# CẤU TRÚC MICROSERVICES

## I. Tổng quan về Microservice

1. Định nghĩa

- Là kiến trúc phần mềm trong đó các ứng dụng được chia nhỏ, độc lập.
- Mỗi service đảm nhận 1 chức năng cụ thể khác nhau.
- Các service giao tiếp với nhau bằng giao thức như: HTTP, gRPC, Message Queue.

2. Mục tiêu

- Tăng tính linh hoạt và khả năng mở rộng.
- Dễ triển khai, bảo trì và phát triển độc lập.
- Cho phép sử dụng nhiều công nghệ khác nhaugiữa service.

3. Ưu & nhược điểm

Ưu:

- Dễ mở rộng theo chiều ngang
- Tách biệt nghiệp vụ rõ rang
- Triển khai độc lập
- Tăng độ tin cậy hệ thống

Nhược:

- Phức tạp trong quản lý và vận hành
- Khó debug và trace lỗi
- Yêu cầu CI/CD và monitoring tốt
- Giao tiếp giữa service có thể gây độ trễ

# CẤU TRÚC MICROSERVICES

## I. Tổng quan về Microservice

4. Tại sao phải dung Microservice

- Tăng tốc độ phát triển: Có thể làm việc song song trên các serive khác nhau
- Triển khai linh hoạt: Không cần deploy toàn bộ hệ thống khi thay đổi 1 phần nhỏ
- Khả năng mở rộng tốt: có thể scale từng service theo nhu cầu
- Chịu lỗi cao: Một service lỗi thì không làm sập toàn bộ hệ thống
- Tự do công nghệ: Có thể dung ngôn ngữ, framwork, database riêng

5. Khi nào nên dùng Microservice

- Hệ thống có quy mô lớn, nhiều nghiệp vụ phúc tạp
- Có nhiều team phát triển, cần tách biệt trách nhiệm
- Cần mở rộng linh hoạt từng phần của hệ thống
- Cần triển khai liên tục (CI/CD) và có hạ tầng hỗ trợ tốt như: Docker, Kubernetes
- Hệ thống cần tính sẵn sàng cao, không bị ảnh hưởng bởi lỗi cục bộ
- Dễ nâng cấp bảo trì, độ tin cậy cao

Ghi chú:

Deploy (triển khai) là quá trình đưa ứng dụng từ môi trường phát triển lên môi trường thực tế để người dùng có thể sử dụng.

Scale (mở rộng) là khả năng tăng hoặc giảm tài nguyên hệ thống để đáp ứng lượng người dùng hoặc xử lý dữ liệu.

Stack (công nghệ stack) là tập hợp các công nghệ bạn dùng để xây dựng ứng dụng

# CẤU TRÚC MICROSERVICES

## I. Tổng quan về Microservice

4. So sánh Monolith và Microservice

| | Monolith | Microservice |
|---|---|---|
| Cấu trúc hệ thống | Một khối duy nhất, tất cả chức năng gộp chung | Chia thành nhiều service nhỏ, mỗi service đảm nhận 1 chức năng |
| Triển khai | Deploy toàn bộ hệ thống mỗi lần | Deploy từng service độc lập |
| Khả năng mở rộng | Khó mở rộng từng phần, phải scale từng phần | Dễ dàng scale từng service theo nhu cầu |
| Độ phức tạp | Đơn giản, dễ quản lý ban đầu | Phức tạp, cần quản lý giao tiếp và vận hành |
| Tính chịu lỗi | Mỗi lỗi có thể ảnh hưởng toàn hệ thống | Service lỗi không ảnh hưởng đến service khác |
| Công nghệ | Hạn chế, thường dung chung 1 stack | Tự do chọn công nghệ cho từng service |
| Debug & Trace lỗi | Dễ dàng | Khó, cần công cụ hỗ trợ |
| Phát triển & bảo trì | Khó chia team, dễ xung đột khi làm việc chung | Dễ chia team theo chức năng, phát triển song song |
| Hạ tầng | Đơn giản, ít yêu cầu | Cần CI/CD, monitoring, |

# CẤU TRÚC MICROSERVICES

## II. Cấu trúc tổng thể của Microservice

1. API Gateway

- Là cổng vào duy nhất cho client, định tuyến và bảo mật request.
- Điểm tiếp xúc duy nhất giữa client với hệ thống microservice

Thực hiện:

- Định tuyền (routing)
- Xác thực, phân quyền (authentication)
- Gộp nhiều service thành một response

2. Service

- Mỗi service đảm nhận 1 chức năng nghiệp vụ riêng.
- Có thể viết bằng ngôn ngữ khác nhau

Mỗi service cần đảm bảo

- Độc lập triển khai
- Độc lập dữ liệu
- Single Responsibility (thực hiện 1 chức năng)

Giao tiếp giữa service

- Đồng bộ: dùng REST API, gRPC
- Bất động bộ: dung Kafka, RabbitMQ

# CẤU TRÚC MICROSERVICES

## II. Cấu trúc tổng thể của Microservice

3. CSDL (Database per service)

- Mỗi service có 1 DB riêng để đảm bảo tính độc lập
- Các service không được truy cập DB của nhau
- Khi cần chia sẽ thông tin thì dung API hoặc message events

4. Service Discovery

- Tự động tìm kiếm và kết nối giữa các service trong hệ thống

Trường hợp dùng

- Khi hệ thống có nhiều instance
- Khi các service có thể thay đổi địa chỉ IP
- Khi muốn giảm độ phụ thuộc vào cấu hình tĩnh và tăng tính linh hoạt.

Thành phần chính:

- Service Registry: lưu trữ thông tin về các service đang hoạt động (tên, địa chỉ, port…).
- Service Provider: các service đăng ký thông tin của mình vào registry khi khởi động.
- Service Consumer: các service khác truy vấn registry để tìm địa chỉ của service cần gọi.

Hai mô hình chính:

- Client-side discovery: client tự hỏi registry để lấy địa chỉ service
- Server-side discovery: client gửi request đến một load balancer, load balancer hỏi registry (chưa hiểu)

Công cụ: Eureka, Consul

# CẤU TRÚC MICROSERVICES

## II. Cấu trúc tổng thể của Microservice

5. Message Broker

- Giao tiếp bất đồng bộ giữa các service (RabbitMQ, Kafka)
- Thay vì gọi trực tiếp API giữa các service, các service có thể phát sinh sự kiện và gửi vào hệ thống Message Broker
- Giảm độ phụ thuộc giữa các service, tăng khả năng mở rộng, và cải thiện độ tin cậy của hệ thống.

6. Cấu hình trung tâm (Centralized Configuration)

- Quản lý cấu hình tập trung cho toàn bộ hệ thống microservices.
- Cho phép cập nhật cấu hình mà không cần khởi động lại service.
- Mỗi service có thể lấy cấu hình từ một nguồn duy nhất, giúp dễ dàng kiểm soát và đồng bộ.
- Hỗ trợ thay đổi cấu hình động, giúp hệ thống linh hoạt hơn khi triển khai hoặc vận hành.

Công cụ: Spring Cloud Config, Consul Config

7. CI/CD Pipeline (chưa hiểu gì)

- Tự động build, test, deploy cho từng service
- CI (Continuous Integration – tích hợp liên tục): Khi push code lên repository (thường là Git), hệ thống sẽ tự động kiểm tra, build và test code đó.
- CD (Continuous Delivery / Continuous Deployment – Triển khai liên tục) Nếu quá trình kiểm tra thành công, hệ thống sẽ tự động triển khai code lên môi trường staging (kiểm thử) hoặc production (sản phẩm).
  - Continuous Delivery: Code được chuẩn bị sẵn để deploy, nhưng cần thao tác thủ công để triển khai.
  - Continuous Deployment: Code được triển khai hoàn toàn tự động sau khi vượt qua các bước kiểm tra.

# CẤU TRÚC MICROSERVICES

## II. Cấu trúc tổng thể của Microservice

8. Monitoring & Logging

- Theo dõi hiệu năng hệ thống, truy vết lỗi
- Công cụ phổ biến:
- Prometheus: Thu thập dữ liệu: CPU, RAM, số lượng request
- Grafana: Tạo biểu đồ dashboard
- ELK Stack (Elasticsearch, Logstash, Kibana)
  - Logstash: thu thập và xử lý log.
  - Elasticsearch: lưu trữ và tìm kiếm log.
  - Kibana: hiển thị log và tạo dashboard.

9. Security Layer

- Chỉ người dùng hợp lệ mới được truy cập tài nguyên, và dữ liệu được bảo vệ an toàn.
- Authentication – Xác thực người dùng (OAuth2)
  - Sử dụng giao thức OAuth2 để xác thực người dùng qua bên thứ ba (Google, Facebook,…)
  - Sau khi xác thực, hệ thống cấp Access Token cho client
  - Token chứng minh người dùng đã đăng nhập hợp lệ
- Authorization – Phân quyền truy cập (JWT)
  - Dùng JWT (JSON Web Token) để mã hóa thông tin phân quyền
  - Token chứa thông tin như: user_id, role, permissions
  - Server kiểm tra JWT để xác định quyền truy cập tài nguyên

# CẤU TRÚC MICROSERVICES

## II. Cấu trúc tổng thể của Microservice

10. Load Balancer

- Load Balancer giúp phân phối lưu lượng truy cập đều giữa các instance của service, đảm bảo tính sẵn sàng và hiệu năng.
- Cách hoạt động:
  - Nhận request từ client và chuyển đến instance phù hợp.
  - Theo dõi tình trạng các instance để tránh gửi request đến node bị lỗi.
  - Có thể hoạt động ở tầng Layer 4 (TCP) hoặc Layer 7 (HTTP).
- Công cụ phổ biến:
  - Nginx hoặc HAProxy: Load balancer truyền thống, cấu hình linh hoạt.
  - Cloud Load Balancer (AWS ELB, Azure Load Balancer): Tích hợp sẵn trong hạ tầng cloud.
  - Service Mesh (Istio, Linkerd): Load balancing thông minh trong môi trường microservices.
- Lợi ích:
  - Tăng khả năng mở rộng (scalability).
  - Đảm bảo tính sẵn sàng cao (high availability).
  - Giảm thiểu downtime khi có instance gặp sự cố.

# CẤU TRÚC MICROSERVICES

## III. Giao tiếp giữa các Microservice

1. REST API (HTTP)

- Giao tiếp đồng bộ qua HTTP
- Sử dụng JSON hoặc XML
- Mỗi microservice cung cấp các endpoint RESTful (Get, Post, Put, Delete)

Trường hợp dung

- Hệ thống nhỏ - vừa
- Đơn giản dễ debug
- Giao tiếp yêu cầu phản hồi ngay lập tức

Ưu

- Phổ biến, dễ dùng
- Hỗ trợ framework: Spring boot,…
- Dễ test: Post,…

Nhược

- Phù thuộc vào mạng
- Nếu service B chết => service A treo
- Không tối ưu cho hệ thống lớn cần hiệu suất cao

# CẤU TRÚC MICROSERVICES

## III. Giao tiếp giữa các Microservice

2. gRPC

- Giao tiếp hiệu suất cao, dùng protobuf
- Gọi qua stub đã được sinh ra tự động trong .proto
- Dựa vào HTTP/2, hỗ trợ multiplexing và streaming

Trường hợp dung

- Hệ thống lớn, cần hiệu suất cao
- Các service nội bộ giao tiếp với nhau
- Các ứng dụng yêu cầu truyền dữ liệu nhanh và liên tục

Ưu

- Nhanh hơn REST API (Ít tốn băng thông)
- Hỗ trợ song song và multiplexing qua HTTP/2
- Hỗ trợ nhiều ngôn ngữ
- Có khả năng streaming dữ liệu 2 chiều

Nhược

- Phải biết thêm về proto
- Debug khó hơn so với REST API
- Không tối ưu cho hệ thống lớn cần hiệu suất cao

Ghi chú:

- HTTP/2 là phiên bản nâng cấp của giao thức HTTP/1.1 – giao thức nền tảng cho web

# CẤU TRÚC MICROSERVICES

## III. Giao tiếp giữa các Microservice

3. Message Queue

- Giao tiếp bất đồng bộ qua hàng đợi
- 1 service gửi message vào hàng đợi => service khác nhận message và xử lý

Trường hợp dung

- Khi không cần phản hồi tức thì (ví dụ: gửi email, …)
- Khi cần scale xử lý (nhiều consumer xử lý song song)
- Tách biệt logic giữa service

Ưu

- Không bị gián đoán nếu 1 service lỗi
- Có thể retry, delay, xử lý hàng loạt
- Decoupling giữa các service

Nhược

- Cần quản lý hàng đợi
- Message backlog nếu consumer chậm
- Debug khó

Ghi chú:

**Scale** nghĩa là mở rộng khả năng xử lý của hệ thống khi lượng công việc tăng lên.

**Consumer** là thành phần nhận và xử lý message từ hàng đợi.

**Message backlog** là tình trạng **message bị tồn đọng** trong hàng đợi vì chưa được xử lý kịp.

# CẤU TRÚC MICROSERVICES

## III. Giao tiếp giữa các Microservice

4. Event Streaming

- Giao tiếp bất đồng bộ kiểu phát sóng
- Các service phát (publish) sự kiện => các service khác nghe (subscribe) sự kiện và xử lý

Trường hợp dung

- Kiến trúc hướng sự kiện (Event-Driven Architecture).
- Hệ thống lớn, nhiều service phản ứng với 1 sự kiện
- Khi cần lưu vết lịch sử hành động

Ưu

- Không rằng buộc giữa các service
- Có thể mở rộng ( nhiều cosumer nghe cùng 1 event)
- Lưu lịch sử event
- Tăng khả năng phản ứng

Nhược

- Cần thiết kế tốt event
- Event sai => khó debug
- Kỹ thuật cao

# CẤU TRÚC MICROSERVICES

## IV. Đặc điểm củaMicroservice

- Độc lập triển khai:  các microservice có thể build, deloy, scale riêng biệt
- Phân tách theo chức năng: mỗi service thực hiện 1 chức năng riêng
- Giao tiếp qua giao thức: REST, gRPC, Message queue
- CSDL riêng biêt:  mỗi service có 1 db riêng
- Tính chịu lỗi cao: Nếu 1 service lỗi thì service vẫn rất OK
- Dễ mở rộng: Có thể mở rộng từng service
- Hướng DevOps: Dễ áp dụng CI/CD, tự động kiểm thử, container hóa
- Công nghệ đa dạng: có thể voeetts bằng nhiều ngôn ngữ khác nhau hoặc framwork khác

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

1. Theo cách triển khai dịch vụ (Deployment Style)

1.1 Triển khai độc lập từng service

- Mỗi microservice được build và chạy như 1 tiến trình (process) riêng biệt
- Có thể chạy trên các máy chủ vật lý hoặc máy áo khác nhau

Ví dụ:

- ServiceA chạy trên post 8080
- ServiceB chạy trên post 8081

Trường hợp dung

- Hệ thống có quy mô vừa – lớn, nhiều service có chức năng riêng, mở rộng độc lập
- Yêu cầu cao về linh hoạt và khả năng phát triển liên tục (CI/CD)
- Đội ngũ phát triển phân chia theo từng service
- Cần khả năng mở rộng riêng biệt
- Ưu tiên ổn định và cô lập lỗi

Ưu

- Tách biệt rõ ràng giữa các service, tránh ảnh hưởng đến nhau
- Có thể cập nhật, triển khai 1 service mà không ảnh hưởng
- Dễ mở rộng từng service theo nhu cầu

Nhược

- Tốn tài nguyên hệ thống (Ram, Cpu) do service chạy trên 1 tiến trình riêng

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

1. Theo cách triển khai dịch vụ (Deployment Style)

1.2 Dùng Docker để container hóa

- Mỗi microservice được đóng gói vào 1 container Docker
- Tất cả container có thể chạy trên cùng 1 máy hoặc phân phối thành nhiều máy

Trường hợp dung

- Cần đóng gói dịch vụ triển khai trên nhiều môi trường khác nhau (test, dev, prod)
- Khi cần chuyển đổi môi trường nhanh chóng và đảm bảo tính nhất quản

Ưu

- Nhẹ hơn máy ảoVM, khởi động nhanh hơn
- Dễ chuyển đổi và triển khai các môi trường khác nhau
- Đóng gói đầy đủ dependencies, giảm lỗi do khác môi trường

Nhược

- Cần hiết biết Docker và containerization
- Cần quản lý vòng đời dịch vụ: cập nhật, giám sát, mở rộng (scale)
- Số lượng container tăng, thì cần thêm công cụ: Kubernetes để điều phối

Ghi chú

- Container là đơn vị phần mềm nhỏ ngọn, chưa tất cả những gì cần  để chạy 1 ứng dụng: mã nguồn, thư viện, cấu hình, và các phụ thuộc (dependencies)
- Docker là một nền tảng phần mền giúp tạo, triển khai và chạy các container một cách dễ dàng

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

1. Theo cách triển khai dịch vụ (Deployment Style)

1.3 Dùng Kubernetes để quản lý dịch vụ

- Kubernetes là nền tảng mã nguồn mở dùng để quản lý và điều phối các container,
- Đặc biệt phù hợp với microservice

Trường hợp dung

- Hệ thống lớn, phức tạp cần khả năng tự mở rộng (auto-scale)
- Tính  sẵn sàng cao
- Yêu cầu quản lý vong đời dịch vụ một cách tự động và hiệu quả

Ưu

- Tự động hóa toàn bộ vòng đời microservice
- Khả năng mở rộng linh hoạt, dễ scale
- Rolling update và rollback
- Quản lý cầu hình và bí mật thông qua ConfigMap và Secret

Nhược

- Phức tạp cao: cần thời gian để học, làm chủ các khái niệm: Pod, Service, Deployment,….
- Cần nhân sự chuyên môn: DevOps hoặc Plaform Engineer quản lý
- Chi phí vận hành có thể cao

Ghi chú

- Rolling Update: là cập nhập phiên bản mới của ứng dụng từng phần 1 (từng Pod), giúp hệ thống không bị gián đoạn

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

1. Theo cách triển khai dịch vụ (Deployment Style)

Ý kiến phù hợp với bài toán (chat nội bộ) – Chọn: Triển khai độc lập từng service

- Dùng nội bộ chỉ phục vụ trong công ty, tổ chức, số lượng người dùng giới hạn
- Mỗi service chạy trực tiếp trên service hoặc máy ảo nên có thể deploy thủ công
- Update 1 service thì chỉ cần build lại và restart lại service đó.
- Mỗi service chạy như một tiến trình (process) riêng, dễ mở log, debug trực tiếp

Cách làm

- Mỗi service là 1 ứng dụng riêng biệt, build thành file chạy
- Chạy trên nhiều cổng  khác nhau trên cùng 1 máy
- Dùng API Gateway để quản lý endpoint
- Khi update service chỉ cần restart service đó

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

2. Theo cách giao tiếp

2.1 REST API (đồng bộ)

- Các service gọi nhau thông qua HTTP (thường dùng JSON hoặc XML)

Trường hợp dung

- Giao tiếp trực tiếp giữa các service
- Phù hợp với hệ thống đơn giản, dễ triển khai

Ưu

- Dễ dùng, phổ biến rộng rãi
- Debug đơn giản và kiểm tra
- Tích hợp tốt với nhiều ngôn ngữ và nên tảng

Nhược

- 1 Service lỗi => Service liên quan ảnh hưởng
- Các service phụ thuộc lẫn nhau
- Khó mở rộng trong hệ thống lớn hoặc yêu cầu hiệu năng cao

Ghi chú

- JSON (javascript object notation): là một định dạng dữ liệu nhẹ, dễ đọc và ghi, thường dung để trao đổi dữ liệu giữa client – server
- XML (eXtensible Markup Language): là ngôn ngữ đánh dấu dung để mô tả dữ liệu. Nó được thiết kể để có thể mở rộng và linh hoạt

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

2. Theo cách giao tiếp

2.2 gRPC(đồng bộ)

- Giao tiếp nhanh hơn REST, dùng protobuf thay vì JSON
- Hoạt động trên nền HTTP/2, hỗ trợ multiplexing và streaming
- Phù hợp với các hệ thống yêu cầu phản hồi nhanh và định nghĩa rõ ràng

Trường hợp dung

- Hệ thống yêu cầu hiệu năng cao, độ trễ thấp
- Cần định nghĩa rõ rang các API giữa các microservice
- Phù hợp với giao tiếp nội bộ giữa các service trong cùng hệ thống

Ưu

- Nhanh hơn REST (nhẹ, hiệu quả hơn JSON)
- Định nghĩa rõ ràng qua .proto
- Hỗ trợ streaming 2 chiều, rất hữa ích trong các ứng dụng real-time

Nhược

- Cần them bước biên dịch .proto
- Phức tạp hơn REST, đặc biệt khi tích hợp với hệ thống bên ngoài k dùng gRPC
- Debug khó hơn do dữ liệu không ở dạng dễ đọc như JSON

Ghi chú

- streaming 2 chiều là client và server có thể gửi dữ liệu cho nhau liên tục, đồng thời mà k cần đợi bên kia hoàn tất

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

2. Theo cách giao tiếp

2.3 Message Queue (bất đồng bộ)

- Service gửi message không cần chờ phản hồi
- Tách biệt dịch vụ: các service giao tiếp gián tiếp qua hàng đợi, giảm coupling
- Đảm bảo truyền tải: message k bị mất nếu được cấu hình đúng
- Xử lý hàng đợi: Message đước xử lý tuần tự hoặc theo nhóm
- Dùng Message Queue (RabbitMQ, Apeche Kafka)

Trường hợp dung

- Hệ thống lớn, cần scale, không cần phản hồi ngay
- Phù hợp với các tác vụ nền, xử lý hàng loạt hoặc truyền dữ liệu dữ các service

Ưu

- Không cần phản hồi ngay, tăng hiệu suất
- Dễ mở rộng, phù hợp hệ  thống lớn
- Giảm độ phụ thuộc giữa các service
- Có thể lưu trữ message tạm thời nếu service nhận chưa OK

Nhược

- Cần cơ chế retry, xử lý lỗi phức tạp
- Debug khó
- Yêu cầu giám sát tốt: Cần theo dõi queue, consumer, và xử lý lỗi hiệu quả

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

2. Theo cách giao tiếp

Ý kiến phù hợp với bài toán (chat nội bộ) – Chọn: REST API

- Dễ triển khai và debubg hơn gPRC, không cần học thêm .proto
- Dễ kết với các FE, chỉ cần gửi HTTP request

Cách làm

- Kiến trúc các service
  - Auth: quản lý đăng nhập đăng kí, quản lý thông tin, xác thực jwt
  - Chat: quản lý xử lý gửi/ nhận tin nhắn, tạo nhóm, thêm thành viên
- Giao tiếp giữa các service
  - Gọi REST API của nhau qua HTTP dùng Feign Client

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

3. Theo cách tổ chức database

3.1 Database riêng

- Mỗi microservice có database riêng, không chia sẽ database cho service khác
- Tăng tính độc lập, giảm rang buộc giữa các thành phần trong hệ thống
- Giao tiếp giữa các service thông qua API hoặc message broker (như Kafka, RabbitMQ)

Trường hợp dung

- Hệ thống lớn, nhiều domain nghiệp vụ
- Cần mở rộng linh hoạt từng service mà k ảnh hướng đến hệ thống
- Yêu cầu bảo mật và phân quyền dữ liệu

Ưu

- Tách biệt hoàn toàn giữa các service
- Không bị khóa dữ liệu chéo
- Tăng tính bảo mật và kiểm soát dữ liệu từng service
- Phù hợp với kiến trúc hướng domain (DDD)

Nhược

- Không join giữa các bảng service
- Cần dùng API – Message Queueđể trao đổi dữ liệu
- Phức tạp hơn trong việc đảm bảo tính nhất quán dữ liệu
- Tốn tài nguyên

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

3. Theo cách tổ chức database

3.2 Database chung

- Nhiều service cùng truy cập một database lớn
- Không tuân thủ theo nguyên lý “Data per service” chuẩn của microservice
- Dễ gây ra tình trạng “tight coupling” giữa các service
- Phù hợp với monolith chuyển sang microservice từng phần

Trường hợp dung

- Hệ thống nhỏ, ít service

Ưu

- Dễ truy vấn phức tạp, đặc biệt là join dữ liệu được
- Phù hợp với hệ thống nhỏ ít thay đổi
- Quản lý dữ liệu tập trung, dễ kiểm soát

Nhược

- Tăng độ phụ thuộc giữa các service
- Khó mở rộng khi hệ thống phát triển
- Một service lỗi hoặc truy vấn sai có thể ảnh hưởng hệ thống
- Khó triển khai các chiến lực: scaling độc lập,  CI/CD riêng, versioning database

Ghi chú

- Tight coupling là liên kết chặt chẽ, khi các thành phần trong hệ thống phụ thuộc lẫn nhau quá nhiều

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

3. Theo cách tổ chức database

Ý kiến phù hợp với bài toán (chat nội bộ) – Chọn: Database riêng

- Mỗi service độc lập về dữ liệu: dễ triển khai, bảo trì mở rộng
- Tránh tình trạng ảnh hưởng lẫn nhau
- Dễ thay đổi công nghệ DB theo như cầu từng service

Cách làm

- Auth: dùng Oracle: lưu thông tin người dùng: username, password, email, fullname,..
- Chat: dung Oracle: lưu tin nhắn, thông tin tin nhắn, nhóm, thành viên,,,,,,

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

4. Theo cách chia nhỏ theo nghiệp vụ

4.1 Subdomain (DDD – Bounded Context)

- Chia theo Bounded Context dựa trên mô hình Domain-Driven Design
- Mỗi service tương ứng một nghiệp vụ rõ ràng

Trường hợp dung

- Hệ thống lớn, nghiệp vụ phức tạp, nhiều team, cần bám sát mô hình nghiệp vụ lâu dài

Ưu

- Rõ ràng về phạm vi nghiệp vụ;
- Bám sát nghiệp vụ, giảm chống chéo
- Dễ mở rộng, bảo trì

Nhược

- Phân tích ban đầu tốn nhiều thời gian
- Khó áp dụng dự án nhỏ hoặc MVP
- Đòi hỏi team có kinh nghiệm DDD

Ghi chú

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

4. Theo cách chia nhỏ theo nghiệp vụ

4.2 Business Capability

- Chia theo chức năng kinh doanh mà doanh nghiệp cung cấp.

Trường hợp dung

- Khi bắt đầu thiết kế, hoặc tách monolith, cần dễ hiểu cho cả kỹ thuật và business

Ưu

- Dễ hiểu cho cả kỹ thuật và business
- Nhanh xác định phạm vị dịch vụ
- Dễ giải thích với stakerholder

Nhược

- Chưa tối ưu và kỹ thuật
- Dễ bị chống chéo nếu k refine theo DDD
- Có thể thiếu tính module hóa

Ghi chú

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

4. Theo cách chia nhỏ theo nghiệp vụ

4.3 Workflow / Business Process

Chia theo luồng xử lý nghiệp vụ hoặc quy trình làm việc

Trường hợp dung

- MVP, POC, startup cần qua sản phẩm nhanh

Ưu

- Nhanh mapping với yêu cầu thực tế
- Phù hợp với MVP/POC
- Dễ giúp team hiểu rõ quy trình nghiệp vụ

Nhược

- Couping cao nếu nhiều luồng dung chung dữ liệu
- Khó mở rộng về lâu dài
- Có thể trùng lặp logic

Ghi chú

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

4. Theo cách chia nhỏ theo nghiệp vụ

4.4 Transantion Boundarise

Chia theo phạm vi giao dịch để đảm bảo tính nhất quán dễ liệu trong mỗi service

Trường hợp dung

- Hệ thống giao dịch tài chính, thanh toán, thương mại điện tử

Ưu

- Đảm bảo tính nhất quán dữ liệu
- Giảm rollback phức tạp
- Rõ ràng về phạm vi giao dịch

Nhược

- Có thể làm service lớn hơn mong muốn
- Tăng độ phức tạp khi giao tiếp
- Giảm tính module hóa

Ghi chú

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

4. Theo cách chia nhỏ theo nghiệp vụ

4.5 Entity/ Event-Driven

Chia theo Entity chính hoặc Event trong hệ thống, tập trung vào xử lý bất đồng bộ

Trường hợp dung

- Hệ thống có nhiều luống bất đồng bộ, real-time, event streaming (Kafka, Rabbit MQ)

Ưu

- Giảm coupling giữa service
- Tăng khả năng mở rộng
- Hỗ trợ xử lý bất đồng bộ, real-time

Nhược

- Khó debug khi lỗi
- Đòi hỏi message broker ổn định
- Tăng chi phí hạ tầng

Ghi chú

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

4. Theo cách chia nhỏ theo nghiệp vụ

So sánh

| | Ưu điểm | Nhược điểm |
|---|---|---|
| Subdomain (DDD) | Bám sát nghiệp vụ; Tách biệt trách nghiệm; Dễ mở rộng, bảo trì | Tốn thời gian phân tích; Yêu cầu kiến thức DDD; Khó áp dụng dự án nhỏ |
| Business Capability | Dễ hiểu cho kỹ thuật & business; Nhanh xác định phạm vi | Chưa tối ưu kỹ thuật; Dễ chống chéo; Thiếu module hóa |
| Workfrom / Business Process | Nhanh mapping với yêu cầu; Phù hợp MVP/POC | Coupling cao; Khó mở rộng; Trùng lặp logic |
| Transaction Boundaries | Đảm bảo nhất quán dữ liệu; Giảm rollback | Service có thể quá lớn; Tăng phức tạp giao tiếp |
| Entity / Event-Driven | Giảm coupling; Tăng mở rộng; Hỗ trợ real-time | Khó debug; Cần broker ổn định; Tăng chi phí hạ tầng |

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

4. Theo cách chia nhỏ theo nghiệp vụ

Ý kiến phù hợp với bài toán (chat nội bộ) – Chọn: DDD – Bounded Context

- Mỗi subdomain (miền nghiệp vụ): là một Bounded Context độc lập: code, DB không bị rối
- Dễ mở rộng từng module mà không ảnh hưởng module khác
- Phù hợp với microservice vì giúp database riêng, team tách biệt, gieo tiếp qua API

Chia subdomain

- Auth: đăng ký, đăng nhập, xác thực JWT, quản lý thông tin người dung
- Chat: gửi/ nhận tin nhắn, quản lý nhóm

Cách làm

- 1 subdomain: 1 service độc lập: code, db riêng
- Giao tiếp: REST API để auth và chat trao đổi dữ liệu

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

5. Theo cách xử lý lỗi – giám sát – an toàn hệ thống

5.1 Xử lý lỗi

- Retry: Tự động thử lại khi lỗi nhẹ
  - Kết hợp với Exponential Backoff (tăng dần thời gian giữa các dần thử)
  - Cần giới hạn số lần retry để tránh vòng lặp vô hạn
- Trường hợp dùng
  - Gọi API bị lỗi tạm thời: mạng chập chờn, service quá tải
  - Giao tiếp với hệ thống bên ngoài: payment, gateway, email service
- Ưu
  - Tăng khả năng phục hồi khi gặp lỗi tạm thời
  - Giảm số lượng lỗi trả về cho người dùng cuối
- Nhược
  - Có thể gây quá tải retry không giới hạn
  - Làm chậm hệ thống nếu retry nhiều lần

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

5. Theo cách xử lý lỗi – giám sát – an toàn hệ thống

5.1 Xử lý lỗi

- Time out: hủy request nếu quá thời gian
  - Nếu cấu hình time out phù hợp với từng loại service
  - Kết hợp với retry để tăng độ tin cậy
- Trường hợp dùng
  - Gọi đến service khác mà không phản hồi đúng thời gian
  - Tranh treo luồng xử lý trong hệ thống
- Ưu
  - Giúp hệ thống phản hồi nhanh, không bị treo
  - Giảm nguy cơ lan truyền lỗi giữa các service
- Nhược
  - Nếu time out quá ngắn, có thể hủy nhầm request đang xử lý
  - Nếu quá dài, gây lãng phí tài nguyên

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

5. Theo cách xử lý lỗi – giám sát – an toàn hệ thống

5.1 Xử lý lỗi

- Circuit Breaker: ngắt gọi nếu 1 service lỗi liên tục để không gây sập hệ thống
  - Có 3 trạng thái: Closed, Open, Half-Open
  - Khi ở trạng thái Open, các request bị chặn ngay lập tực
- Trường hợp dùng
  - Service A gọi service B nhưng B liên tục lỗi
  - Tránh việc A tiếp tục gọi B và gấy quá tải
- Ưu
  - Bảo vệ hệ thống khỏi lỗi làn truyền
  - Cho phéo service lỗi có thời gian phục hồi
- Nhược
  - Cần cấu hình ngưỡng chính xác để tránh ngắt mạch sớm hoặc muộn

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

5. Theo cách xử lý lỗi – giám sát – an toàn hệ thống

5.1 Xử lý lỗi

- Fallback: thay thế khi lỗi
  - Thường kết hợp với Circuit Breaker
  - Có thể dung cache, mock data, hoặc service phụ
- Trường hợp dùng
  - Khi service chính không hoạt động, dung dữ liệu cache hoặc service phụ
  - Ví dụ: Nếu k lấy được dữ liệu từ service sản phẩm thì hiển thị dữ liệu cũ
- Ưu
  - Tăng trải nghiệm người dung
  - Giảm tác động của lỗi đến toàn hệ thống
- Nhược
  - Dữ liệu có thể không chính xác hoặc lỗi tạm thời
  - Tăng độ phức tạp trong thiết kế

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

5. Theo cách xử lý lỗi – giám sát – an toàn hệ thống

5.2 Giám sát – logging – truy vết

- Logging: ghi log tập trung  dung mô hình ELK Stack (Elasticseach + Logstask + Kibana)
  - Elasticsearch: lưu trữ và tìm kiếm log hiệu quả
  - Logstash: thu thập và xử lý log từ service
  - Kibana: giao diện trực quan để phân tích log
  - Ghi log theo định dạng chuẩn (JSON, strucred logging)
  - Gắn metadata (service name, request ID, timestamp,…)
  - Log theo cấp độ: INFO, DEBUG, WARN, ERROR
- Trường hợp dung
  - Phân tích lỗi, sự cố hệ thống
  - Theo dõi hành vi người dung hoặc hiệu năng service
  - Audit và bảo mật
- Ưu
  - Tập trung hóa dữ liệu log: dễ tìm kiếm, phân tích
  - Hỗ trợ truy vến mạnh mẽ qua Elasticseach
  - Giao diện trực quan qua Kibana
- Nhược
  - Tốn tài nguyên: CPU, RAM, storage

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

5. Theo cách xử lý lỗi – giám sát – an toàn hệ thống

5.2 Giám sát – logging – truy vết

- Tracing: theo dõi request đi qua nhiều service, để xác định bottleneck, lỗi hoặc độ trễ
  - Gắn trace ID và span ID vào mỗi request
  - Mỗi service ghi lại thời gian xử lý, matadata
  - Hiển thị luồng request dạng cây hoặc biểu đồ thời gian
- Trường hợp dung
  - Debug lỗi phức tạp trong hệ thống phân tán
  - Phân tích hiệu năng từng service
  - Đảm bảo SLA và tối ưu latency
- Ưu
  - Hiển thị rõ rang luồng request, dễ xác định điểm nghẽn
  - Tích hợp tốt với các hệ thống giám sát khác (Prometheus, Grafana)
  - Hỗ trợ chuẩn OpenTelemtry
- Nhược
  - Cần tích hợp vào từng service, tang độ phức tạp
  - Tốn tài nguyên lưu trữ trace
  - Có thể gây ảnh hưởng hiệu năng nếu k cầu hình hợp lý

  Công cụ: Jaeger (do Uber phát triển), Zipkin (do Twitter phát triển)

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

6. Theo cách kiểm thử - kiểm soát phiên bản

- Đảm bảo từng service hoạt động đúng logic riêng biệt
- Đảm bảo các service tương tác đúng với nhau
- Đảm bảo hợp đồng giao tiếp giữa các service không bị phá vỡ khi thay đổi

6.1 Kiểm thử

Unit test: kiểm thử logic nội bộ từng service một cách độc lập

- Chạy nhanh, dễ tự động hóa
- Không phụ thuộc vào các service khác
- Thường kiểm thử cacgs hàm, class hoặc module riêng lẻ
- Trường hợp dung
  - Kiểm thử các hàm xử lý nghiệp vụ, tính toán, xử lý dữ liệu trong service
- Ưu
  - Phát hiện lỗi sớm
  - Dễ viết, duy trì
  - Tăng độ tin cậy cho từng service
- Nhược
  - Không kiểm thử sự tương thích giữa các service
  - Có thể bỏ sót lỗi tích hợp

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

6. Theo cách kiểm thử - kiểm soát phiên bản

- Đảm bảo từng service hoạt động đúng logic riêng biệt
- Đảm bảo các service tương tác đúng với nhau
- Đảm bảo hợp đồng giao tiếp giữa các service không bị phá vỡ khi thay đổi

6.1 Kiểm thử

Intergration test: kiểm thử sự tích hợp giữa service, đảm bảo chúng gọi nhau đúng cách

- Kiểm thử toàn bộ luồng xử lý có sự tham gia của nhiều service
- Có thể cần nhiều môi trường giản lập hoặc thực tế
- Trường hợp dung
  - Gọi API từ service A sang service B
  - Kiểm tra truy cập database, message queue hoặc các hệ thống bên ngoài
- Ưu
  - Phát hiện lỗi khi các service tương tác sai
  - Đảm bảo hệ thống hoạt động đúng như tích hợp
- Nhược
  - Chạy chậm hơn unit test
  - Khó cô lập lỗi
  - Phụ thuộc vào môi trường

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

6. Theo cách kiểm thử - kiểm soát phiên bản

- Đảm bảo từng service hoạt động đúng logic riêng biệt
- Đảm bảo các service tương tác đúng với nhau
- Đảm bảo hợp đồng giao tiếp giữa các service không bị phá vỡ khi thay đổi

6.1 Kiểm thử

Contract test: đảm bảo rằng các service tuân thủ hợp đồng giao tiếp (API, dữ liệu vào/ ra)

- Kiểm thử ở mức giao tiếp giữa các service
- Dùng để kiểm tra xem service A có trả đúng dữ liệu mà servie B cần không
- Trường hợp dung
  - Khi có nhiều team phát triển các service khác nhau
  - Khi cần đảm bảo back compatibility giữa các phiên bản API
- Ưu
  - Giảm rủi ro khi thay đổi API
  - Tăng độ tin cậy khi triển khai độc lập từng service
- Nhược
  - Cần định nghĩa hợp đồng rõ rang
  - Có thể phức tạp khi có nhiều phiên bản hoặc nhiều bên liên quan

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

6. Theo cách kiểm thử - kiểm soát phiên bản

6.2 Quản lý phiên bản API

- Đảm bảo tính tương thích ngược khi thay đổi API
- Cho phép phát triển và triển khai linh hoạt các phiên bản mới mà k ảnh hướng đến hệ thống đang hoạt động
- Hỗ trợ kiểm thử, giám sát, và rollback dễ dàng
- Mỗi phiên bản API được định danh rõ rang
- Có thể tồn tại song song với nhiều phiên bản để phục vụ client khác nhau
- Thường kết hợp với công cụ mô tả API như Swagger hoặc OpenAPI

Trường hợp dung

- Khi có nhiều client sử dụng api với yêu cầu khác nhau
- Khi cần thay đổi cấu trúc dữ liệu hoặc logic xử lý mà k ảnh hướng đến client cũ
- Muốn duy trì ổn định hệ thống trong quá trình nâng cấp

Ưu

- Tăng tính linh hoạt và khả năng mở rộng
- Giảm rủi ro khi triển khai thay đổi
- Hỗ trợ kiểm thử và rollback dễ dàng

Nhược

- Tăng độ phức tạp trong quản lý và bảo trì
- Có thể gây trùng lặp logic giữa các phiên bản

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

7. Theo luồng xử lý dữ liệu

7.1 Orchestration

- Có một làm điều phối trung tâm chịu trách nhiệm kiểm soát và ra lệnh cho các service khác thực hiện

Trường hợp dung

- Kiểm soát luồng công việc rõ rang và có thứ tự
- Phù hợp với các quy trình nghiệp vụ phức tạp, có nhiều xử lý tuần tự
- Dễ dàng áp dụng trong các hệ thống có yêu cầu kiểm tra, giám sát, rollback

Ưu

- Luồng xử lý dễ hiểu, dễ theo dõi
- Dễ debug và kiểm tra lỗi
- Có thế tái sử dụng logic điều phối

Nhược

- Coupling cao giữa orchestrator và các service
- Khó mở rộng khi số lượng service quá lớn
- Orchestrator có thể trở thành bottleneck hoặc single point of failure nếu k được thiết kế tốt

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

7. Theo luồng xử lý dữ liệu

7.2 Choreography

- Dùng mô hình Event-driven: mỗi service "tự động phản ứng" với các sự kiện liên quan, k có service trung tâm điều phối
- Các service gửi và lắng nghe sự kiện, thông qua 1 hệ thống message broker (Kafka, RabbitMQ)

Trường hợp dung

- Phù hộ với hệ thống phân tán, dễ mở rộng
- Khi các service độc lập và có thể xử lý theo luồng sự kiện riêng biệt

Ưu

- K có điểm điều phối trung tâm, giảm bottleneck
- Dễ mở rộng service độc lập
- Các service k phụ thuộc trực tiếp nhau

Nhược

- Debug khó,
- khó đảm bảo thứ tự xử lý đúng
- Khó kiểm soát lỗi

# CẤU TRÚC MICROSERVICES

## V. Cách triển khai và tổ chức Microservice

8. Theo cách đóng gói CI/CD

- Tự động hóa quá trình build, test, deploy từng microservice 1 cách độc lập
- Đảm bảo tính nhất quán trong quuy trình phát triển và triển khai
- Tăng tốc độ phát hành phần mềm, giảm rủi ro khi release
- Tách biệt trách nhiệm giữa nhóm phát triển, giúp dễ mở rộng hệ thống
- Mỗi microservice có thể cps pipeline riêng biệt hoặc dung chung tùy theo mô hình tổ chức
- Sử dụng công cụ: Jenkins, GitHub actions, GitLab CI,….
- Đóng gói service dưới dạng container (Docker) hoặc artifact (JAR, WAR,…)
- Các bước chuẩn: Build -> Test -> Package -> Deploy -> Monitor
- Hỗ trợ triển khai theo môi trường: Dev, Staging, Production

Ưu

- Tự động hóa toàn bộ quy trình, giảm lỗi thủ công
- Triển khai nhanh chóng, dễ rollback nếu có lỗi
- Tăng tính linh hoạt, mỗi service có thể được cập nhật độc lập
- Dễ mở rộng, phù hợp với kiến trúc microservice có nhiều nhóm phát triển
- Tích hợp kiểm thử liên tục giúp phát hiện lỗi sớm

Nhược

- Chi phí thiết lập ban đầu cao, cần đầu tư vào hạ tầng CI/CD và cấu hình pipeline
- Quản lý phức tạp khi số lượng service lớn

# CẤU TRÚC MICROSERVICES

## VI. Các kiểu của cấu trúc Microservice

1. Microservice truyền thống

- Tách biệt chức năng thành các service độc lập để tăng khả năng mở rộng, phát triển song song và triển khai linh hoạt
- Mỗi service là một tiến trình độc lập, có thể có CSDL riêng
- Giao tiếp giữa các service thông qua: HTTP, gRPC hoặc message queue
- Thường sử dụng API Gateway để quản lý truy cập từ bên ngoài

Trường hợp dung

- Hệ thống lớn nhiều team phát triển song song
- Các service có vòng đời và logic đặc biệt

Ưu

- Dễ mở rộng và phát triển độc lập từng service
- Phù hợp với tổ chức lớn, nhiều nhóm phát triển song song
- Tăng tính linh hoạt, tái sử dụng và khả năng chịu lỗi

Nhược

- Phức tạp trong giao tiếp, bảo mật và giám sát
- Yêu cầu hệ thống CI/CD và DevOps mạnh
- Tốn chi phí vận hành và hạ tầng

# CẤU TRÚC MICROSERVICES

## VI. Các kiểu của cấu trúc Microservice

2. Serverless Microservice

- Tách biệt chức năng thành các service độc lập để tăng khả năng mở rộng, phát triển song song và triển khai linh hoạt
- Mỗi service là một tiến trình độc lập, có thể có CSDL riêng
- Giao tiếp giữa các service thông qua: HTTP, gRPC hoặc message queue
- Thường sử dụng API Gateway để quản lý truy cập từ bên ngoài

Trường hợp dung

- Hệ thống lớn nhiều team phát triển song song
- Các service có vòng đời và logic đặc biệt

Ưu

- Dễ mở rộng và phát triển độc lập từng service
- Phù hợp với tổ chức lớn, nhiều nhóm phát triển song song
- Tăng tính linh hoạt, tái sử dụng và khả năng chịu lỗi

Nhược

- Phức tạp trong giao tiếp, bảo mật và giám sát
- Yêu cầu hệ thống CI/CD và DevOps mạnh
- Tốn chi phí vận hành và hạ tầng

# CẤU TRÚC MICROSERVICES

## VI. Các kiểu của cấu trúc Microservice

3. Event-Driven Microservice

- Tăng tính linh hoạt và giảm độ phụ thuộc giữa các service bằng giao tiếp thông qua sự kiện
- Các service giao tiếp thông qua message broker: Kafka, RabbitMQ
- Service phản ứng với sự kiện thay vì gọi trực tiếp
- Thiếu kế hướng bất đồng bộ, phù hợp với hệ thống có nhiều tương tác

Trường hợp dung

- Hệ thống nhiều luồng xử lý song song, bất đồng bộ
- Cần xử lý sự kiện theo thời gian thực
- VD: hệ thống thanh toán, tracking đơn hang, phân tích dữ liệu

Ưu

- Tăng khả năng scale và tính linh hoạt
- Giảm coupling giữa các service
- Phù hợp với hệ thống phức tạp và nhiều luồng xử lý

Nhược

- Khó kiếm soát luồng dữ liệu và xử lý lỗi
- Debug phực tạp do tính bất đồng bộ
- Cần thiết kế schame sự kiện và quản lý thứ tự xử lý cẩn thận

# CẤU TRÚC MICROSERVICES

## VI. Các kiểu của cấu trúc Microservice

4. Domain-Oriented Microservice

- Thiết kế hệ thống theo bounded context trong Domain-Driven Design (DDD), phản ánh đúng nghiệp vụ thực tế
- Mỗi service gắn với một domain nghiệp vụ cụ thể
- Giao tiếp giữa các service theo ngữ cảnh rõ rang
- Tạch biệt dự liệu và logic theo từng domain

Trường hợp dung

- Hệ thống tài chính, bảo hiểm, quản lí chuỗi cung ứng
- Các tổ chức cõ nghiệp vụ rõ rang và phân chia theo domain

Ưu

- Phù hợp với hệ thống nghiệp vụ phức tạp
- Dễ mở rộng theo từng domain
- Tăng tính rõ rang và khả năng bảo trì

Nhược

- Cần hiểu sâu về DDD và kiến trúc hướng domain
- Thiết kế ban đầu phức tạp và tốn thời gian
- Dễ phát sinh trùng lặp nếu k kiểm soát tốt

# CẤU TRÚC MICROSERVICES

## VI. Các kiểu của cấu trúc Microservice

5. Hybrid Microservice

- Kết hợp nhiều kiểu microservice để tận dụng ưu điểm của từng loại, phù hợp với hệ thống đa dạng
- Một số service dung REST, số khác dung event hoặc serverless
- Linh hoạt trong cách triển khai và giao tiếp
- Tùy biến theo nhu cầu từng module hoặc nhóm nhiệp vụ

Trường hợp dung

- Hệ thống tích hợp nhiều dịch vụ hoặc nền tảng
- Các tổ chức cần xử lý cả tác vụ đồng bộ và bất đồng bộ
- VD: nền tảng thương mai điện tử tích hợp thanh toán, kho, vận chuyển

Ưu

- Tối ưu hiệu năng, chi phí và khả năng mở rộng
- Phù hợp với hệ thống có nhiều loại tác vụ khác nhau
- Dễ thích nghi với thay đổi công nghệ

Nhược

- Khó quản lý tổng thể hệ thống
- Cần chiến lược rõ rang để tránh hỗn loạn kiến trúc
- Yêu cầu đội ngũ kỹ thuật có kinh nghiệm đa nên tảng

# CẤU TRÚC MICROSERVICES

## VI. Các kiểu của cấu trúc Microservice

Ghi chú:

Modular Monolith: Thiết kế hệ thống theo mô-đun rõ rang trong một khối đơn, nhằm đơn giản hóa phát triển và chuẩn bị cho việc chuyển sang microservice sau này

- Một tiến trình duy nhất, chi thành các mô-đun logic
- Giao tiếp nội bộ qua hàm hoặc module
- Ưu
  - Dễ phát triển, kiểm thử và debug
  - Không cần xử lý các vấn đề phát tán
- Nhược
  - Khó mở rộng theo chiều ngang
  - Không tận dụng được các lợi ích của kiến trúc phân tán
- Trường hợp dung
  - Startup hoặc dự án nhỏ cần phát triển nhanh
  - Hệ thống có ít người dung, ít dịch vụ phưc tạp
  - Giai đoạn đầu của sản phẩm chưa cần scale lớn

Service Mesh

# CẤU TRÚC MICROSERVICES

## VI. Các kiểu của cấu trúc Microservice

Ghi chú:

Service Mesh: lớp hạ tầng quản lý giao tiếp giữa các service một cách tự động, bảo mật và có thể quan sát

- Sử dụng sidecar proxy để kiểm soát giao tiếp
- Tách biệt logic giao tiếp khỏi logic nghiệp vụ
- Ưu
  - Tăng cường bảo mật, giám sát và quản lý traffic
  - Không cần viết them code cho các chức năng giao tiếp
- Nhược
  - Tăng độ phức tạp hệ thống
  - Tốn tài nguyên và chi phí vận hành
- Trường hợp dung
  - Hệ thống microservice đã triển khau ở quy mô lớn
  - Cần quản lý traffic, bảo mật và giám sát chi tiết
  - VD: nên tảng SaaS, hệ thống tài chính, cloud-native apps

# CẤU TRÚC MICROSERVICES

## VII. Các mẫu thiết kế trongMicroservice

1. API Gateway (Quản lý truy cập và routing)

- Làm điểm vào duy nhất cho toàn hệ thống microservice.
- Đóng vai trò trung gian giữa client và các service backend, quản lý truy cập và điều phối luồng dữ liệu
- Chức năng
  - Routing: Định tuyến yêu cầu đến đúng service backend
  - Xác thực và phân quyền: Kiểm tra token, quyền truy cập
  - Caching: Lưu trữ tạm thời để giảm tải backend
  - Rate limiting: Giới hạn số lượng request từ client
  - Versioning: hỗ trợ nhiều phiên bản API
  - Load balancing: Phân phối tải đều giữa các instance backend

Trường hợp dung

- Khi có nhiều backend và muốn ẩn chi tiết triển khai với client
- Khi cần tang cường bảo mật, kiểm soát truy cập và tối ưu hiệu suất

Ưu

- Tăng độ bảo mật và kiểm soát: Giấu các service nội bộ, kiếm soát truy cập tập trung
- Giảm tải cho các service backend: nhờ caching, xác thực tập trung
- Hộ trợ versioning và load balancing: dễ dàng quản lý nhiều phiên bản API và phân phối tải

Nhược

- Điểm nghẽn tiềm ẩn: Nếu k tối ưu, API Gateway có thể trở thành bottlenck
- Tăng độ vận hành phức tạp cấu hình: Cần cấu hình và giám sát kỹ lưỡng

# CẤU TRÚC MICROSERVICES

## VII. Các mẫu thiết kế trongMicroservice

2. Circuit Breaker (Ngăn lỗi lan truyền)

- Ngăn không cho gọi service lỗi liên tục
- Tránh làm hệ thống quả tải hoặc sập dây chuyền
- Tăng khả năng tự phục hồi và ổn định của hệ thống

Trường hợp dung

- Một service phụ thuộc vào service khác
- Khi có nguy có cơ lỗi lan truyền giữa các service

Ưu

- Tăng độ ổn định hệ thống: Giảm thiểu tác động của lỗi cục bộ đến toàn bộ hệ thống
- Giúp hệ thống tự phục hồi: sau 1 khoảng thời gian, circuit breaker sẽ thử lại để xem service ok chưa

Nhược

- Cần cấu hình ngưỡng hợp lý: nếu sai có thể kịch hoạt quá circuit breaker sớm hoặc muôn
- Có thể bỏ qua các request hợp lệ: nếu circuit breaker đang mở, các request hợp hệ cũng sẽ bị tự chối

Các trạng thái

- Closed: Mọi request đều đc gửi đến service, nếu có lỗi vượt ngưỡng, chuyển sang trạng thái Open
- Open: Ngăn k có cho gửi request đến service, Sau 1 time, chuyển sang trạng thái Half-Open
- Half-Open: Cho phép một số request thử nghiệm, nếu thành công => Closed, nếu thất bại => Open

# CẤU TRÚC MICROSERVICES

## VII. Các mẫu thiết kế trongMicroservice

3. Saga (Quản lý giao dịch phân tán)

- Quản lý giao dịch phân tán giữa nhiều service
- Đảm bảo tính nhất quán mà k dung đến database transation

Trường hợp dung

- Khi một nhiệm vụ cần nhiều service cùng xử lý
- Khi k thể hoặc k muốn dung transation truyền thống

Ưu

- Giải quyết giao dịch phấn tán hiệu quả
- Tăng tính linh hoạt và mở rộng
- Có thể triển khai 2 mô hình: choreography và Orchestraion

Nhược

- Logic phức tạp, dễ sai nếu k kiểm soát tốt luồng nghiệp vụ và roll back
- Khó debug khi có nhiều rollback hoặc xử lý bất đồng bộ
- Cần thiết kế kỹ lưỡng để đảm bảo tính nhất quán cuối cùng

# CẤU TRÚC MICROSERVICES

## VII. Các mẫu thiết kế trongMicroservice

4. CQRS (Command Query Reponsibility Segregation – Tách đọc/ghi để tối ưu hiệu năng)

- Tạch riêng phần ghi (Command) và phần đọc (Query) để tối ưu hiện năng và mở rộng
- Cho phép sử dụng hai mô hình riêng biệt, phù hợp với từng mục đích

Trường hợp dung

- Khi hệ thống lượng đọc và ghi k cân bằng
- Khi cần scale phần đọc riêng biệt để đáp ứng nhu cầu truy cập
- Khi cần tối ưu hóa hiệu năng cho từng loại thao tác
- Khi hệ thống có logic nghiệp vụ phức tạp ở phần ghi nhưng phần đọc lại đơn giản

Ưu

- Tối ưu hóa hiệu năng đọc/ ghi: có thể tối ưu riêng cho phần đọc( cache, database chuyên cho truy vấn) và phần ghi (xử lý nghiệp vụ, xác thực)
- Dễ mở rộng từng phần: có thể scale độc lập từng phần
- Tách biệt rõ rang logic nghiệp vụ: giúp hệ thống dễ bảo trì và phát triển

Nhược

- Tăng độ phức tạp cho hệ thống: quản lý 2 mô hình dữ liệu và 2 luồng xử lý riêng biệt
- Khó đồng bộ dữ liệu: cần cơ chế đồng bộ giữa ghi/đọc
- Yêu cầu hiểu biết sâu: đội ngũ phát triển có kinh nghiệm để triển khai đúng cách

# CẤU TRÚC MICROSERVICES

## VII. Các mẫu thiết kế trongMicroservice

5. Event Sourcing (Lưu trạng thái qua sự kiện)

- Lưu trạng thái hệ thống bằng cách ghi lại chuỗi sự kiện đã xảy ra thay vì lưu trực tiếp trạng thái hiện tại
- Mỗi thay đổi trong hệ thống được biểu diễn bằng một sự kiện và trạng thái hiện tại có thể được tía tạo bằng cách replay các sự kiện này

Trường hợp dung

- Khi cần audit (kiểm tra lịch sử thay đổi)và rollback(quay lại trạng thái trước đó)
- Khi cần khôi phục trạng thái từ lịch sử sự kiển
- Khi xây dựng hệ thống event-driven hoặc CQRS

Ưu

- Dễ audit và rollback
- Tăng khả năng mở rộng và tích hợp event-driven
- Lưu được toàn bộ lịch sử thay đổi, phục vụ phân tích và kiểm tra

Nhược

- Khó hiểu và phức tạp trong quá trình phát triển, đặc biệt là xử lý logic hi replay sự kiện
- Tốn tài nguyên lưu trữ sự kiện
- Cập nhập hoặc sửa lỗi trong sự kiện cũ có thể gây rắc rối

# CẤU TRÚC MICROSERVICES

## VII. Các mẫu thiết kế trongMicroservice

6. Bulkhead (Cô lập lỗi giữa các phần)

- Tránh lỗi lan truyền trong hệ thống để tránh lan truyền sự cố
- Giới hạn tài nguyên cho từng service nhằm đảm bảo tính ổn định

Trường hợp dung

- Khi có nhiều service chạy song song và có thể ảnh hướng lẫn nhau
- Khi muốn giới hạn tài nguyên (CPU, RAM,…) cho từng service để tránh tình trạng một service tiêu tốn quá nhiều tài nguyên gây ảnh hướng đến service khác

Ưu

- Tăng độ ổn định cho hệ thống
- Giảm rủi ro lan truyền lỗi khi một service bị lỗi
- Dễ kiểm soát và giám sat từng phần riêng

Nhược

- Cần cấu hình tái nguyên hợp lý
- Tăng độ phức tạp vận hành

# CẤU TRÚC MICROSERVICES

## VII. Các mẫu thiết kế trongMicroservice

7. Service Discovery

- Cho phép các microservice tự động tìm thấy nhau mà k cần cấu hình địa chỉ IP hoặc hostname cố định

Trường hợp dung

- Triển khai hệ thống trên môi trường động:
  - Kubernetes
  - Docker Swam
  - Cloud (AWS, GCP, Azure)

Ưu

- Tự động hóa việc định tuyến giữa các service
- Giảm lỗi cấu hình thủ công
- Tăng khả năn mở rộng: dễ them/ bớt instance

Nhược

- Cần thêm thành phần trung gian:
  - Consul, Eureka, Zookeeper hoặc Kubernetes DNS
- Tăng độ phức tạp
- Phải xử lý các vấn đề: load balancing, health check, failover

# CẤU TRÚC MICROSERVICES

## VII. Các mẫu thiết kế trongMicroservice

8. Database per Service

- Mỗi microservice có CSDL riêng biệt, k chia sẻ schema hay table

Trường hợp dung

- Khi muốn tách biệt domain logic rõ rang
- Khi cần triển khai độc lập từng service
- Khi muốn tránh coupling dữ liệu giữa các service

Ưu

- Tăng tính độc lập: service có thể deloy, scale, rollback riêng
- Bảo mật tốt hơn: giới hạn quyền truy cập dữ liệu
- Dễ mở rộng: mỗi service có thể dung DB phù hợp

Nhược

- Khó thực hiện truy vấn liên bảng: JOIN giữa các service
- Phức tạp khi cần đồng bộ dữ liệu
- Cần Saga hoặc Event Sourcing để xử lý giao dịch phân tán

# CẤU TRÚC MICROSERVICES

## VII. Các mẫu thiết kế trongMicroservice

9. Logging & Monitoring

- Theo dõi hoạt động, hiệu năng và lỗi của hệ thống microservice

Trường hợp dung

- Khi hệ thống có nhiều service cần
  - Debug lỗi
  - Phân tích hiệu năng
  - Cảnh báo sự số

Ưu

- Hiển thị toàn bộ cảnh hệ thống: dễ phát hiện bottleneck
- Tăng độ tin cậy: phát hiện lỗi sớm
- Hỗ trợ Devv Ops/ORS: dễ tích hợp CI/CD, cảnh báo tự động

Nhược

- Tốn tài nguyên lư trữ và xử lý log
- Cần thiết kế log format chuẩn hóa giữa các service
- Cần triển khai thêm công cụ:
  - Prometheus + Grafana
  - ELK Stack (Elasticsearch, Logstash, Kibana)
  - Jaeger / Zipkin (tracing)

# 1. Kiến trúc Microservice

- Service chính: auth-service, chat-service, api-gateway, eureka, config
  - Auth: quản lý xác thực, đăng nhập, đăng ký, quản lý thông tin người dung
  - Chat: quản lý tin nhắn, lưu trữ
  - Eureka: Cho phép các service đăng ký, tìm kiếm lẫn nhau
  - Config: quản lý cấu hình
  - Api-gateway: định tuyền các request đến service

# 2. Eureka Server (Eureka-Service)

- Vai trò: Nơi các microservice đăng kí khi khởi động.
- Cách hoạt động:
  - Mỗi service khác đều có annotiation @EnableEurekaClinet
  - Eureka tự động cập nhật trạng thái của các service
- Công dụng: Giúp các service tìm thấy nhay mà k cần hardcode IP/port
- Ví dụ: chat-service gọi auth-service để xác thực user. K cần biết auth chạy ở đâu, chỉ cần gọi http://auth-service

# 3. Config Server (Cấu hình tập trung)

- Vai trò: lưu trữ cấu hình cho tất cả service ở 1 nơi.
- Cách hoạt động:
  - Các service sử dụng annotation @EnableConfigServer ở server và spring.config.import =optional:configserver: ở client
- Công dụng
  - Dễ cấu hình
  - Thay đổi cấu hình k cần build lại

# 4. API Gateway

- Vai trò: trung tâm định tuyến, bảo mật, logging, load balancing
- Cách hoạt động:
  - Cấu hình trong application
  - Nhận request từ client kiểm tra token, và chuyển tiếp request để service tương ứng
- Công dụng:
  - Bảo mật tập trung.
  - Giấu thông tin nội bộ service
  - Giảm tải cho service con
- Ví dụ:

# 5. Bảo mật & Phân quyền với JWT

- Cách hoạt động:
  - Đăng nhập → cấp token chứa username, role,...
  - Client gửi token
  - Service ktra token xác định phân quyền
- Công dụng:
  - Tránh phải lưu trên session
  - Gửi kèm mọi request để xác thực
- Sử dụng @PreAuthorize và cấu hình security

# 6. Các chức năng chính

- Auth-service
  - Đăng ký / đăng nhập
  - Cập nhật user
  - Xóa user (role-admin)
  - Tìm kiếm (role-admin)
- Chat-service
  - Gửi tin nhắn
  - Xem lịch sử

# So sánh 4 phương pháp chia Microservices phổ biến

| Phương pháp | Đặc điểm chính | Trường hợp dùng | Ưu điểm | Nhược điểm |
|---|---|---|---|---|
| Subdomain (DDD) | Chia theo Bounded Context dựa trên mô hình Domain-Driven Design | Hệ thống lớn, nghiệp vụ phức tạp, nhiều team, cần bám sát mô hình nghiệp vụ lâu dài | Rõ ràng về phạm vi nghiệp vụ; Giảm phụ thuộc giữa service; Dễ mở rộng, bảo trì | Cần hiểu sâu nghiệp vụ; Thời gian phân tích ban đầu lâu; Đòi hỏi team có kinh nghiệm DDD |
| Business Capability | Chia theo chức năng nghiệp vụ mà doanh nghiệp cung cấp | Khi bắt đầu thiết kế hoặc tách monolith, cần dễ hiểu cho cả kỹ thuật và business | Dễ trình bày với stakeholder; Nhanh xác định phạm vi dịch vụ; Dễ mapping sang Subdomain | Có thể chưa tối ưu về kỹ thuật; Dễ bị chồng chéo nếu không refine theo DDD |
| Transaction Boundaries | Chia theo phạm vi giao dịch để đảm bảo tính nhất quán dữ liệu | Hệ thống giao dịch tài chính, thanh toán, thương mại điện tử, nơi dữ liệu phải đồng bộ và chính xác | Đảm bảo tính nhất quán dữ liệu; Giảm rollback phức tạp | Có thể dẫn tới service lớn hơn mong muốn; Tăng độ phức tạp khi giao tiếp giữa service |
| Entity/Event-Driven | Chia theo Entity hoặc Event chính trong hệ thống, tập trung vào sự kiện | Hệ thống có nhiều luồng bất đồng bộ, realtime, event streaming (Kafka, RabbitMQ, IoT, log system) | Giảm coupling giữa service; Dễ mở rộng theo hướng sự kiện; Tăng khả năng xử lý bất đồng bộ | Khó debug khi lỗi; Yêu cầu hệ thống message broker ổn định; Tăng chi phí hạ tầng |