**Chủ đề: Tìm hiểu về kiến trúc microservice**

Người thực hiện: Hoàng Mạnh Dũng – LA 31 PM

Hà Nội – 2023

PHỤ LỤC

[MỞ ĐẦU 3](#_Toc52237949)

[CHƯƠNG I: TỔNG QUÁT KIẾN TRÚC MICROSERVICE 4](#_Toc52237950)

[1. Kiến trúc Monolithic 4](#_Toc52237951)

[2. Kiến trúc Microservice 5](#_Toc52237952)

[2.1. Tổng quan kiến trúc Microservice 5](#_Toc52237953)

[2.2. Đặc điểm của kiến trúc Microservice 10](#_Toc52237954)

[CHƯƠNG II: ỨNG DỤNG KIẾN TRÚC MICROSERVICE 12](#_Toc52237955)

[1. Một số lưu ý khi sử dụng kiến trúc Microservice 12](#_Toc52237956)

[2. Ứng dụng kiến trúc Microservice 13](#_Toc52237957)

[KẾT LUẬN 24](#_Toc52237958)

[TÀI LIỆU THAM KHẢO 25](#_Toc52237959)

# **MỞ ĐẦU**

Thời gian gần đây, đi liền với sự nổi dậy của cloud computing, microservice cũng đang dần trở thành một từ khóa hot. Microservice hiện nay được đề cập tới trong thế giới phần mềm, công nghệ được kỳ vọng cao và đánh giá như một xu hướng cho tương lai (Open API, service provider, …) và đang nhanh chóng hướng tới đỉnh cao của kỳ vọng tăng cao trên chu kỳ Gartner Hype. Một số cho rằng , microservice không có gì mới lạ, chẳng qua nó là SOA (kiến trúc hướng dịch vụ) được đánh bóng, đổi tên mà thôi.

Mặc cho mọi lời khen chên thì thực sự hiện nay kiến trúc Microservice vẫn đem lại lới ích khi nó giúp phương pháp agile thực sự hiệu quả và xây dựng được giải pháp phần mềm doanh nghiệp phức tạp.

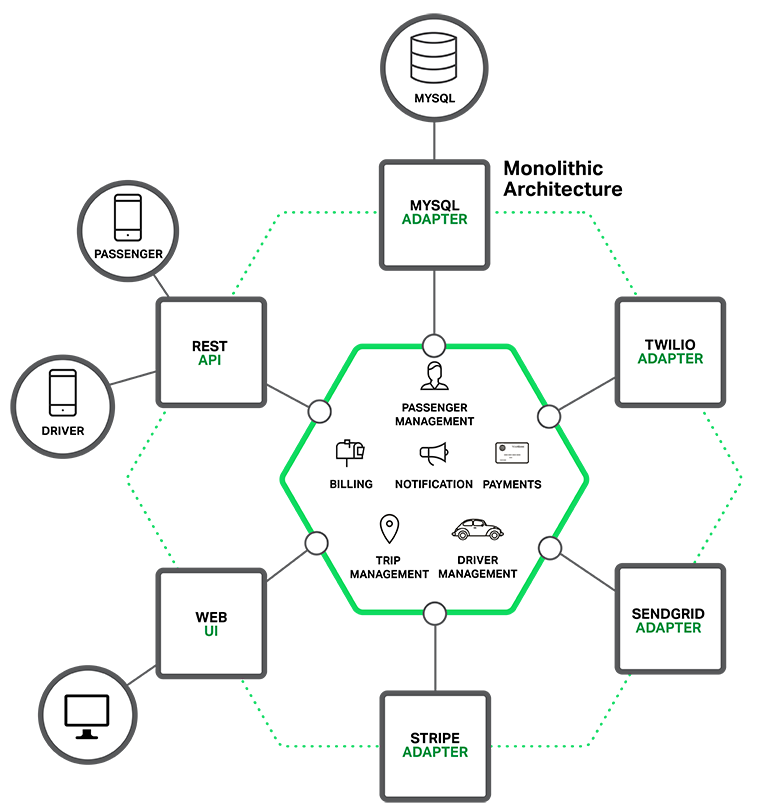
Mong rằng bài tìm hiểu dưới đây giúp cho mọi người có cái nhìn tổng quan nhất về kiến trúc Microservice và những lợi ích cũng như hạn chế mà nó mang lại.

# **CHƯƠNG I: TỔNG QUÁT KIẾN TRÚC MICROSERVICE**

1. **Kiến trúc Monolithic**

Trong quá khứ, người ta xây dựng hệ thống theo một khối (Monolithic Applications). Ứng dụng phần mềm doanh nghiệp được thiết kế để đáp ứng nhiều yêu cầu kinh doanh của doanh nghiệp. Do đó, các phần mềm cung cấp hàng trăm các tính năng và tất cả những tính năng này đều được gói trong một ứng dụng monolithic. Ví dụ, ERPs, CRMs hay nhiều phần mềm khác chứa hàng trăm tính năng. Việc triển khải, sử lỗi, mở rộng và nâng cấp những phần mềm khổng lồ này trở thành một cơn ác mộng.

Thử tưởng tượng bạn đang phải xây dựng một dịch vụ gọi taxi qua di động nhằm cạnh tranh với Uber và Grab. Sau một số buổi họp thu thập yêu cầu và phân tích thiết kế, bạn sẽ chọn công nghệ (technology stack) rồi tạo dự án bằng các framework đại loại như Rails, Spring Boot, Play. Dự án này sẽ có kiến trúc chia khối lục giác (hexagol architecture) hay tổng quát hơn là khối đa diện. Kiến trúc đa diện giúp ứng dụng chuyên biệt mô hình dữ liệu.



Như hình trên là một sơ đồ mô tả dịch vụ gọi taxi qua di động, được mô tả qua các khối dịch vụ: đối tượng cho vùng nghiệp vụ (domain objects) và các sự kiện (events: khách đặt xe, khách hủy xe, xe nhận khách...) Xung quanh lõi là bộ chuyển đổi (adapter) ví dụ như kết nối vào cơ sở dữ liệu, gửi nhận thông điệp (messaging), web service hoặc giao diện web front end.

Trong nhiều trường hợp, người ta có thể xây dựng các services độc lập nhưng chúng lại được triển khai chung. Mặc dù có cấu trúc module hóa hợp lý, nhưng ứng dụng kiểu này sẽ đóng gói và cài đặt thành một khối (monolithic). Mã chạy cụ thể tùy thuộc vào ngôn ngữ lập trình hay thư viện framework.

Tương tự như ứng dụng monolithic, những services này to và phức tạp lên theo thời gian vì thường xuyên thêm các tính năng. Và thế là những ứng dụng này lại trở thành một mớ các services monolithic, cũng không còn khác mấy so với kiến trúc một khối thông thường. Hình trên thể hiện một ứng dụng gồm nhiều services. Những services này được triển khai cùng một lúc vào 1 ứng dụng lớn. Dù bên trong có gồm các services thì đây là một ứng dụng monolithic. Một số tính chất của kiến trúc monolithic:

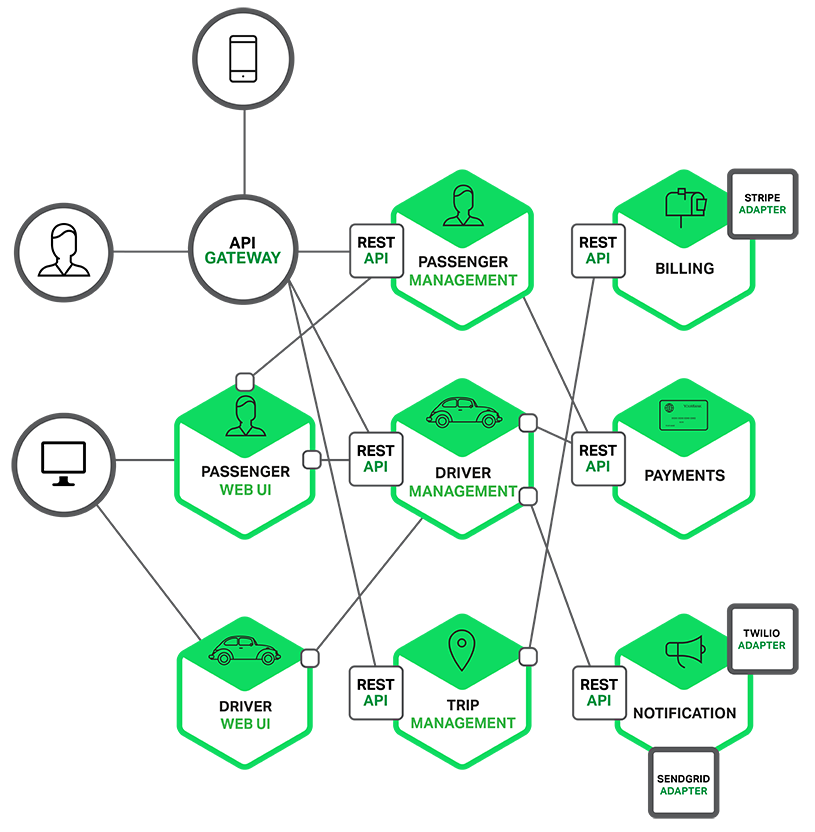
* Được thiết kế và phát triển theo 1 khối duy nhất
* Ứng dụng monolithic thức tạp và to gây sẽ khó khăn cho việc bảo trì( triển khia toàn bộ hệ thống dù chỉ cập nhật hay bảo trì một phần nhỏ…)
* Khi mở rộng có thể phải yêu cầu thêm nhiều tài nguyên khác nhau (như service này phải thêm cpu, service kia phải thêm memory..)
* Một lỗi nhỏ có thể đánh sập cả hệ thống
* Khó đổi mới vì cả hệ thống (tất cả các service) dùng chung một công nghệ

1. **Kiến trúc Microservice**
   1. Tổng quan kiến trúc Microservice

Nền tảng cả kiến trúc microservices là xây dựng một ứng dụng mà ứng dụng này là tổng hợp của nhiều services nhỏ và độc lập có thể chạy riêng biệt, phát triển và triển khai độc lập.

Quay lại ứng dụng gọi xe bên trên, ta có thể giải quyết các vấn đề của ứng dụng một khối bằng kiến trúc microservices (nhiều dịch vụ nhỏ). Ý tưởng là chia nhỏ ứng dụng lớn ra thành các dịch vụ nhỏ kết nối với nhau.

Mỗi dịch vụ nhỏ thực hiện một tập các chức năng chuyên biệt như quản lý đơn hàng, quản lý khách hàng. Mỗi dịch vụ là một ứng dụng nhỏ có kiến trúc đa diện lõi là business logic kết nối ra các adapter khác nhau. Một số dịch vụ nhỏ lộ ra giao tiếp lập trình API cho dịch vụ nhỏ khác hay ứng dụng client gọi tới. Khi vận hành, mỗi dịch vụ nhỏ được chạy trong một máy ảo hoặc Docker container.

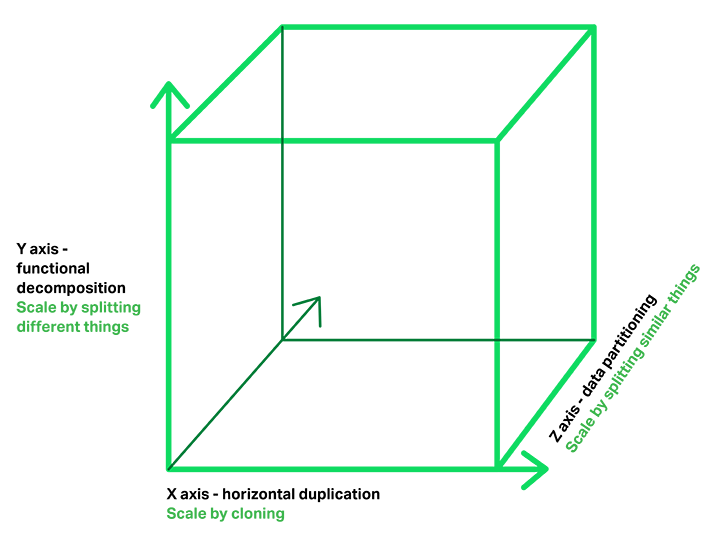


Mỗi vùng chức năng được chia nhỏ và thực thi bởi một ứng dụng nhỏ. Ứng dụng web cũng có thể chia nhỏ hơn chuyên cho từng đối tượng(khách hàng/tài xế). Thiết kế giao diện cho tứng loại người dùng để có trải nghiệm tốt hơn, tốc độ nhanh hơn.

Điều quan trọng chính là nhìn vào các tính năng trong một ứng dụng monolithic, ta có thể nhận biết, xác định các yêu cầu và khả năng cần thiết để đáp ứng một nghiệp vụ. Sau đó từng nghiệp vụ này sẽ được xây dựng thành những service nhỏ, độc lập. Những services này có thể sử dụng các nền tảng công nghệ khác nhau và phục vụ một mục đích cụ thể và có giới hạn.

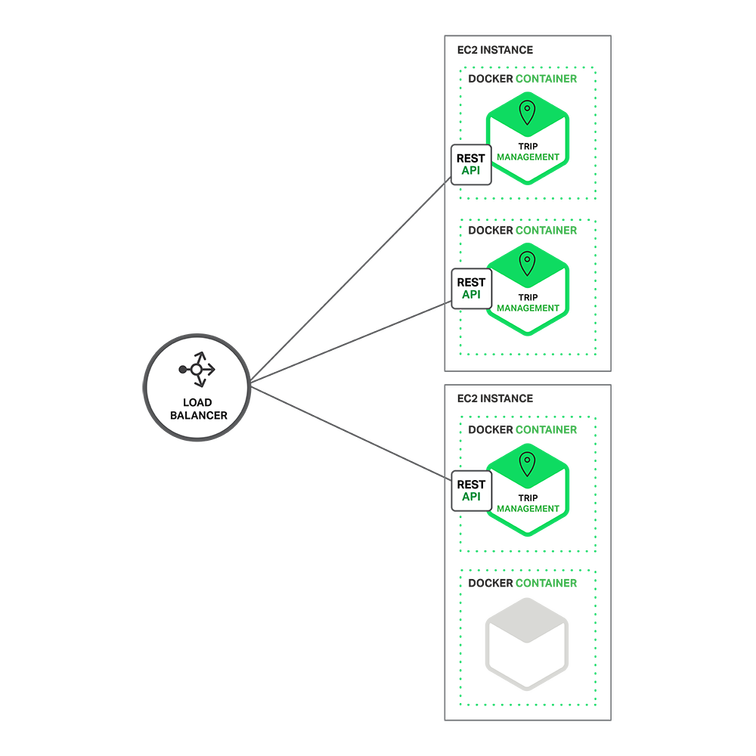
Mỗi dịch vụ phụ trợ cho thấy các REST API và hầu hết các dịch vụ đều consume các API do các dịch vụ khác cung cấp. Ví dụ: Driver Management sử dụng Notification server để cho tài xế đang rảnh biết về chuyến đi tiềm năng. Các dịch vụ UI gọi các dịch vụ khác để hiển thị các trang web. Các dịch vụ cũng có thể sử dụng giao tiếp không đồng bộ, dựa trên thông điệp (message-based communication)

Một số API REST cũng được tiếp xúc với các ứng dụng dành cho thiết bị di động dành cho tài xế và hành khách. Tuy nhiên, các ứng dụng không có quyền truy cập trực tiếp vào các dịch vụ phụ trợ. Thay vào đó, giao tiếp được trung gian bởi một thành phần trung gian được gọi là API Gateway. API Gateway chịu trách nhiệm về các nhiệm vụ như cân bằng tải, bộ nhớ đệm, kiểm soát truy cập, đo lường API và monitoring



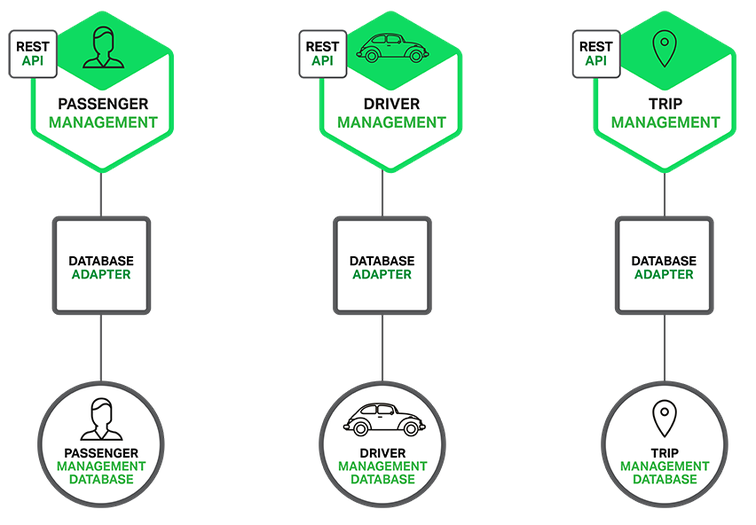
Mẫu Kiến trúc Microservices tương ứng với tỷ lệ trục Y của Scale Cube, là mô hình 3D có khả năng mở rộng từ cuốn sách tuyệt vời có tên *The Art of Scalability*. Hai trục tỷ lệ khác là chia tỷ lệ trục X, bao gồm chạy nhiều bản sao giống hệt nhau của ứng dụng phía sau bộ cân bằng tải và chia tỷ lệ trục Z (hoặc phân vùng dữ liệu), trong đó thuộc tính của yêu cầu (ví dụ, khóa chính của một hàng hoặc danh tính của một khách hàng) được sử dụng để định tuyến yêu cầu tới một máy chủ cụ thể.

Các ứng dụng thường sử dụng ba loại chia tỷ lệ với nhau. Trục Y chia tỷ lệ decomposes các ứng dụng thành microservices như được hiển thị ở trên trong hình đầu tiên trong phần này. Khi chạy, quy mô trục X sẽ chạy nhiều instances của từng dịch vụ phía sau bộ cân bằng tải cho thông lượng và tính khả dụng. Một số ứng dụng cũng có thể sử dụng chia tỷ lệ trục Z để phân vùng dịch vụ. Diagram sau đây cho thấy Trip Management service có thể được triển khai với Docker chạy trên Amazon EC2 như thế nào.



Khi chạy, Trip Management service bao gồm nhiều service instances. Mỗi service instance là một Docker container. Để có tính sẵn sàng cao, các containers đang chạy trên nhiều Cloud VMs. Ở phía trước của các service instances là một bộ cân bằng tải phân phối các yêu cầu trên các cá thể. Trình cân bằng tải cũng có thể xử lý các mối quan tâm khác như caching, access control, API metering, and monitoring.

Mô hình Microservices tác động đáng kể đến mối quan hệ giữa ứng dụng và cơ sở dữ liệu. Thay vì chia sẻ một lược đồ cơ sở dữ liệu duy nhất với các dịch vụ khác, mỗi dịch vụ có lược đồ cơ sở dữ liệu riêng của nó. Một mặt, cách tiếp cận này là mâu thuẫn với ý tưởng của một mô hình dữ liệu toàn doanh nghiệp. Ngoài ra, nó thường dẫn đến trùng lặp một số dữ liệu. Tuy nhiên, có một lược đồ cơ sở dữ liệu cho mỗi dịch vụ là điều cần thiết nếu bạn muốn hưởng lợi từ microservices, bởi vì nó đảm bảo các khớp nối lỏng lẻo (loose coupling). Diagram sau đây cho thấy database architecture cho ứng dụng ví dụ.



Mỗi dịch vụ đều có cơ sở dữ liệu riêng. Hơn nữa, một dịch vụ có thể sử dụng một loại cơ sở dữ liệu phù hợp nhất với nhu cầu của nó, cái gọi là kiến ​​trúc bền bỉ đa điểm (polyglot persistence architecture). Ví dụ: Driver Management cần tìm tài xết gần với hành khách tiềm năng, phải sử dụng cơ sở dữ liệu hỗ trợ truy vấn địa lý hiệu quả.

Nhìn bề ngoài, mô hình Microservices tương tự như SOA. Với cả hai cách tiếp cận, kiến ​​trúc bao gồm một tập hợp các dịch vụ. Tuy nhiên, có thể coi Microservices như là SOA nhưng không có sự thương mại hóa và dấu hiệu nhận biết về các đặc tả dịch vụ web (WS-\*) và Service Service Bus (ESB). Các ứng dụng dựa trên microservice có các giao thức đơn giản, nhẹ hơn như REST, chứ không phải WS-\*. Microservices cũng rất tránh sử dụng ESB và thay vào đó thực hiện chức năng giống như ESB trong bản thân microservices. Mô hình Microservices cũng loại bỏ các phần khác của SOA, chẳng hạn như khái niệm lược đồ chuẩn (canonical schema).

* 1. Đặc điểm của kiến trúc Microservice

a) Đặc điểm

* Tập hợp một nhóm nhỏ các service: mức độ chi tiết của một service là nhỏ và mỗi service này sẽ chịu một trách nhiệm cụ thể (single responsiblity) và chỉ tập trung vào nhiệm vụ đó. Ví dụ: storage service sẽ chịu riêng trách nhiệm về lưu trữ
* Việc phát triển và mở rộng một service là hoàn toàn độc lập. Điều này mang lại tính linh hoạt cho hệ thống . Quá trình deliver feature, release version sẽ dễ dàng và nhanh chóng. Hơn nữa sẽ không còn tình trạng bị block như ở mô hình monolithic
* Giảm tải được các mối quan ngại về công nghệ sử dụng. Chọn một công nghệ phù hợp với vấn đề của doanh nghiệp có thể được giải quyết dễ dàng. Các service giap tiếp với nhau thông qua API, do vậy mỗi service có thể dùng một ngôn ngữ riêng biệt. Serivce A dùng Java, Service B dùng Javascript ...
* Đối với team, microservice đem lại tính độc lập và tự quản lí cho team. Một team sẽ có trách nhiệm toàn bộ với life-cycle của một hay nhiều service. Họ làm việc trong việc context biệt lâp, có thể tự quản lí các quyết định của mình.

b) Các thuộc tính của mô hình Microservice

* Autonomous (tính tự trị): 1 service sẽ là 1 đơn vị chức năng, cung cấp API để thực hiện việc trao đổi, giao tiếp với các service khác
* Isolated (tính biệt lập): 1 serivce sẽ là 1 đơn vị triển khai. Nó có thể được chỉnh sửa, test và deployed như 1 đơn vị mà không ảnh hưởng đến những khía cạnh khác.
* Elastic: 1 service là phi trạng thái (stateless) vì vậy nó có thể scale tùy ý khi cần thiết.
* Resilient: 1 microservice sẽ được thiết kế để chấp nhận các lỗi, các rủi ro có thể xảy ra, các lỗi này là các lỗi có thể chấp nhận được
* Responsive: respond cho các request trong khoảng thời gian hợp lý.
* Intelligent: Tính thông minh ở đây tức là muốn nhắc đến việc hệ thống có thể tìm thấy các endpoint của các microservice đã được đăng kí.
* Message Oriented: Mô hình micro-service hoạt động dựa trên giao thức HTTP hoặc message bus để tạo nên sự giao tiếp giữa các service. Điều này đảm bảo tính loose coupling, tính biệt lập và có thể cung cấp lỗi dưới dạng message
* Programmable: Cung cấp API's cho phép truy cập bởi developer và administrator.
* Composable: Bao gồm nhiều microservices.
* Automated: Lifecycle của Microservice được quản lý thông qua automation bao gồm development, build, test, staging, production và distribution.)

c) Ưu điểm của kiến trúc Microservice

* Dễ nâng cấp và scale, đây là điều quan trọng nhất. Giả sử với một trang web bán hàng cần xuất nhiều hóa đơn, chỉ việc nâng cấp server cho service order và report. Việc nâng cấp server với mô hình microservice rất dễ dàng thực hiện. Điều này rất khó thực hiện với monolithic
* Do tách biệt nên nếu một service bị lỗi, toàn bộ hệ thống vẫn hoạt động bình thường. Với monolith, một module bị lỗi có thể sẽ kéo theo toàn bộ hệ thống bị sập
* Với monolithic, các module sử dụng chung 1 ngôn ngữ/framework. Với microservice, các service nằm tách biệt nhau, bạn có thể thoải mái sử dụng ngôn ngữ lập trình riêng, database riêng. VD service xử lý ảnh có thể viết bằng C++, service tổng hợp data có thể viết bằng Python
* Khả năng testing dễ dàng hơn - các services nhỏ hơn và nhanh hơn để test
* Cải thiện khả năng bảo trì - mỗi service tương đối nhỏ do đó dễ hiểu và thay đổi hơn
* Dễ dàng hơn trong việc tích hợp 3rd-party
* Mỗi service có dung lượng lưu trữ riêng và có thể có cơ sở dữ liệu riêng.

d) Nhược điểm của kiến trúc Microservice

* Các module giao tiếp qua mạng nên có thể tốc độ không cao bằng monolith. Ngoài ra, mỗi module phải tự giải quyết các vấn đề về bảo mật, transaction, lỗi kết nối, quản lý log files.
* Mỗi service sử dụng một database riêng, việc đảm bảo tính đồng nhất trong dữ liệu sẽ trở nên phức tạp.
* Sử dụng nhiều service nên việc theo dõi, quản lý các service này sẽ phức tạp hơn. Do vậy, một số tool/công nghệ ra đời để phục vụ cho việc này (Docker, Ansible, Salt, Octopus, …).
* Cần một đội ngũ có khả năng và trình độ: Software Architect để phân tách module, Techinical Leader để setup workflow, IT/DevOps để setup CI/CD , deploy lên cloud….

**CHƯƠNG II: ỨNG DỤNG KIẾN TRÚC MICROSERVICE**

1. **Một số lưu ý khi sử dụng kiến trúc Microservice**

Kiến trúc Microservice đã được nhiều công ty lớn áp dụng như Amazon, Netflix, chứng tỏ rằng nó hiệu quả, giải quyết được vấn đề. Tuy nhiên, điều đó không có nghĩa chúng ta nên làm theo các công ty lớn. Để đơn giản, trước hết hãy tập trung phát triển ứng dụng dạng monolith, với nhiều module riêng biệt trước. Khi vai trò của các module đã rõ ràng, ta có thể dần tách cách module này ra thành các service riêng

Phải biết giới hạn chức năng của từng service, tập trung vào một nhiệm vụ giúp quá trình phát triển và triển khai dịch vụ trở nên nhanh tróng hơn

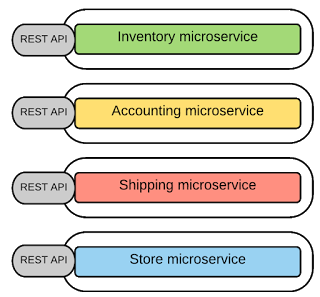
Chúng ta thường sử dụng kiến trúc microservice khi:

* Ứng dụng có phạm vi lớn và bạn xác định các tính năng sẽ được phát triển rất mạnh theo thời gian. Ví dụ: cửa hàng thương mại điện tử trực tuyến, dịch vụ truyền thông xã hội, dịch vụ truyền phát video với số lượng người dùng lơn, dịch vụ cung cấp API,…
* Team-size lớn, có đủ thành viên để phát triển các component riêng lẻ một cách hiệu quả.
* Mặt bằng kỹ năng của team tốt và các thành viên tự tin về các mẫu thiết kế microservice nâng cao.
* Thời gian để đem đi marketing không quan trọng. Kiến trúc microservice sẽ mất nhiều thời gian hơn để hoạt động được.
* Bạn sẵn sàng chi nhiều hơn cho cơ sở hạ tầng, giám sát,… để nâng cao chất lượng sản phẩm.
* Tiềm năng về người dùng lớn và bạn kỳ vọng số lượng người dùng sẽ phát triển. Ví dụ một phương tiện truyền thoong xã hội nhắm mục tiêu là người dùng trên toàn thế giới.

1. **Ứng dụng kiến trúc Microservice**

*Liên lạc giữa Microservices*: Trong ứng dụng monolithic, các chức năng khác nhau nằm trong các component khác nhau được kết nối bằng cách gọi hàm hay phương thức. Trong SOA, việc này được chuyển sang một chế độ tách rời hơn với kiểu nhắn tin qua các dịch vụ web (web service messaging), phần lớn dùng SOAP trên nền phương thức HTTP, JMS. Những webservices này khá phức tạp. Với microservices, yêu cầu là phải có một cơ chế truyền tin đơn giản và nhẹ.

*Gửi Tin Đồng Bộ - REST, Thrift*:Với truyền tin đồng bộ (người gửi - client sẽ chờ một khoảng thời gian để nhận kết quả từ service), REST là sự lựa chọn hàng đầu vì nó cung cấp hệ thống truyền tin đơn giản qua giao thức HTTP dạng request - response. Do đó, nhiều microservices sử dụng HTTP với API. Mỗi chức năng xuất ra API.



Ngoài REST, [Thrift](https://thrift.apache.org/) cũng được sử dụng để truyền tin đồng bộ.

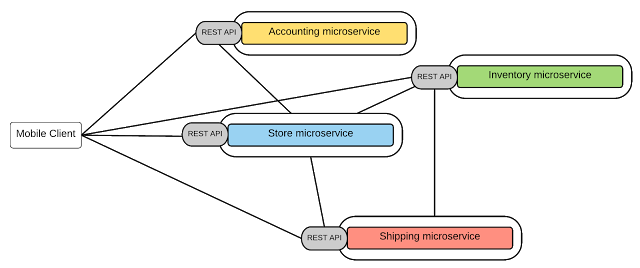
*Gửi Tin Bất Đồng Bộ - AMQP, STOMP, MQTT*: Trong một số hoàn cảnh, truyền tin bất đồng bộ là cần thiết (client không mong đợi response ngay lập tức, hoặc không cần response). Các giao thức truyền tin bất đồng bộ như AMQP, STOMP hay MQTT được sử dụng rộng rãi.

*Các Kiểu Tin Nhắn - JSON, XML, Thrift, ProtoBuf, Avro*: Quyết định kiểu tin nhắn phù hợp cho microservices cũng là một yếu tố quan trọng. Với phần lớn các ứng dụng microservices, họ sử dụng những kiểu tin nhắn dạng chữ như JSON và XML trên nền giao thức HTTP với API. Trong trường hợp cần truyền tin dạng nhị phân, microservices có thể dùng dạng Thrift, Proto hay Avro.

*Service Contracts – Định Nghĩa Service Interfaces – Swagger, RAML*: Bởi vì chúng ta xây dựng microservices trên kiểu kiến trúc REST, ta có thể sử dụng cùng kiểu REST API để định nghĩa hợp đồng của microservices. Do đó, microservices sử dụng các ngôn ngữ định nghĩa REST API tiêu chuẩn như Swagger, RAML để định nghĩa hợp đồng dịch vụ.

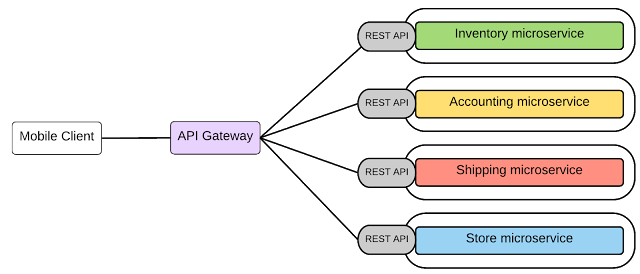
*Kết Nối Microservices (Giao Tiếp Giữa Các Services)*: Trong kiến trúc microservices, ứng dụng phần mềm được cấu thành từ các dịch vụ độc lập. Để hoàn thành một tác vụ của người dùng trên phần mềm, kết nối và giao tiếp giữa các microservices là cần thiết vì tác vụ gồm nhiều tác động khác nhau lên các services. Vì vậy, giao tiếp giữa các microservices là một vấn đề cực kì quan trọng. Bởi vì microservices sử dụng giao thức tiêu chuẩn HTTP, JSON, ..., yêu cầu để kết nối một giao thức khác biệt được tối giản. Một lựa chọn khác cho giao tiếp giữa microservices là sử dụng một message bus nhẹ hay gateway với khả năng định tuyến tối thiểu hoạt động như những đường truyền đơn thuần không xử lý business logic gì hết (dump pipe). Dựa vào những phong cách này, có một số kiểu mẫu giao tiếp trong microservices như dưới đây.

* Point-to-point - Kết Nối Trực Tiếp Giữa Các Services: Với kiểu điểm nối điểm, toàn bộ logic của việc định tuyến truyền tin nhắn nằm trong mỗi điểm cuối (endpoint) hay chính là các services. Và services nói chuyện trực tiếp với nhau. Mỗi service mở ra một REST APIs và bất kì service hay khách hàng bên ngoài nào cũng có thể gọi service qua REST API của nó.



Rõ ràng là mô hình này đơn giản và hoạt động ổn với ứng dụng microservices tương đối nhỏ nhưng khi số lượng services tăng lên, việc giao tiếp trở nên cực phức tạp. Lí do này cũng chính là tác dụng của ESB trong SOA truyền thống, chính để loại bỏ kết nối trực tiếp phức tạp và rối rắm. Tổng hợp một số vấn đề của kiểu giao tiếp trực tiếp.

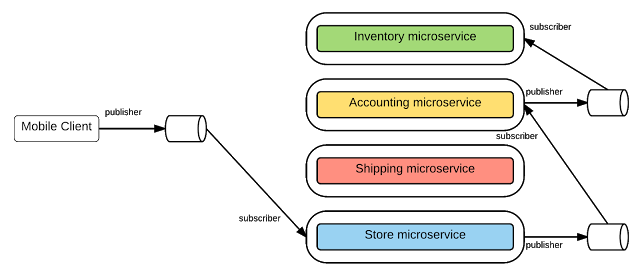
* Những yêu cầu như xác thực người dùng, điều tiết, giám sát,...phải được xây dựng tại tất cả microservices.
* Việc trên dẫn đến lập các tính năng chung, mỗi microservices có thể trở nên phức tạp.
* Không có cách quản lý, kiểm soát giao tiếp giữa các services
* Thường việc kết nối trực tiếp trong microservices được coi là anti-pattern khi áp dụng cho ứng dụng to.
* API-Gateway: cho phép bạn sử dụng một API được quản lý qua REST/HTTP. Vì thế, chúng ta có thể cung cấp tất cả các chức năng nghiệp vụ được phát triển thành microservices qua API Gateway như một APIs tập trung được quản lý.



Như hình trên tất cả microservices đều được xuất qua API Gateway và đây là một chốt đầu vào duy nhất cho người dùng. Nếu một microservice muốn giao tiếp với một microservice khác thì cần đi qua API Gateway.

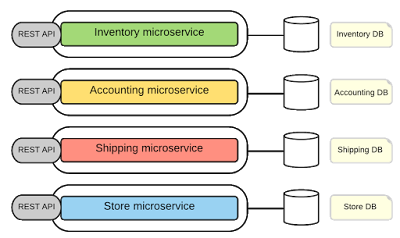
API Gateway cung cấp các lợi thế dưới đây:

* Cung cấp một lớp trừu tượng hóa các microservices. Ví dụ thay vì cung cấp một API cho tất cả khách hàng, API gateway có thể xuất ra hay hiển thị API khác nhau cho mỗi khách hàng.
* Định tuyến và chuyển đổi tin nhắn gọn nhẹ ở cấp gateway.
* Một điểm tập trung cho các chức năng chung không mang tính nghiệp vụ kinh doanh như bảo mật, giám sát và điều tiết.
* Với API Gateway. microservices trở nên càng gọn nhẹ vì các chức năng chung không mang tính nghiệp vụ đều chuyển sang Gateway.
* API Gateway có thể là kiểu mẫu được sử dụng rộng rãi nhất trong triển khai microservices.
* Message Broker - Người truyền tin trung gian: Microservices có thể được kết nối qua truyền tin bất đồng bộ như yêu cầu một chiều (requests) hay thông báo/ đăng kí nhận thông báo (publish/ subscribe) qua queues hay topics. Với publish/ subscribe, một service có thể tạo tin nhắn và gửi bất đồng bộ đến một hàng chờ (queue hay topic). Sau đó service khác có thể nhận tin này từ queue hay topic. Phong cách này tách biệt người gửi và người nhận, và người truyền tin trung gian sẽ lưu tin nhắn đến khi người nhận có thể xử lý. Người gửi hoàn toàn không biết gì về người nhận.



Giao tiếp giữa người gửi/ người nhận được tạo ra bởi message broker qua các tiêu chuẩn truyền tin bất đồng bộ như AMQP, MQTT,...

*Quản Lý Cơ Sở Dữ Liệu Phân Tán*: Trong microservices, các chức năng được tách thành nhiều microservices và nếu sử dụng cùng một CSDL trung tâm thì microservices không còn độc lập. Thay đổi dữ liệu của một service có thể làm hỏng các services khác. Do đó, mỗi microservice phải có CSDL riêng.



Việc phân tách quản lý dữ liệu cho phép bạn hoàn toàn phân tách các microservices và có thể chọn các công nghệ quản lý dữ liệu khác nhau (SQL hay NoSQL,...nhiều CSDL khác nhau cho các microservices). Tuy nhiên, với các giao dịch phức tạp liên quan đến nhiều microservices, giao dịch phải được thực hiện qua API.

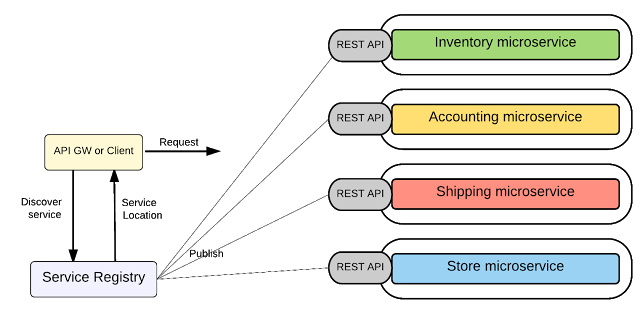
*Quản trị phân tán*: Microservices được xây dựng động lập và phân tán với nền tảng công nghệ khác nhau. Do đó, việc thiết lập một tiêu chuẩn thiết kế và phát triển chung là không cần thiết. Chúng ta có thể kết luận về khả năng quản trị phân tán trong kiến trúc microservices như sau:

* Không cần phải có một hệ quản trị tập trung khi thiết kế
* Microservices có thể tự quyết định thiết kế và phát triển của nó
* Kiến trúc ủng hộ và hỗ trợ việc chia sẻ các services chung hay có thể tái sử dụng

` Một số mặt của quản trị trong quá trình hoạt động như SLAs. điều phối, giám sát, bảo mật hay tìm kiếm dịch vụ (service discovery) có thể được phát triển ở cấp API Gateway

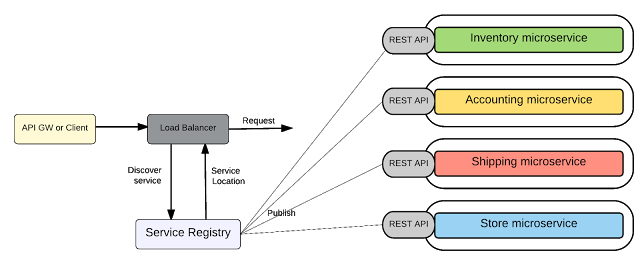
*Service Registry & Service Discorvery (Truy Tìm Dịch Vụ)*: Trong microservices, số lượng services mà bạn cần xử lý khá lớn. Và địa điểm của chúng có thể thay đổi nhanh và thường xuyên vì tính chất của việc phát triển microservices là nhanh. Nên bạn cần phải xác định được vị trí của một service trong quá trình chạy.

* Service Registry: Service Registry giữ các thực thể microservices và địa chỉ của chúng. Thực thể microservices được đăng kí với service registery khi bắt đầu chạy và hủy đăng kí khi tắt. Người dùng có thể tìm các services đang tồn tại và địa chỉ của chúng qua service registry.
* Service Discovery: Để tìm các microservices đang tồn tại và địa điểm của chúng, chúng ta cần một quy trình truy tìm dịch vụ. Có hai mô hình là Client-side Discovery và Server-side Discovery:
* Client-side Discovery: Với mô hình này, client hay API Gateway lấy thông tin địa điểm của một thực thể service bằng cách truy vấn Service Registry.



Client hay API Gateway phải thực hiện logic tìm kiếm service bằng cách gọi vào Service Registry.

* Server-side Discovery: Với cách này, client hay API Gateway gửi yêu cầu lên một component (có thể là một Load Balancer). Component này sẽ gọi Service Registry và quyết định địa điểm của các microservices.

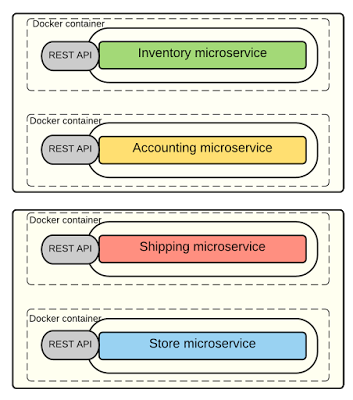


Một số giải pháp triển khai microservices như Kubernetes cung cấp mô hình server-side discovery

*Deloyment*: Việc triển khai các dịch vụ có một trò trọng yếu và gồm các yêu cầu

* Khả năng triển khai/ gỡ xuống độc lập mà không ảnh hưởng đến dịch vụ khác
* Có thể mở rộng theo cấp microservices, chỉ mở rộng microservices cần thiết
* Phát triển và triển khai microservices nhanh chóng
* Một microservice ngắt kết nối hay sập thì không ảnh hưởng các dịch vụ khác
* Docker (một công cụ mã nguồn mở cho phép lập trình viên và quản trị viên hệ thống triển khai các ứng dụng thành các containers trên môi trường Linux) cung cấp một công cụ tuyệt vời để triển khai microservices đáp ứng đủ các yêu cầu trên. Các bước chính gồm:
* Đóng gói mỗi microservice thành một ảnh Docker (docker image)
* Triển khai mỗi thực thể của service là một Docker container
* Mở dộng dựa vào số lượng thực thể
* Phát triển, triển khai và khởi động microservices trở nên nhanh hơn với Docker (nhanh hơn nhiều các máy ảo thông thường - VM)

Kubernetes mở rộng khả năng của Docker bằng cách quản lý một cụm các Linux container như một hệ thống duy nhất, quản lý và chạy các Docker containers trên nhiều hosts, cung cấp service discovery và kiểm soát việc nhân rộng. Như bạn có thể thấy, những tính năng này cần thiết cho kiến trúc microservices. Vì vậy sử dụng Kurbernetes (trên nền Docker) để triển khai microservices đang trở thành một phương pháp cực kì hữu dụng, đặc biệt với ứng dụng lớn.

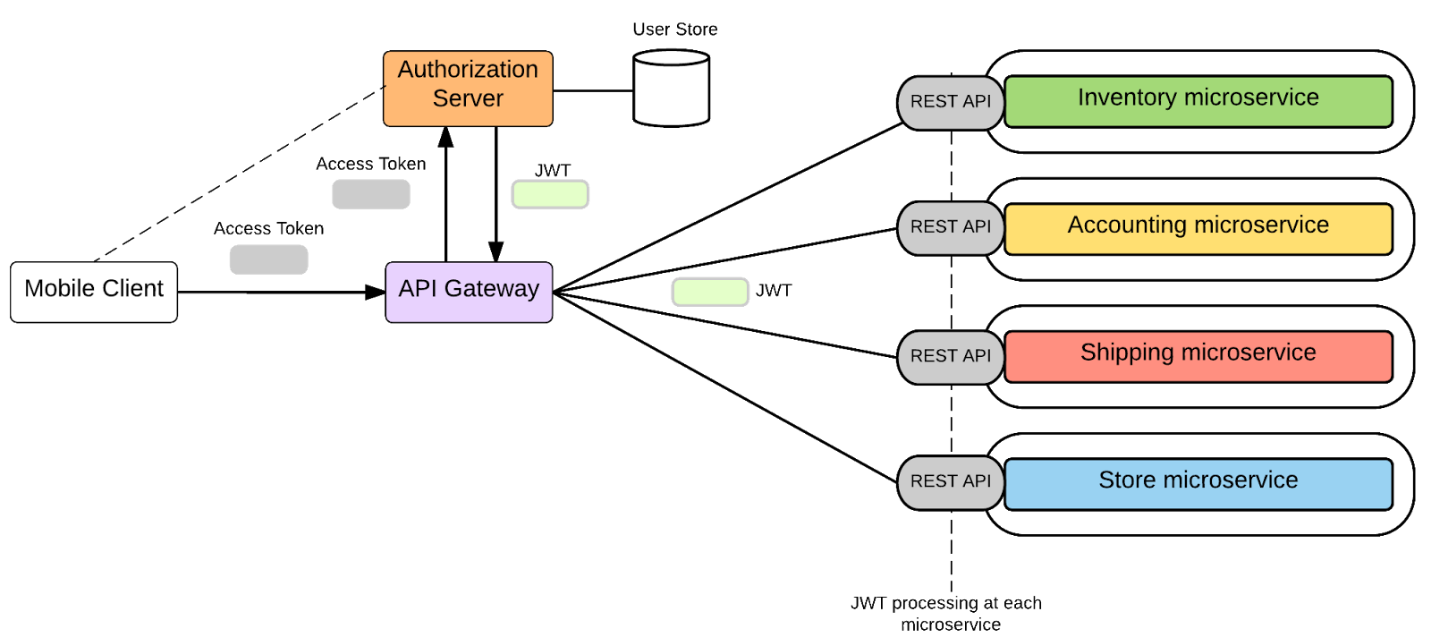


Hình trên thể hiện tổng quát việc triển khai các microservices. Mỗi thực thể của microservices được triển khai thành một container và có hai containers trên mỗi host. Bạn có thể thay đổi số lượng containers chạy trên một host theo ý mình.

*Security-Bảo mật*: Không chỉ microservice bất kì một kiến trúc hay hệ thống nào đều cần bảo mật, đó là một yêu cầu phổ biến hay bắt buộc.

Chúng ta có thể tận dụng các API bảo mật tiêu chuẩn như OAuth2 và OpenID Connect để tìm một giải pháp tốt hơn cho vấn đề bảo mật. Trước khi nói sâu hơn, chúng ta sẽ tóm tắt mục tiêu của mỗi tiêu chuẩn và cách sử dụng chúng.

* OAuth2 - một phương thức chứng thực kiểu ủy quyền. Client xác thực với server cấp quyền (authorization server) và nhận một token gọi là "Access token". Access token không chứa bất kì thông tin gì về client. Nó chỉ là một tấm vé tham chiếu đến thông tin người dùng mà server cấp quyền có thể truy xuất đến. Do đó, đây cũng được gọi là token kiểu tham chiếu "by-reference token" và an toàn để sử dụng trên mạng lưới mở và internet.ư
* OpenID Connect hoạt động tương tự OAuth nhưng ngoài Access Token, server cấp quyền còn phát một ID token chứa thông tin người dùng. Token này thường dạng JWT (JSON Web Token) và được kí bởi server cấp quyền. JWT token do đó được coi là token kiểu tham trị "by-value token" bởi vì nó chức thông tin về người dùng và có thể trở nên không an toàn.



Với ví dụ như hình trên, các bước chính để thực hiện bảo mật microservices gồm:

* Chuyển việc xác thực cho server dạng OAuth và OpenID Connect (Authorization server)
* Sử dụng API Gateway để có một điểm đầu vào duy nhất cho yêu cầu từ clients
* Client kết nối với server cấp quyền và nhận Access token. Sau đó gửi token này đến API Gateway cùng với yêu cầu
* Dịch token tại Gateway - API Gateway lấy ra access token và gửi đến server cấp quyền để lấy JWT
* Gateway truyền JWT cùng với yêu cầu đến microservices
* JWT chứa thông tin người dùng cần thiết để lưu user sessions,....
* Ở mỗi lớp microservice, bạn có thể có một component xử lý JWT, việc này khác đơn giản

**KẾT LUẬN**

Cho dù kiến trúc Microservice đã được nhiều công ty lớn áp dụng và phát huy tính hiêu quả của nó. Tuy nhiên, điều đó không có nghĩa là chúng ta nên mù quáng, cắm đầu theo công ty lớn.

Kiến trúc Microservice không nhỏ như cái tên của nó. Mà nó là một kiến trúc dành cho hệ thống và ứng dụng lớn. Nếu ứng dụng của bạn chưa đủ lớn và cũng không có ý định mở rộng trong tương lai thì hướng phát triển theo kiến trúc nguyên khối vẫn là một lựa chọn hợp lý.

Kiến trúc Microservice giải quyết được rất nhiều vấn đề của kiến trúc nguyên khối, nó tỏ ra vượt trội kiến trúc nguyên khối ở nhiều khía cạnh. Tuy nhiên mỗi loại kiến trúc đều có những mặt ưu nhược khác nhau và Microservice cũng vậy. Chưa ai có thể khẳng định nó có thể thay thế Monolithic.

Việc áp dụng mô hình kiến trúc Microservice cần một kiến trúc sư có cái nhìn thật tốt về dịch vụ mình sẽ cung cấp trong hiện tại, tương lai. Việc lựa chọn Microservice cần phải được đánh giá thông qua mục đích, mục tiêu của dự án.

Trên đây là một số ý kiến đánh giá chủ quan của chúng tôi về mô hình kiến trúc Microservice có thể đúng hoặc sai xong rất cảm ơn các bạn đã đọc nó.

**TÀI LIỆU THAM KHẢO**

1. https://www.nginx.com/blog/introduction-to-microservices/

2. https://www.cnblogs.com/wintersun/p/6219259.html

3. https://www.redhat.com/en/topics/microservices/what-are-microservices

4. https://toidicodedao.com/2017/02/21/tong-quan-micro-service/

5. https://dzone.com/articles/microservices-in-practice-1

6. https://viblo.asia/p/tim-hieu-ve-microservice-Do754PD45M6

7. https://techmaster.vn/posts/34410/microservices-thuc-tien-tu-thiet-ke-den-trien-khai