DevOps VietNam

MICROSERVICES CHEAT SHEET

Microservices là kiến trúc chia ứng dụng thành các dịch vụ nhỏ, độc lập, giao tiếp qua API, dễ mở rộng và triển khai riêng lẻ.

NGUYÊN TẮC THIẾT KẾ (DESIGN PRINCIPLES)

NGUYÊN TẮC	MÔ TẢ	VÍ DỤ
Đơn trách nhiệm	Mỗi dịch vụ chỉ làm một việc duy nhất	Dịch vụ "Order" chỉ xử lý đơn hàng
Độc lập triển khai	Có thể cập nhật một dịch vụ mà không ảnh hưởng dịch vụ khác	Cập nhật "Payment" không dừng "User"
Phi trạng thái	Dịch vụ không lưu trạng thái, dùng cơ sở dữ liệu bên ngoài	Trạng thái phiên lưu trong Redis
Giao tiếp nhẹ	Dùng API REST hoặc message queue thay vì giao thức nặng	REST qua HTTP hoặc RabbitMQ
Cơ sở dữ liệu riêng	Mỗi dịch vụ có database riêng, không chia sẻ	"Order" dùng PostgreSQL, "User" dùng MongoDB

CÔNG CỤ VÀ CÔNG NGHỆ PHỔ BIẾN (TOOLS & TECHNOLOGIES)

CÔNG CỤ	MỤC ĐÍCH	VÍ DỤ SỬ DỤNG
Docker	Đóng gói dịch vụ thành container	`docker run -d my-service`
Kubernetes	Quản lý và mở rộng container	`kubectl apply -f deployment.yaml`
API Gateway (Kong, NGINX)	Định tuyến và bảo mật API	Chuyển "/orders" đến dịch vụ Order
RabbitMQ/Kafka	Hàng đợi tin nhắn cho giao tiếp bất đồng bộ	Gửi sự kiện "OrderCreated" qua Kafka
Prometheus + Grafana	Giám sát và trực quan hóa hiệu suất	Theo dõi thời gian phản hồi API

TRIỂN KHAI MICROSERVICES (DEPLOYMENT)

Ví dụ Dockerfile

```
FROM node:16
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["node", "server.js"]
```

***Giải thích**: Đóng gói dịch vụ Node.js, mở cổng 3000.*

Ví dụ Kubernetes Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: order
  template:
    metadata:
      labels:
        app: order
    spec:
      containers:
      - name: order
        image: order-service:1.0
        ports:
        - containerPort: 3000
```

***Giải thích**: Triển khai 3 bản sao của dịch vụ "Order".*

GIAO TIẾP GIỮA CÁC DỊCH VỤ (SERVICE COMMUNICATION)

PHƯƠNG THỨC	MÔ TẢ	ƯU ĐIỂM	NHƯỢC ĐIỂM
REST API	Giao tiếp đồng bộ qua HTTP	Dễ triển khai, phổ biến	Độ trễ cao nếu nhiều dịch vụ
gRPC	Giao tiếp hiệu suất cao dùng Protocol Buffers	Nhanh, hỗ trợ đa ngôn ngữ	Phức tạp hơn REST
Message Queue	Giao tiếp bất đồng bộ qua hàng đợi	Khả năng chịu lỗi tốt	Cần quản lý hàng đợi
Event Sourcing	Lưu trữ và phát sự kiện thay vì trạng thái	Dễ theo dõi lịch sử	Khó đồng bộ dữ liệu

QUẢN LÝ DỮ LIỆU (DATA MANAGEMENT)

KỸ THUẬT	MÔ TẢ	VÍ DỤ
Database per Service	Mỗi dịch vụ có cơ sở dữ liệu riêng	"Payment" dùng MySQL, "Inventory" dùng DynamoDB
CQRS	Tách Command (ghi) và Query (đọc)	Ghi vào PostgreSQL, đọc từ Elasticsearch
Saga Pattern	Quản lý giao dịch phân tán qua các bước	Choreography: Dịch vụ tự phát sự kiện
Eventual Consistency	Đảm bảo dữ liệu nhất quán cuối cùng	Cập nhật "Stock" sau khi "Order" hoàn tất

GIÁM SÁT VÀ XỬ LÝ SỰ CỐ (MONITORING & TROUBLESHOOTING)

CÔNG CỤ/LỆNH	MÔ TẢ	VÍ DỤ
`docker logs [container]`	Xem nhật ký container	`docker logs order-service`
`kubectl get pods`	Liệt kê trạng thái pod	Kiểm tra pod bị crash
`Prometheus Query`	Truy vấn số liệu hiệu suất	`rate(http_requests_total[5m])`
`Distributed Tracing (Jaeger)`	Theo dõi yêu cầu qua nhiều dịch vụ	Xác định độ trễ giữa "Order" và "Payment"

MẪU TRIỂN KHAI THỰC TẾ (PRACTICAL DEPLOYMENT EXAMPLE)

Cấu trúc hệ thống

- **Order Service**: REST API, PostgreSQL, port 3000.
- **Payment Service**: gRPC, Redis, port 4000.
- **API Gateway**: NGINX, định tuyến /orders → Order, /payments → Payment.
- **Message Queue**: Kafka xử lý sự kiện "OrderPlaced".

Docker Compose

```
version: '3'
services:
  order:
    image: order-service:1.0
    ports:
      - "3000:3000"
    environment:
      - DB_HOST=postgres
  payment:
    image: payment-service:1.0
    ports:
      - "4000:4000"
    environment:
      - REDIS_HOST=redis
  postgres:
    image: postgres:13
    environment:
      - POSTGRES_USER=order
      - POSTGRES_PASSWORD=secret
  redis:
    image: redis:6
```

***Giải thích**: Triển khai Order và Payment với database riêng.*

MẸO TRIỂN KHAI VÀ XỬ LÝ SỰ CỐ

- **Circuit Breaker**: Dùng Hystrix hoặc Resilience4j để tránh lỗi lan truyền.
- **Health Checks**: Thêm endpoint `/health` cho mỗi dịch vụ.
- **Logging**: Dùng ELK (Elasticsearch, Logstash, Kibana) để tập trung log hệ thống.
- **Rollback**: Chuẩn bị phiên bản trước trong CI/CD (ví dụ: blue-green deployment).
- **Test**: Dùng Chaos Engineering (Chaos Monkey) để chủ động kiểm tra độ bền hệ thống.
