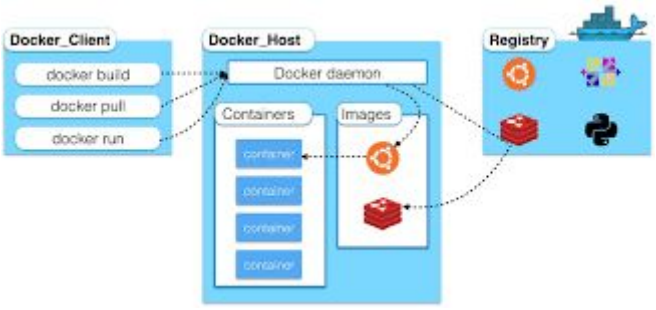
Docker



Khái niệm

- Là công cụ giúp chạy ứng dụng dễ dàng bằng việc tạo ra các container.Container là môi trường mà lập trình viên đưa vào các thành phần cần thiết để ứng dụng chạy được,được đóng gói trong container.

Ưu điểm

- Nhanh
- Linh hoạt
- Khả năng mở rộng
- Luân chuyển nhanh chóng
- Sử dụng tài nguyên tốt hơn máy ảo.

Nhược điểm

- Việc ảo hóa hoàn toàn không được cung cấp bởi docker vì nó phụ thuộc vào nhân linux. Được cung cấp bởi máy chủ cục bộ
- Không chạy trên các máy 32 bit,chỉ hỗ trợ máy 64-bit
- Cần phải có kiến thức chuyên sâu mới có thể giải quyết các lỗi phát sinh khi sử dụng docker.

Dockerfile

Khái niệm:

- Là một tập tin dạng text chứa một chuỗi các câu lệnh, chỉ thị để Docker đọc và chạy theo chỉ thị đó để tạo nên 1 image theo yêu cầu.
- Nó dùng 1 image cơ bản để xây dựng lớp image ban đầu.Một số image cơ bản: python, ubuntu,...Sau đó nếu có các lớp bổ sung thì được xếp chồng lên lớp cơ bản đó

Các chỉ thị trong dockerfile:

FROM: chỉ ra image gốc(cơ sở)
LABEL: cung cấp metadata cho image
ENV: thiết lập biến môi trường
RUN: dùng để cài các package vào trong container
COPY hoặc **ADD**: sao chép các file và thư mục vào container

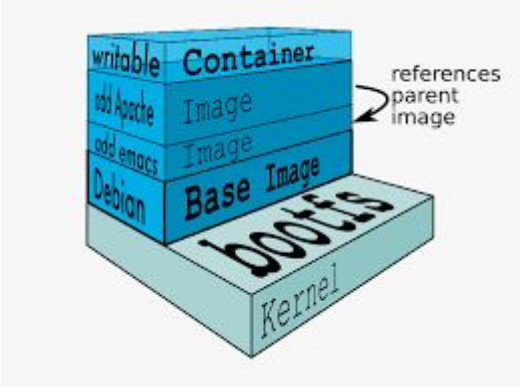
CMD: cung cấp lệnh để cho container thực thi.
WORKDIR: thiết lập thư mục làm việc cho các chỉ thị khác như: RUN, CMD, COPY, ADD,..
ARG: Định nghĩa giá trị biến được dùng trong lúc build image
EXPOSE: thiết lập cổng lắng nghe của container.
VOLUME: gắn thư mục để truy cập và lưu trữ dữ liệu.

Docker image

Khái niệm:

- Là một template dùng để tạo ra các container.
- Được xây dựng từ một loạt các layers.Mỗi layer là một kết quả đại diện cho một câu lệnh trong Dockerfile.
- Lưu trữ dưới dạng read-only template.

Docker container



Khái niệm:

- Được khởi chạy từ docker image
- Là một gói phần mềm thực thi lightweight, độc lập và chứa mọi thứ: code, runtime, các thư viện,..Và đặc biệt luôn chạy ổn định bất kể môi trường nào.
- Container chạy trực tiếp trên môi trường máy chủ như 1 tiến trình

Docker compose

Khái niệm:

- Là công cụ để tạo, xác định và chạy nhiều container có sự liên quan với nhau trong cùng 1 thời điểm; được khai báo trong một file với định dạng YAML. Khởi động tất cả các dịch vụ chỉ với 1 câu lệnh duy nhất.

Các bước thực hiện:

- Định nghĩa các ứng dụng thông qua Dockerfile
- Định nghĩa các ứng dụng chạy tách biệt và khởi động cùng nhau trong docker-compose.yml

- Thực thi câu lệnh docker-compose up để hoàn tất.

Lợi ích:

- Tạo nhiều môi trường riêng biệt trên cùng 1 host

Các chỉ thị trong docker compose:

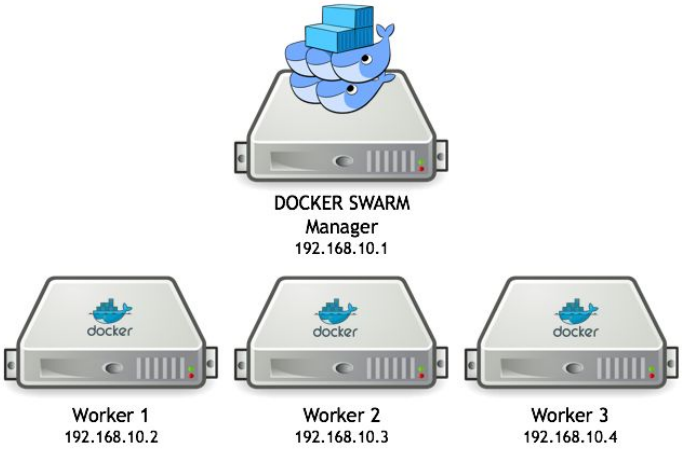
version: chỉ ra phiên bản docker-compose đã sử dụng.
services: thiết lập các services(containers) muốn cài đặt và chạy.
image: chỉ ra image được sử dụng trong lúc tạo ra container.
build: dùng để tạo container.
ports: thiết lập ports chạy tại máy host và trong container.
restart: tự động khởi chạy khi container bị tắt.
environment: thiết lập biến môi trường (thường sử dụng trong lúc config các thông số của db).
depends_on: chỉ ra sự phụ thuộc. Tức là services nào phải được cài đặt và chạy trước thì service được config tại đó mới được chạy.
volumes: dùng để mount hai thư mục trên host và container với nhau.

Docker machine

Ý nghĩa:

- Khi cài đặt Docker, nó chạy trên máy của ta (gọi nó là một docker daemon), máy đó là Docker Host, trong trường hợp có nhu cầu chạy nhiều docker daemon nhiều Docker Host như tạo hệ thống Docker Swarm, lúc đó là bạn cần tạo ra các máy ảo, trong máy ảo đó có cài đặt Docker thì lúc đó ta cần docker-machine để tạo ra các máy ảo Docker.

Docker swarm



Khái niệm:

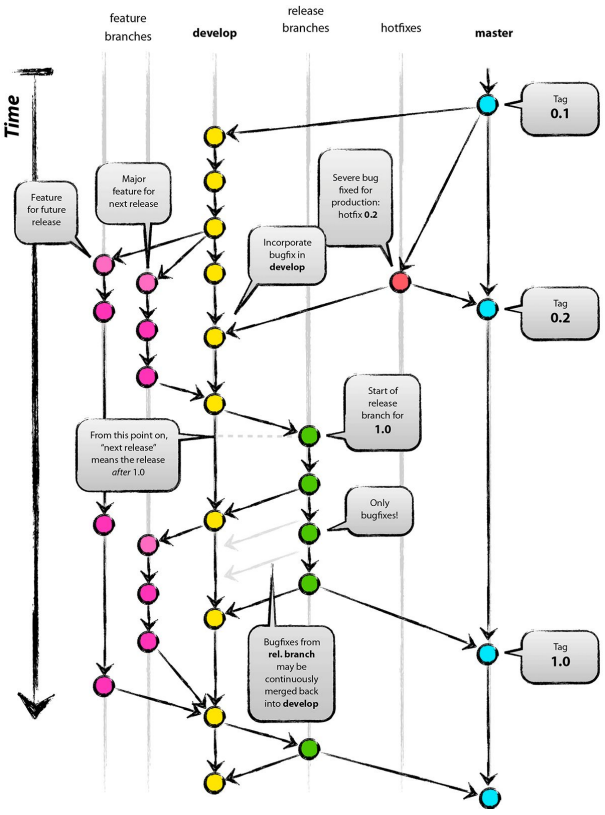
- Docker swarm là một công cụ giúp tạo ra một cụm Docker. Nó giúp gom nhiều Docker Engine lại với nhau và ta có thể "nhìn" nó như duy nhất một virtual Docker Engine **duy nhất**.

Lợi ích:

- Kết nối các máy vật lý, máy ảo với nhau thành 1 cụm để chạy dịch vụ trên cụm đó.

Git

Git flow



❖ Các branch trong git flow:

- Master branch: là branch dùng cho sản phẩm chính thức. Đây là branch ổn định nhất và nó chứa lịch sử các lần release của dự án.
- Develop branch: là nhánh dùng cho sản phẩm trong quá trình phát triển
- Feature: mỗi tính năng mới cho sản phẩm sẽ được tạo và phát triển trên một branch mới với tên quy ước feature/tên_branch. Các feature này sẽ tạo ra từ develop branch và khi được hoàn thiện sẽ được gộp trở lại với develop branch
- Release: khi develop branch đã có đủ số tính năng cần thiết để có thể release, ta có thể tạo branch mới với tên quy ước release/tên_version. Branch này sau khi được tạo và thực hiện xong sẽ tiến hành debug, test và fix các lỗi nhỏ (không code thêm tính năng) sau đã trơn tru, merge nó với đồng thời cả master branch và develop branch
- Hotfix branch: khi sản phẩm trên master branch của chúng ta gặp phải trục trặc và cần có bản vá ngay lập tức thì ta sẽ tạo ra hotfix branch. Branch này tương tự như release branch nhưng nó được tạo ra từ master branch thay vì từ develop branch như release (*Chú ý hotfix branch cũng cần được gộp lại với master branch với develop branch)

❖ Các lệnh trong git flow

- Khởi tạo :Sau khi di chuyển tới thư mục dự án,chạy lệnh khởi tạo git flow:

git flow init
(nên sử dụng các giá trị mặc định khi trở lời câu hỏi)

- Bắt đầu một feature:
  git flow feature start <tên-feature>
- Công bố tính năng: công bố source code lên remote để người khác cập nhật được.
  git flow feature publish <tên-feature>
- Kết thúc một feature:Sau khi phát triển xong một feature.
  git flow feature finish <tên-feature>
- Lấy về feature:pull mã nguồn của feature được cập nhật bởi những người khác.
  git flow feature pull REMOTE_NAME <tên-feature>
- Để tạo một bản release:
  git flow release start <release-version>
- Để công bố phần code release:
  git flow release publish <release-version>
- Kết thúc release:
  git flow release finish <release-version>
- Sửa đổi nóng,bắt đầu hotflix:
  git flow hotflix start <tên-hotfix>
- Kết thúc hotflix:
  git flow hotfix finish <tên-hotfix>

Cấu trúc commit message

<type>[optional scope]: <description>
[optional body]
[optional footer]
trong đó :

- Các thành phần type, description là bắt buộc có trong commit message, optional là tùy chọn, có hoặc không cũng được.
- type: từ khóa để phân loại commit là feature, fix bug, refactor.. Chú ý ngay sau type có dấu :
- scope: được dùng để phân loại commit, nhưng trả lời câu hỏi: commit này refactor|fix cái gì? được đặt trong cặp ngoặc đơn ngay sau type.
  VD: feat(authentication):, fix(parser):.
- description: là mô tả ngắn về những gì sẽ bị sửa đổi trong commit đấy.
- body: là mô tả dài và chi tiết hơn, cần thiết khi description chưa thể nói rõ hết được.
- footer: một số thông tin mở rộng như số ID của pull request, issue.. được quy định theo conventional.

ví dụ : *fix(player): uiza player can not initialize*

Microservice

Ưu điểm:

-Code base sáng sủa do hệ thống được tách nhỏ thành các service nên việc đọc hiểu dễ dàng hơn.

-Việc phát triển diễn ra đồng thời: do mỗi service là tách biệt nên mỗi người sẽ đảm nhiệm một service nào đó.
-Hệ thống có thể sử dụng nhiều công nghệ: do cho từng service là tách biệt nên có thể sử dụng công nghệ riêng.Và việc nâng cấp hệ thống trở nên dễ dàng hơn.
-Khả năng bảo trì tốt hơn: vì mỗi service có cấu trúc nhỏ gọn nên việc đọc hiểu tốt hơn.
-Việc triển khai tốt hơn: vì mỗi service được triển khai độc lập lẫn nhau
-Việc testing dễ dàng hơn: do mỗi service nhỏ do đó việc test trở nên nhanh chóng, thuận tiện
-Lỗi hệ thống tách biệt hơn: do mỗi service là độc lập nhau nên nếu một service mà bị lỗi thì sẽ không ảnh hưởng đến các service khác.

Nhược điểm

-Phải gặp phải đối mặt thêm sự phức tạp của một hệ thống phân tán:

- Cần implement việc communication giữa các inter-services
- Xử lý lỗi riêng phần là rất phức tạp vì một luồng sẽ phải đi qua nhiều services.
- Việc thực hiện các requests trải rộng trên nhiều services đòi hỏi sự cẩn thận giữa những người phát triển.
- Khó khăn trong việc đảm bảo toàn vẹn CSDL nếu triển khai theo kiến trúc cơ sở dữ liệu phân vùng.

-Triển khai và quản lý phức tạp do hệ thống bao gồm nhiều services khác nhau.
-Tăng chi phí sử dụng tài nguyên bộ nhớ.

Khi nào cần sử dụng đến kiến trúc microservice

-Khi ứng dụng của chúng ta quá phức tạp, cồng kềnh nếu như xây dựng bằng mô hình đơn nhất.
-Khi hệ thống vẫn còn mở rộng thêm theo chiều ngang trong tương lai.

Quản lý dữ liệu trong microservice

Yêu cầu:

-Các service cần đảm bảo có mối quan hệ lỏng lẻo để việc phát triển, triển khai, scale độc lập lẫn nhau
-Một số business transaction cần phải thực thi nhất quán và bất biến trên nhiều service.
-Một số queries cần join data thuộc sở hữu của các services khác nhau.
-Database đôi khi phải được nhân rộng và phân chia để mở rộng.
-Các service khác nhau có nhu cầu lưu trữ dữ liệu khác nhau

Giải pháp

Shared database

-Đây là dạng một Database được chia sẻ cho nhiều services. Các services được tự do truy cập các bảng của nhau để đảm bảo ACID transaction.

Lợi ích:

- Một single database là đơn giản hơn để hoạt động.

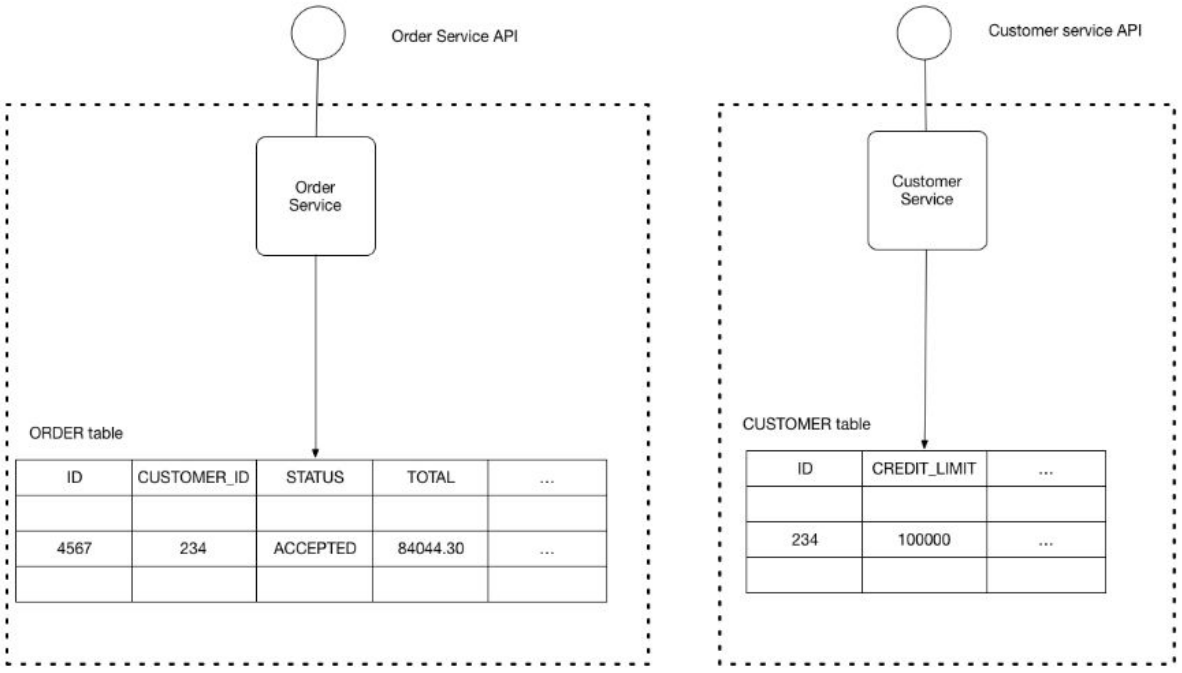
- Các developers dễ dàng sử dụng ACID transaction để thực thi tính nhất quán dữ liệu.

Hạn chế:

- Development time coupling.Ví dụ: Sự thay đổi schema của DB sẽ gây ảnh hưởng đến nhiều services do chúng truy cập trực tiếp vào các tables, và trong khi các lập trình viên cho mỗi service là khác nhau thì rõ ràng coupling này làm chậm quá trình development.
- Runtime coupling: tất cả các services truy cập vào cùng một cơ sở dữ liệu, chúng có khả năng can thiệp lẫn nhau. Ví dụ: nếu một CustomerService transaction chạy mất nhiều thời gian và lock ORDER table thì OrderService sẽ bị chặn.
- Thực tế đối với các hệ thống lớn, một DB đơn lẻ không thể đáp ứng đủ yêu cầu về lưu trữ và truy cập dữ liệu của tất cả các services.

Database per service

-Data của mỗi service chỉ được sử dụng bởi service đó, chỉ có thể lấy data thông qua API của nó. Và giao dịch của service đó chỉ liên quan đến database của nó.



-Có một số cách để giữ cho database của service chỉ được truy cập bởi service đó.Nếu sử dụng cơ sở dữ liệu quan hệ thì có những lựa chọn sau:

- Private-tables-per-service: mỗi service sở hữu một tập tables thì các table chỉ được truy cập bởi service đó.
- Schema-per-service: mỗi service có database schema riêng
- Database server-per-service: mỗi service có một database server riêng.

-Private-tables-per-service và schema-per-service thì có chi phí thấp nhất.và việc sử dụng Schema-per-service là tốt hơn vì nó làm cho quyền sở hữu trở nên rõ ràng. Trong khi đó, một số services có lượng truy cập cao sẽ cần có database server của riêng nó (Database-server-per-service).

Lợi ích:

- Giúp đảm bảo các services có mối quan hệ lỏng lẻo.Việc thay đổi database của service này không làm ảnh hưởng tới các service khác.
- Mỗi service có thể sử dụng database phù hợp nhất.

Nhược điểm:

- Việc thực hiện các business transaction trên nhiều services không đơn giản.
- Thực hiện các truy vấn cũng như join data trên nhiều database là khá thách thức.
- sự phức tạp trong việc quản lý nhiều cơ sở dữ liệu SQL và NoSQL.

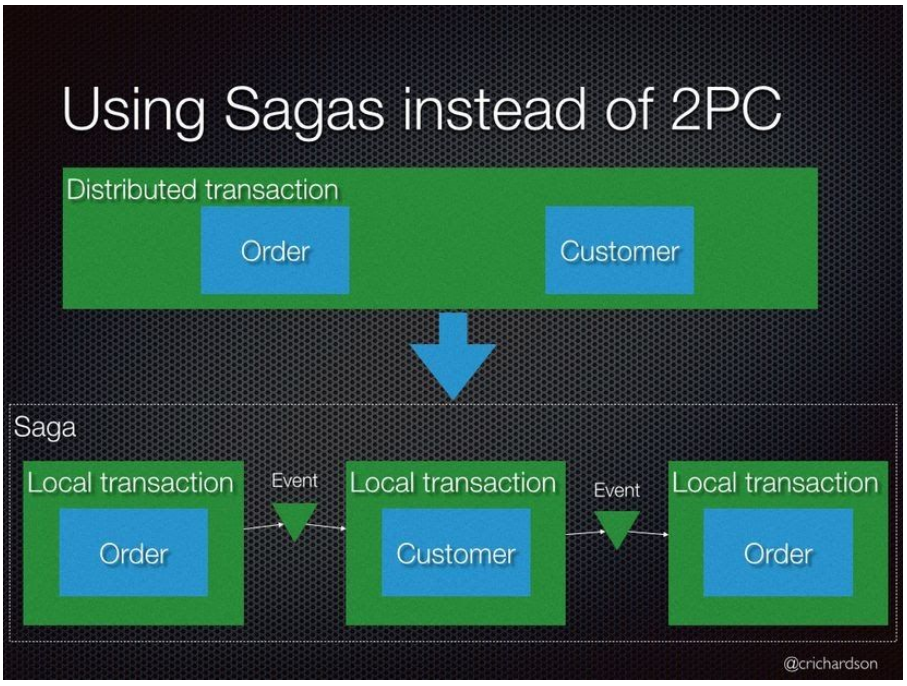
Có một số giải pháp để thực hiện các giao dịch và truy vấn trên nhiều service:

- Thực hiện các giao dịch trên nhiều service sử dụng **Saga pattern**.
- Thực hiện các truy vấn trên nhiều service:
  - ➢ **API Composition**: ứng dụng thực hiện join chứ không phải là database. Ví dụ, một service (hoặc API gateway) có thể truy xuất một khách hàng và đơn đặt hàng của họ bằng cách truy xuất khách hàng đầu tiên từ customer service và sau đó truy vấn order service để trả về các đơn đặt hàng gần nhất cho khách hàng.
  - ➢ **Command Query Responsibility Segregation (CQRS)**-duy trì một hoặc nhiều khung nhìn(views) cụ thể chứa data từ nhiều services. Các khung nhìn được giữ bởi các services subscribe các event mà mỗi service publish khi cập nhật dữ liệu của nó.Ví dụ : cửa hàng trực tuyến có thể thực hiện truy vấn tìm kiếm khách hàng ở một khu vực cụ thể và các đơn đặt hàng gần đây của họ bằng cách duy trì khung nhìn (view) để joins các customers và orders. Khung nhìn được cập nhật bởi một service subscribers các events của khách hàng và đặt hàng

Thực hiện các giao dịch trên nhiều service

Saga pattern

-Đối với các hệ thống lựa chọn mô hình Database per Service, mỗi service sẽ có một Database riêng. Tuy nhiên một số transaction cần trải rộng trên nhiều services. Do đó cần một cơ chế để đảm bảo tính thống nhất của dữ liệu trên các services.
-Giải pháp được đưa ra như sau: Ta sẽ coi mỗi một transaction trải rộng trên nhiều services là một Saga. Và mỗi một Saga là một chuỗi các transaction cục bộ trên từng service khác nhau. Nếu một transaction cục bộ thất bại thì Saga sẽ thực hiện một loạt các transactions để rollback lại các thay đổi đã được thực hiện trước đó.



-Có 2 cách để triển khai Saga.

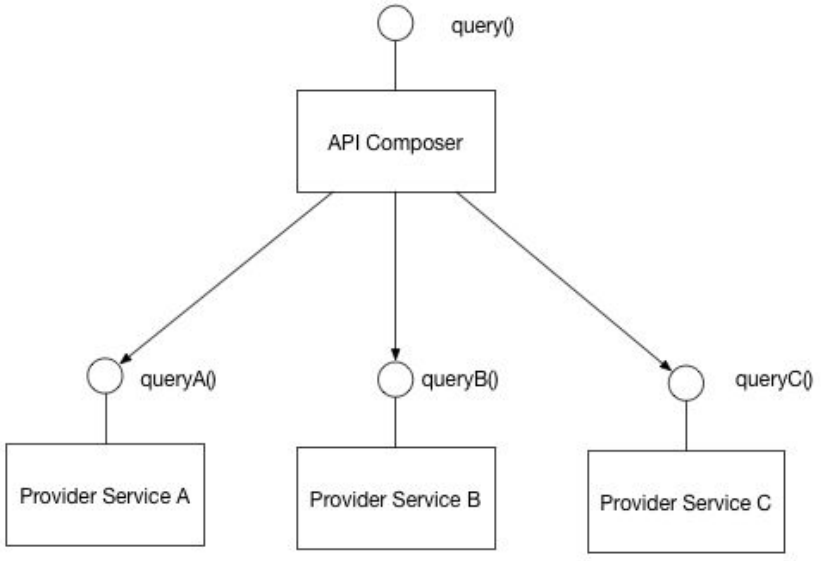
Events/Choreography-based saga

Command/Orchestration-based saga

Thực hiện query trên nhiều services

API Composition

-Thực hiện truy vấn bằng cách định nghĩa một API Composer, nó gọi các services sở hữu data và thực hiện join trong bộ nhớ để đưa ra kết quả.



Ví dụ : API composition thường thấy trong một API gateway.

Ưu điểm :

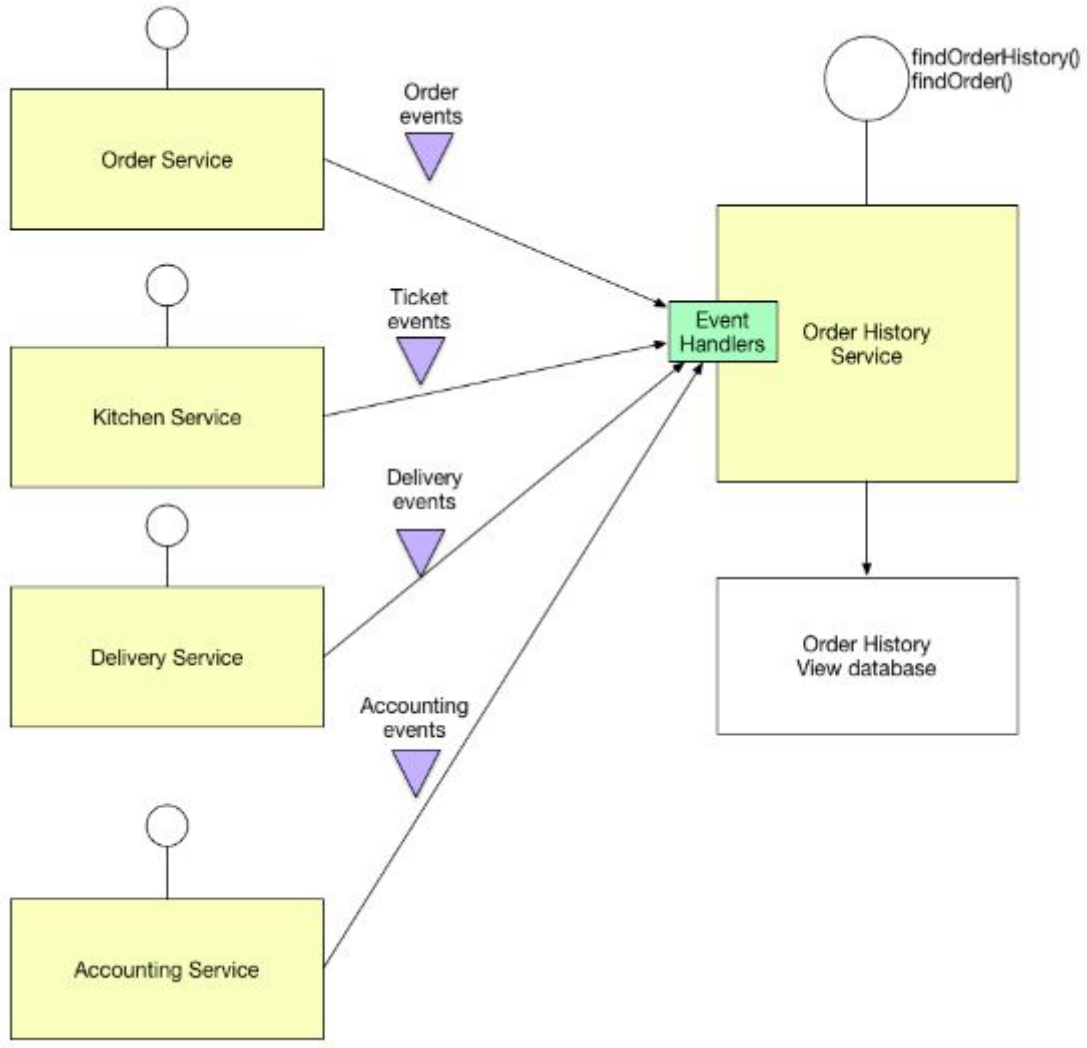
- Đây là một cách làm đơn giản để query data trong một kiến trúc microservice

Nhược điểm :

- Một số truy vấn sẽ dẫn đến việc join trong bộ nhớ trở nên không hiệu quả khi tập dữ liệu lớn.

Command Query Responsibility Segregation (CQRS) pattern

-Việc triển khai các truy vấn join data từ nhiều service không còn đơn giản nữa.Ngoài ra, nếu sử dụng Event sourcing pattern thì dữ liệu không còn dễ dàng truy vấn nữa.
Cách hoạt động :
-Định nghĩa một khung nhìn database, là bản sao chỉ đọc được thiết kế để hỗ trợ truy vấn đó. Ứng dụng giữ bản sao tối đa dữ liệu bằng cách subscribing các events được publish bởi service sở hữu dữ liệu.



Ưu điểm :

- Hỗ trợ nhiều khung nhìn không chuẩn hóa có thể mở rộng và hiệu quả.
- Cải thiện phân tách mối quan tâm = các mô hình truy vấn và lệnh đơn giản hơn.
- Cần thiết trong một kiến trúc có nguồn gốc sự kiện.

Nhược điểm :

- Sự phức tạp tăng lên
- Tạo bản sao code tiềm ẩn.

Các patterns sau là các cách để atomically update trạng thái và publish các messages/events:

- Event sourcing (https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing)
- Transactional Outbox (https://medium.com/engineering-varo/event-driven-architecture-and-the-outbox-pattern-569e6fba7216)

Một choreograph dựa trên saga có thể publish các events bằng việc sử dụng Aggregates và Domain Events

Authentication và authorization trong microservice

(tìm hiểu sau)

Security trong microservice

(tìm hiểu sau)

External API trong microservice

(tìm hiểu sau)

Service discovery trong microservice

(tìm hiểu sau)

Các kiểu giao tiếp trong microservice

Remote Procedure Invocation

(tìm hiểu sau)

Message

(tìm hiểu sau)

Domain-specific protocol

(tìm hiểu sau)