo.
- Se genere tormenta de reintentos que agrave la situación.

7.5 Coreografía vs. Orquestación:

Coreografía: cada servicio reacciona a eventos sin coordinador central; pro: bajo acoplamiento, escalable; contra: mayor complejidad de seguimiento y depuración. Orquestación: un coordinador (p.ej., Order Service o un workflow engine) dirige pasos; pro: visibilidad y control; contra: riesgo de centralización y acoplamiento.

Aspecto	**Coreografía**	**Orquestación**
Acoplamiento	Bajo entre servicios (solo contratos de eventos)	Mayor acoplamiento lógico al coordinador
Visibilidad / trazabilidad	Difícil (necesita trazas distribuidas y correlación)	Mejor visibilidad (el orquestador conoce el flujo)
Control secuencial / compensaciones complejas	Complicado de garantizar	Sencillo de expresar y controlar
Escalabilidad	Muy escalable (workload distribuido)	Orquestador puede ser cuello de botella si no se diseña bien
Simplicidad conceptual	Simpler para flujos simples	Más simple para flujos complejos y long-running
Tolerancia a fallos	Resiliente por diseño si se manejan idempotencia/outbox	Recuperable si el orquestador persiste estado (durable)
Testing	Testing end-to-end más complejo	Más fácil de testear el flujo en el orquestador

Aspecto	Coreografía	Orquestación
Latencia	Menor emparejado (asíncrono) pero eventual consistencia	Puede ser más rápido para secuencias cortas (sin esperar eventos)

7.6 Patrón Strangler Fig:

Es una estrategia de modernización de aplicaciones heredadas que consiste en reemplazar gradualmente un sistema monolítico (legacy) con una arquitectura nueva (por ejemplo, microservicios), sin tener que hacer un "big bang" de migración.

Estrategia de modernización progresiva: se coloca un proxy/gateway delante del monolito y se desvían funcionalidades nuevas y/o extraídas a servicios modernos, estrangulando gradualmente el monolito hasta reemplazarlo, es ideal cuando:

- Tienes un monolito grande y crítico en producción que no puedes apagar de golpe.
- Quieres reducir riesgos de un cambio radical (vs. reescribir todo de cero).
- Necesitas entregar valor de negocio incremental mientras modernizas.
- Hay funcionalidades prioritarias o de alto cambio que conviene mover primero (ej. Catálogo en e-commerce).
- Quieres aplicar estrategia de coexistencia: monolito + microservicios conviviendo hasta completar la transición.

7.7 Gestión de Datos en Microservicios – Base de Datos por Servicio:

Principio: Cada servicio es dueño exclusivo de su esquema.
Desafíos: consistencia distribuida, duplicación de datos para lecturas, transacciones distribuidas.
Mitigación: eventos de dominio, outbox, SAGA, CQRS, idempotencia, versionado de esquemas y contratos.
Este enfoque garantiza autonomía de cada microservicio, reduce el acoplamiento y permite escalar y evolucionar servicios de manera independiente, aunque introduce retos de consistencia que deben gestionarse con patrones de integración de datos

7.8 Observabilidad – Tres Pilares:

Logs : detalle de eventos específicos.
Métricas: valores agregados y cuantificables para monitoreo/alertas.
Trazas: seguimiento de una solicitud a través de múltiples servicios.
Logs son útiles para diagnóstico puntual, pero no ofrecen visibilidad proactiva ni permiten detectar tendencias.
Métricas permiten alertar en tiempo real sobre anomalías (ej. latencia promedio, error rate, saturación de recursos).

Trazas son críticas en microservicios, donde una petición puede recorrer decenas de servicios, y los logs aislados no muestran el flujo extremo a extremo

7.9 Infraestructura como Código (IaC):

Iac: Definir y versionar infraestructura con código declarativo (Terraform, CloudFormation).

Beneficios: reproducibilidad, revisión por pares, automatización CI/CD, reducción de drift entre entornos y auditoría de cambios.

IaC es fundamental porque permite al arquitecto garantizar escalabilidad, seguridad y gobernanza en entornos híbridos y multinube, alineando a los equipos de desarrollo y operaciones bajo prácticas DevOps.

7.10 Service Discovery:

Mecanismo para resolver ubicaciones dinámicas de instancias de servicios.
Lado del cliente: el cliente consulta el registry y decide a qué instancia llamar.
Lado del servidor: un proxy/balanceador (p.ej., gateway/ingress) resuelve y enruta.

7.11 Bounded Context (Contexto Delimitado):

Un Bounded Context en el marco de Domain-Driven Design (DDD) es una frontera explícita dentro del dominio donde un modelo tiene un significado bien definido y consistente. Dentro de ese límite se establecen reglas de negocio, terminología y lógica que no se mezclan con otros contextos. Cada contexto delimitado puede tener su propio lenguaje ubicuo (ubiquitous language) que evita ambigüedades y mantiene la coherencia interna.

En una arquitectura de microservicios, el concepto de Bounded Context es la guía natural para definir los límites de un microservicio. Idealmente, cada microservicio se alinea con un Bounded Context y es responsable exclusivo de su propio modelo y de su base de datos. Por ejemplo, en un e-commerce:

- El Catálogo (productos, precios, inventario) es un Bounded Context → se implementa como el *Catalog Service*.
- Las Cuentas de usuario forman otro Bounded Context → se implementa como el *User Service*.
- Las Órdenes constituyen un tercer Bounded Context → se implementa como el *Order Service*.

Esta correspondencia permite que cada microservicio evolucione de forma independiente, con límites claros de responsabilidad, minimizando acoplamientos y facilitando escalabilidad, mantenibilidad y autonomía de los equipos.

8. Entregable:

1. Clonar el repositorio https://github.com/angelicadr/prueba-arquitecta-microservicios/
2. Archivo fuente diagrama arquitectura_ecommerce_microservicios.puml.
3. imágenes con los diagramas.
4. Documento con la decision_arquitectura.docx. y PDF