# Chương I

## Phần 1: Kiến trúc Microservice

### Lịch sử:

Theo Wikipedia:

Ngay từ năm 2005, Peter Rodgers đã giới thiệu thuật ngữ "Micro -Web-Services " trong một bài thuyết trình tại hội nghị Web Services Edge. Chống lại suy nghĩ thông thường của kiến trúc SOAP SOA, ông đã lập luận cho "REST-services", ông thảo luận về " Các thành phần phần mềm là Micro-Web-Services". Các dịch vụ được tạo bằng cách sử dụng các thành phần Unix (Web đáp ứng Unix - web kết hợp lỏng lẻo với nhau). Các service có thể gọi các service (có thể sử dụng nhiều ngôn ngữ lập trình khác nhau. Một nền tảng microservices được thiết kế tốt phải áp dụng các nguyên tắc kiến trúc cơ bản của dịch vụ Web và REST cùng với sử dụng các thành phần Unix để linh hoạt và cải thiện tính đơn giản trong service.

Từ năm 1999 Rodgers với dự án nghiên cứu Dexter tại Hewlett Packard Lab, với mục đích là làm cho code ít phức tạp và làm cho các hệ thống phần mềm phức tạp, quy mô lớn có thể thay đổi mạnh mẽ và nhanh chóng. Cuối cùng, con đường nghiên cứu này đã dẫn đến sự phát triển của máy tính hướng tài nguyên (ROC) và giao thức REST.

Một hội thảo của các kiến trúc sư phần mềm được tổ chức gần Venice vào tháng 5 năm 2011 đã sử dụng thuật ngữ "microservice" để mô tả kiến trúc phổ biến mà nhiều người trong số họ đã khám phá. Vào tháng 5 năm 2012, đã quyết định chọn "microservices" là tên thích hợp nhất. James Lewis đã trình bày một số ý tưởng đó như một nghiên cứu vào tháng 3 năm 2012 tại Bằng cấp 33 ở Kraków về các microservice - Java, phương thức Unix. Cùng khoảng thời gian đó Adrian Cockcroft, cựu giám đốc Hệ thống đám mây tại Netflix, đã mô tả cách tiếp cận này là "SOA fine-grained" (làm mịn SOA), đi tiên phong trong phong cách web, cũng như nhiều người khác - Joe Walnes, Dan North, Evan Bottcher và Graham Tackley.

Microservices là một giải pháp cho các kiến trúc hướng dịch vụ (SOA) được sử dụng để xây dựng các hệ thống phần mềm linh hoạt, có thể triển khai độc lập trên nhiều máy chủ khác nhau. Microservices đang trở nên phổ biến hơn để xây dựng các hệ thống được triển khai liên tục, phù hợp với các mô hình phatr triển phần mềm tiến hóa.

Vào tháng 2 năm 2020, Báo cáo Nghiên cứu Thị trường Cloud Microservices dự đoán rằng quy mô thị trường kiến trúc dịch vụ vi mô toàn cầu sẽ tăng với tốc độ CAGR là 21,37% từ năm 2019 đến năm 2026 và đạt 3,1 tỷ đô la vào năm 2026.

### Kiến trúc của microservice:

#### Service Registry và Service Discovery

Trong MSA (microservice achitechture), số lượng các service cần để làm việc sẽ là khá nhiều. Chúng thay đổi vị trí linh hoạt do tính chất cần phát triển nhanh của microservice. Do đó, cần tìm vị trí của microservice trong suốt thời gian runtime. Giải pháp cho vấn đề này là sử dụng Service Registry.

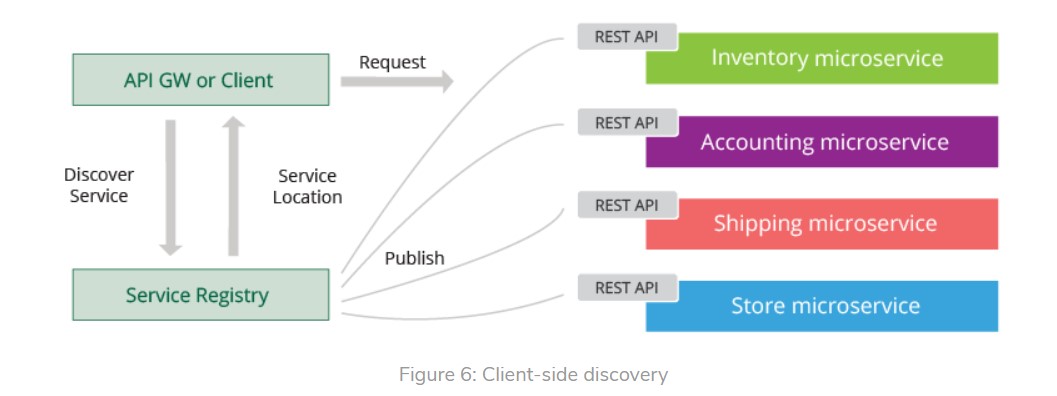
##### Service Registry

Service registry là nơi để chứa các metadata của các microservice instances (bao gồm vị trí location, host port, …). Các microservice instance được đăng ký với service registry khi khởi động và sẽ hủy đăng ký khi bị shut down. Các thành phần khác cần tìm thông tin của một microservice nào đó thì sẽ tìm thông qua service registry.

##### Service Discovery

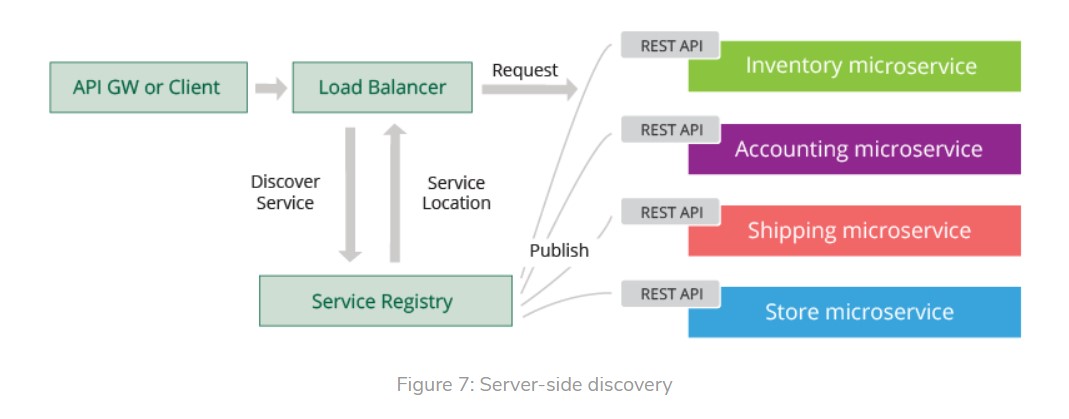
Để tim các microservice đang hoạt động và vị trí location của nó, chúng ta cần đến cơ chế service discovery. Có 2 loại cơ chế service discovery là client-side discovery và server-side discovery. Chúng ta sẽ xem xét kỹ hơn về các cơ chế này:

Client-side discovery Theo cách tiếp cận này, client hoặc API-gateway sẽ có được vị trí của một service instance bằng cách truy vấn một service registry.



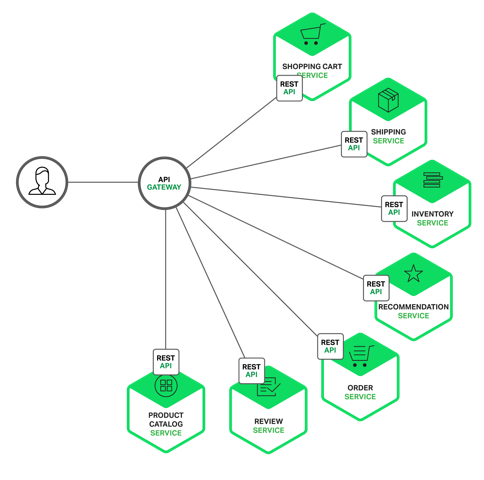
Hình . kiến trúc Client-side discovery

Server-side discovery Với phương pháp này, client/API gateway sẽ gửi một request đến một component (ví dụ như một load balancer) chạy trên một location đã biết. Component đó sẽ gọi đến service registry và xác định vị trí location mà request cần đến. Trong hệ thống Spring Cloud Netflix thì Server-side discovery có tên gọi riêng là Eureka server



Hình . Kiến trúc Server-side discovery

#### API Gateway:



Hình . Cách hoạt động của API gateway

Đối với hầu hết các ứng dụng dựa trên microservices, việc triển khai API Gateway, hoạt động như một điểm vào duy nhất vào hệ thống là rất hợp lý. API Gateway chịu trách nhiệm định tuyến yêu cầu, thành phần và giao thức. Nó cung cấp cho mỗi client của ứng dụng một API tùy chỉnh. API Gateway cũng có thể che dấu các lỗi trong các dịch vụ bằng cách trả về dữ liệu mặc định hoặc dữ liệu lưu trong bộ nhớ cache

API Gateway chịu trách nhiệm định tuyến yêu cầu, thành phần và dịch giao thức. Tất cả các yêu cầu từ khách hàng trước tiên đi qua API Gateway. Sau đó, nó định tuyến các yêu cầu đến microservice thích hợp. API Gateway sẽ xử lý yêu cầu bằng cách gọi nhiều dịch vụ nhỏ và tổng hợp kết quả. Nó có thể dịch giữa các giao thức web như HTTP và WebSocket và các giao thức web không thân thiện được sử dụng nội bộ.

##### Lợi ích và hạn chế của API Gateway

Lợi ích chính của việc sử dụng API Gateway là nó đóng gói cấu trúc bên trong của ứng dụng. Thay vì phải gọi các dịch vụ cụ thể, Client chỉ cần nói chuyện với cổng. API Gateway cung cấp cho mỗi loại Client một API cụ thể. Làm giảm số lượng tương tác giữa Client và ứng dụng. Nó cũng đơn giản hóa mã của Client.

API Gateway cũng có một số nhược điểm. Nó là một thành phần có tính khả dụng cao phải được phát triển, triển khai và quản lý. Các nhà phát triển phải cập nhật API Gateway để hiển thị các điểm cuối của mỗi microservice. Với quy trình cập nhật API Gateway càng nhẹ càng tốt. Nếu không, các nhà phát triển sẽ buộc phải xếp hàng đợi để cập nhật. Tuy nhiên, mặc dù có những nhược điểm này, đối với hầu hết các ứng dụng việc sử dụng API Gateway là hợp lý.

Lời mời dịch vụ

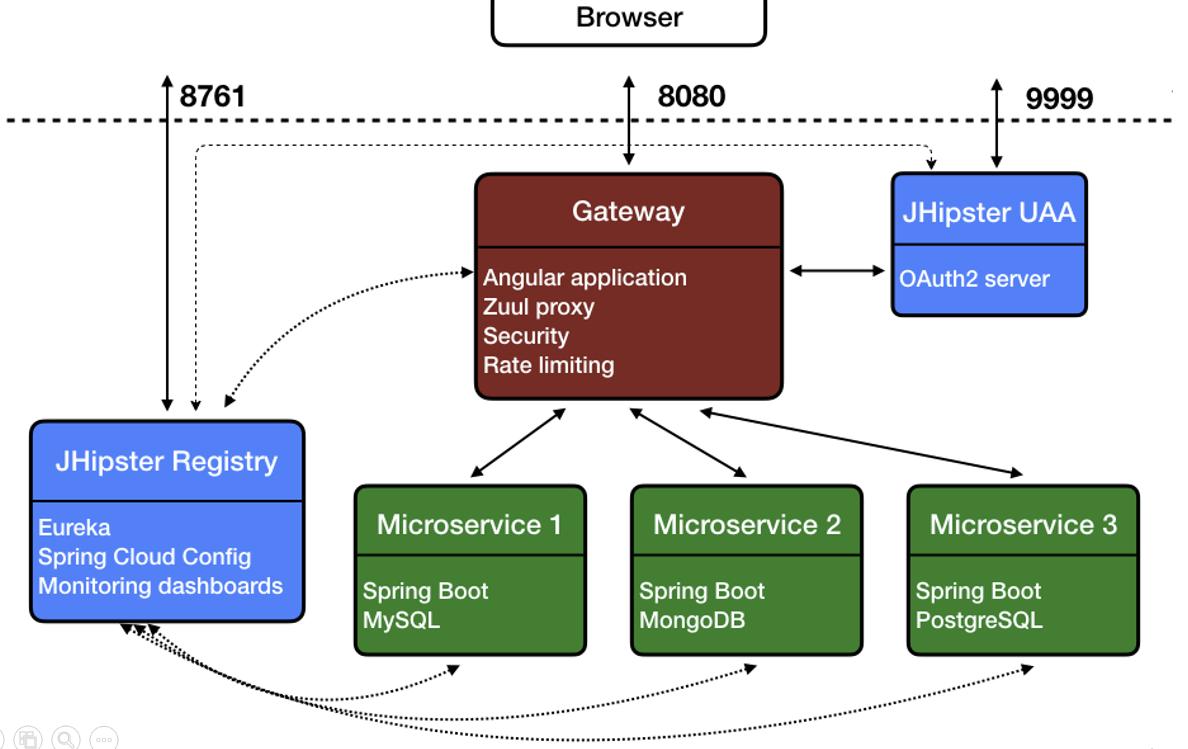
Ứng dụng dựa trên microservices là một hệ thống phân tán và phải sử dụng cơ chế giao tiếp giữa các quá trình. Có hai cách giao tiếp giữa các quá trình. Một là sử dụng cơ chế không đồng bộ, dựa trên tin nhắn. Một số sử dụng trình môi giới thông báo như JMS hoặc AMQP. Những loại khác, chẳng hạn như Zeromq, không có môi giới và các dịch vụ giao tiếp trực tiếp. Cách khác của giao tiếp giữa các quá trình là một cơ chế đồng bộ như HTTP hoặc Thrift. Một hệ thống thường sẽ sử dụng cả kiểu không đồng bộ và kiểu đồng bộ. Nó thậm chí có thể sử dụng nhiều cách triển khai của mỗi kiểu. Do đó, API Gateway sẽ cần hỗ trợ nhiều cơ chế giao tiếp.

Khám phá dịch vụ

API Gateway cần biết vị trí (địa chỉ IP và cổng) của từng microservice mà nó giao tiếp. Trong một ứng dụng truyền thống, có thể cố định địa chỉ, nhưng trong một ứng dụng microservices thì đó là 1 vấn đề lớn. Các dịch vụ cơ sở thường sẽ có một vị trí tĩnh, có thể được chỉ định thông qua các biến môi trường OS. Tuy nhiên, việc xác định vị trí của một dịch vụ không dễ dàng, các dịch vụ có các vị trí động. Ngoài ra, tập hợp các dịch vụ sẽ thay đổi động do việc nâng cấp. Do đó, API Gateway, giống như bất kỳ dịch vụ nào khác trong hệ thống, cần sử dụng cơ chế khám phá dịch vụ của hệ thống: Khám phá phía máy chủ hoặc Client, nếu hệ thống sử dụng Client Side Discovery thì API Gateway phải có khả năng truy vấn Service Registry, đây là cơ sở dữ liệu của tất cả các phiên bản microservice và vị trí của chúng.

Trong mô hình microservice thông thường của Spring Cloud Netflix các microservice sẽ đăng ký với eureka server để định danh chính mình trên eureka server, gateway sẽ khám phá tất cả các dịch vụ có trong eureka server, khi client muốn truy cập một trong số các microservice, chỉ cần gọi đến gateway mà không cần quan tâm microservice muốn sử dụng nằm ở địa chỉ nào, gateway sẽ điều hướng đến microservice đó và trả về dữ liệu chính xác cho client.

#### Uaa (User authenticate and authorization)



Hình . Kiến trúc microservice sử dụng UAA

##### Vai trò của UAA

Về bản chất, UAA cũng mà một microservice, UAA sẽ đăng ký với eureka server để định danh, khi một client truy cập vào một trong tất cả các service hoặc gateway, UAA sẽ chịu trách nhiệm xác thực cho mỗi request đó, nhiệm vụ chính của UAA là lưu trữ thông tin đăng nhập, xác thực và ủy quyền cho client.

##### Lợi ích khi sử dụng UAA

Trong một hệ thống có rất nhiều microservice khác nhau, việc triển khai bảo mật API cho mỗi microservice trở nên phức tạp, đối với các hệ thống lớn có hàng trăm microservice thì việc này là bất khả thi. Thay vào đó ta sẽ sử dụng một microservice duy nhất, đảm bảo xác thực cho toàn bộ các microservice khác, đây cũng là mục đích của microservice đó là làm một việc và làm tốt.

##### Nhược điểm khi sử dụng UAA:

* Khi một hệ thống sử dụng UAA nếu phá được lớp bảo mật của UAA và có đủ quyền truy cập thì có thể truy cập toàn bộ các API của tất cả các microservice.
* Một nhược điểm khác hiếm khi xảy ra đó là là sập microservice UAA, nó hiếm khi xảy ra vì một hệ thống microservice thì mỗi microservice có thể triển khai trên nhiều máy khác nhau, hoạt động chung trên cùng một cơ sở dữ liệu với nhau, khi một microservice chết đi thì vẫn còn các microservice còn đang chạy. Trong trường hợp tất cả các UAA đều sập thì khi đó client có thể truy cập tất cả API của tất cả các microservice vì lúc này không còn service nào bảo bật cho các microservice.

#### Swagger là gì?

Swagger cho phép mô tả cấu trúc các API. Swagger có khả năng để các API mô tả cấu trúc của chính chúng bằng cách yêu cầu API và trả lại một YAML hoặc JSON chứa mô tả về toàn bộ API. Về cơ bản là một danh sách tài nguyên của API trong ứng dụng tuân theo kỹ thuật OpenAPI. Các API tự mô tả yêu cầu bao gồm các thông tin như:

Tất cả các hoạt động mà API hỗ trợ là gì?

Các tham số API là gì và nó trả về những gì?

API có cần ủy quyền không?

Và ngay cả như điều khoản, thông tin liên hệ và giấy phép sử dụng API.

Tóm lại: Swagger là một Ngôn ngữ mô tả interface để mô tả các API RESTful được thể hiện bằng JSON. Swagger được sử dụng cùng với một bộ công cụ phần mềm nguồn mở để thiết kế, xây dựng, lập tài liệu và sử dụng các dịch vụ web RESTful. Swagger bao gồm tài liệu tự động, tạo mã (nhiều ngôn ngữ lập trình) và tạo test-case. Swagger được áp dụng cho gateway để các nhà phát triển có thể dễ dàng tìm kiếm các API cần thiết và phát triển.

### Tổng kết phần 1:

Trong kiến trúc microservice, các microservice sẽ có các mục đích khác nhau, giữa các microserivce có thể gọi lẫn nhau thông qua gateway hoặc gọi trực tiếp để trao đổi hoặc lấy thông tin với nhau. Tuy nhiên, điều này nên hạn chế vì việc các microservice gọi lẫn nhau như vậy làm cho các ứng dụng trở nên phức tạp, phá vỡ đi cấu trúc của microservice, đồng thời, khi một service không được sử dụng nữa sẽ gây ảnh hưởng đến các service liên quan đến nó.

Trong kiến trúc microservice đầy đủ sẽ gồm 4 phần: service registry, gateway, UAA và các microservice.

* Service registry chịu trách nhiệm dùng để cho các service khác đăng ký định danh của mình ở đó.
* Gateway chịu trách nhiệm điều hướng cho mỗi request đến các service mà request đó yêu cầu.
* UAA chịu trách nhiệm xác thực, ủy quyền và bảo mật các API cho các microservice.
* Microservice chịu trách nhiệm cung cấp các dịch vụ mà nó đảm nhiệm, đồng thời đăng ký thông tin định danh của mình trong service registry.

## Phần II: Monolothic, SOA và MSA

### SOA:

#### SOA là gì (Service Oriented Achitecture)

Kiến trúc hướng dịch vụ (SOA) là một kiểu kiến trúc trong đó các thành phần của ứng dụng cung cấp dịch vụ cho các thành phần khác thông qua một giao thức truyền thông, thường là qua mạng.

SOA giúp các thành phần phần mềm trên các mạng khác nhau làm việc với nhau dễ dàng hơn.

Các Web Service được xây dựng theo kiến trúc SOA có xu hướng làm cho Web Service độc lập hơn. Bản thân các Web Service có thể trao đổi dữ liệu với nhau do các nguyên tắc cơ bản mà chúng được tạo ra, chúng không cần bất kỳ loại tương tác nào của con người và cũng không cần bất kỳ sửa đổi mã nào. Nó đảm bảo rằng các Web Service trên mạng có thể tương tác với nhau một cách liền mạch.

Kiến trúc hướng dịch vụ (SOA) là một cách tiếp cận kiến trúc trong đó các ứng dụng sử dụng các dịch vụ có sẵn trong mạng. Trong kiến trúc này, các dịch vụ được cung cấp để tạo thành các ứng dụng, thông qua một cuộc gọi qua internet.

• SOA cho phép người dùng kết hợp các phương tiện từ các dịch vụ hiện có để tạo thành các ứng dụng.

• SOA bao gồm một tập hợp các nguyên tắc thiết kế cấu trúc sự phát triển hệ thống và cung cấp các phương tiện để tích hợp các thành phần vào một hệ thống nhất quán và phi tập trung.

• SOA đóng gói các chức năng thành một tập hợp các dịch vụ có thể tương tác, có thể được tích hợp vào các hệ thống phần mềm khác nhau thuộc các lĩnh vực kinh doanh riêng biệt.

#### Nguyên tắc của SOA:

- Khớp nối lỏng lẻo: Các dịch vụ được thiết kế thành các phần riêng biệt kết nối với nhau, giảm thiểu sự phụ thuộc vào các dịch vụ khác.

- Khả năng tái sử dụng: Được thiết kế thành các thành phần, các dịch vụ có thể được tái sử dụng hiệu quả hơn, giảm thời gian phát triển và các chi phí liên quan.

- Quyền tự chủ: Các dịch vụ có quyền kiểm soát logic bên trong, người dùng không cần quan tâm kiến trúc bên trong có gì, chỉ cần quan tâm nó cung cấp cho người dùng cái gì.

- Khả năng tương thích: Sử dụng các dịch vụ khác để xây dựng nên ứng dụng chính. Các dịch vụ này hỗ trợ cho các dịch vụ khác thực hiện nhiệm vụ của mình dựa trên các dịch vụ này.

#### Ưu điểm của SOA:

• Khả năng tái sử dụng dịch vụ: Trong SOA, các ứng dụng được tạo từ các dịch vụ hiện có. Do đó, các dịch vụ có thể được tái sử dụng để tạo ra nhiều ứng dụng.

• Bảo trì dễ dàng: Vì các dịch vụ độc lập với nhau nên chúng có thể được cập nhật và sửa đổi dễ dàng mà không ảnh hưởng đến các dịch vụ khác.

• Độc lập với nền tảng: SOA cho phép tạo một ứng dụng phức tạp bằng cách kết hợp các dịch vụ từ các nguồn khác nhau, độc lập với nền tảng.

• Tính khả dụng: Các khối SOA dễ dàng có sẵn ở một số trang web được các nhà phát triển cung cấp.

• Độ tin cậy: Các ứng dụng SOA đáng tin cậy hơn vì dễ dàng gỡ lỗi.

• Khả năng mở rộng: Các dịch vụ có thể chạy trên các máy chủ khác nhau trong một môi trường, điều này làm tăng khả năng mở rộng

#### Nhược điểm của SOA:

• Chi phí cao: Việc kiểm tra và lọc lại các tham số đầu vào của các dịch vụ được thực hiện bất cứ khi nào có request đến, điều này làm giảm hiệu suất vì nó làm tăng tải và tắng thời gian phản hồi.

• Đầu tư cao: Cần có một khoản đầu tư ban đầu rất lớn cho SOA.

• Quản lý dịch vụ phức tạp: Khi các dịch vụ tương tác, chúng sẽ trao đổi thông điệp với nhau, số lượng record có thể lên đến hàng triệu.

#### **Ứng dụng thực tế của SOA:**

1. Cơ sở hạ tầng SOA được nhiều quân đội và không quân sử dụng để triển khai các hệ thống nhận thức tình huống.

2. SOA được sử dụng để cải thiện việc cung cấp dịch vụ chăm sóc sức khỏe.

3. Ngày nay, nhiều ứng dụng là trò chơi và chúng sử dụng các chức năng có sẵn để chạy. Ví dụ: một ứng dụng có thể cần GPS để ứng dụng đó sử dụng các chức năng GPS có sẵn của thiết bị.

### MSA

#### MSA là gì

MSA (Microservice Achitechture) - một biến thể của kiểu cấu trúc SOA – biến đổi một ứng dụng thành một tập hợp các dịch vụ được ghép nối lỏng lẻo. Trong kiến ​​trúc microservices, các dịch vụ rất chi tiết và các giao thức có trọng lượng nhẹ. Một Microservice là một phần chức năng nghiệp vụ với các API rõ ràng, có thể thực hiện kiến ​​trúc phân lớp cho các thành phần bên trong. Kiến ​​trúc microservices về cơ bản tuân theo triết lý Unix là "Làm một việc và làm tốt".

#### Tại sao lại sử dụng microservices?

Không có chương trình đơn lẻ nào đại diện cho toàn bộ ứng dụng, vì vậy các dịch vụ có thể thay đổi khuôn khổ (hoặc thậm chí cả ngôn ngữ) mà không tốn kém. Các ứng dụng có thể được viết trên một số nền tảng khác nhau - Java, Ruby, Node, Go, .NET, v.v. - mà không gặp vấn đề gì.

#### Lợi ích của Microservices

- Giải quyết vấn đề phức tạp. Nó phân rã một ứng dụng nguyên khối thành một tập hợp các dịch vụ. Mà số chức năng không thay đổi, ứng dụng đã được chia thành các phần hoặc dịch vụ có thể quản lý được. Mỗi dịch vụ có một ranh giới được xác định rõ ràng dưới dạng API. Mô hình Kiến trúc Microservices thực thi ở mức module mà một ứng dụng nguyên khối không thể làm. Do đó, các dịch vụ riêng lẻ phát triển nhanh hơn nhiều và dễ hiểu và dễ bảo trì hơn nhiều.

- Thứ hai, kiến trúc này cho phép mỗi dịch vụ được phát triển độc lập bởi một nhóm tập trung vào dịch vụ đó. Các nhà phát triển có thể tự do lựa chọn bất kỳ công nghệ nào có, miễn là nó dựa trên API. Tuy nhiên, không bắt buộc phải sử dụng các công nghệ lỗi thời đã tồn tại khi làm một dự án mới. Khi viết một dịch vụ mới, có thể sử dụng công nghệ hiện tại. Hơn nữa, vì các dịch vụ tương đối nhỏ nên việc viết lại một dịch vụ cũ bằng công nghệ hiện tại trở nên khả thi.

- Thứ ba, mô hình Kiến trúc Microservices cho phép từng microservice được triển khai độc lập. Các nhà phát triển không cần phải thay đổi code. Mô hình Kiến trúc Microservices giúp triển khai liên tục, thích hợp cho các mô hình làm việc như Scrum.

- Có thể phát triển và triển khai từng microservice trên nhiều nền tảng khác nhau, sử dụng các ngôn ngữ lập trình và công cụ dành cho nhà phát triển khác nhau. Các microservices sử dụng các API và giao thức truyền thông để tương tác với nhau, nhưng chúng không dựa vào nhau.

- Ưu điểm lớn nhất của kiến ​​trúc microservices là các nhóm có thể phát triển, duy trì và triển khai từng microservice một cách độc lập. Các ứng dụng sử dụng microservices mở rộng quy mô tốt hơn, vì có thể chia tỷ lệ chúng một cách riêng biệt. Microservices cũng làm tăng tốc đường dẫn. Bên cạnh đó, các dịch vụ biệt lập có khả năng chịu lỗi tốt hơn. Việc duy trì và gỡ lỗi một microservice nhẹ sẽ dễ dàng hơn một ứng dụng phức tạp.

- Cuối cùng, kiến trúc Microservices cho phép từng dịch vụ được mở rộng quy mô một cách độc lập. Có thể triển khai số lượng phiên bản của mỗi dịch vụ đáp ứng các hạn chế về khả năng và tính khả dụng của nó.

#### Mặt hạn chế của Microservices

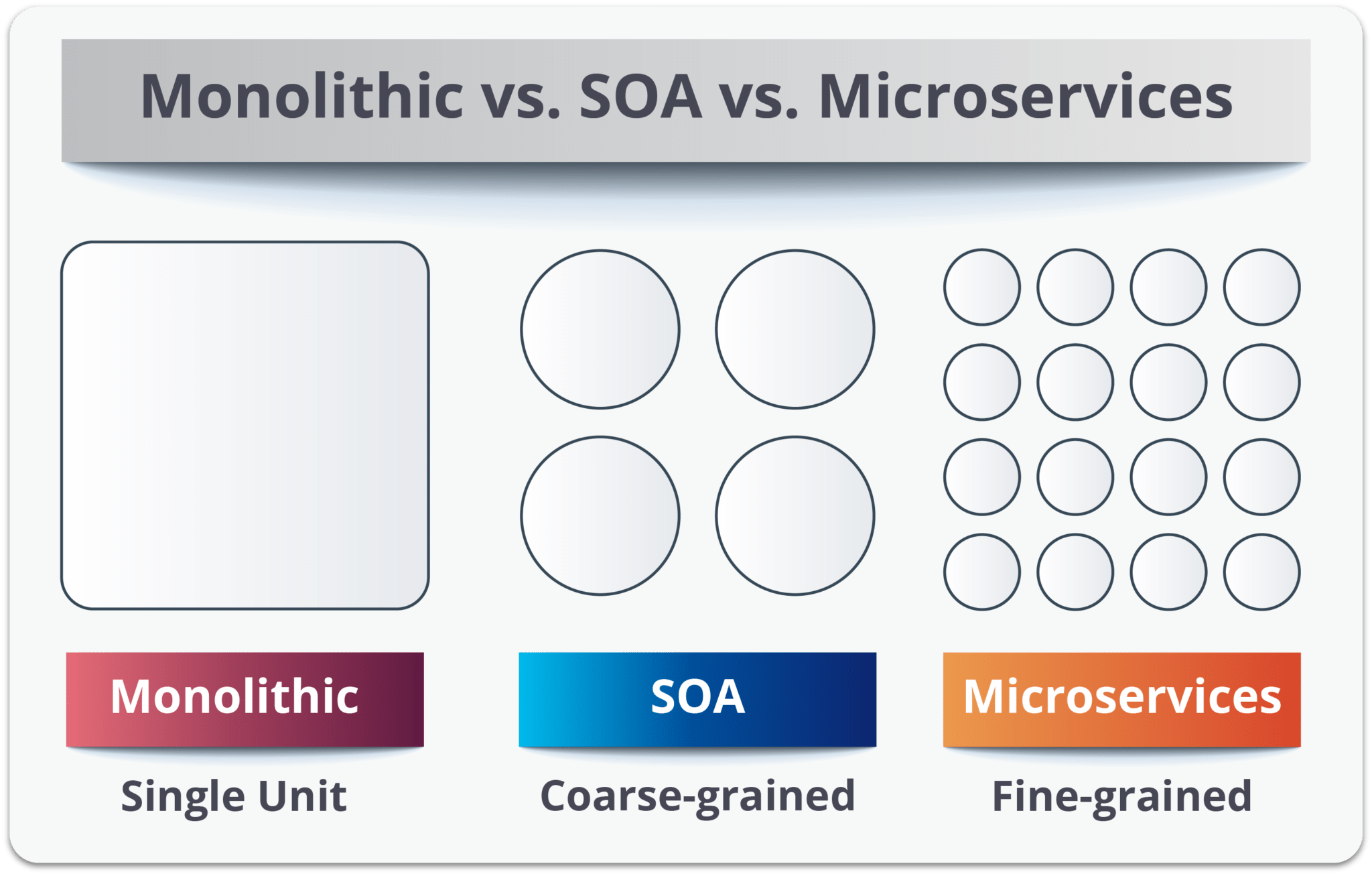
- Một nhược điểm lớn của microservices là sự phức tạp nảy sinh do ứng dụng microservices là một hệ thống phân tán. Các nhà phát triển cần lựa chọn và thực hiện một cơ chế giao tiếp giữa các quá trình dựa trên tin nhắn.

- Một thách thức khác với microservices là cơ sở dữ liệu được phân tán. Các request cập nhật nhiều thực thể là khá nhiều. Những loại yêu cầu này rất khó thực hiện trong một ứng dụng nguyên khối vì chỉ có một cơ sở dữ liệu duy nhất. Tuy nhiên, trong ứng dụng dựa trên microservices, cần cập nhật nhiều cơ sở dữ liệu thuộc sở hữu của các dịch vụ khác nhau. Sử dụng các yêu cầu phân tán thì không khuyến khích vì định lý CAP.

- Việc thử nghiệm một ứng dụng microservices cũng phức tạp hơn nhiều. Ví dụ: với một frameword hiện đại như Spring Boot, việc viết một lớp để thử nghiệm một ứng dụng web nguyên khối và kiểm tra API REST của nó là điều rất dễ dàng. Ngược lại, một lớp thử nghiệm tương tự cho một service sẽ cần khởi chạy service đó và bất kỳ service nào mà nó liên quan đến.

- Một thách thức lớn khác với Kiến trúc Microservices là triển khai các thay đổi trên nhiều dịch vụ. Ví dụ: triển khai một yêu cầu thay đổi đối với các dịch vụ A, B và C, trong đó A phụ thuộc vào B và B phụ thuộc vào C. Trong một ứng dụng nguyên khối, có thể chỉ cần thay đổi các mô-đun tương ứng, tích hợp các thay đổi, và triển khai chúng trong một lần. Ngược lại, trong mẫu Kiến trúc Microservices, cần phải lập kế hoạch và điều phối cẩn thận việc triển khai các thay đổi cho từng dịch vụ. Ví dụ: sẽ cần cập nhật dịch vụ C, tiếp theo là dịch vụ B và cuối cùng là dịch vụ A. Nhưng hầu hết các thay đổi thường chỉ tác động đến một dịch vụ và các thay đổi đa dịch vụ tương đối hiếm.

- Việc triển khai một ứng dụng dựa trên microservices cũng phức tạp hơn nhiều. Một ứng dụng nguyên khối được triển khai đơn giản trên một máy chủ giống bằng bộ cân bằng tải truyền thống. Mỗi phiên bản ứng dụng được định cấu hình với các vị trí (máy chủ và cổng) của các dịch vụ cơ sở hạ tầng như cơ sở dữ liệu. Ngược lại, một ứng dụng microservice thường bao gồm một số lượng lớn các dịch vụ. Ví dụ: Hailo có 160 dịch vụ khác nhau và Netflix có hơn 600. Mỗi dịch vụ sẽ có nhiều phiên bản về thời gian chạy. Nhiều bộ phận khác cần được cấu hình, triển khai, mở rộng và giám sát. Ngoài ra, cũng sẽ cần phải triển khai cơ chế khám phá dịch vụ cho phép dịch vụ phát hiện ra các vị trí (máy chủ và cổng) của bất kỳ dịch vụ nào khác mà nó cần giao tiếp. Việc triển khai thành công một ứng dụng microservices yêu cầu các nhà phát triển kiểm soát nhiều hơn các phương pháp triển khai và mức độ tự động hóa cao.



### SOA monolothic microservice

Các ứng dụng doanh nghiệp ngày nay đang được thiết kế để đáp ứng được số lượng lớn nghiệp vụ kinh doanh. Do đó một ứng dụng phần mềm cần cung cấp hàng trăm chức năng và tất cả các những chức năng như vậy thường được gói gọn trong một ứng dụng nguyên khối duy nhất. Việc triển khai, xử lý sự cố, mở rộng và nâng cấp các ứng dụng như vậy là một cơn ác mộng đối với bất kỳ doanh nghiệp nào.

Kiến trúc hướng dịch vụ (service-oriented architecture - SOA) được thiết kế để khắc phục các vấn đề phát sinh từ ứng dụng một khối (monolithic) bằng cách đưa ra khái niệm service. Do đó, với SOA, một ứng dụng sẽ được thiết kế dưới dạng kết hợp các service khác nhau. Khái niệm SOA không có nghĩa là biến từng service thành một khối riêng, nhưng hầu hết các ứng dụng triển khai theo SOA đều có hướng triển khai từng service dưới dạng một khối có cùng thời gian runtime. Vì vậy, tương tự như các ứng dụng một khối, các service này theo thời gian cũng tích lũy nhiều nghiệp vụ và chức năng khác nhau. Sự tăng trưởng này sẽ sớm biến những service đó thành những khối u nguyên khối, không khác gì những ứng dụng một khối thông thường.

Nền tẳng của kiến trúc microservice (MSA) là về việc phát triển một ứng dụng bằng các service nhỏ và độc lập chạy trong tiến trình riêng của chúng. Các service này được phát triển và deploy một cách độc lập.

Hầu hết các định nghĩa về MSA giải thích nó như là một khái niệm kiến trúc tập trung vào việc tách các service sẵn có trong kiến trúc một khối thành một tập hợp các service độc lập. Tuy nhiên thì microservice không chỉ là làm những việc phân chia như thế.

#### Khác nhau giữa microservice và SOA

|  |  |  |
| --- | --- | --- |
|  | Microservices | SOA |
| Ngành kiến ​​trúc | Được thiết kế để lưu trữ các dịch vụ có thể hoạt động độc lập | Được thiết kế để chia sẻ tài nguyên trên các dịch vụ |
| Chia sẻ thành phần | Thường không liên quan đến chia sẻ thành phần | Thường xuyên liên quan đến việc chia sẻ thành phần |
| Mức độ chi tiết | Dịch vụ chi tiết | Các dịch vụ lớn hơn, nhiều mô-đun hơn |
| Lưu trữ dữ liệu | Mỗi dịch vụ có thể có một bộ lưu trữ dữ liệu độc lập | Liên quan đến việc chia sẻ lưu trữ dữ liệu giữa các dịch vụ |
| Quản trị | Yêu cầu sự hợp tác giữa các nhóm | Các giao thức quản trị chung giữa các nhóm |
| Kích thước và phạm vi | Tốt hơn cho các ứng dụng nhỏ hơn và dựa trên web | Tốt hơn cho tích hợp quy mô lớn |
| Giao tiếp | Giao tiếp thông qua một lớp API | Giao tiếp thông qua ESB |
| Khớp nối và gắn kết | Phụ thuộc vào ngữ cảnh giới hạn để ghép nối | Phụ thuộc vào việc chia sẻ tài nguyên |
| Dịch vụ từ xa | Sử dụng REST và JMS | Sử dụng các giao thức như SOAP và AMQP |
| Triển khai | Triển khai nhanh chóng và dễ dàng | Ít linh hoạt hơn trong triển khai |

### Tổng kết phần II:

Trong một ứng dụng monolothic tất cả các thành phần phần mềm của một ứng dụng được tập hợp lại với nhau và được đóng gói chặt chẽ.

SOA về bản chất là một tập hợp các dịch vụ. Các dịch vụ này giao tiếp với nhau. Giao tiếp có thể liên quan đến việc truyền dữ liệu đơn giản hoặc hai hoặc nhiều dịch vụ điều phối một số hoạt động. Sử dụng một số phương tiện kết nối các dịch vụ.

Microservices, hay còn gọi là kiến trúc microservice, là một kiểu kiến ​​trúc cấu trúc ứng dụng như một tập hợp các dịch vụ tự trị nhỏ được mô hình hóa xung quanh một miền.

## Phần III: Spring Cloud Netflix:

### Spring Cloud Netflix là gì?

Spring Cloud Netflix là 1 bộ thư viện cung cấp tích hợp Netflix OSS cho các ứng dụng Spring Boot thông qua tự động định cấu hình và ràng buộc với Spring Environment và các mô hình lập trình Spring khác. Có thể nhanh chóng cấu hình các mẫu phổ biến bên trong ứng dụng và xây dựng các hệ thống phân tán với các thành phần Netflix bằng các annotation. Các mẫu được cung cấp bao gồm Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) và Client Side Load Balancing (Ribbon) ...

### Spring Cloud Netflix hoạt động như thế nào?

Eureka: Eureka Server là nơi để các service có thể đăng ký thông tin của mình để các service khác có thể discovery và gọi tới.

Ribbon: Định tuyến động và cân bằng tải.

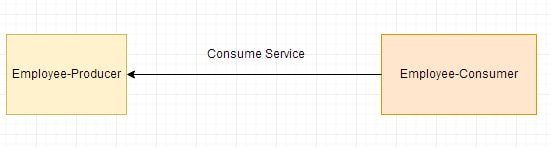
Hystrix: Circuit Breaker có thể cho phép một microservice tiếp tục hoạt động khi một dịch vụ liên quan bị lỗi, ngăn chặn lỗi xảy ra theo tầng và cho phép thời gian phục hồi dịch vụ bị lỗi.

Zuul: Zuul là một dịch vụ cổng cung cấp định tuyến động, giám sát, khả năng phục hồi, bảo mật, nó cấu hình service id của các service trong Eureka Server với context path tương ứng của service id đó, xuất ra một URL để các ứng dụng bên ngoài có thể access tới, sau đó dựa vào context path mà các ứng dụng đang request tới để forward request tới đúng ứng dụng bên trong.

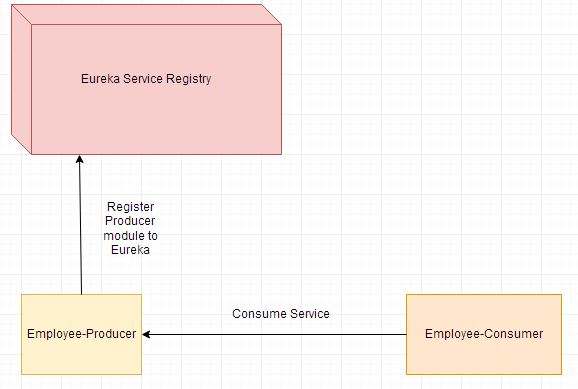
### Eureka Server:

Vì tất cả các dịch vụ đều được đăng ký với máy chủ Eureka và việc tra cứu được thực hiện bằng cách gọi Máy chủ Eureka, bất kỳ thay đổi nào về vị trí dịch vụ không cần được xử lý và giám sát.

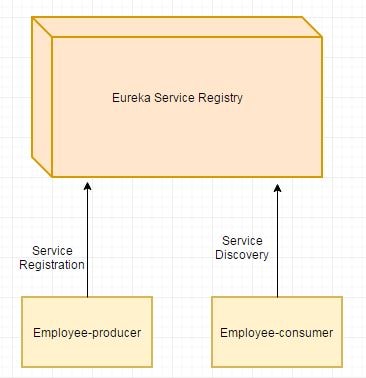
Ví dụ về bài toán:



producer – consumer



Sử dụng eureka server để đăng ký service

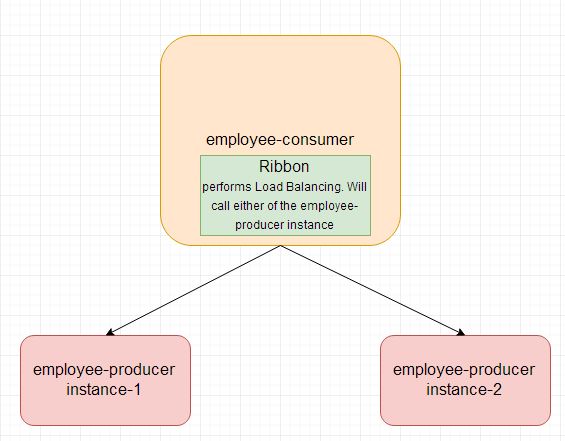


Sử dụng eureka server để khám phá service

### Cân bằng tải (load balancing) là gì? Ribbon Netflix và cân bằng tải

Trong máy tính, cân bằng tải cải thiện việc điều phối khối lượng công việc trên nhiều tài nguyên máy tính, chẳng hạn như máy tính, cụm máy tính, liên kết mạng, đơn vị xử lý trung tâm hoặc ổ đĩa. Cân bằng tải nhằm mục đích tối ưu hóa việc sử dụng tài nguyên, tối đa hóa thông lượng, giảm thiểu thời gian phản hồi và tránh quá tải cho bất kỳ tài nguyên đơn lẻ nào. Sử dụng nhiều thành phần với cân bằng tải thay vì một thành phần duy nhất làm tăng độ tin cậy và tính khả dụng thông qua việc dự phòng tài nguyên. Cân bằng tải thường liên quan đến phần mềm hoặc phần cứng chuyên dụng, ví dụ như multilayer switch hoặc Domain Name System server process

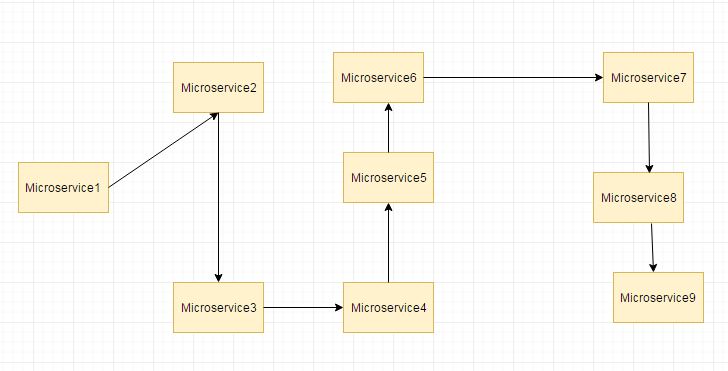
Trong ví dụ service producer – consumer. Giả sử các mô-đun khác cũng đang gọi và sử dụng các dịch vụ của mô-đun producer – consumer. Vì vậy, tải producer – consumer sẽ khá cao. Để giải quyết vấn đề này, cần triển khai nhiều trường hợp producer – consumer. Cần sử dụng Load Balancer để định tuyến bất kỳ yêu cầu nào đến một trong hai dịch vụ này.



### Netflix Hystrix là gì? Dùng nó trong việc gì?

Hystrix là một thư viện về độ trễ và khả năng chịu lỗi, được thiết kế để phân biệt các điểm truy cập vào các hệ thống, dịch vụ từ xa và thư viện của bên thứ 3, ngăn chặn lỗi phân tầng và cho phép khả năng phục hồi trong các hệ thống phân tán phức tạp.

Thông thường đối với các hệ thống được phát triển bằng kiến ​​trúc Microservices, có rất nhiều microservices tham gia. Các microservices này cộng tác với nhau.

Hãy xem xét các microservices sau- 

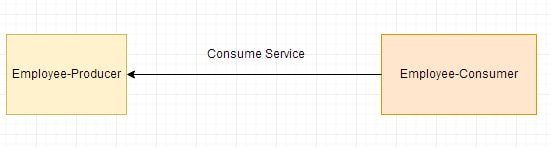
Giả sử nếu microservice 9 trong sơ đồ không thành công, thì sử dụng cách tiếp cận truyền thống, chúng ta sẽ trả về một ngoại lệ. Nhưng điều này vẫn sẽ khiến toàn bộ hệ thống gặp sự cố.

Vấn đề này trở nên phức tạp hơn khi số lượng microservice tăng lên. Số lượng microservices có thể lên đến 1000. Đây là lúc hystrix có tác dụng-

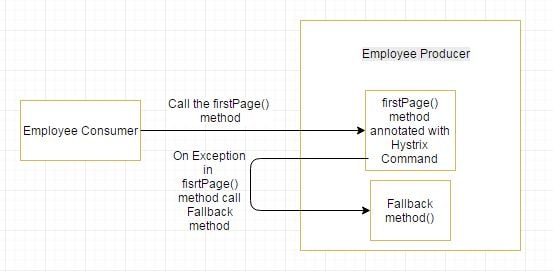
Có hai tính năng của Hystrix:

Phương pháp dự phòng (Fallback method)

Ngắt mạch (Circuit Breaker)

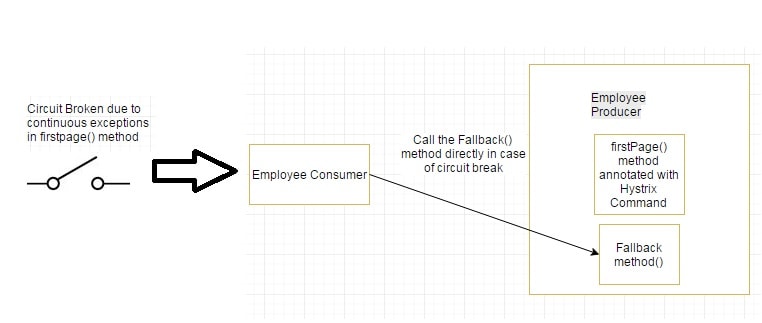


Bây giờ, giả sử do một số lý do mà producer – consumer tiếp xúc với dịch vụ đưa ra một ngoại lệ. Trong trường hợp sử dụng Hystrix - phương pháp dự phòng. Phương thức dự phòng này phải có cùng kiểu trả về với service được hiển thị. Trong trường hợp ngoại lệ trong service được hiển thị, phương thức dự phòng sẽ trả về một số giá trị.



Nếu các ngoại lệ tiếp tục xảy ra trong phương thức firstPage() thì Hystrix sẽ bị ngắt và consumer sẽ bỏ qua tất cả phương thức firtsPage và gọi trực tiếp phương thức dự phòng.

Mục đích của ngắt mạch là cung cấp thời gian cho phương thức firstPage() hoặc các phương thức khác mà phương thức firstPage() có thể đang gọi và khôi phục ngoại lệ. Có thể xảy ra khi tải thấp, sự cố gây ra các ngoại lệ có cơ hội khôi phục tốt hơn

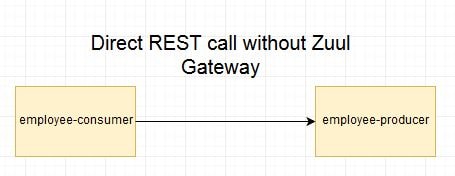


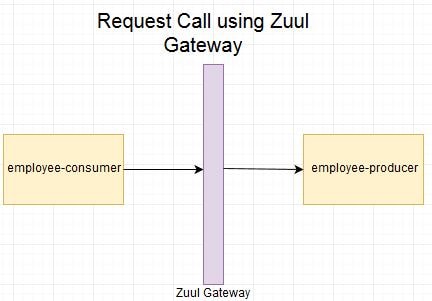
### Netflix Zuul là gì? Tại sao cần nó?

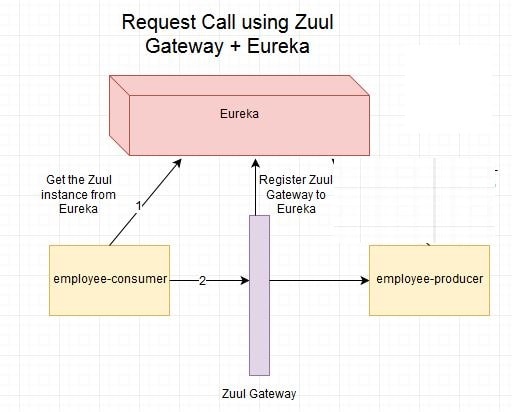
Zuul là bộ định tuyến dựa trên JVM và bộ cân bằng tải phía máy chủ của Netflix.

Nó cho phép trình duyệt, ứng dụng dành cho thiết bị di động hoặc giao diện người dùng khác sử dụng dịch vụ từ nhiều máy chủ mà không cần quản lý chia sẻ tài nguyên cross-origin (CORS) và xác thực cho từng máy chủ. Có thể tích hợp Zuul với các dự án Netflix khác như Hystrix để chịu lỗi và Eureka để khám phá dịch vụ hoặc sử dụng nó để quản lý việc định tuyến, bộ lọc và cân bằng tải trên hệ thống

Cuộc gọi microservice mà không có Netflix Zuul



Cuộc gọi microservice với Netflix Zuul  


Cuộc gọi microservice với Netflix Zuul + Netflix Eureka  


### Tổng kết phần III:

* Spring cloud Netflix là một bộ thư viện tích hợp Netflix OSS với Spring Boot giúp chúng ta dễ dàng hiện thực xây dựng các ứng dụng Microservices
* Eureka Server là nơi các microservice đăng ký dịch vụ của mình để gateway có thể phát hiện.
* Load balancing là cơ chế gọi xen kẽ các service giúp cân bằng tải.
* Netflix Hystrix là một cơ chế kiểm soát lỗi.
* Zuul là một cơ chế định tuyến động, giúp gateway có thể phát hiện – gọi service cần thiết mà không quan tâm đến địa chỉ ip hay url.

## Phần IV: API

### Api là gì:

API được viết tắt bởi cụm từ tiếng Anh Application Programming Interface, tức giao diện lập trình ứng dụng. Đây là phương tiện cho hai hoặc nhiều ứng dụng trao đổi, tương tác với nhau, tạo ra tương tác giữa người dùng với ứng dụng hiệu quả và tiện lợi hơn hoặc API là phương thức để **kết nối**, **trao đổi dữ liệu** giữa các **ứng dụng** với nhau. API có thể được sử dụng cho hệ thống nhúng, phần cứng, phần mềm hay gần đây nhất là các hệ thống microservice với nhiều thành phần trong hệ thống được tách biệt riêng với nhau.

Với API, các lập trình viên có thể tiếp cận, truy xuất dữ liệu từ máy chủ thể hiện chúng trên ứng dụng phần mềm hoặc website của mình một cách dễ dàng hơn.

Vì một phần mềm chứa rất nhiều logic phức tạp, nên người ta tìm cách chia nhỏ nó ra thành nhiều phần, mỗi phần này tạm gọi là một component. Mỗi component sẽ có tính độc lập cao, ít phụ thuộc hoặc có thể không phục thuộc vào các thành phần khác. Tuy là có tính độc lập cao, nhưng để có thể kết nối được với nhau mà một phần mềm hoàn chỉnh, buộc chúng vẫn phải tuân theo một hoặc một số chuẩn nào đó. Thì mỗi cái chuẩn đó được gọi là một giao diện lập trình ứng dụng – hay chính là một API.

API hiện có 2 chính sách bảo vệ cơ bản gồm:

* Các công ty bảo vệ API của chính mình
* Các công ty chuyên cung cấp API miễn phí

Với loại thứ 1, ở đây các công ty thường sẽ chủ trương bảo vệ API và thu lời từ các nhà phát triển phần mềm thứ 3 đăng ký, xin phép họ. Họ thường là những công ty sản xuất thiết bị, game, công nghệ, ví dụ như Sony cùng hệ thống playstation.

Với loại thứ 2, họ là các công ty chuyên cung cấp các sản phẩm API miễn phí. Người dùng có thể thoải mái sử dụng API này để viết lên phần mềm bên thứ 3. Tuy nhiên, người dùng vẫn cần mua thêm phần mềm để sử dụng. Đây cũng chính là nguồn lợi của các nhà cung cấp API miễn phí, đồng thời vừa có thể đem tới hiệu quả lan tỏa và marketing mạnh mẽ hơn.

Điển hình cho loại thứ 2 đó là hệ sinh thái của Microsoft, Google hay Apple. Họ hầu như đều cung cấp các API miễn phí, đổi lại là các lập trình viên sẽ viết phần mềm trên hệ điều hành, người dùng phải mua của Windows, Google để có thể sử dụng phần mềm đó. Đặc biệt trong đó, API của Google chính là ứng dụng trong hệ sinh thái giúp họ ngày càng phổ biến và dễ dàng tiếp cận mọi người khắp thế giới hơn.

Hiện nay, xu hướng người dùng chủ yếu thiên về loại thứ 2. Điều này khiến cho các API miễn phí ngày càng trở nên phổ biến hơn, vận hành và có mặt ở mọi nơi trên internet và cuộc sống.

### API hoạt động như thế nào?

API cho phép các thành phần của hai ứng dụng nói chuyện với nhau bằng cách sử dụng một tập hợp các lệnh đơn giản. Về cơ bản, API thực hiện việc gửi yêu cầu và phản hồi giữa các ứng dụng.

Trong mỗi lần tương tác, có một **máy chủ** (ứng dụng cung cấp tài nguyên) và một **máy khách** (ứng dụng đưa ra yêu cầu). Nếu máy chủ có thể thực hiện những gì máy khách yêu cầu, thì API sẽ trả về các tài nguyên có liên quan.

Nếu máy khách yêu cầu một tài nguyên không tồn tại trên máy chủ hoặc máy khách không có quyền truy cập, thì thay vào đó, API sẽ cung cấp thông báo lỗi. Các API có thể kiểm soát truy cập có chọn lọc đối với máy chủ.

### Lợi ích:

* Bằng cách sử dụng lại các thành phần thông qua API, các developer không phải lặp lại công việc đã thực hiện.
* **Tự do sử dụng bên ngoài:** API cho phép sử dụng kiến ​​thức chuyên môn của người khác (trên toàn cầu) và tích hợp vào hệ thống mà không cần phải viết lại các module mà người khác đã phát triển.
* **Trải nghiệm người dùng tốt hơn:** Nhóm phát triển có thể là những người giỏi nhất, nhưng họ không thể biết được mọi tính năng mà khách hàng muốn. Bằng cách sử dụng các API bên ngoài, có thể tối ưu hóa và mở rộng dần cho ứng dụng. Tạo trải nghiệm người dùng tốt nhất - với chi phí thấp.

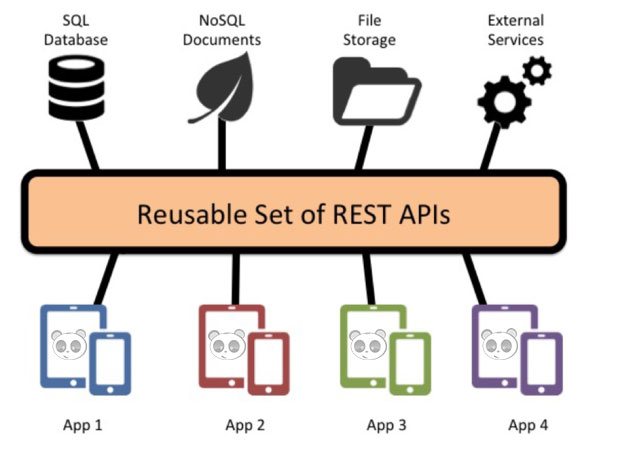
## SOAP-REST – RESTFul API trong thiết kế website

### ****SOAP web service****

**S**imple **O**bject **A**ccess **P**rotocol là một dạng giao thức (cũng có thể coi là một chuẩn). SOAP sử dụng XML làm cấu trúc dữ liệu trả về. Tuy nhiên SOAP không có quy ước về cách viết url cũng như http method. Nhưng bù lại, SOAP lại có WS-Security SOAP – là một chuẩn giúp an toàn dữ liệu, giải quyết được vấn đề **an toàn dữ liệu**

### ****SOAP vs RESTful****

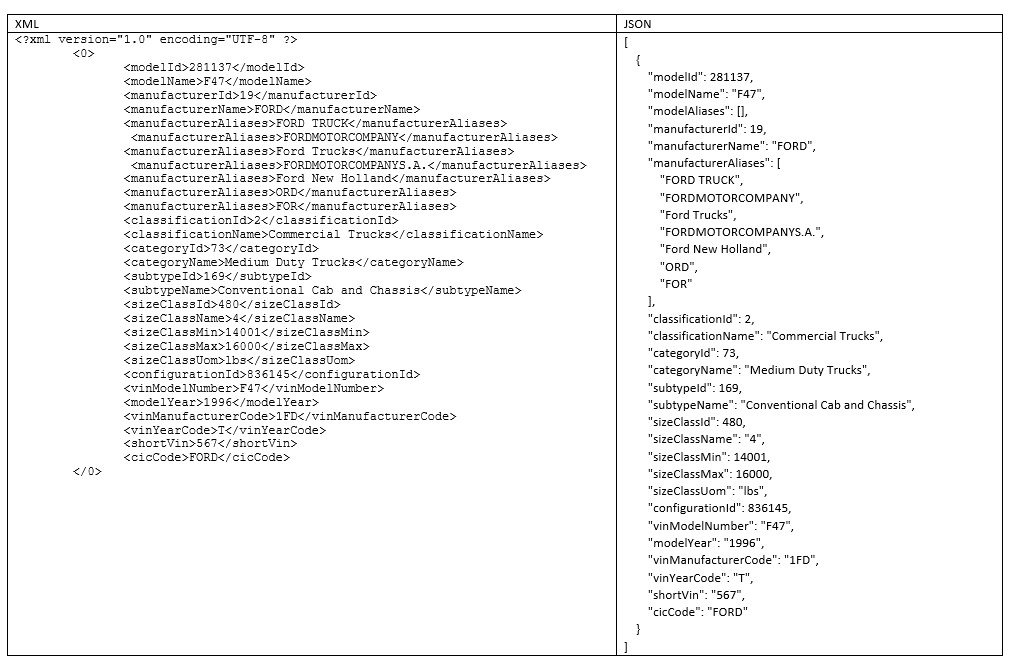
Ngày nay các dự án web service đa phần (thậm chí gần như tất cả) đều sử dụng RESTful thay vì sử dụng SOAP. RESTful có quy ước rõ ràng hơn hẳn SOAP. Mặt khác RESTful có thể sử dụng nhiều loại dữ liệu để trả về, trong đó có cả XML, vậy xét ở góc độ nào đó có thể nói rằng RESTful bao gồm cả SOAP cũng không sai.



**SOAP** – (Simple Object Access Protocol: Giao thức truy cập đối tượng đơn giản) - là một giao thức được chuẩn hóa với các quy tắc nghiêm ngặt. Nó khá nặng khiến tốc độ tải chậm hơn trong việc truyền dữ liệu. Các API dựa trên SOAP nặng hơn so với API dựa trên RESTful, môi trường SOAP sẽ phải mạnh hơn nhiều để cung cấp cùng tốc độ.

**REST** – (REpresentational State Transfer- Chuyển trạng thái đại diện). Nó không phải là một giao thức mà là một kiểu kiến ​​trúc, ban đầu nó được tạo ra để giải quyết các vấn đề về tốc độ và độ khó của SOAP. Trong khi SOAP chỉ cho phép một định dạng dữ liệu trả về là XML; REST với kiến ​​trúc nhẹ và hiệu suất tốt hơn nhiều, có thể truyền nội dung ở nhiều định dạng như HTML, JSON, XML, YAML và văn bản. JSON là một trong những định dạng trao đổi dữ liệu phổ biến nhất.

Ví dụ sau đây cho thấy sự khác biệt giữa SOAP và REST. Với ví dụ cột bên trái được viết bằng SOAP XML và ví dụ cột bên phải về cùng một chuỗi mã bằng JSON.



Nói chung, JSON là lựa chọn tốt nhất cho các hệ thống hướng dữ liệu với dữ liệu là tài nguyên hệ thống với khả năng mở rộng, tính linh hoạt, thân thiện với trình duyệt, tốc độ truyền dữ liệu cao hơn và hiệu suất tốt hơn. Các API RESTful dựa trên JSON là lựa chọn tốt nhất cho các doanh nghiệp muốn có tốc độ truyền dữ liệu cao.

https://openinsurance.io/tech-lab/what-are-apis-why-do-we-need-them/

### REST và RESTful API là gì?

REST được viết tắt bởi Representational State Transfer, là dạng cấu trúc quy định các ứng dụng giao tiếp và tương tác với nhau. Nó bao gồm 3 bộ phận cơ bản như: bộ máy chủ ngoài chứa dữ liệu (external server), máy chủ API và máy chủ khách (client). Trong đó, máy khách có thể là bất cứ thứ gì, ứng dụng trên nền web, thư viện hoặc thậm chí là các phần mềm khác nhau của một phần mềm máy tính.

REST cho phép các máy khách truy cập máy chủ API và thực hiện các lệnh lấy về, chỉnh sửa hay xóa dữ liệu từ external server. Các lập trình viên có thể thoải mái truy xuất, chỉnh sửa dữ liệu từ máy chủ mà không cần biết hệ thống hoạt động như thế nào.

Giao thức chính của REST sử dụng là HTTP, một giao thức phổ biến với hầu hết các ứng dụng hay dịch vụ web hiện nay. Nó đem tới hiệu quả nhanh chóng trong bối cảnh đường truyền mạnh mẽ và khiến cho REST có tốc độ nhanh hơn.

Ngoài ra, REST cũng có ưu điểm khi sử dụng giao thức stateless (không trạng thái). Hệ thống này không sử dụng session, cookie, không cần biết những thông tin đó trong mỗi lần request đến máy chủ ngoài. Điều này giúp REST giảm tải cho máy chủ ngoài, nâng cao hiệu suất làm việc.

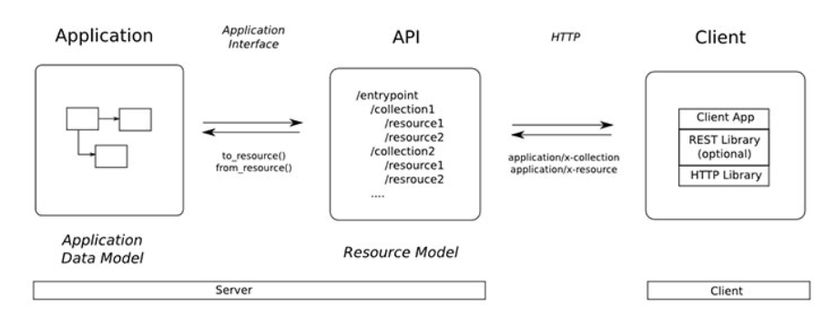
REST ban đầu được thiết kế để sử dụng cho các dịch vụ web. Tuy nhiên, bất cứ phần mềm nào cũng có thể ứng dụng REST làm cầu nối giao tiếp với các ứng dụng khác tốt và hiệu quả hơn. Đó cũng là lý do giúp cho REST trở thành tiêu chuẩn mặc định cho hầu hết các giao thức. Và những API được thiết kế theo cấu trúc REST được gọi là RESTful API.

RESTful API là những API của web service sử dụng theo chuẩn RESTful. Trước khi áp dụng RESTful để tạo API, người ta sẽ đưa ra các chuẩn (API) trước. Ví dụ quy định nếu thực hiện thêm users thành công, thì sẽ phải trả về header status là 200, kèm một tin nhắn có nội dung là “thành công” chẳng hạn, ai mà làm sai theo quy tắc này tức là sai API, và endpoint đó sẽ chỉ được coi là RESTful endpoint chứ không được coi là RESTful API.

Kiến trúc REST có 1 số đặc điểm như sau

* Nhất quán xuyên suốt các API. Ví dụ: luôn luôn sử dụng danh từ số nhiều thay vì khi số nhiều, khi số ít
* Tồn tại mà không lưu trạng thái (Stateless existence), ví dụ như không sử dụng session của server. Đây chính là lý do vì sao lại có cụm State Transfer (ST) trong REST. Tính chất này đảm bảo rằng mỗi khi client submit 1 request, request ấy sẽ bao gồm toàn bộ các thông tin mà server cần để xử lý request. Đây là điều cần thiết để build 1 ứng dụng phục vụ lên tới hàng triệu user.
* Sử dụng HTTP status code khi có thể
* Sử dụng URL Endpoint có phân tầng logic
* Đánh version trong URL thay vì trong HTTP Headers

## RESTful API hoạt động như thế nào?



REST hoạt động chủ yếu dựa vào giao thức HTTP. Các hoạt động cơ bản sẽ sử dụng những phương thức HTTP riêng:

### 4.1. Quy ước về các method:

* GET: được sử dụng để lấy thông tin từ sever theo URI đã cung cấp.
* POST: gửi thông tin tới sever thông qua các biểu mẫu http (đăng kí chả hạn ...)
* HEAD: giống với GET nhưng response trả về không có body, chỉ có header
* PUT: ghi đè tất cả thông tin của đối tượng với những gì được gửi lên
* PATCH: ghi đè các thông tin được thay đổi của đối tượng.
* DELETE: xóa tài nguyên trên server.
* CONNECT: thiết lập một kết nối tới server theo URI.
* OPTIONS: mô tả các tùy chọn giao tiếp cho resource.
* TRACE: thực hiện một bài test loop – back theo đường dẫn đến resource.

Trên thực tế, ta thường hay chỉ sử dụng các phương thức GET, POST, PUT, DELETE. Những phương thức hay hoạt động này thường được gọi là CRUD tương ứng với Create, Read, Update, Delete – Tạo, Đọc, Sửa, Xóa.

### 4.2. Quy ước về resource, endpoint

**Resource** chính là dữ liệu mà chúng ta phải quản lý, có thể là customers, products, posts, images, videos… Mặt khác, tên resource cũng sẽ xuất hiện trong cách viết endpoint, nên nếu đặt tên cho resource một cách khoa học, thì endpoint cũng trở nên dễ hiểu và dễ tiếp cận hơn.

Xem ví dụ trong bảng sau để hiểu rõ đâu là resource, và resource thường được viết như thế nào trong mỗi endpoint

|  |  |
| --- | --- |
| Endpoint | Resource |
| http://api.example.com/users | users |
| http://api.example.com/users/1/accounts | accounts |
| http://api.example.com/users/1/images | images |

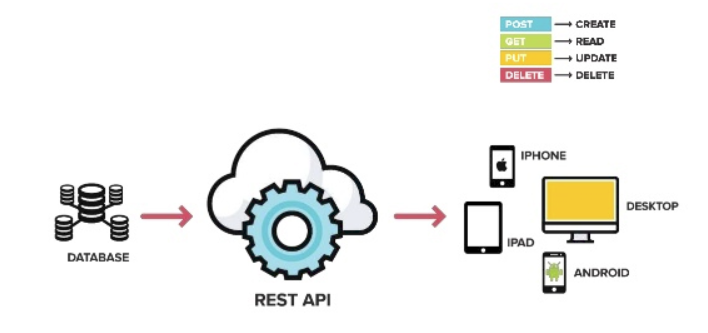
#### Status code

Khi chúng ta request một API nào đó thường thì sẽ có vài status code để nhận biết sau:

* 200 OK – Trả về thành công cho những phương thức GET, PUT, POST hoặc DELETE.
* 201 Created – Trả về khi một Resouce vừa được tạo thành công.
* 204 No Content – Trả về khi Resource xoá thành công.
* 304 Not Modified – Client có thể sử dụng dữ liệu cache.
* 400 Bad Request – Request không hợp lệ
* 401 Unauthorized – Request cần có auth.
* 403 Forbidden – bị từ chối không cho phép.
* 404 Not Found – Không tìm thấy resource từ URI
* 405 Method Not Allowed – Phương thức không cho phép với user hiện tại.
* 410 Gone – Resource không còn tồn tại, Version cũ đã không còn hỗ trợ.
* 415 Unsupported Media Type – Không hỗ trợ kiểu Resource này.
* 422 Unprocessable Entity – Dữ liệu không được xác thực
* 429 Too Many Requests – Request bị từ chối do bị giới hạn.

Khi viết api cho app ios hay client side, chúng ta nên đặt version cho các api. Ví dụ như endpoint sau: api/v1/users. Điều này sẽ giúp hệ thống sau khi nâng cấp lên version mới vẫn hộ trợ các api của version cũ, cũng như giúp việc bảo trì, sửa chữa dễ dàng hơn.

### 4.3. Ưu điểm, nhược điểm của RESTFUL API:



#### 4.3.1. Ưu điểm:

* Giúp cho ứng dụng rõ ràng hơn
* REST URL đại diện cho resource chứ không phải hành động
* Dữ liệu được trả về với nhiều định dạng khác nhau như: xml, html, json….
* Code đơn giản và ngắn gọn
* REST chú trọng vào tài nguyên của hệ thống

Những trang web ngày nay thường sử dụng REST API để cho phép kết nối đến dữ liệu của họ. Trong đó, facebook cũng cung cấp các REST API để giúp các ứng dụng bên ngoài kết nối đến dữ liệu của họ

khi ứng dụng của bạn không đặt nặng vào vấn đề bảo mật, hoặc khi máy chủ của bạn không đủ mạnh để có thể kiểm soát tài nguyên thì bạn hãy sử dụng REST.

#### 4.3.2. Nhược điểm:

* Bảo mật kém hơn so với SOAP
* Do không lưu trạng thái giữa Client - Server nên khối lượng thông tin truyền tải lớn -> tốn tài nguyên

### Tổng kết phần IV:

REST: là một dạng chuyển đổi cấu trúc dữ liệu, một kiểu kiến trúc để viết API. Nó sử dụng phương thức HTTP đơn giản để tạo cho giao tiếp giữa các máy. Vì vậy, thay vì sử dụng một URL cho việc xử lý một số thông tin người dùng, REST gửi một yêu cầu HTTP như GET, POST, DELETE, vv đến một URL để xử lý dữ liệu.

RESTful: là một tiêu chuẩn dùng trong việc thiết kế các API với kiến trúc REST cho các ứng dụng web để quản lý các resource. RESTful là một trong những kiểu thiết kế API được sử dụng phổ biến ngày nay để cho các ứng dụng (web, mobile…) khác nhau giao tiếp với nhau.

API: là một tập các quy tắc và cơ chế mà theo đó, một ứng dụng hay một thành phần sẽ tương tác với một ứng dụng hay thành phần khác. API có thể trả về dữ liệu cần cho ứng dụng của mình ở những kiểu dữ liệu phổ biến như JSON hay XML.

RESTful API: Những API được thiết kế theo cấu trúc REST được gọi là RESTful API

## Phần V các cơ chế bảo mật:

### JWT

JSON web token (JWT) là một tiêu chuẩn mở định nghĩa về việc truyền thông tin an toàn giữa các bên dưới dạng đối tượng JSON. Thông tin này có thể được xác minh và độ tin cậy cao vì nó có chữ ký điện tử. JWT có thể được ký bằng cách sử dụng mã hõa (với thuật toán **HMAC**) hoặc cặp khóa public/private bằng **RSA** hoặc **ECDSA**.

**Ủy quyền**: Đây là tình huống phổ biến nhất để sử dụng JWT. Khi người dùng đã đăng nhập, mỗi yêu cầu tiếp theo sẽ bao gồm JWT, cho phép người dùng truy cập dịch vụ và tài nguyên được phép với token đó. Đăng nhập một lần là một tính năng được sử dụng rộng rãi JWT ngày nay, vì chi phí nhỏ và khả năng dễ dàng sử dụng trên các miền khác nhau.

**Trao đổi thông tin**: JSON web token là một cách tốt để truyền thông tin giữa các bên một cách an toàn. Bởi vì JWT có thể được ký

ví dụ: sử dụng cặp khóa public/private

chữ ký được tính bằng cách sử dụng tiêu đề và nội dung, bạn cũng có thể xác minh rằng nội dung không bị giả mạo.

Cấu trúc JSON web token:

bao gồm ba phần được phân tách bằng dấu chấm (.), đó là:

Header (tiêu đề)

Payload (khối hàng)

Signature (chữ ký)

Do đó, JWT thường trông giống như sau.

xxxxx.yyyyy.zzzzz

#### Header

Thường bao gồm hai phần: loại mã và thuật toán đang được sử dụng, chẳng hạn như HMAC SHA256 hoặc RSA.

Ví dụ:

{

"alg": "HS256",

"typ": "JWT"

}

Sau đó, JSON này được mã hóa **Base64Url** để tạo thành phần header JWT.

#### Payload

Xác nhận quyền và dữ liệu bổ sung. Có ba loại xác nhận quyền: xác nhận quyền đã đăng ký, public và private.

[**Các xác nhận quyền đã đăng ký**](https://tools.ietf.org/html/rfc7519#section-4.1) : cung cấp một tập hợp các xác nhận quyền. Một số trong số đó là:**Iss** (nhà phát hành),**exp** (thời gian hết hạn),**sub** (chủ đề),**aud** (audience) và [những thứ khác](https://tools.ietf.org/html/rfc7519#section-4.1) .

[**Public clame**](https://tools.ietf.org/html/rfc7519#section-4.2): Được xác định theo ý muốn của những người sử dụng JWT. Nhưng để tránh tranh chấp, nên được xác định trong [IANA Json Web T](https://www.iana.org/assignments/jwt/jwt.xhtml)oken Registry hoặc một URI có chứa không gian tên để chống tranh chấp.

[**Private**](https://tools.ietf.org/html/rfc7519#section-4.3) clame : Đây là những xác nhận quyền sở hữu tùy chỉnh được tạo ra để chia sẻ thông tin giữa các bên đồng ý sử dụng chúng và không phải làxác nhận quyền sở hữu đã đăng ký hoặc công khai .

Ví dụ:

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

sau đó được mã hóa **Base64Url** để tạo thành payload của JSON Web token.

Thông tin này, mặc dù được bảo vệ để chống giả mạo, nhưng bất kỳ ai cũng có thể đọc được. Không nên tiết lộ các thông tin bí mật trong các phần tử payload hoặc header của JWT trừ khi nó được mã hóa.

#### Signature (chữ ký)

Để tạo phần chữ ký phải lấy header được mã hóa, payload được mã hóa, khóa, thuật toán được chỉ định trong header và chữ ký đó.

Ví dụ: nếu sử dụng thuật toán HMAC SHA256, chữ ký sẽ được tạo theo cách sau:

HMACSHA256(

base64UrlEncode(header) + "." +

base64UrlEncode(payload),

secret)

Chữ ký được sử dụng để xác minh thông điệp không bị thay đổi trong quá trình thực hiện và, trong trường hợp các token được ký bằng khóa cá nhân, nó cũng có thể xác minh rằng đối tượng đang gửi JWT là đối tượng đang giao tiếp.

Kết hợp tất cả lại với nhau

Đầu ra là ba chuỗi Base64-URL được phân tách bằng dấu chấm có thể dễ dàng chuyển trong môi trường HTML và HTTP, đồng thời nhỏ gọn hơn khi so sánh với các tiêu chuẩn dựa trên XML như SAML.

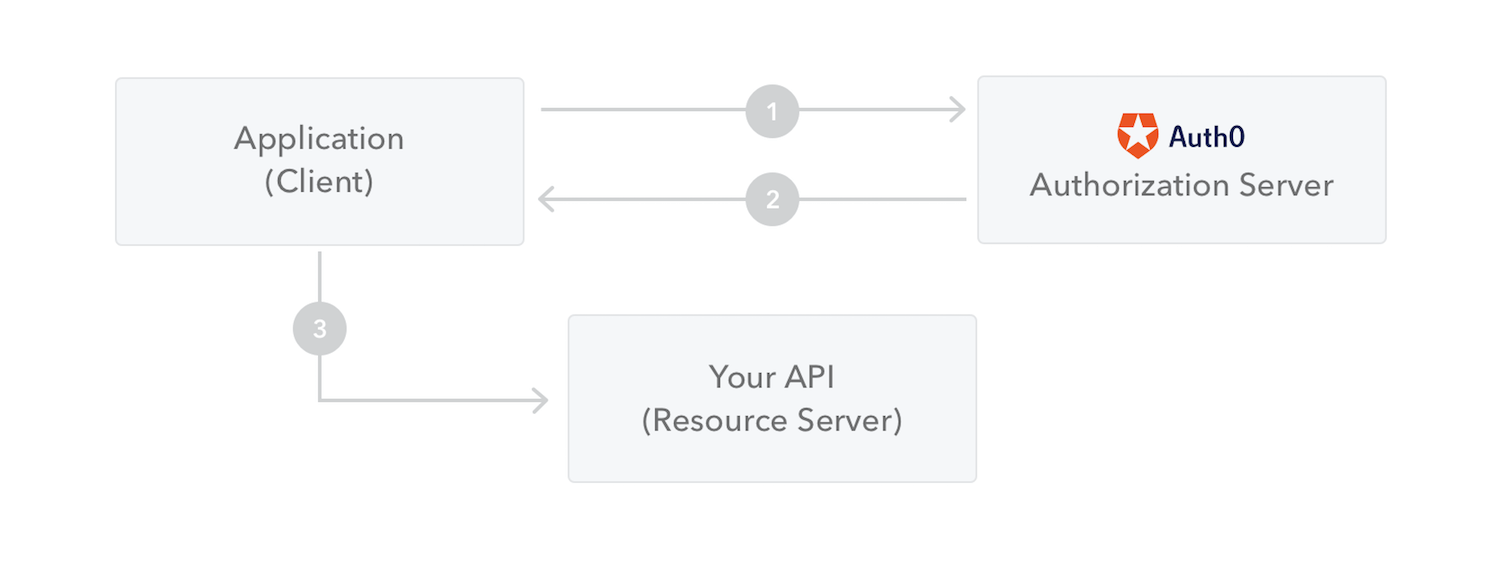
Trong xác thực, khi người dùng đăng nhập thành công bằng thông tin đăng nhập, JWT sẽ được trả lại. Vì token là thông tin xác thực nên cần cẩn thận để ngăn chặn các vấn đề về bảo mật.

Bất cứ khi nào người dùng truy cập tài nguyên được bảo vệ, người dùng phải gửi JWT, thường là trong header Authorization bằng cách sử dụng lược đồ Bearer . Nội dung của tiêu đề sẽ giống như sau:

Authorization: Bearer <token>

Các lớp vảo vệ của máy chủ sẽ kiểm tra JWT hợp lệ trong Authorization header và nếu nó hợp lệ, người dùng sẽ được phép truy cập các tài nguyên được bảo vệ. Nếu JWT chứa dữ liệu cần thiết, nhu cầu truy vấn cơ sở dữ liệu cho các hoạt động nhất định có thể được giảm bớt.

Sơ đồ sau cho thấy cách JWT được lấy và sử dụng để truy cập các API hoặc tài nguyên:



Ứng dụng hoặc client yêu cầu ủy quyền cho authorization server. Khi ủy quyền được cấp, authorization server trả về access token cho ứng dụng. Ứng dụng sử dụng access token để truy cập tài nguyên được bảo vệ (API).

### Open ID

OpenID là một hệ thống đăng nhập một lần không có tính tập trung. Đối với những trang web có sử dụng OpenID thì người sử dụng không cần phải nhớ các thông tin về username và password cho riêng trang đó nữa. Thay vào đó họ chỉ cần đăng ký trước 1 tài khoản OpenID tại một trong những nhà cung cấp OpenID, hay thường gọi là i-broker. Do OpenID không mang tính tập trung nên bất kỳ trang web nào cũng có thể sử dụng được OpenID như là một cách đăng nhập cho người dùng.

OpenID hiện đang được ứng dụng rộng rãi trong các trang web lớn như AOL, Yahoo, Google, Facebook, Youtube... Các mã nguồn mở như NukeViet, Joomla... được hỗ trợ đăng nhập OpenID cũng cho phép người dùng dần quen với việc sử dụng openid trong các dịch vụ.

Việc ứng dụng OpenID vào phát triển ứng dụng (A) về bản chất thực ra chính là việc ứng dụng A thông qua một tải khoản đã có của người dùng U được đăng ký trên OpenID provider để thay cho việc đăng ký tài khoản mới trên A. Thông qua OpenID Provider, ứng dụng A xác thực được người dùng U mà không cần phải đăng ký tài khoản trên A. Như vậy sử dụng một tài khoản duy nhất được đăng ký bởi OpenID Provider người dùng có thể sử dụng để xác thực (đăng nhập) trên nhiều ứng dụng khác nhau mà không cần đăng ký tài khoản trên từng ứng dụng này.

Khi ứng dụng A sử dụng OpenID để xác thực người dùng thì quy trình diễn ra khá đơn giản như sau:

* Ứng dụng A chuyển tiếp người dùng về một URL của OpenID provider để đăng nhập.
* Nếu đăng nhập thành công OpenID Provider sẽ chuyển tiếp người dùng về trang của ứng dụng A và thông báo với ứng dụng này rằng người dùng U đã được xác thực.
* Cuối cùng ứng dụng A thực hiện việc xác thực người dùng U mà không cần đăng nhập (vì A tin vào kết quả trả về của OpenID Provider).

### SSO

Single Sign-on (SSO) xảy ra khi người dùng đăng nhập vào một ứng dụng và sau đó tự động đăng nhập vào các ứng dụng khác, bất kể platform, technology hoặc domain mà người dùng đang sử dụng. Người dùng chỉ đăng nhập một lần, do đó tên của tính năng này là Single Sign-on.

#### Lợi ích

SSO cung cấp trải nghiệm liền mạch cho người dùng khi sử dụng các ứng dụng và dịch vụ cho người dùng. Thay vì phải nhớ thông tin đăng nhập riêng cho từng ứng dụng hoặc dịch vụ, người dùng chỉ cần đăng nhập một lần và truy cập các ứng dụng khác.

Bất cứ khi nào người dùng truy cập một domain mà yêu cầu xác thực, họ sẽ được chuyển hướng đến domain khác để xác thực, tại đây có thể yêu cầu đăng nhập. Nếu người dùng đã đăng nhập, họ có thể được chuyển hướng ngay lập tức đến domain truy cập ban đầu mà không cần đăng nhập lần nữa.

Single Sign-on và Single Logout một lần có thể sử dụng thông qua session. Có thể có tối đa 3 session khác nhau cho người dùng có SSO:

* Local session được duy trì bởi ứng dụng
* Authorization Server session, nếu SSO được kích hoạt
* Identity Provider session, nếu người dùng chọn đăng nhập thông qua Identity Provider (như Google, Facebook hoặc 1 Identity Provider dạng SAML)

Với SSO, một domain trung tâm thực hiện xác thực và sau đó chia sẻ session với các domain khác. Cách một session được chia sẻ có thể khác nhau giữa các protocol SSO, nhưng khái niệm chung là giống nhau.

Ví dụ: Domain xác thực có thể tạo 1 JSON Web Token (JWT) (Được mã hóa bằng JSON Web Encryption (JWE)) đã được signed, chứa tất cả thông tin cần thiết để xác định người dùng cho bất kỳ domain khác cần xác thực. Token này được truyền cho client, nhưng vì đã signed, nên client không thể sửa đổi nó theo bất kỳ cách nào. Mã thông báo có thể được chuyển đến domain ban đầu bằng chuyển hướng và được sử dụng bởi domain xác thực và 1 vài domain khác để xác định người dùng.

#### Luồng chạy chính của hệ thống sử dụng SSO

Client redirect người dùng tới Provider cho việc xác minh

Người dùng đăng nhập vào Provider

Provider redirect người dùng trở lại Client với một token được sinh ra ngẫu nhiên

Client sử dụng token đó để tạo lời gọi API tới Provider cùng với ID và Secret Key tạo nên Access Token

Những request sau được xác minh thông qua Access Token

Đăng xuất xóa bỏ session ở Client cũng như Provider và database

### Oauth2

#### Oauth2 là gì

Trong mô hình xác thực client – server truyền thống, client yêu cầu một tài nguyên bị hạn chế truy cập (tài nguyên được bảo vệ) trên server bằng cách xác thực với server các thông tin xác thực. Để cung cấp cho các ứng dụng của bên thứ ba quyền truy cập vào tài nguyên bị hạn chế đó, chủ sở hữu tài nguyên phải chia sẻ thông tin đăng nhập đó với bên thứ ba. Điều này tạo ra một số vấn đề và hạn chế:

* Các ứng dụng của bên thứ ba được yêu cầu lưu trữ thông tin đăng nhập của chủ sở hữu để sử dụng trong tương lai, thường là mật khẩu.
* Server được yêu cầu để hỗ trợ xác thực mật khẩu.
* Các ứng dụng của bên thứ ba có được quyền truy cập quá rộng vào tài nguyên của chủ sở hữu, khiến chủ sở hữu không có khả năng hạn chế truy cập.
* Chủ sở hữu tài nguyên không thể thu hồi quyền truy cập đối với bên thứ ba, chỉ có cách thay đổi mật khẩu của bên thứ ba.
* Sự can thiệp của bất kỳ ứng dụng nào của bên thứ ba dẫn đến sự can thiệp đến mật khẩu của người dùng và tất cả dữ liệu được bảo vệ bởi mật khẩu.

OAuth giải quyết những vấn đề này bằng cách kết hợp với lớp ủy quyền và tách vai trò của client khỏi vai trò của chủ tài nguyên (resource owner). Trong OAuth, ứng dụng client yêu cầu quyền truy cập vào các tài nguyên được kiểm soát bởi chủ sở hữu tài nguyên (resource owner) và được lưu trữ bởi máy chủ tài nguyên (resource server) và cấp một bộ thông tin xác thực khác với bộ thông tin của resource owner.

Thay vì sử dụng thông tin đăng nhập của resource owner để truy cập tài nguyên, client nhận được 1 mã access token - một chuỗi biểu thị phạm vi, thời gian tồn tại và các thuộc tính truy cập khác. Token được cấp cho client bên thứ ba bởi một máy chủ ủy quyền (authorization server) với sự chấp thuận của chủ sở hữu tài nguyên. Khách hàng sử dụng mã token này để truy cập các tài nguyên được bảo vệ do resource server lưu trữ.

Tóm lại: OAuth là một phương thức chứng thực giúp các ứng dụng có thể chia sẻ tài nguyên với nhau mà không cần chia sẻ thông tin username và password. Từ Auth ở đây mang 2 nghĩa:

* Authentication: xác thực người dùng thông qua việc đăng nhập.
* Authorization: cấp quyền truy cập vào các Resource.

#### Ứng dụng

- Xác thực người dùng

- Cho phép người dùng ủy quyền để ứng dụng truy cập tài nguyên

- Cho phép các ứng dụng khác đăng nhập bằng tài khoản facebook, google, twitch, …

- Tiết kiệm chất xám, không cần phải nhớ các tài khoản ở các ứng dụng khác nhau.

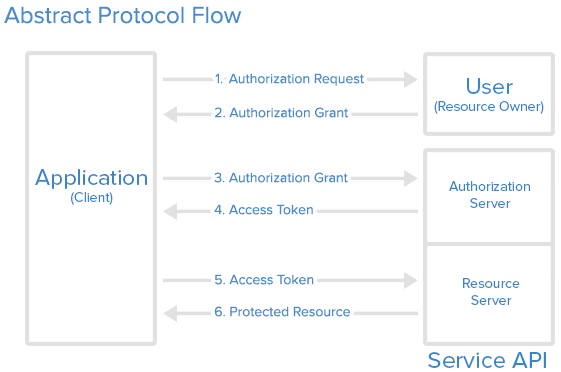
#### Lịch sử

* Năm 2006, Twitter đưa ra chuẩn OAuth đầu tiên có tên là OpenID, điểm yếu đó là yêu cầu người dùng phải cung cấp thông tin cá nhân (username + password).
* Năm 2010, phát hành phiên bản chính thức đầu tiên của Oauth 1.0 ([RFC 5849](https://tools.ietf.org/html/rfc5849)). Sau đó lỗi bảo mật nghiêm trọng được phát hiện với tên gọi [Session Fixation](http://hueniverse.com/2009/04/23/explaining-the-oauth-session-fixation-attack/) cho phép Hacker chiếm quyền truy cập vào tài nguyên của người dùng.
* Năm 2012, OAuth2 ra đời, tuy vẫn còn những lỗi bảo mật như dùng Chrome để [Hack Facebook](http://homakov.blogspot.co.uk/2013/02/hacking-facebook-with-oauth2-and-chrome.html) nhưng hiện vẫn đang được sử dụng khá rộng rãi.

#### Các tác nhân

* Resource Owner (User): Một thực thể có khả năng cấp quyền truy cập vào tài nguyên được bảo vệ. Khi Resource owner là một người, nó được gọi là người dùng cuối.
* Client (Application): Là những ứng dụng mong muốn truy cập vào dữ liệu người dùng. Trước khi được phép tương tác với dữ liệu thì ứng dụng này phải qua bước ủy quyền của User, và phải được kiểm tra xác nhận thông qua API. => Có thể hiểu là các ứng dụng sử dụng Facebook, Twitter, Google API.
* Resource Server (API): Nơi lưu trữ thông tin tài khoản của User, có khả năng chấp nhận và phản hồi các yêu cầu tài nguyên được bảo vệ bằng cách sử dụng mã thông báo truy cập
* Authorization Server (API): làm nhiệm vụ kiểm tra thông tin user, sau đó cấp quyền truy cập cho Application thông qua việc phát sinh "access token". Resource server và Authorization server chính là điểm khác biệt cơ bản giữa OAuth2 và OAuth1 khi tách biệt được hai thao tác: chứng thực (Authorization) và cung cấp thông tin người dùng (Resource) thành 2 server.

#### Cách hoạt động



1. Application yêu cầu ủy quyền để truy cập vào Resource Server thông qua User
2. Nếu User ủy quyền cho yêu cầu trên, Application sẽ nhận được giấy ủy quyền từ phía User (dưới dạng một token string nào đó chẳng hạn)
3. Application gửi thông tin định danh (ID) của mình kèm theo giấy ủy quyền của User tới Authorization Server
4. Nếu thông tin định danh được xác thực và giấy ủy quyền hợp lệ, Authorization Server sẽ trả về cho Application access token. Đến đây quá trình ủy quyền hoàn tất.
5. Để truy cập vào tài nguyên (resource) từ Resource Server và lấy thông tin, Application sẽ phải đưa ra access token để xác thực.
6. Nếu access token hợp lệ, Resource Server sẽ trả về dữ liệu của tài nguyên đã được yêu cầu cho Application.

Luồng hoạt động thực tế có thể sẽ khác nhau tùy thuộc vào việc ứng dụng sử dụng loại ủy quyền (authorization grant type) nào, đây chỉ là ý tưởng chung để thực hiện.

#### Đăng ký thông tin cho ứng dụng

Trước khi sử dụng Oauth cho ứng dụng, phải đăng ký ứng dụng với bên cung cấp dịch vụ, một số thông tin cơ bản cần đăng ký như sau:

* Tên ứng dụng (Application Name)
* Website của ứng dụng (Application Website)
* Redirect URI hoặc Callback URL: chính là địa chỉ sẽ quay về sau khi quá trình ủy quyền hoàn tất (cho phép hoặc từ chối từ phía User), chính vì thế mà địa chỉ quay về đó chính là nơi bạn sẽ phải thực hiện xử lý cho authorization codes hoặc access tokens.

Khi ứng dụng được đăng ký, bên dịch vụ sẽ phát hành "thông tin chứng thực client" (client credentials) bao gồm thông tin:

* Client Identifier: là một chuỗi ký tự được sử dụng bởi Service API để định danh ứng dụng, đồng thời cũng được dùng để xây dựng "authorization URL" hiển thị phía User.
* Client Secret: là một chuỗi ký tự được sử dụng để xác thực định danh (ID) của ứng dụng khi ứng dụng yêu cầu truy cập thông tin tài khoản của User. Chuỗi này được giữ bí mật giữa Application và API.

### Các loại ủy quyền của Oauth2

Loại ủy quyền phụ thuộc vào phương thức mà Application sử dụng để yêu cầu ủy quyền, Oauth2 định nghĩa ra 4 loại:

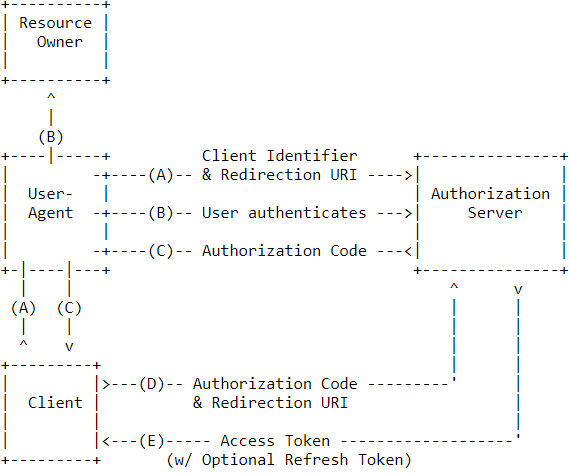
* Authorization Code: sử dụng với các server-side Application.
* Implicit: sử dụng với các Mobile App (ứng dụng chạy trên thiết bị của User) hoặc Web App (có thể hiểu là Browser).
* Resource Owner Password Credentials: sử dụng với các Trusted Application, kiểu như những ứng dụng của chính Service.
* Client Credentials: sử dụng với các ứng dụng truy cập thông qua API

#### Authorization Code:

Authorization Code được lấy bằng cách sử dụng máy chủ ủy quyền (authorization server) làm trung gian giữa client và resource owner. Thay vì yêu cầu ủy quyền trực tiếp từ resource owner, client hướng resource owner đến authorization server, sau đó điều hướng resource owner trở lại client với mã ủy quyền.

Trước khi điều hướng resource owner trở lại client với mã ủy quyền (authorization code), authorization server cần xác thực resource owner và nhận được ủy quyền. Vì resource owner chỉ xác thực với authorization, nên tài nguyên về thông tin đăng nhập không được chia sẻ với client.

Authorization code cung cấp một số lợi ích bảo mật quan trọng, chẳng hạn như khả năng xác thực client, cũng như truyền trực tiếp access token đến client mà không cần chuyển nó qua tác nhân người dùng của resource owner và có khả năng chia sẻ với những thứ khác, bao gồm cả resource owner.



(A) Client bắt đầu luồng bằng cách hướng tác nhân người dùng của resource owner đến điểm cuối là authorization server. Yêu cầu gồm client identifier, phạm vi được yêu cầu, trạng thái và URI điều hướng mà authorization server sẽ gửi lại tác nhân người dùng sau khi quyền truy cập được cấp (hoặc bị từ chối).

(B) authorization server xác thực resource owner (thông qua tác nhân người dùng) và xác định xem resource owner cấp hay từ chối yêu cầu truy cập của client.

(C) Giả sử resource owner cấp quyền truy cập, authorization server chuyển hướng tác nhân người dùng trở lại client bằng cách sử dụng URI chuyển hướng được cung cấp trước đó (trong yêu cầu hoặc trong quá trình đăng ký client). URI chuyển hướng bao gồm

Authorization code và bất kỳ trạng thái cục bộ nào được client cung cấp trước đó.

(D) client yêu cầu access token từ authorization server bằng cách bao gồm authorization code đã nhận được ở bước trước. Khi đưa ra yêu cầu, client xác thực với authorization server. Yêu cầu bao gồm URI chuyển hướng, authorization code để xác minh.

(E) authorization server xác thực client, xác thực authorization code và đảm bảo rằng URI chuyển hướng nhận được khớp với URI được sử dụng để chuyển hướng client trong bước (C). Nếu hợp lệ, resource server sẽ phản hồi lại bằng access token và refresh token

* Yêu cầu ủy quyền gồm: response type (code), client id, redirect uri, phạm vi, trạng thái
* Phản hồi yêu cầu ủy quyền gồm: code, trạng thái.
* Yêu cầu access token gồm: grant type (authority code), code (authorization code), redirect uri, client id
* Phản hồi yêu cầu access token gồm: access token, refresh token.

Ví dụ phản hồi access token thành công:

HTTP / 1.1 200 OK

Content-Type: application / json; charset = UTF-8

Cache-Control: no-store

Pragma: no-cache

{

"access\_token": "2YotnFZFEjr1zCsicMWpAA",

"token\_type": "example",

"expires\_in": 3600,

"refresh\_token": "tGzv3JOkF0XG5Qx2TlKWIA",

"example\_parameter": "example\_value"

}

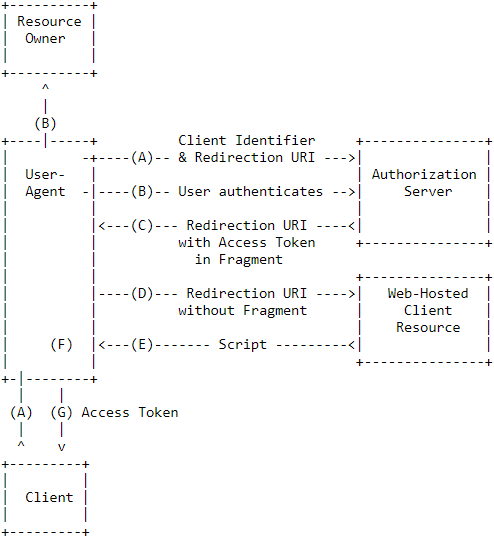
Lưu ý: Authorization code phải hết hạn ngay sau khi được cấp để giảm thiểu rủi ro rò rỉ

#### Implicit

Là authorization code được tối ưu hóa cho các ứng dụng client triển khai trong trình duyệt sử dụng ngôn ngữ kịch bản như JavaScript. Nó ngầm định, thay vì cấp cho client authorization code, client được cấp trực tiếp access token (do sự cho phép ủy quyền của resource owner). Implicit không có xác thực trung gian (chẳng hạn như authorization code) được cấp (và sau đó được sử dụng để lấy access token).

Khi phát hành access token trong luồng của implicit, authorization server không xác thực ứng dụng client. Trong một số trường hợp, định danh của client có thể xác minh thông qua URI được sử dụng để cung cấp access token cho client. Access token có thể được hiển thị cho resource owner hoặc các ứng dụng khác có quyền truy cập vào tác nhân người dùng của resource owner.

Implicit cải thiện khả năng phản hồi và hiệu quả của ứng dụng client (chẳng hạn như ứng dụng client được triển khai dưới dạng ứng dụng trong trình duyệt), vì nó làm giảm số lượng chuyến đi khứ hồi cần thiết để có được access token. Tuy nhiên, các tác động của implicit, mã thông báo truy cập được truyền trong URI, có thể bị. authorization server phải đảm bảo rằng access token không được được tạo, sửa đổi hoặc đoán để tạo ra các access token hợp lệ bởi các bên trái phép.



(A) Client bắt đầu quy trình bằng cách hướng tác nhân người dùng của resource owner đến authorization server. Yêu cầu bao gồm client id, phạm vi được yêu cầu, trạng thái cục bộ và URI chuyển hướng mà authorization server sẽ gửi lại tác nhân người dùng sau khi quyền truy cập được cấp (hoặc bị từ chối).

(B) Authorization server xác thực resource owner (thông qua tác nhân người dùng) và xác định xem resource owner cấp hay từ chối yêu cầu truy cập của client.

(C) Giả sử resource owner cấp quyền truy cập, authorization server chuyển hướng tác nhân người dùng trở lại client bằng cách sử dụng URI chuyển hướng được cung cấp trước đó. URI chuyển hướng bao gồm access token trong phân đoạn URI.

(D) Tác nhân người dùng được chuyển hướng bằng cách đưa ra yêu cầu đối với tài nguyên máy khách được lưu trữ trên web.

(E) Tài nguyên client được lưu trữ trên web trả về một trang web (thường là tài liệu HTML có tập lệnh nhúng) có thể truy cập URI chuyển hướng có thể trích xuất access token (và các tham số khác) có trong đoạn URI.

(F) Tác nhân người dùng thực thi tập lệnh được cung cấp bởi tài nguyên client được lưu trữ trên web cục bộ, mã này trích xuất access token.

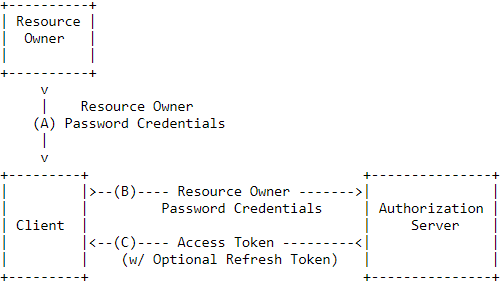
(G) Tác nhân người dùng chuyển access token cho client.

* Yêu cầu ủy quyền gồm: response type(token), client id, redirect uri, phạm vi, trạng thái.
* Phản hồi access token: access token, token type, expires in (thời gian tồn tại của access token), phạm vi và trạng thái

#### Resource Owner Password Credentials

Thông tin đăng nhập của resource owner (tức là tên người dùng và mật khẩu) có thể được sử dụng trực tiếp như cấp phép để lấy access token. Thông tin xác thực chỉ nên được sử dụng khi có mức độ tin cậy cao giữa resource owner và client (ví dụ: client là một phần của hệ điều hành hoặc một ứng dụng có đặc quyền cao) và khi các loại cấp phép khác không khả dụng (chẳng hạn như authorization code).

Mặc dù nó yêu cầu client truy cập trực tiếp vào thông tin đăng nhập của resource owner, nhưng thông tin xác thực của resource owner được sử dụng cho một yêu cầu duy nhất và được cấp access token. Nó loại bỏ nhu cầu client phải lưu trữ thông tin đăng nhập của resource owner để sử dụng trong tương lai, bằng cách trao đổi thông tin đăng nhập với access token tồn tại lâu dài hoặc refresh token.



(A) resource owner cung cấp cho client tên người dùng và mật khẩu.

(B) Client yêu cầu access token của authorization server bằng các thông tin xác thực nhận được từ resource owner. Khi thực hiện yêu cầu, client xác thực với authorization server

(C) authorization server xác thực client và xác thực thông tin xác thực của resource owner nếu hợp lệ, sẽ cấp access token

* Yêu cầu access token gồm: grant type (giá trị là password), tên người dùng, mật khẩu, phạm vi

Ví dụ: client thực hiện yêu cầu HTTP sau:

POST /token HTTP/1.1

Host: server.example.com

Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

Content-Type: application/x-www-form-urlencoded

grant\_type=password&username=johndoe&password=A3ddj3w

Phản hồi yêu cầu access token: Nếu yêu cầu access token hợp lệ và được ủy quyền, authorization server sẽ cấp access token và refresh token.

Ví dụ:

HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8

Cache-Control: no-store

Pragma: no-cache

{

"access\_token":"2YotnFZFEjr1zCsicMWpAA",

"token\_type":"example",

"expires\_in":3600,

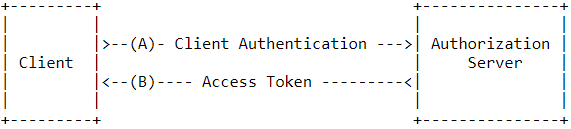
"refresh\_token":"tGzv3JOkF0XG5Qx2TlKWIA",

"example\_parameter":"example\_value"

}

#### Client Credentials

Client Credentials (hoặc các hình thức xác thực ứng dụng client khác) có thể được sử dụng để cấp phép khi phạm vi ủy quyền để sử dụng các tài nguyên bị giới hạn dưới sự kiểm soát của ứng dụng client hoặc các tài nguyên được bảo vệ đã được sắp xếp trước đó với authorization server. Thông tin đăng nhập của client được sử dụng để cấp phép khi client thực hiện trong chính nó (client cũng là chủ sở hữu tài nguyên) hoặc đang yêu cầu quyền truy cập vào các tài nguyên được bảo vệ dựa trên quyền đã được quy định trước đó với authorization server.



(A) Client xác thực với authorization server và yêu cầu access token.

(B) Authorization server xác thực client và nếu hợp lệ, sẽ cấp access token.

Client yêu cầu access token bằng cách thêm các tham số theo định dạng "application / x-www-form-urlencoded" với mã hóa ký tự là UTF-8 trong phần request HTTP, thuộc loại "client\_credentials".

* Yêu cầu access token:

Nếu yêu cầu access token hợp lệ và được ủy quyền, authorization server sẽ cấp access token.

Ví dụ:

POST / mã thông báo HTTP / 1.1

Máy chủ: server.example.com

Cấp phép: Cơ bản czZCaGRSa3F0MzpnWDFmQmF0M2JW

Nội dung -Type: application / x-www-form-urlencoded

Grant\_type = client\_credentials

Authorization server phải xác thực client.

* Phản hồi yêu cầu access token: Nếu yêu access token hợp lệ và được ủy quyền, authorization server sẽ cấp access token

Ví dụ về phản hồi thành công:

HTTP / 1.1 200 OK

Content-Type: application / json; charset = UTF-8

Cache-Control: no-store

Pragma: no-cache

{

"access\_token": "2YotnFZFEjr1zCsicMWpAA",

"token\_type": "ví dụ ",

" expires\_in ": 3600,

"example\_parameter": "example\_value"

}

### Access token

Access token là thông tin xác thực được sử dụng để truy cập các tài nguyên được bảo vệ. Access token là một chuỗi đại diện cho việc ủy quyền đã được cấp cho client. Chuỗi thường không rõ ràng đối với client. Access token đại diện cho phạm vi và thời lượng truy cập, được cấp bởi resource owner và được thực thi bởi resource server và authorization server. Access token hiển thị một số dữ liệu được sử dụng để truy xuất thông tin ủy quyền hoặc có thể tự chứa thông tin ủy quyền theo cách có thể xác minh được (tức là chuỗi mã thông báo bao gồm một số dữ liệu và chữ ký).

Access token cung cấp một lớp trừu tượng, thay thế các cấu trúc ủy quyền khác nhau (ví dụ: tên người dùng và mật khẩu) bằng một mã thông báo duy nhất được resource server hiểu. Nó cho phép cung cấp access token hạn chế hơn so với cấp phép được sử dụng để lấy chúng, cũng như loại bỏ nhu cầu của resource server để hiểu nhiều phương pháp xác thực.

Access token có thể có các định dạng, cấu trúc và phương pháp sử dụng khác nhau (ví dụ: mật mã) dựa trên các yêu cầu bảo mật của resource server.

### Refresh token

Là thông tin xác thực sử dụng để lấy access token. Refresh token được authorization server cấp cho client và được sử dụng để lấy access token mới khi access token hiện tại không hợp lệ hoặc hết hạn hoặc để nhận access token bổ sung có phạm vi giống hoặc hẹp hơn (access token có thể có tuổi thọ ngắn hơn và ít quyền hơn so với quyền của resource owner). Việc cấp refresh token tùy chọn theo quyết định của authorization server. Nếu máy chủ ủy quyền phát hành refresh token, nó sẽ được gói lại cùng với access token. Refresh token là một chuỗi đại diện cho việc ủy quyền được cấp bởi resource owner. Refresh token chỉ được sử dụng với authorization server và không bao giờ được gửi đến resource owner.

### Đăng ký thông tin

Trước khi sử dụng **Oauth** cho ứng dụng, phải đăng ký ứng dụng với bên cung cấp dịch vụ, một số thông tin cơ bản cần đăng ký như sau:

* Tên ứng dụng (**Application Name**)
* Website của ứng dụng (**Application Website**)

**Redirect URI** hoặc **Callback URL**: chính là địa chỉ sẽ quay về sau khi quá trình ủy quyền hoàn tất (cho phép hoặc từ chối từ phía User), địa chỉ quay về là nơi thực hiện xử lý cho authorization codes hoặc acc

### Tổng kết phần V:

Các cơ chế bảo mật giúp cho việc xác thực, ủy quyền, và bảo mật cho API một cách an toàn cho mỗi request, các cơ chế bảo mật này tuy sẽ làm tiêu tốn một phần tài nguyên để xử lý nhưng nó sẽ đảm bảo về độ bảo mật hơn cũng như chiếm ít tải, chiếm ít tài nguyên hơn việc duy trì session.

## Phần VI: Jhipter

### Jhipster là gì:

### Lịch sử phát triển:

### Ưu điểm:

### Nhược điểm:

### Các thành phần trong Jhipster:

#### Cơ chế bảo mật:

#### Code java:

#### Angular:

#### Jdl:

### Tổng kết phần VI

# Chương II: Phân tích hệ thống đánh giá điểm rèn luyện

## Phần 1: Tổng quan

### Khảo sát hiện trạng

Hiện trên thực tế nhiều trường đại học vẫn còn sử dụng cách đánh giá rèn luyện cho sinh viên thủ công, chưa có hệ thống cụ thể. Nhưng với tình hình dịch bệnh như hiện nay, việc học online đang ngày càng được áp dụng nhiều hơn, tuy nhiên, sinh viên vẫn phải nộp lại các phiếu rèn luyện trực tiếp cho nhà trường thông qua ban cán sự. Điều này không thuận tiện và gây nhiều khó khăn trong công tác kiểm soát lẫn quản lý. Một vài trường đã triển khai theo cách đánh giá online nhưng vẫn tự đánh giá trên excel, không có hệ thống quản lý hay thống kê cụ thể.

### Bài toán thực tế:

#### a) Quy mô:

Hiện nay trường Đại học Quy Nhơn có hơn 12 khoa, đào tạo 38 ngành với quy mô xấp xỉ 16.000 sinh viên.

#### b) Công tác quản lý:

- Quy trình đánh giá:

* Từng khoa gửi bản mềm về cho từng lớp
* Lớp in và photo chia cho sinh viên trong lớp để sinh viên tự đánh Giá
* Ban cán sự lớp thu lại và tiến hành đánh giá lần hai
* Ban cán sự vào điểm ở file tổng hợp
* Cố vấn học tập duyệt kết quả đánh giá lần 2
* Ban cán sự lớp nộp lại các phiếu đánh giá và in + nộp file tổng hợp
* Khoa nhận, đánh giá lần 3 và vào điểm ở hệ thống.
* Hồ sơ dữ liệu:

+ Hồ sơ sinh viên

+ Phiếu đánh giá

+ Phiếu tổng hợp đánh giá

### Mục tiêu:

#### a) Phát biểu bài toán

Bộ Giáo dục và Đào tạo đã ban hành quyết định về việc ban hành quy chế đánh giá kết quả rèn luyện của học sinh – sinh viên các cơ sở giáo dục đại học và trường trung cấp chuyên nghiệp hệ chính quy. Kết quả đánh giá rèn luyện từng học kỳ, từng năm học của người học được lưu trong hồ sơ quản lý người học của trường, được sử dụng trong việc xét duyệt học bổng, xét khen thưởng – kỷ luật, xét thôi học, ngừng học, xét lưu trú ký túc xá, xét giải quyết việc làm thêm, xét miễn giảm chi phí, dịch vụ và sinh hoạt trong ký túc xá, ... Kết quả đánh giá rèn luyện toàn khóa của người học được lưu trong hồ sơ quản lý người

học của cơ sở giáo dục đại học, làm căn cứ để xét tốt nghiệp, làm luận văn hoặc khóa luận tốt nghiệp. Kết quả đánh giá rèn luyện toàn khóa được ghi chung vào bảng điểm kết quả học tập và lưu trong hồ sơ của người học khi tốt nghiệp ra trường. Người học có kết quả rèn luyện xuất sắc được cơ sở giáo dục đại học xem xét biểu dương, khen thưởng. Người học bị xếp loại rèn luyện yếu, kém trong hai học kỳ liên tiếp thì phải tạm ngừng học ít nhất một học kỳ ở học kỳ tiếp theo và nếu bị xếp loại rèn luyện yếu, kém hai học kỳ liên tiếp lần thứ hai thì sẽ bị buộc thôi học, và điểm rèn luyện còn là cơ sở để xét duyệt việc trao học bổng. Vì vậy, việc tổ chức đánh giá và lưu trữ điểm rèn luyện của sinh viên cần được thực hiện liên tục mỗi học kỳ dẫn đến kho dữ liệu ngày càng khổng lồ, việc truy xuất gặp không ít khó khăn, đòi hỏi mỗi trường phải có biện pháp cải thiện. Biện pháp cải thiện hiệu quả nhất chính là tin học hóa quy trình đánh giá điểm rèn luyện cho sinh viên ở trường.

Cũng trên tinh thần ấy, Trường Đại học Quy Nhơn hằng năm vẫn tổ chức xét điểm rèn luyện cho sinh viên vào cuối mỗi học kỳ. Tuy nhiên, toàn bộ quy trình hiện tại đều được thực hiện thủ công, tốn rất nhiều thời gian và chi phí. Bên cạnh đó, với số lượng sinh viên tại trường ngày càng đông làm cho công tác tổ chức đánh giá, lưu trữ, sử dụng điểm rèn luyện gặp nhiều bất cập là khoa đi đầu về công nghệ thông tin tại trường, Công nghệ thông tin mong muốn số hóa quá trình đánh giá và quản lý điểm rèn luyện để từ đó có thể mở rộng sử dụng ra toàn trường. Với mong muốn đạt được mục tiêu chung của khoa, em đề xuất đề tài “Tìm hiểu kiến trúc phần mềm Microservice và ứng dụng xây dựng hệ thống đánh giá rèn luyện sinh viên”. Để tiếp cận công nghệ mới và đem nó ứng dụng trong một bài toán thực tế trong nhà trường, hệ thống sử dụng kiến trúc microservice để phát triển.

#### b) Mục tiêu:

- Mục tiêu chung: Đáp ứng việc đánh giá điểm rèn luyện cho sinh viên và việc quản lý, khai thác điểm rèn luyện của sinh viên cho cán bộ quản lý.

- Mục tiêu cụ thể 1: Sinh viên tự đánh giá điểm rèn luyện.

- Mục tiêu cụ thể 2: Giáo viên chủ nhiệm và ban cán sự dựa vào kết quả đánh giá của sinh viên để xét duyệt điểm rèn luyện cho sinh viên.

- Mục tiêu cụ thể 3: Cán bộ phụ trách ở khoa sẽ xét duyệt điểm rèn luyện cho sinh viên dựa vào bảng tự đánh giá của sinh viên, bảng đánh giá của giáo viên chủ nhiệm và ban cán sự. Sau đó, cán bộ phụ trách sẽ tiến hành thực hiện các báo cáo tổng hợp để thống kê và phân loại kết quả đánh giá theo từng lớp, ...

## Phần 2: Phân tích, thiết kế hệ thống:

### Yêu cầu chức năng

* 1. Sơ đồ usecase:
  2. abc

### Yêu cầu phi chức năng

### Các mô hình phân tích

### Lược đồ cơ sở dữ liệu:

#### Lược đồ cơ sở dữ liệu cho microservice UAA:

#### Lược đồ cơ sở dữ liệu cho Microservice Evaluation Criteria:

### Đặc tả giao diện màn hình

# Chương III: Cài đặt và triển khai

Tài liệu tham khảo: