

# Kiến thức lý thuyết về Microservices & Các Pattern phổ biến

---

## 1. Microservices là gì?

Microservices là một kiến trúc phần mềm trong đó ứng dụng được chia thành nhiều dịch vụ nhỏ, hoạt động độc lập và giao tiếp với nhau qua các giao thức nhẹ như HTTP hoặc gRPC.

Mỗi microservice chịu trách nhiệm cho một chức năng cụ thể trong toàn bộ hệ thống. Các nhóm phát triển có thể làm việc độc lập trên từng service, triển khai và scale mà không ảnh hưởng đến phần còn lại.

Ví dụ, trong một hệ thống thương mại điện tử, bạn có thể có các service riêng cho người dùng, đơn hàng, sản phẩm và thanh toán. Mỗi service này hoạt động như một ứng dụng riêng biệt với database riêng và có thể được phát triển bằng công nghệ riêng.

## 2. Lợi ích và Thách thức

Kiến trúc microservices giúp tăng tính linh hoạt, khả năng mở rộng, và khả năng triển khai liên tục. Nhưng cũng mang lại nhiều thách thức như quản lý dịch vụ phân tán, đồng bộ dữ liệu, và tăng độ phức tạp hệ thống.

### 2.1. Lợi ích


- Phát triển độc lập: Các nhóm có thể phát triển và triển khai từng service riêng biệt.
- Tăng khả năng mở rộng: Có thể scale từng service theo nhu cầu sử dụng.
- Khả năng chịu lỗi tốt hơn: Nếu một service gặp sự cố, các service khác vẫn hoạt động.
- Dễ bảo trì và phát triển: Các service nhỏ dễ quản lý và kiểm thử hơn.
- Hỗ trợ đa ngôn ngữ và công nghệ: Mỗi service có thể được viết bằng ngôn ngữ khác nhau.

### 2.2. Thách thức

- Quản lý phân tán phức tạp hơn: Phải xử lý networking, logging, tracing...
- Tốn kém về tài nguyên hệ thống khi triển khai nhiều service nhỏ.
- Cần thiết lập pipeline CI/CD chuyên biệt cho từng service.
- Yêu cầu service discovery và balancing hiệu quả.

- Phức tạp trong việc kiểm thử tích hợp toàn hệ thống.

## Chi tiết các Pattern trong Microservices

 Chi tiết các Pattern trong Microservices

### 3.1. API Gateway

Chức năng chính:

- Là cổng vào duy nhất cho toàn bộ hệ thống backend.
- Phân phối request đến đúng service backend tương ứng.
- Hỗ trợ xử lý authentication, routing, logging, caching và rate limiting.

Công cụ phổ biến:

- Express Gateway, Kong, NGINX, Zuul (Netflix), API Gateway (AWS)

Best Practices:

- Tránh thực hiện business logic tại gateway.
- Sử dụng rate limit để chống abuse.
- Kết hợp với service discovery để dynamic routing.

### 3.2. Service Discovery

Mục tiêu:

- Cho phép các service tự động tìm kiếm nhau thông qua registry mà không cần cấu hình cứng.

Cách triển khai:

- Registry-based: Service đăng ký với registry (Eureka, Consul).
- DNS-based: Trong K8s, dùng DNS nội bộ để resolve tên service.

Công cụ phổ biến:

- Consul, Eureka, K8s CoreDNS, Etcd

Best Practices:

- Đăng ký và health check định kỳ.
- Tích hợp với load balancer hoặc gateway.

### 3.3. Authentication & Authorization (JWT)

Mô hình phổ biến:

- Stateless Auth với JWT: Không lưu session trên server.

JWT gồm:

- Header: Loại token và thuật toán (HS256, RS256).
- Payload: Thông tin người dùng (ID, role).
- Signature: Bảo vệ integrity.

Ưu điểm:

- Không cần session store.
- Dễ tích hợp với API Gateway.

Best Practices:

- Đặt thời gian hết hạn (exp) ngắn và refresh token.
- Không lưu thông tin nhạy cảm trong payload.
- Sử dụng HTTPS để bảo mật truyền tải.

### 3.4. Caching

Các dạng cache:

- Response Cache (HTTP-level)
- Data/Object Cache (thường dùng Redis)
- Computation Cache (tính toán nặng)

Chiến lược cache:

- TTL (Time To Live)
- Write-through / Write-back cache
- Cache invalidation bằng sự kiện

Công cụ:

- Redis, Memcached, CDN (Cloudflare, Akamai)

Best Practices:

- Xác định rõ dữ liệu nào nên cache.
- Cẩn thận khi cache dữ liệu mutable.
- Tránh stale data bằng chiến lược invalidation hợp lý.

### 3.5. Retry, Timeout, Circuit Breaker

Retry:

- Tự động gửi lại các request bị lỗi do sự cố tạm thời.
- Thường áp dụng backoff strategy (exponential, jittered).

Timeout:

- Giới hạn thời gian tối đa để tránh block hệ thống.
- Áp dụng cho cả outbound và inbound requests.

Circuit Breaker:

- Phát hiện service downstream bị lỗi liên tục.
- Ngắt luồng tạm thời để bảo vệ hệ thống.

Công cụ:

- Resilience4j, Hystrix (deprecated), Istio

Best Practices:

- Đặt ngưỡng hợp lý cho failure rate.
- Log chi tiết khi circuit mở/đóng.
- Kết hợp fallback logic.

### 3.6. Event-Driven Architecture (EDA)

Lợi ích:

- Tăng khả năng mở rộng.
- Giảm coupling giữa service.
- Dễ theo dõi chuỗi sự kiện.

Cách hoạt động:

- Service phát sự kiện lên message bus (Kafka, RabbitMQ).

- Các service subscriber lắng nghe và xử lý tương ứng.

Types:

- Choreography: Services phản ứng với event.
- Orchestration: Có một service điều phối chính (less common).

Best Practices:

- Thiết kế schema sự kiện rõ ràng (sử dụng Avro, JSON Schema).
- Versioning sự kiện để đảm bảo backward compatibility.
- Dùng dead-letter queue (DLQ) để xử lý lỗi.

### 3.7. Database per Service

Tại sao cần tách DB?

- Đảm bảo tính độc lập và đóng gói.
- Tránh lock hoặc conflict khi nhiều service truy cập cùng DB.

Khó khăn:

- Giao dịch phân tán (Distributed Transaction)
- Truy vấn kết hợp phức tạp

Giải pháp:

- Dùng CQRS hoặc Event Sourcing để đồng bộ dữ liệu.
- Hoặc Aggregator Service kết hợp dữ liệu từ nhiều nguồn.

Best Practices:

- Không chia sẻ DB hoặc bảng giữa các service.
- Dùng migration tool riêng cho mỗi DB service.

### 3.8. Centralized Logging & Distributed Tracing

Vấn đề:

- Request có thể đi qua nhiều service → khó debug nếu không có trace chung.

Giải pháp:

- Centralized logging (ELK stack, Fluentd)

- Distributed tracing (OpenTelemetry, Jaeger, Zipkin)

Cách triển khai:

- Mỗi service gán trace ID cho request.
- Gửi log/trace về một hệ thống tập trung để phân tích.

Best Practices:

- Gán trace ID vào tất cả log line.
- Monitor latency và error rate dựa vào trace.
- Kết hợp với alerting để phát hiện sớm vấn đề.