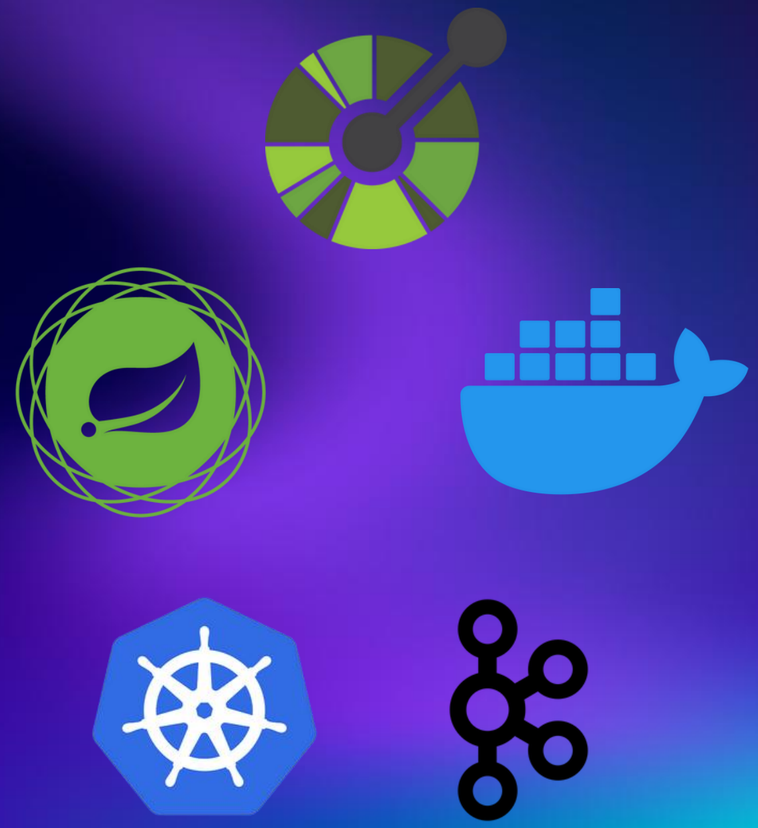
codearti

Curso
ARQUITECTURA DE MICROSERVICIOS CON SPRING BOOT Y KUBERNETES

Spring Boot, WebFlux, Docker, Open API, Kafka, Prometheus, Elastic Stack, Jaeger, K8s,...



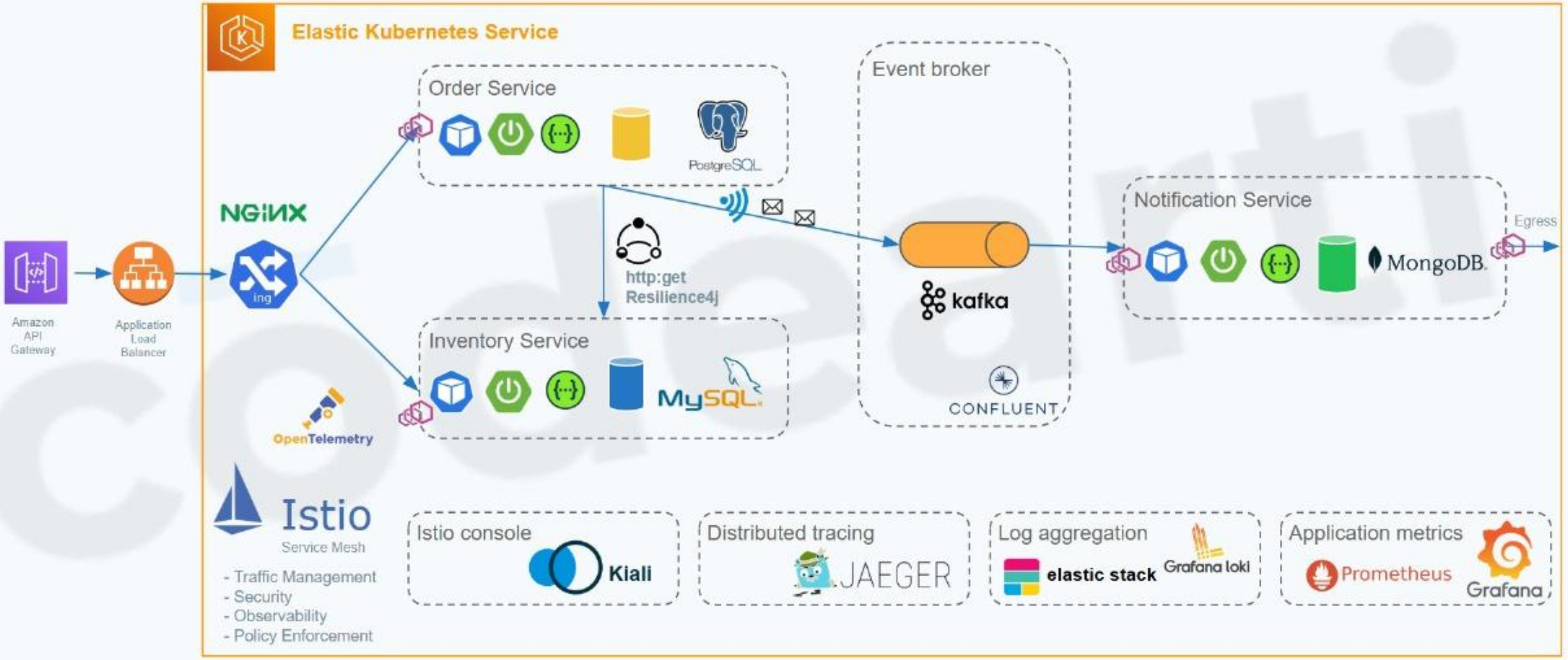
KUBERNETES (K8S)

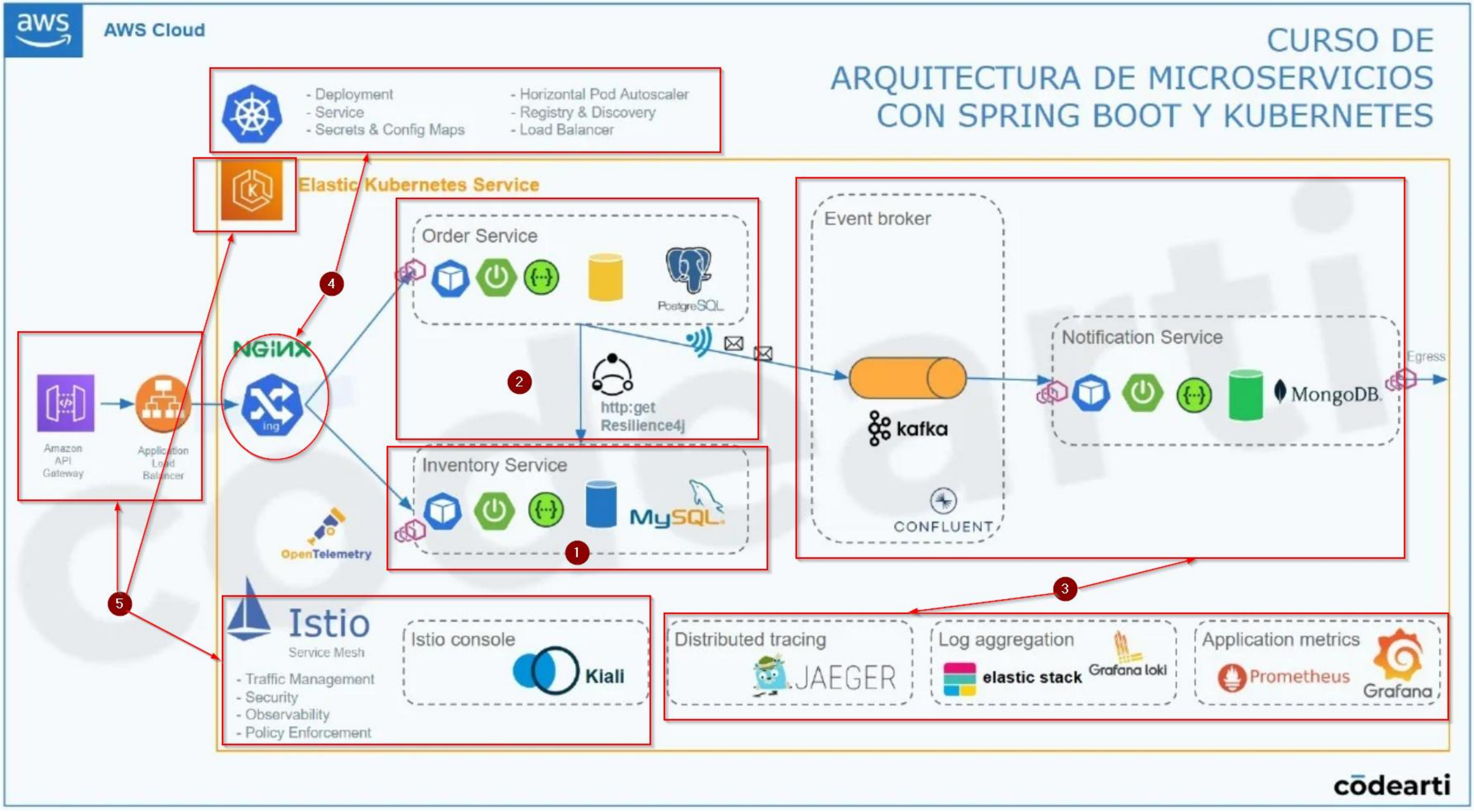
Stack Tecnológico

CURSO DE ARQUITECTURA DE MICROSERVICIOS CON SPRING BOOT Y KUBERNETES

AWS Cloud

- Deployment
- Service
- Secrets & Config Maps
- Horizontal Pod Autoscaler
- Registry & Discovery
- Load Balancer




AWS Cloud
CURSO DE
ARQUITECTURA DE MICROSERVICIOS
CON SPRING BOOT Y KUBERNETES
- Deployment
- Service
- Secrets & Config Maps
- Horizontal Pod Autoscaler
- Registry & Discovery
- Load Balancer
Elastic Kubernetes Service
Order Service
PostgreSQL
Event broker
Notification Service
MongoDB
Egress
NGINX
ing
Amazon API Gateway
Application Load Balancer
http:get
Resilience4j
Inventory Service
MySQL
kafka
CONFLUENT
OpenTelemetry
Istio
Service Mesh
- Traffic Management
- Security
- Observability
- Policy Enforcement
Istio console
Kiali
Distributed tracing
JAEGER
Log aggregation
elastic stack
Grafana loki
Application metrics
Prometheus
Grafana
1
2
3
4
5
codearti

Requisitos

- Editor IDE. ⇢ https://www.jetbrains.com/es-es/idea/download/?section=windows
- Configurado el JDK 21 + Paths de variable de entorno. ⇢ https://learn.microsoft.com/es-es/java/openjdk/download
- Postman. ⇢ https://www.postman.com/downloads/
- Docker Desktop (WSL 2) + Hyper-V. ⇢ https://docs.docker.com/desktop/install/windows-install/
- Muchas ganas de aprender.

¿Qué aprenderemos en esta sesión 1?

Introducción a la arquitectura de microservicios:

- Arquitectura de microservicios
- Principios, ventajas y desventajas
- Revisión de los patrones para microservicios
- Construcción del servicio de inventario (inventory-service)
- Despliegue de base de datos sobre docker (Mysql)

Sesión 1

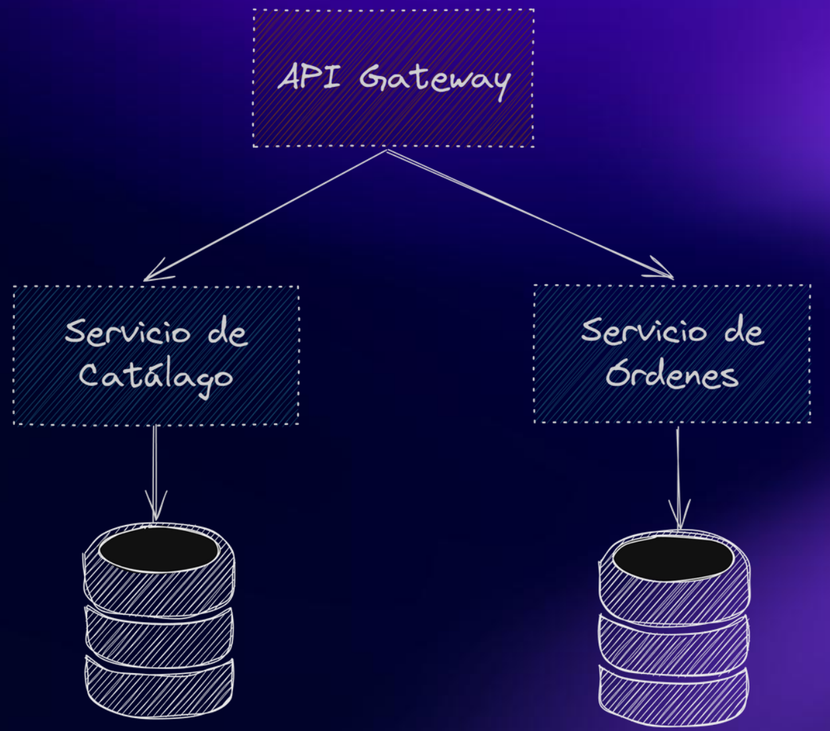
01

Arquitectura de Microservicios

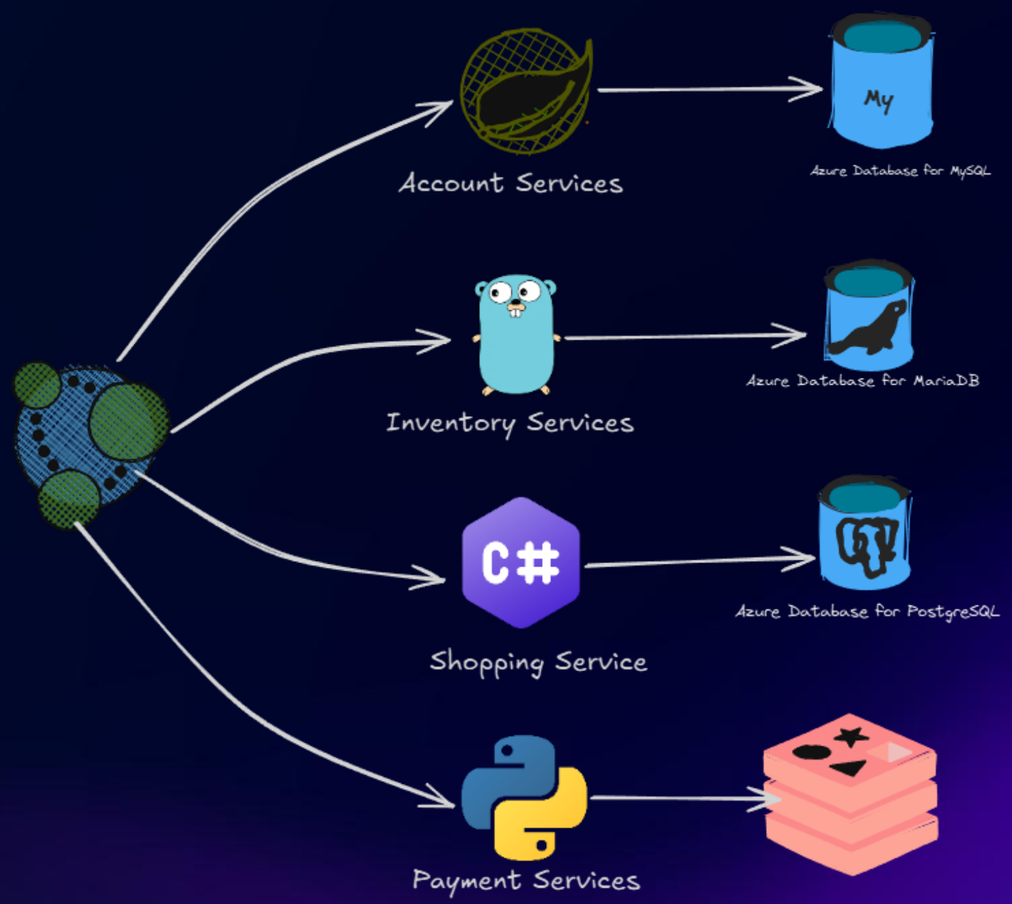
Servicio de E-Commerce

codearti

¿Aplicación monolítica vs aplicación en microservicios?



Arquitectura de Microservicios

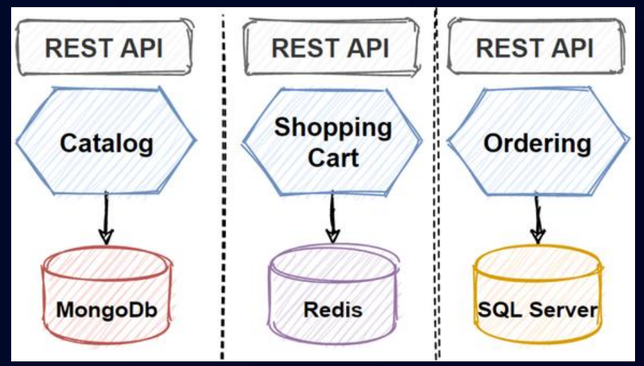


Caracteristicas

- Varios servicios de componentes.
- Muy fáciles de mantener y probar.
- Pertenecen a equipos pequeños.
- Se organizan en torno a capacidades empresariales.
- Infraestructura automatizada.

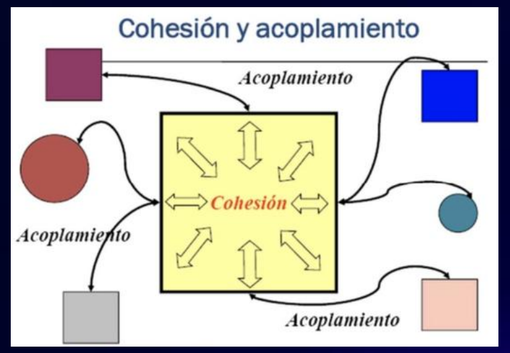
Principios

Independencia de servicios



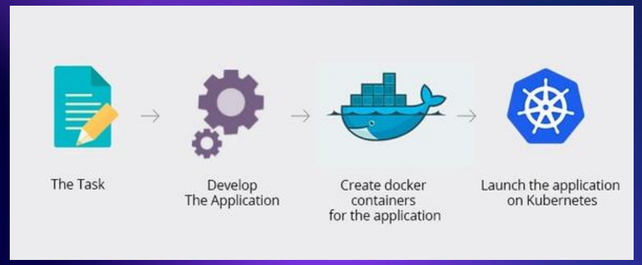
Cada microservicio es autónomo y se despliega independientemente.

Bajo acoplamiento y alta cohesión



Servicios que están lo menos acoplados posible entre sí y que tienen una responsabilidad clara y específica.

Despliegue independiente



Capacidad para actualizar, escalar o desplegar cada servicio sin afectar a otros.

Escalabilidad

Servicios pueden escalarse de manera individual según sus necesidades de carga.

Descentralización de datos

Cada servicio maneja su propia base de datos y modelo de datos, evitando una única base de datos compartida.

Ventajas

- Escalabilidad independiente
- Despliegues más rápidos
- Tolerancia a fallos
- Flexibilidad tecnológica
- Equipos pequeños y autónomos

Desventajas

- Complejidad en la gestión
- Comunicación entre servicios
- Pruebas integrales más complejas
- Gestión de datos
- Sobrecarga operativa

Patrones para Microservicios

Patrones de Descomposición

- Descomposición por Funcionalidad de Negocio
- Descomposición por Subdominio de Negocio (Domain-Driven Design - DDD)

Patrones de Gestión de Datos

- Base de Datos por Servicio
- Saga
- CQRS

Patrones de Comunicación entre Servicios

- API Gateway
- Event-Driven
- Outbox Pattern

Patrones de Resiliencia

- Circuit Breaker
- Retry
- Bulkhead
- Timeout

Patrones de Seguridad

- Access Token
- API Rate Limiting
- Service Mesh

Patrones Observabilidad

- Logging Centralizado
- Tracing Distribuido
- Health Check
- Metric Collection

Patrones de Infraestructura

- Service Discovery
- Ambassador (Proxy)

Patrones de Estabilidad

- Autoscaling
- Load Balancer

Patrones de Descomposición

Descomposición por Funcionalidad de Negocio

Dividir la aplicación en microservicios basados en capacidades o funcionalidades del negocio, asegurando que cada microservicio tenga un propósito específico y claro.

Descomposición por Subdominio de Negocio (Domain-Driven Design - DDD)

Descomponer la aplicación siguiendo los límites de subdominios definidos en el diseño del dominio, usando conceptos como Bounded Contexts.

Patrones de Gestión de Datos

Base de Datos por Servicio

Cada microservicio tiene su propia base de datos independiente para mantener la autonomía.

Saga

Coordinación de transacciones distribuidas utilizando eventos para manejar la consistencia eventual.

CQRS

Separar la parte de comandos (escritura) de la parte de consultas (lectura), optimizando cada lado para sus tareas específicas.

Patrones de Comunicación entre Servicios

API Gateway

Patrón que actúa como punto único de entrada para todas las solicitudes de clientes, gestionando el enrutamiento, la seguridad, el balanceo de carga y la transformación de peticiones hacia los microservicios correspondientes.

Event-Driven

Arquitectura donde los microservicios se comunican a través de eventos asincrónicos, permitiendo un desacoplamiento entre servicios y mejor escalabilidad, donde los componentes reaccionan a eventos publicados por otros servicios.

Patrones de Resiliencia

Circuit Breaker

Detener temporalmente las solicitudes a un servicio fallido para evitar la sobrecarga del sistema.

Retry

Reintentar operaciones fallidas después de un tiempo de espera, con o sin aumento exponencial del tiempo entre intentos.

Bulkhead

Aislar fallas en ciertas partes del sistema para evitar que afecten a otras áreas.

Timeout

Establecer límites de tiempo en las operaciones para evitar que bloqueen indefinidamente.

Rate Limit

Limita la cantidad de solicitudes / request de los servicios.

Patrones de Seguridad

Access Token

Uso de tokens de acceso (por ejemplo, JWT) para la autenticación y autorización.

API Rate Limiting

Controlar la cantidad de solicitudes permitidas por usuario para evitar abusos.

Service Mesh

Implementación de seguridad, enrutamiento, y observabilidad en la capa de red mediante herramientas como Istio o Linkerd.

Patrones de Observabilidad

Logging Centralizado

Centralizar los logs de todos los microservicios para monitorear y depurar.

Tracing Distribuido

Seguir una solicitud a lo largo de múltiples microservicios para entender el flujo y detectar problemas de rendimiento.

Health Check

Monitorizar la salud de los microservicios usando endpoints específicos.

Metric Collection

Recopilar métricas sobre rendimiento y uso de recursos para monitoreo.

Patrones de Escalabilidad

Autoscaling

Ajustar automáticamente el número de instancias de microservicios según la demanda.

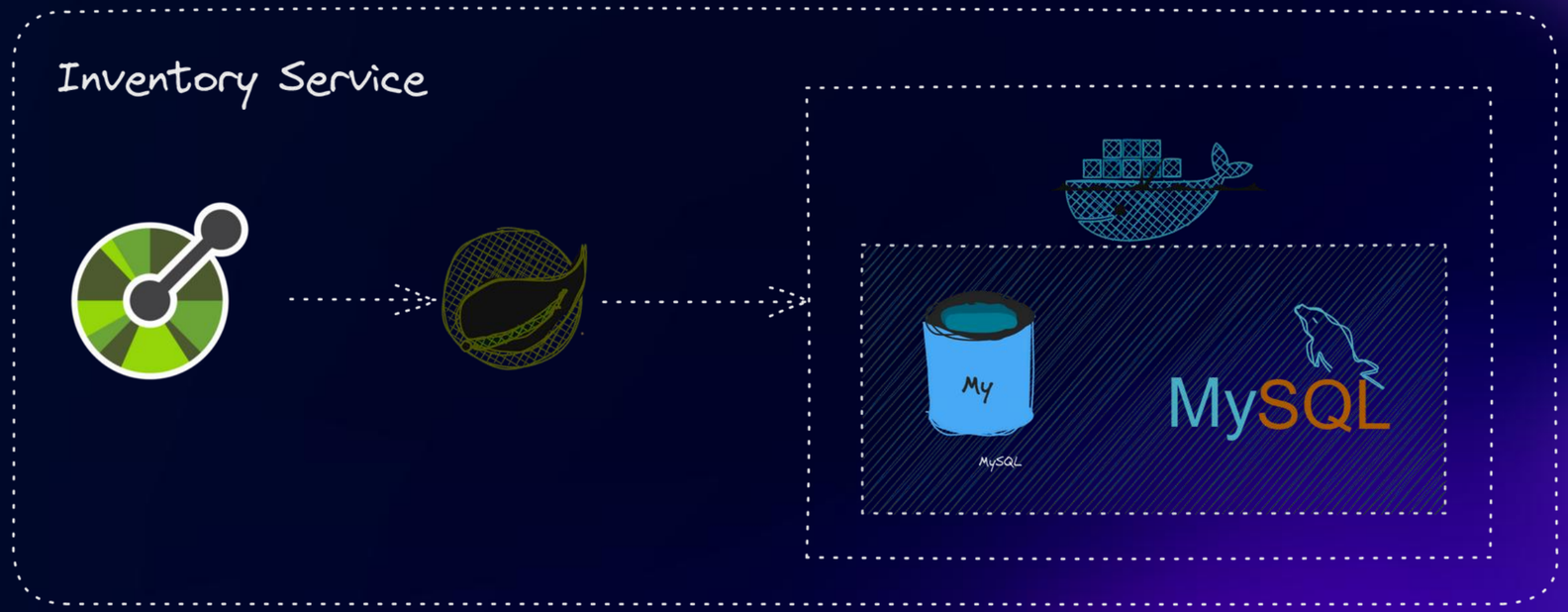
Load Balancer

Distribuir el tráfico de entrada entre múltiples instancias de un microservicio.

Implementación

Servicio de E-Commerce

API - Servicio de Inventario



- Open API
- Spring Boot + Spring WebFlux
- Docker
- MySql

Open API

- Es una especificación estándar para describir APIs RESTful
- Permite documentar endpoints, parámetros, respuestas y modelos de datos en formato JSON/YAML

Contract First

- Es un enfoque de diseño de APIs donde primero se define el contrato/interfaz
- El contrato se crea antes de implementar el código

https://openapi-generator.tech

Spring Boot

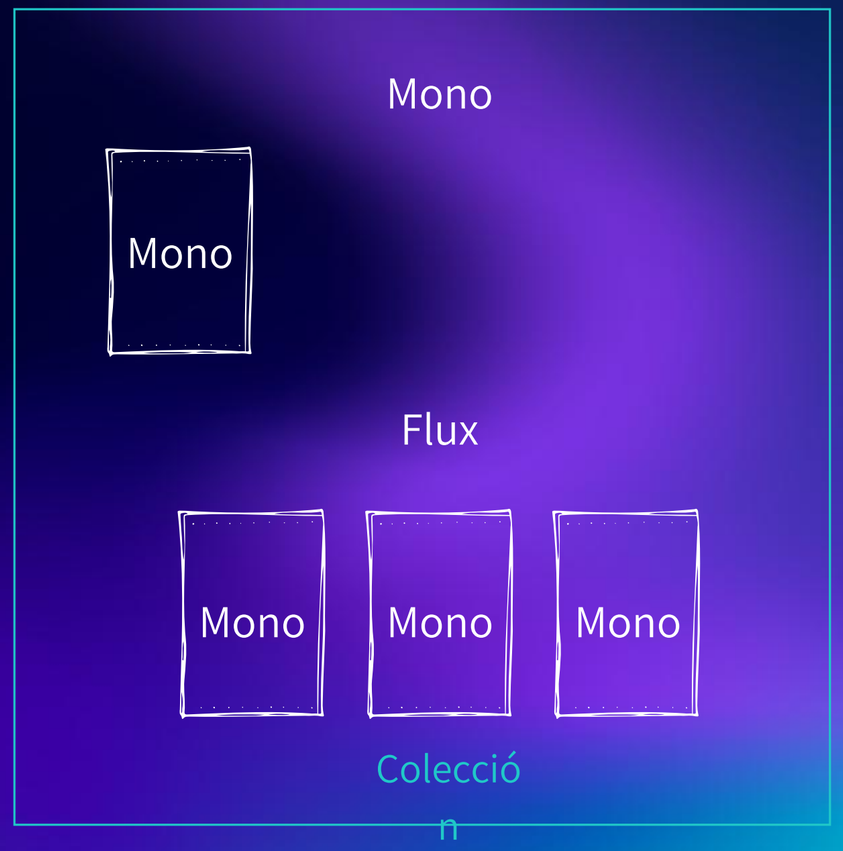
- Framework de Java que simplifica el desarrollo de aplicaciones
- Características principales:
  - Configuración automática (auto-configuration)
  - Servidor embebido (Tomcat por defecto)
  - Gestión de dependencias simplificada
  - Aplicaciones standalone (no requiere servidor externo)
- Beneficios:
  - Desarrollo rápido y productivo
  - Configuración mínima requerida
  - Monitoreo y métricas incorporadas
  - Ideal para microservicios

Spring WebFlux

- Framework reactivo para aplicaciones web no bloqueantes
- Basado en Project Reactor
- Características clave:
  - Programación reactiva
  - Manejo asíncrono de peticiones
  - Soporte para streams de datos
  - Event-loop en lugar de thread-per-request
- Beneficios:
  - Mayor rendimiento con menos recursos
  - Mejor manejo de concurrencia
  - Ideal para aplicaciones con alta carga
  - Excelente para servicios en tiempo real

Elección basada en requisitos:
Boot: Aplicaciones tradicionales/síncronas
WebFlux: Alta concurrencia/tiempo real

Reactividad



Docker

- Plataforma de contenedores para aplicaciones

- Elementos principales:
  - Contenedores: Entornos aislados y ligeros
  - Imágenes: Plantillas para crear contenedores
  - Dockerfile: Script para construir imágenes
  - Registry: Repositorio de imágenes (ej: Docker Hub)

- Beneficios:
  - Consistencia entre entornos
  - Aislamiento de aplicaciones
  - Despliegue rápido
  - Eficiencia en recursos

Docker Compose

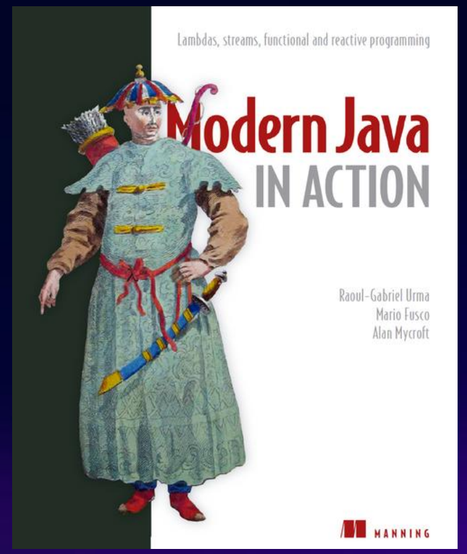
- Herramienta para definir y ejecutar aplicaciones multi-contenedor

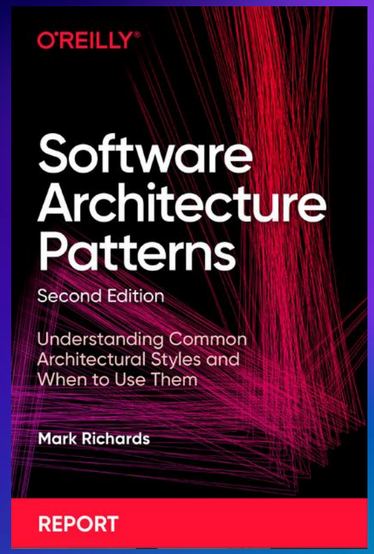
- Características principales:
  - Archivo YAML para configuración
  - Gestión de servicios relacionados
  - Redes y volúmenes compartidos
  - Variables de entorno

- Beneficios:
  - Orquestación simplificada
  - Reproducibilidad del entorno
  - Gestión de dependencias entre servicios
  - Desarrollo local más sencillo

Referencias

- https://www.atlassian.com/es/microservices/microservices-architecture
- https://learn.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices
- https://aws.amazon.com/es/microservices/
- https://docs.spring.io/spring-framework/reference/web/webflux.html
- https://www.openapis.org/what-is-openapi
- https://www.redhat.com/es/topics/containers/what-is-docker





Gracias!

¿Tienes alguna Pregunta?

contacto@codearti.com
+51 953 888 029
codearti.com

codearti