# Giới thiệu về docker

* + Được tạo ra như là Internal tool trong PaaS company (công ty dotCloud).
  + 3/2013=> open source.
  + Kubernetes là 1 tool được phát triển bởi Google cho việc deploy các container dọc các cluster.

# Ý tưởng đằng sau Docker

* + Đóng gói ứng dụng cùng tất cả các phụ thuộc của nó vào 1 thứ duy nhất, đơn vị tiêu chuẩn cho việc deployment.Các phụ thuộc đó có thể là: binaries,lib,JAR files, configuration files,....Docker sẽ bọc tất cả chúng vào 1 tập tin hoàn chỉnh chứa tất cả thứ mà ứng dụng của bạn cần để chạy.
  + Đóng gói tất cả đảm bảo nó sẽ có tính portable,nó sẽ chạy giống hệt nhau, không quan trọng môi trường nó chạy.
  + Docker là 1 tool giúp giải quyết các vấn đề như install, distributing, management phần mềm. Nó cho phép các dev và DevOps build, ship, và chạy code ở bất kì máy nào
  + So sánh giữa virtualization và containerization
  + Công nghệ Containerization: Đó chính là việc công nghệ này cho phép một máy chủ có nhiều các máy ảo cùng các hệ điều hành ảo khác. Lúc này công nghệ Containerization lưu trữ các hệ điều hành này, các Containerization chạy bằng các container nhỏ. Các container nhỏ này chứa các phần mềm và có sự liên hệ giữa các containers này bằng các kênh chuyên biệt.
  + Virtualization:
    - Ảo hóa mức phần cứng.Về bản chất nó là 1 hoàn chỉnh, máy vật lý ảo hóa với BIOS và hệ điều hành được cài đặt cùng.
    - Nặng nề,rườm rà khi cài đặt,và nhiều khi rất là khó khăn khi cài đặt.
    - Tiêu tốn nhiều tài nguyên.
  + Containerization:
    - Các ứng dụng chạy trên Docker trong 1 môi trường cô lập gọi là container.Contiainer không phải là 1 máy ảo theo nghĩa phổ biến. Nó có các đặc điểm như một máy ảo.Nhưng nó k ảo tất cả.
    - Sự khác biệt là trong khi image máy ảo truyền thống chạy trên 1 hệ điều hành khách độc lập, Docker container chạy trong cùng kernel trên máy host.Container là tự cung cấp và bị cô lập không chỉ OS, mà còn từ các container khác.
    - Nó sở hữu hệ thống file và các biến môi trường riêng biệt.Một cách tự nhiên, các container có thể giao tiếp với nhau và có thể chia sẻ files trên ổ đĩa.
    - Sử dụng ít tài nguyên hệ thống hơn do container chạy bên trong cùng kernel.
    - Các phần mềm cốt lõi của hệ điều hành bị xóa bỏ khỏi Docker image.Do đó container có thể rất là nhẹ. Không hề có thêm chi phí cho việc hypervisor hay một hệ điều hành khách.
    - Thời giạn khởi động app cũng sẽ nhanh hơn.
  + Lợi ích của việc sử dụng Docker:
    - Hiệu năng cao.Tạo và hủy nhanh, dễ.
    - Các ứng dụng trở nên lightweight.
    - Deploy nhanh hơn, dễ dàng migration, và thời gian khởi động app nhanh.
    - Docker cho phép deploy ready-to-run phần mềm, portable và dễ dàng distribute. Các app chạy một cách đơn giản bên trong conatiner, khôgn cần cài đặt. Việc không cài đặt có một lợi thế rất lớn, nó giúp loại bỏ các vấn đề như là sự xung đột phần mềm và thư viện hoặc thậm chí các vấn đề về tương thích.
    - Docker container là portable, có thể chạy ở bất kì đâu : trên máy local, remote server, cloud.
    - Dev có thể tạo ra các phần mềm mà k cần quan tâm tới nền tảng nó sẽ chạy về sau.
    - Docker phù hợp với continuous integration or continuous delivery flow do việc sử dụng Docker's repoducible build environment.

# Docker concepts

## Image

* + Read-only template, là nền tảng cơ bản để tạo ra 1 container từ nó.Nó giống như một công thức chứa các định nghĩa của tất cả các thứ mà ứng dụng của bạn cần để hoạt động. Nó có thể là Linux với một server và 1 ứng dụng Java. Mọi image bắt đầu từ 1 base image, như là Ubuntu, Linux image.Bạn có thể tự xây dựng từ đầu hoặc sử dụng một số image có sẵn.
  + Image dc tạo bởi việc sử dụng 1 chuỗi các command, gọi là instruction.Instruction được viết trong Dockerfile.Docker file chỉ đơn giản là 1 text file, chứa một bộ sưu tập có thứ tự các thay đổi của hệ thống tập tin root(giống như chạy 1 command để chạy server, tạo file hoặc thư mục, tạo biến môi trường...) và các tham số thực thi tương ứng để sử dụng bên trong container runtime.
  + Docker sẽ đọc Dockerfile khi bạn chạy tiến trình để build image và thực thi các instruction từng lệnh một. Kết quả cuối cùng sẽ ra 1 image. Mỗi instruction sẽ tạo ra 1 layer mới trong image.
  + Image layer sẽ trở thành cha cho layer được tạo bởi instruction tiếp theo. Image có thể chạy trong Docker container trên bất kì host nào chạy Docker. DOcker sử dụng image để chạy code của bạn, không phải Dockerfile.
  + Dockerfile được sử dụng để tạo ra image khi chạy câu lệnh docker build.

## Layer

* + Mỗi image chứa các layer được cấu trúc dạng stack.Mỗi lớp là 1 image trung gian. Bằng việc sử dụng union filesystem,Docker kết hợp tất cả các layer vào 1 thực thể image duy nhất. union filesystem cho phép các file và các thư mục của các hệ thống file riêng biệt chồng lên nhau trong suốt, tạo thành duy nhất.Nội dung và cấu trúc của thư mục có cùng đường dẫn bên trong các hệ thống file riêng biệt sẽ được nhìn thấy nhau trong 1 thưc mục sáp nhập duy nhất, nằm trong một thứ goi là hệ thống file ảo. Nói cách khác, cấu trúc hệ thống file ở layer trên cùng sẽ được gộp voiứ các cấu trúc file ở dưới.
  + Để duy trì trật tự các lớp, Docker sử dụng ID và pointer. Mỗi layer sẽ có ID và pointer trỏ tới layer cha. Layer mà k có con trỏ trỏ tới cha là layer ở đầu tiên, layer base.
  + Layer có một số tính năng hay:
    - Có thể tái sử dụng và được cache.Khi Docker xử lý Dockerfile, nó nhìn vào 2 thứ: 1 là các instruction đc thực thi và image cha. Docker sẽ quét tất cả các con của layer cha và tìm kiếm 1 sao cho command trùng với instruction hiện tai. Nếu tìm thấy,
    - Docker sẽ bỏ qua instruction tiếp theo và lặp lại quá trình. Nếu k tìm thấy layer trong cache, layer mới sẽ được tạo.Ví dụng với instruction thêm 1 file vào image, Docker sẽ tạo ra 1 mà checkssum cho mỗi nội dung file. Trong quá trình build, checksum đó được sử dụng để so sánh với checksum của image đang tồn tại kiểm tra xem nếu layer đó có thể được tái sử dụng từ cache không.Nếu 2 imge khác nhau có thành phần chung,Ví dụ như Linux shell hoặc Java runtime, Docker sẽ track tất cả các layer được kéo về và sẽ tái sử dụng shell layer trong cả 2 image.Khi download 1 image, layer sẽ được tải sử dụng và chỉ phần khác biệt mới được kéo mới về từ Docker Hub.
    - Nó sẽ tiết kiệm được thời gian, băng thông. Một khi Docker cache image layer cho một instruction, nó sẽ không cần build lại. Thay vì chúng ta cần distribute toàn bộ image, bạn chỉ cẩn push phần được chỉnh sửa. Điều này khiến quá trình trở nên đơn giản và nhanh hơn. Nó đặc biệt hữu ích nếu bạn sử dụng Docker luồng CD (continuous deployment flow):đẩy một git branch sẽ trigger việc build một image và sau đó pulish ứng dụng cho mọi người.
    - Việc reuse layer là nguyên nhân tại sao Docker lại lightweight khi so sánh với máy ảo.Đó là nhờ vào layer mà khi bạn kéo một image về, bạn k cần phải download tất cả các hệ thống tập tin của nó. Nếu bạn đã có một image khác mà có một vài layer của image mà bạn muốn pull, chỉ những layer thiếu mới được download về.
    - Layer tạo ra một history về việc image được build như thế nào. Do đó chúng ta sẽ có thể sử dụng thêm 1 feature mới: Rollback tới một điểm trong image's history. Vì mỗi hình ảnh chứa tất cả các bước build, chúng ta có thể dễ dàng quay trở lại một step trước đó nếu như ta muốn. Điều này có thể thực hiện được bằng cách gắn thẻ một lớp nhất định. Layer và image liên quan chặt chẽ tới nhau.
    - Image được lưu trữ như là một loadj các read-only layer. Nghĩa là một khi image được tạo, nó k thể thay đổi.

## Container

* + Nó là 1 running instance của image.
  + Có thể có nhiều running container của cùng 1 image.
  + Để chạy container, ta sử dụng command: docker run [options] IMAGE [COMMAND] [ARG...]
  + Khi chạy dòng lệnh trên, Docker sẽ check xem image có trên máy local của bạn không? Nếu không, nó sẽ được kéo về từ remote repository. Docker engine lấy image và thêm một writeable layer lên trên cùng của layer stack của image đó. Tiếp theo, nó sẽ khởi tạo tên image, ID, giới hạn tài nguyên như CPU, bộ nhớ. Trong phase này, Docker cũng sẽ thiết lập địa chỉ IP cho container bằng cách tìm kiếm và đính một địa chỉ IP có sẵn từ pool.
  + Bước cuối cùng của quá trình thực thi sẽ là các command thực tế, được truyền thông qua các tham sôs cuối của lệnh docker run. Nếu option "it" được sử dụng, Docker sẽ capture và cung cấp container output, sẽ được hiển thị trên màn hình console.
  + Bây giờ chúng ta có thể làm những việc như bình thường làm khi chuẩn bị OS để chạy các ứng dụng. Nó có thể là cài đặt các package, kéo source về từ Git, build ứng dụng Java sử dụng Maven,... Tất cả các action đó sẽ thay đổi hệ thống file trên cùng, chính là writeable layer.
  + Nếu bạn sau đó thực thi commit command, một image mới chứa tất cả thay đổi của bạn sẽ được tạo, sẵn sàng để chạy về sau.Để dừng một container, sử dụng command: docker stop Container khi bị dừng sẽ giữ lại tất cả cài đặt và thay đổi của hệ thống file (ở trên layer trên cùng chính là writeable layer). Tất cả tiến trình chạy trong container sẽ bị dừng và bạn sẽ mất tất cả mọi thứ trong bộ nhớ.
  + Để liệt kê các container bạn có trên hệ thống của bạn,sử dụng câu lệnh:docker ps -a
  + Kết quả sẽ là 1 bảng chứa danh sách các container với ID của nó, ngày tạo, câu lệnh dc sử dụng để chạy container, status, exposed ports, tên. Để xóa 1 container, có thể sử dụng câu lệnh docker rm. Nếu bạn muốn xóa bỏ nhiều một lúc, có thể sử dụng 1 danh sach container và 1 filter:docker rm $(docker ps -a -q -f status=exited)
  + Khi container đc bắt đầu, writeable layer trên cùng của layer stack là dành cho chúng ta xử lý. Chúng ta có thể thực sự thay đổi một conatiner đang chạy, nó có thể là thêm, thay đổi file, cài đặt gói ứng dụng, cấu hình OS,... Nếu bạn thay đổi file trong container đang chạy, các tập tin sẽ được đưa ra và đặt lên trên cùng, ở writeable layer. các thay đổi của chúng ta chỉ có thể ở trên top layer. Thông qua câu lệnh docker commit, bạn sẽ tạo ra 1 read-only image từ running container (và tất cả thay đổi trong writeable layer): docker commit <container-id> <image-name>
  + Câu lệnh docker commit lưu tất cả thay đổi bạn gây ra với container vào trong writeable layer. Để tránh việc hỏng dữ liệu hay tính không nhất quán dữ liệu. Docker sẽ tạm dừng container bạn đang commit thay đổi vào trong đấy. Kết quả của câu lệnh docker commit là 1 brand mới, read-only image, thứ bạn có thể tạo ra container mới từ nó.
  + Trong kết quả trả về từ việc commit thành công, Docker sẽ cho ta thấy ID của phiên bản mới image. Nếu bạn xóa container mà k commti trước và sau đó chạy lại cùng image đó, Docker sẽ chạy lại container mới mà k có bất cứ thay đổi nào từ running container trước đó.Trong cả 2 trường hợp, có commit và k có commit, thay đổi của bạn tới hệ thống file sẽ k bao giờ ảnh hưởng tới image base. Tạo ra image bằng cách thay đổi top writeable layer trong container rất hữu ích khi debug và thử nghiệp, nhưng nó thường tốt hơn khi sử dụng Dockerfile để quản lý các image của bạn.

## Docker registry,repository. index

### Docker registry

* + Là service lưu trữ Docker images. Docker Hub là 1 ví dụ về Docker registry.Repository là 1 tập hợp (namespace) của các image có liên quan, thường cung cấp các phiên bản khách nhau của cùng 1 ứng dụng hay service.Nó là tập hợp các image khác nhau với cùng tên và khác tag.
  + Ví dụ: nếu app của bạn tên là hello-world-java và username( hoặc namespace) cho Registry là dockerJavaDeveloper thì image sẽ được đặt ở dockerJavaDeveloper/hello-world-java repository. Bạn có thể gắn nhẵn image và lưu trữ nhiều phiên bản khách nhau của image với các ID khác nhau trong cùng 1 repository và truy cấp các phiên bản với tag khác nhau của image với cú pháp đặc biệt: username/image\_name:tag. Docker repository tương tự như Git repository:{registryAddress}/{namespace}/{repositoryName}:{tag}
  + Docker Hub là registry mặc định và Docker sẽ kéo image từ DockerHub nếu bạn không cụ thể địa chỉ registry.
  + Để tìm kiếm image trong registry, sử dụng command: docker search Ex: docker search hello-java-world.
  + Nếu k cụ thể remote registry, Docker sẽ tìm kiếm trên Docker Hub và đưa ra danh sách các image phù hợp với điều kiện tìm kiếm.

### Index

Index quản lý việc tìm kiếm và gắn thể và các tài khoản người dùng, quyền. Khi thực thư remote command như push hoặc pull, index đầu tiên sẽ nhìn vào tên của image và sau đó check xem nếu nó có repository tương ứng, Nếu có index verify xem nếu bạn có quyền truy cập hoặc thay đổi image không.Nếu bạn có, hoạt động sẽ được approve và registry lấy hoặc gửi image.

# Networking và persistent Storage

## Networking

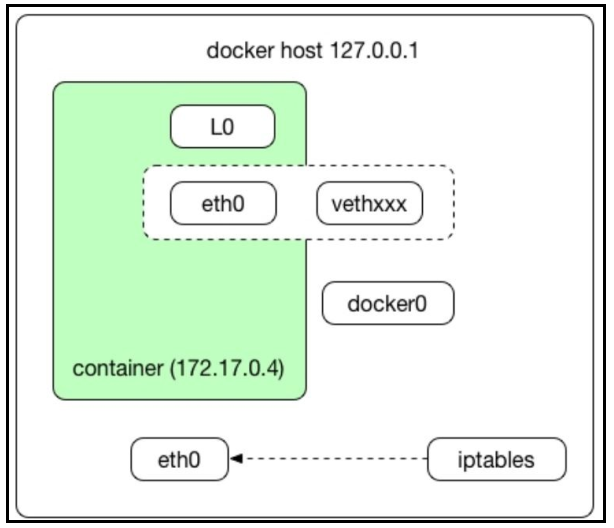
Cho phép container có khả năng giao tiếp với bên ngoài thế giới, cac server khác, container khác.Docker cung cấp một số cách khách nhau để cấu hình network.

Docker network type: Có 3 loại. để liệt kê danh sách các loại , ta sử dụng command: docker network ls.

### Bridge

Được dùng làm mặc định trong Docker. Khi tiến trình dịch vụ Docker chạy, nó cấu hình một bridge ảo, đc đặt tên là docker0. Nếu chúng ta k đưa tham số cấu hình mạng trong câu lệnh docker run -net=<NETWORK>, tiến trình docker sẽ kết nối container tới bridge network theo mặc định. Do đó, nếu bạn tạo mới 1 container, nó sẽ được kết nối tới bridge network.

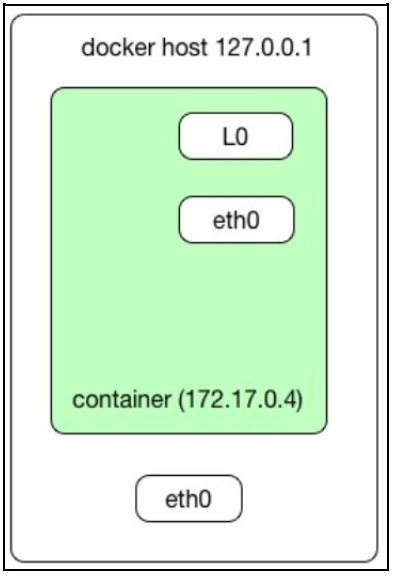
Với mỗi container được Docker tạo ra, nó cấp phát 1 thiết bị Ethernet ảo, thứ sẽ được gắn vào bridge.Thiết bị Ethernet ảo được ánh xạ tới eth0 trong container.



Interface eth0 được đưa ra với 1 địa chỉ IP từ dải địa chỉ của bridge. Nói cách khác, Docker sẽ tìm ra 1 địa chỉ IP free từ dải địa chỉ sẵn có trên bridge và sẽ cấu hình interface eth0 của container với địa chỉ IP đó. Từ giờ trở đi, nếu một container mới muốn, ví dụ kết nối tới Internet, nó sẽ sử dụng bridge, chính là địa chỉ IP của máy chủ. Bridge sẽ tự động chuyển tiếp các packet giữa bất kì network interface mà được gắn với nó và cũng cho phép container có thể giao tiếp với máy host, tương tự như với các container trên cùng host.Bridge network được sử dụng khá là thường xuyên

### Host

Kiểu network này chỉ đơn giản là đặt container vào trong stack network của host. Đó là, tất cả các network interface được định nghĩa trên host sẽ có thể truy cập đến container



Nếu bạn chạy container sử dụng option -net=host, container sẽ sử dụng host network. Tốc độ mạng sẽ nhanh như mạng bình thường: Không hề có bridge, không có translation. Vì thế rất hữu ích khi bạn cần một mạng có hiệu năng cao. Container chạy trong stack network của host sẽ đạt được hiệu nặng mạng cao hơn khi so sánh với các container sử dụng mạng bridge, sẽ k cần việc duyệt docker0 bridge và ánh xạ cổng trên iptables,trực tiếp cho nó ra thế giới bên ngoài. Bằng cách sử dụng -net=host, container của bạn sẽ có thể được truy cập thông qua địa chỉ IP của host. Tuy nhên nó khá là nguy hiểm. Nếu bạn có một ứng dụng chạy như là ứng dụng root và nó có một số lỗ hổng, có thể sẽ xảy ra một nguy hại về bảo mật, và ai đó có thể kiểu khiển từ xa host network của bạn thông qua Docker container. Sử dụng kiểu mạng này đồng nghĩa là bạn sẽ cần ánh xạ port để tìm tới service đang chạy trong container.

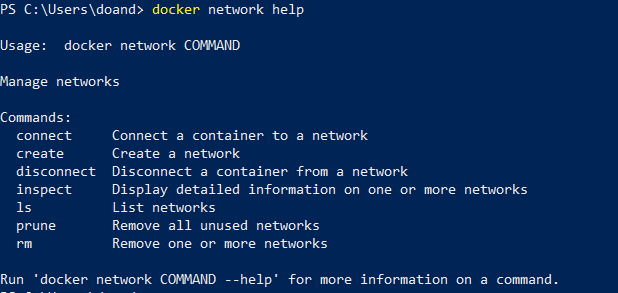
### None

None network tức là k cấu hình mạng. K có driver nào được sử dụng bởi kiểu này. Nó sẽ hữu ích khi bạn không muốn container của mình có khả năng truy cập mạng. Sử dụng option -net=none để vô hiệu hóa network.

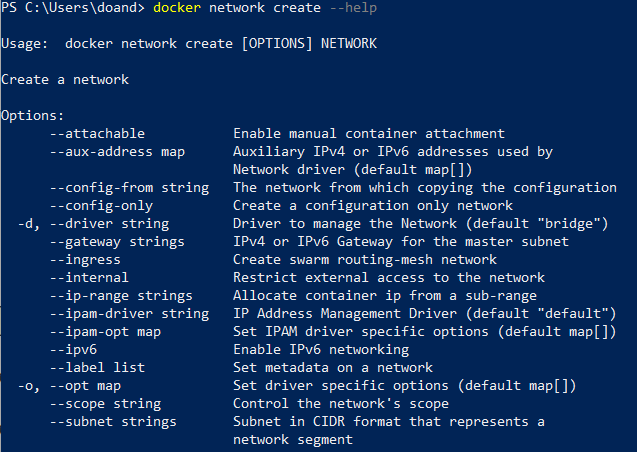
### Networking commands

Là các sub của câu lệnh docker network.

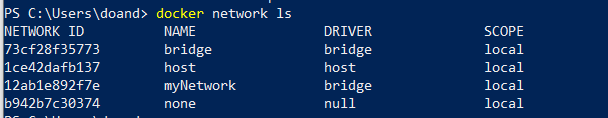
Docker network help :liệt kê tất cả câu lệnh



Ddocker network <COMMAND> --help: xem chi tiết cú pháp và miêu tả về câu lệnh COMMAND. EX: docker network create --help:



docker network ls: Liệt kê danh sách các network có sẵn cho container của bạn.



docker network create: tạo mới 1 network.

docker network rm: xóa network.

docker network connect : connect container tới mạng cụ thể

docker network disconnect: disconnect container khỏi một mạng cụ thể

docker network inspect: hiển thị chi tiết thông tin về mạng.

VD:

Tạo một mạng mới: docker network create myNetwork

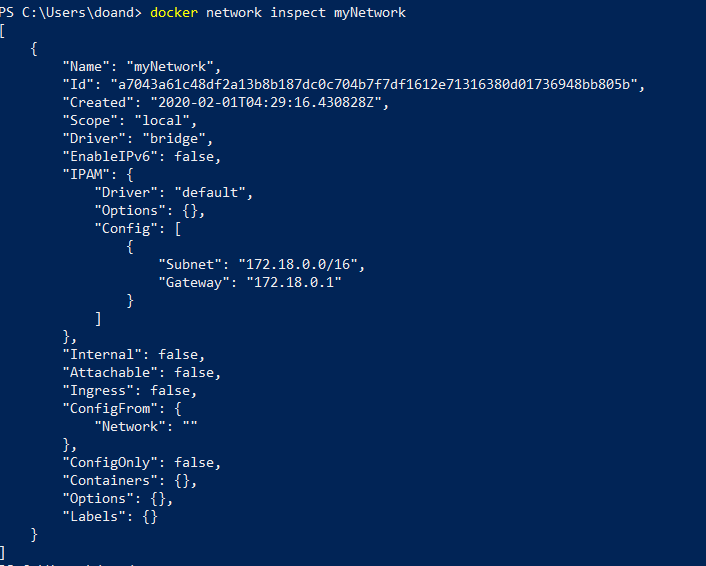


Một số tham số của câu lệnh docker network create:

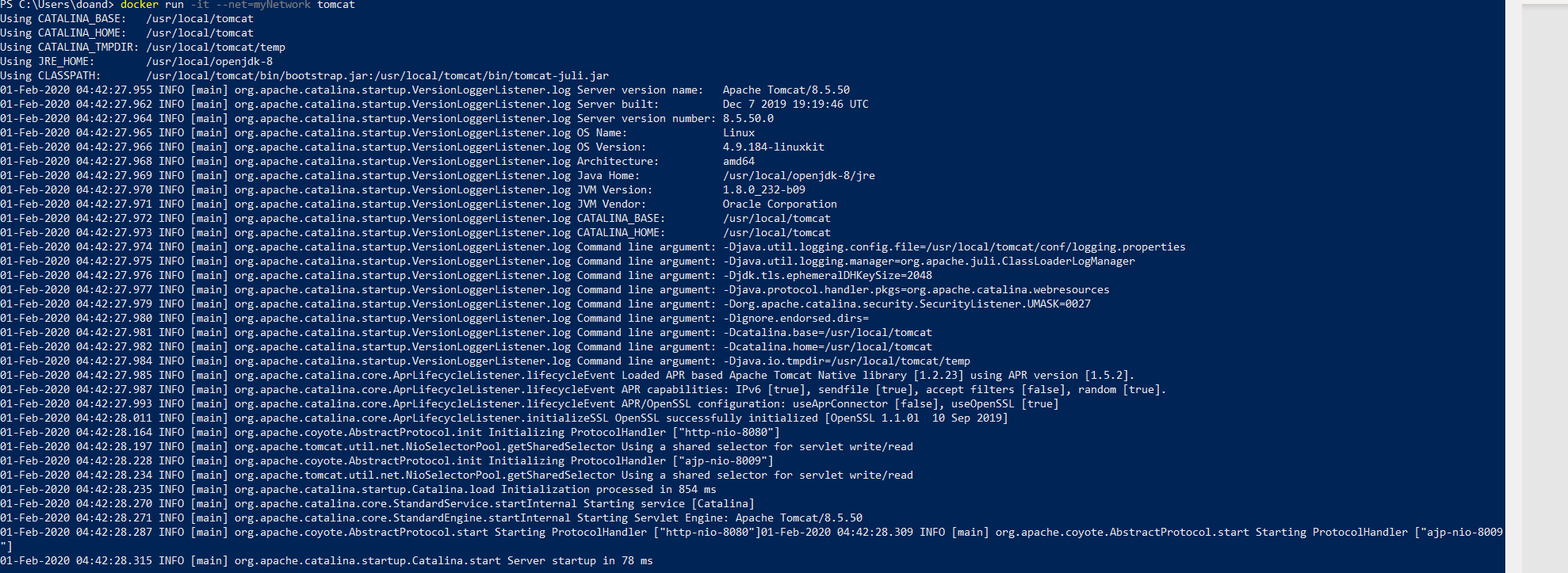
|  |  |
| --- | --- |
| OPTION | DESCRIPTION |
| -d, -driver=”bridge” | Driver quản lý mạng |
| -aux-address=map[] | Đia chỉ phụ trợ IPv4 hoặc IPv6 được sử dụng bởi network driver |
| -gateway=[] | IPv4 hay IPvv6 gateway cho master subnet |
| -ip-range=[] | Cấp phát địa chỉ IP container từ sub-range |
| -ipam-driver=default | Driver quản lý địa chỉ IP |
| -o, -opt=map[] | Thiết lập các options cho driver |
| -subnet=[] | Subnet ở định dnạg CIDR đại diện một network segment |

Tham số quan trọng trong danh sách trên chính là -d options, được sử dụng để cụ thể loại network.

Thông số chi tiết của 1 mạng : docker network inspect myNetwork



Kết nối container với network: ta sử dụng câu lệnh docker run –net=<NETWORK>, <NETWORK> chính là tên của mạng mà bạn muốn sử dụng. Ex: docker run -it –net=myNetwork tomcat



Bạn có thể sử dụng cách khác, connect tới mạng tương tự container khác đang sử dụng.

Ex:

docker run -it --net=bridge myTomcat

docker run -it --net=container:myTomcat myPostgreSQL

### Exposing port và mapping port

Khi bạn muốn ứng dụng trong container của bạn chấp nhận các kết nối tới, hay từ các container khác hoặc thậm chỉ từ bên ngoài Docker. Nó cũng có thể là khi ứng dụng server của bạn lắng nghe trên cổng 80 hay co sở dữ liệu chấp nhận các yêu cầu tới.

Image có thể expose các port. Expose port tức là ứng dụng của bạn sẽ lắng nghe trên các port đó. Ví dụ như Tomcat server sẽ lắng nghe trên port 8080 theo mặc định. Tất cả container chạy trên cùng host và trên cùng 1 mạng có thể giao tiếp với Tomcat trên port đó. Expose port có thể được thực hiện bởi 2 cách sau:

* Sử dụng EXPOSE instruction trong Dockerfile
  + Ex: FROM openjdk:8-jre-alpine

ENV CATALINA\_HOME /usr/local/tomcat

ENV PATH $CATALINA\_HOME/bin:$PATH

RUN mkdir -p "$CATALINA\_HOME"

WORKDIR $CATALINA\_HOME

EXPOSE 8080

CMD ["catalina.sh", "run"]

* Sử dụng –expose option khi chạy lệnh docker run.
  + Ex: docker run -it --name myTomcat --net=myNetwork tomcat

Khi bạn muốn giao tiếp giữa container và thế giới bên ngoài, ta cần ánh xạ port. Chúng ta có thể ánh xạ 1 port, expose nó bởi Docker container,vào trong port của máy host. Điều này sẽ khiến port trên máy host sẽ được ánh xạ tới port cụ thể của container đang chạy. Để thực hiện được việc gán port từ máy host tới container, chúng ta sử dụng cờ -p trong câu lệnh docker run.

Ex: docker run -it --name myTomcat2 --net=myNetwork -p 8080:8080 tomcat

Câu lệnh trên có nghĩa: chúng ta ánh xạ port 8080 của container tới port 8080 của máy host.

Cấu trúc lệnh: docker run -p <hostPort>:<containerPort> <image ID or name>

Docker image có thể expose một dải port sử dụng EXPOSE instruction trong Dockerfile hoặc sử dụng docker run

Ex: docker run --expose=7000-8000 <container ID or name>

Sau đó, bạn có thể ánh xạ cả dải port đó từ máy host tới container bằng câu lệnh docker run

Ex: docker run -p 7000-8000:7000-8000 <container ID or name>

Có sự khác biệt giữa việc expose và ánh xạ port. Expose sẽ có tác dụng vào thời điểm runtime, nhưng k tạo ra ánh xạ tới host. Việc expose có tác dụng tới các container chạy cùng mạng, cùng host. Còn việc ánh xạ port tương tự với việc publish: nó sẽ tạo 1 quy tắc ánh xạ port, ánh xạ port trên container với port trên host. Việc ánh xạ sẽ có tác dụng ngoài Docker. Nếu chúng ta chỉ sử dụng -p mà k có EXPOSE trong Dockerfile, Docker sẽ ngầm định sử dụng EXPOSE port đó .

Có một cách giúp bạn ánh xạ tất cả các port được expose trong image một lần duy nhất, tự động trong quá trình khởi động container. Đó là sử dụng -P. Nó sẽ có tác dụng ánh xạ một port trên host ngẫu nhên được cấp phát tới tất cả các port của container mà được expose trong Dockerfile ở câu lệnh EXPOSE.

## Pesistent storage

Chúng ta thường mong muốn việc dữ liệu được tách biệt, không liên quan tới life cycle của container, để có thê sử dụng lại. Một ví dụ điển hình là về web application server: Docker image chứa phần mềm web server, ví dụ như là Tomcat, với một ứng dụng Java được deploy, cấu hình và sẵn sàng để sử dụng. Nhưng dữ liệu của server sẽ được dung nên được tạch biệt khỏi image. Việc đó có thể thực hiện được thông qua volume. Volume không phải là 1 phần của hệ thống file hợp nhất, nó tồn tại như một thư mục và file trên hệ thống file của host. Có 3 mục tiêu chính của volume mà Docker sử dụng:

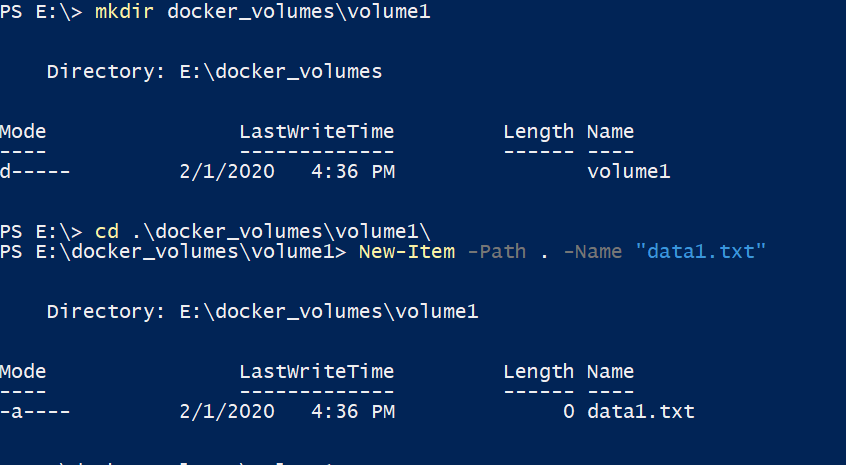
* Chia sẻ dữ liệu giữa hệ thống file host và Docker container
* Giữ lại dữ liệu khi container bị xóa bỏ
* Chia sẻ dữ liệu giữa các Docker container

Một số câu lệnh:

|  |  |
| --- | --- |
| Command | Ý nghĩa |
| docker volume create | Tạo một volume |
| docker volume inspect | Hiển thị chi tiết thông tin của 1 hay nhiều volume |
| docker volume ls | Liệt kê danh sách volume |
| docker volume rm | Xóa một hay nhiều volume |
| docker volume prune | Xóa bỏ tất cả các volume không sử dụng, những volume k được ánh xạ tới bất kì container nào cả |

### Creating a volume

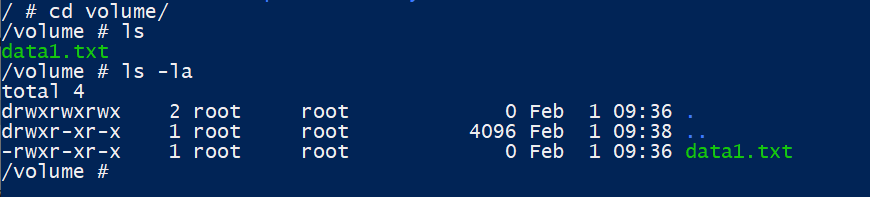
Ta sẽ tạo ra volume docker\_volumes/volume1 trong ổ E của máy host.



Có 2 cách để tạo ra volume. Motọ là cụ thể -v option khi chạy image.

Ex: docker run -v E:/docker\_volumes/volume1:/volume -it busybox

Câu lệnh trên chúng ta tạo ra volume sử dụng -v option và chỉ cho Docker biết rằng thư mục trên máy host E:/docker\_volumes/volume1 sẽ được ánh xạ vào thư mục /volume của container đang chạy.

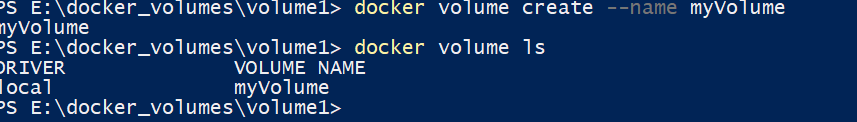


Ta cũng có thể ánh xạ với các file. Điều này rất hữu ích trong việc bạn có các file cấu hình.

Cách khác để tạo ra một volume trước khi chạy bất kì container nào, đó là sử dụng câu lệnh docker volume create.

Ex: docker volume create --name myVolume

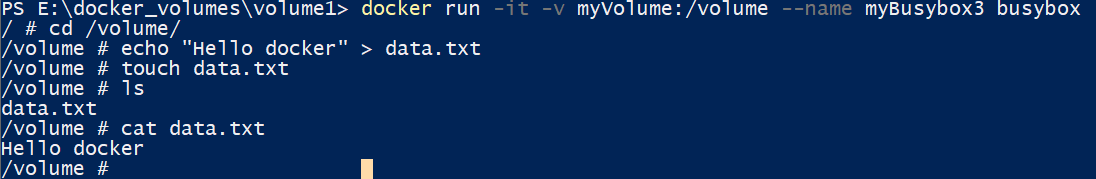
Để hiển thị tất cả volume hiện có, sử dụng câu lệnh docker volume ls.



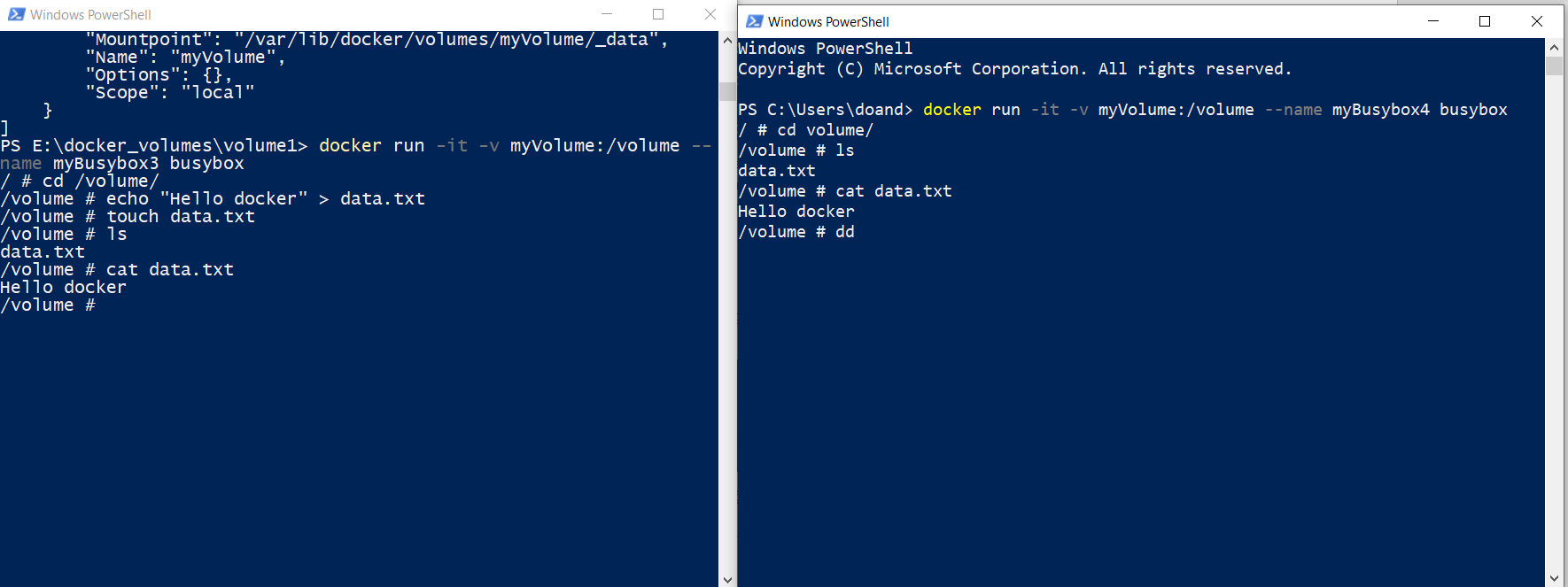
Volume được tạo theo cách này sẽ chưa được ánh xạ vào bất cứ đường dẫn nào trong máy host. Nếu image cơ bản của container chứa dữ liệu tai tại thời điểm mount, dữ liệu đó sẽ được copy vào trong volume mới khi volume được khởi tạo.Nó khác bietẹ với việc cụ thể thư mục trên máy host. Việc này giúp cho khi bạn tạo ra image, bạn không cần quan tâm tới thư mục của volume trên máy host.Để ánh xạ container với volume, ta làm như sau:

docker run -it -v myVolume:/volume --name myBusybox3 busybox

Câu lệnh trên nói cho Docker biết rằng chúng ta ánh xạ volume vừa tạo myVolume với thư mục /volume trên container.



Giờ ta sẽ khởi tạo một container khác, cũng ánh xạ vào volume đó



Như trên ta có thể thấy, 2 container chia sẻ dữ liệu và có thể truy cập như nhau.

Một cách khác để chia sẻ volume giữa các container là sử dụng -volumes-from trong câu lệnh docker run.Câu lệnh này nhằm mục đích nếu container của bạn đã có voulume đc mount, bạn có thể chỉ cho Docker sử dụng volume này vào container khác.

Ex: docker run -it -volumes-from myBusybox4 --name myBusybox5 busybox

Và cách cuối cùng đó là sử dụng VOLUME CREATE trong Dockerfile.

### Removing a volume

Có 2 cách.Đầu tiên, bạn có thể remove volume bằng tham chiếu tới tên của container và thực thi câu lệnh docker rm -v

docker rm -v <containerName or ID>

Khi đó, bạn sẽ có 1 volume không có tham chiếu bởi bất kì container nào. Bạn có thể sử dụng câu lệnh prune để xóa nó đi.

Cách khác, đó là sử dụng câu lệnh docker volume rm

docker volume rm <volumeName or ID>

Nếu volume đang được sử dụng bởi container nào, bạn sẽ không thể xóa và Docker sẽ bắn ra thông báo.

### Volume drivers

Tương tự như driver mạng, volume cũng có các driver.Ví dụ: Docker volume driver for Azure file storage,

IPFS, Keywhiz

# Các instruction trong docker file

## From

* + Instruction đầu tiên.
  + Thiết lập base image cho tất cả các instruction sau.
  + Syntax: FROM <image>, or FROM <image>:<tag>, or FROM <image>@<digest>

## MAINTAINER

* + Set Author field cho generated image.
  + Có thể là tên, username, ....
  + Đặt ở bất kì đâu trong file docker.
  + Syntax:MAINTAINER author\_name

## WORKDIR

* + Thêm working directory cho các instruction : CMD,RUN,ENTRYPOINT,COPY,ADD.
  + Syntax: WORKDIR /PATH
  + Có thể có multipe WORKDIR instruction/1 file docker.
  + Nếu relative path dc proviđe, nó sẽ trở thành relative với path của workdir instruction trước đó

## ADD

* + Copy files từ source->destination
  + Syntax: ADD <source path or URL> <destination path>
    - Source: có thể là đường dẫn file, thư mục hoặc URL. Là đường dẫn tương đối.
    - Source và destination path có thể chứa wild cards. (\*,?).
    - Source có thể là multipe paths,cách bởi dấu ",", đều là tương đối so với build context
    - Nếu trong path chứa dấu cách, sử dụng square brackets around : ADD ["<source path or URL>" "<destination path>"]
    - Path không kết thúc bởi / => Single file.
    - Path kết thúc bở / =>Directory.Tất cả content sẽ được copy nhưng thư mục sẽ k dcd create ở destination path.
    - Nếu là file compresed như ZIP,TAR, nó sẽ được decompress vào trong destination path
    - Destination directory có thể là absolute path hoặc relative path với directory specific bởi WORKDIR instruction.
      * Example :
        + ADD config.json projectRoot/ : add file config.json to <WORKDIR>/projectRoot/
        + ADD config.json /absoluteDirectory/ : add file config to /absoluteDirectory/
    - Permission của file: Trùng với permission của file nguồn. Nếu file từ URL-> permission được gán giá trị 600. Nếu muốn thay đổi giá trị permission. Cần viết thêm các instruction sau ADD instruction

-COPY:

* + Tương tự ADD instruction
  + Syntax: COPY <source path or URL> <destination path>
  + Chỉ copy file k làm gì khác.
  + File được copy tới image thể được cache.

-RUN:

* + Là central executing instruction của Dockerfile.
  + Thực thi command ( nhiều command) trong layer mới nhất trên cùng của current image sau đó commit các kết quả.
  + kết quả image đã commit sẽ dc sử dụng như là base cho instruction tiếp theo trong Dockerfile.
  + RUN sẽ lấy command như là tham số và chạy nó để tạo ra layer mới.

🡺 các câu lệnh COPY và ENTRYPOINT thiết lập các parameter có thể bị ghi đè tại thời điểm runtime. Nếu k thay đổi bất cứ gì sau khi chạy container, kết quả sẽ luôn luôn như vậy. RUN chạy vào thời điểm build time.

* + RUN chạy theo thứ tự được viết trong Dockerfile
  + Cache busting:Sử dụng "version pinning" để tránh cache. hoặc gộp các câu lệnh thành 1.
  + EX:

FROM ubuntu

RUN apt-get update

RUN apt-get install -y openjdk-8-jre

RUN apt-get install -y nodejs

==>

RUN apt-get update \

&& apt-get install -y openjdk-8-jre \

&& apt-get install -y nodejs \

&& apt-get clean

## CMD

* + Truyền một lệnh của Linux mỗi khi thực hiện khởi tạo một container từ image.
  + cung cấp mặc định cho việc thực thi container.
  + Có thể nghĩ đấy là starting point của image, khi container được chạy sau này.
  + Có thể là một excutable, hoặc, nếu bạn cụ thể ENTRYPOINT instruction, bạn có thể bỏ qua excutable và cung cấp các giá trị parameter default.
    - Syntax:
      * CMD ["executable","parameter1","parameter2"]
        + Đây là dạng exec form, được yêu thích và khuyên dùng. Các tham số là json array và được bao trong dấu "".Dạng này sẽ k gọi một command shell khi container được chạy.
        + Nó chỉ chạy excutable được cung cấp như là tham số đầu tiên.Nếu ENTRYPOINT instruction được sử dụng trong file Dockerfile, CMD cung cấp 1 tập các tham số mặc định cho ENTRYPOINT instruction
      * CMD command parameter1 parameter2
        + Đây là dạng shell form. Shell sẽ sử lý command.
        + Nên sử dụng exec form bởi vì: Mọi thứ được bắt đầu thông qua shell sẽ được bắt đầu như là 1 subcommand của /bin/sh -c, nó se k truyền signals, do đó excutable sẽ k có PID 1 của container, và sẽ k nhận được Unix signals, excutable sẽ k nhận dc SIGTERM từ docker stop <container>.
  + RUN và CMD có sự khác biệt
    - RUN thực thi vào thời điểm build,được sử dụng để build image,bằng việc tạo ra new layer.
    - CMD được thực thi khi container được chạy bởi thực thi dòng lệnh docker run .

## ENTRYPOINT

* + Cho phép bạn cấu hình container sẽ chạy như 1 excutable.
  + Cụ thể 1 command sẽ luôn luôn được chạy khi container start.
  + CMD cụ thể các tham số sẽ được ENTRYPOINT sử dụng.
  + Docker có 1 ENTRYPOINT mặc định là /bin/sh -c nhưng không giống mặc định CMD.
  + Syntax: tương tự CMD.
  + Example: ENTRYPOINT [ "sh", "-c", "echo $HOSTNAME" ]
  + Cho phép ta đặt các lệnh và tham số sau đó sử dụng một trong 2 dạng:
    - Một là CMD để đặt các tham số bổ sung
    - Hai là các đối số ENTRYPOINT
  + Các đối số CMD có thể đc ghi đè bằng các tham số dòng lệnh được cung cấp khi Docker container chạy
  + Example:
    - ENTRYPOINT ["/bin/echo", "Hello"]

CMD ["world"]

* + - * Nếu chạy :docker run -it < image > => Hello world
      * Nếu chạy :docker run -it < image > John => Hello John
      * Shell form của ENTRYPOINT ignore tất CMD và các tham số dòng lệnh
      * Có thể ghi đè các setting của ENTRYPOINT bằng cách sử dụng --entrypoint khi start container.

## EXPOSE

* + Container lăngs nghe trên các port cụ thể tại thời điểm runtime.
  + Dùng để cho phép incoming traffic tới container.EXPOSE không khiến các port của container tự động đc chấp nhận trên máy host. Phải dử dụng -p flag để public range port hoặc -P flag để publc tất cả các port một lần

## VOLUME

* + Syntax:VOLUME ["/volumeName"]
  + Example: VOLUME ["/var/lib/tomcat8/webapps/"]
    - VOLUME /var/log/mongodb /var/log/tomcat
  + Tạo ra một mount point với tên cụ thể và đánh dấu nó như là một externally mounted volumes của native host hoặc các container khác.
  + Câu lênh VOLUME sẽ mount directory trong container của bạn và store bất kì file nào đc tạo hoặc sửa vào trong directory đó trên ổ đĩa của host đó nằm ngoài cấu trúc file của container
  + Sử dụng VOLUME trong Dockerfile làm cho Docker biết thư mục đó chứa dữ liệu vĩnh viễn. Docker sẽ tạo volume đó để chứa data và k bao giờ xóa nó, trừ khi bạn xóa tất cả container sử dụng nó.
  + Nó cx bỏ qua union file system, do đó volume là một directory thực sự đc mount,hoặc đọc-ghi hoặc chỉ đọc, ở đúng nơi, ở trong tất cả các container được chia sẻ với nhau.
  + Example: FROM ubuntu
  + VOLUME /var/myVolume
  + VOLUME và -v là tương tự nhau. Điểm khác biệt làm ở bạn có thể sử dụng -v một cách động và mount thư mục trên máy host trên container khi chạy container bởi câu lệnh docker run.

## LABEL

* + Thêm metadata vào image.
  + LABEL dạng key-value.
  + Được sử dụng thêm, bao gồm tất cả các nhãn lấy từ một image là base của image của bạn.Nếu docker gặp 1 nhãn đã tồn tại, nó sẽ ghi đè nhãn đó với cùng key và giá trị mới.
  + RULE: các key chỉ có thể bao gồm các kĩ tự chữ thường và số, dấu chấm, gạch ngang và phải bắt đầu và kết thức với kí tự chữ và số. Để ngăn chặn naming conflic, Docker khuyên sử dụng không gian tên nhãn keys sử dụng kí hiếu tên miền ngược.Nói cách khác, keys mà k có namespaces (dấu chấm) được dành riêng cho việc sử dụng dòng lệnh
  + Syntax: LABEL "key"="value"

Multipe value:

LABEL description="This is my \

multiline description of the software."

Multipe label in a single image

LABEL key1="value1" key2="value2" key3="value3"

LABEL key1="value1" \

key2="value2" \

key3="value3

## ENV

* + Set các biến môi trường <key> với <value> của nó.
  + Có 2 option cho việc sử dụng ENV:
    - ENV <key> <value>
      * Example:ENV JAVA\_HOME /var/lib/java8
    - ENV <key>=<value>
      * Example:ENV CONFIG\_TYPE=file CONFIG\_LOCATION="home/Jarek/my \app/config.json"
  + Có thể sử dụng ENV để cập nhật biến môi trường PATH, và sau đó các thông số CMD sẽ nhận thức được điều đó.
    - Example:ENV PATH /var/lib/tomcat8/bin:$PATH => Nó sẽ đảm bảo rằng CMD ["startup.sh"] sẽ chạy, bởi vì nó se tìm file startup.sh trong system PATH.
  + Bạn cũng có thể sử dụng ENV để thiết lập số phiên bản thường xuyên thay đổi do đó việc upgrade sẽ dễ dàng:
    - Example:ENV TOMCAT\_VERSION\_MAJOR 8

ENV TOMCAT\_VERSION 8.5.4

RUN curl -SL <http://apache.uib.no/tomcat/tomcat-$TOMCAT_VERSION_MAJOR/v$TOMCAT_VERSION/bin/apache-tom> ENV PATH /usr/Jarek/apache-tomcat-$TOMCAT\_VERSION/bin:$PATH

## USER

* + Thiết lập username hoặc UID để sử dụng khi chạy image.Nó sẽ ảnh hưởng tới user cho bất cứ câu lệnh RUN,CMD, hay ENTRYPOINT nào được viết sau nó trong Dockerfile.
  + Syntax: USER tomcat
  + Bạn có thể sử dụng USER command nếu một excutable có thể chạy mà k vướng đặc quyền.Dockerfile có thể chứa instruction về user và group như: RUN groupadd -r tomcat && useradd -r -g tomcat tomcat
  + Thay đổi USER qua lại thường xuyên sẽ làm tăng số lớp trong resuling image và do đó sẽ khiến Dockerfile càng phức tạp.

## ARG

* + Sử dụng để truyền argument tới Docker daemon trong quá trình build :docker build.
  + Một định nghĩa biến ARG sẽ có hiệu lực từ dòng mà nó được định nghĩa trong Dockerfile. Bằng cách sử dụng --build-arg, bạn có thể gán giá trị cho các biến đã được định nghĩa:
    - Example:docker build --build-arg <variable name>=<value> .
  + Giá trị từ câu lệnh --build-arg sẽ được truyền vào daemon building image.Bạn có thể cụ thể multiline arguments sử dụng multiline ARG instructions.Nếu bạn cụ thể 1 tham số build time mà chưa được định nghĩa bởi ARG, quá trình build sẽ lỗi.Nhưng giá trị mặc định có thể được định nghĩa trong Dockerfile.
    - Example:FROM ubuntu

ARG user=jarek

* + Nếu k giá trị nào đc gán trước khi chạy build, giá trị mặc định sẽ được sử dụng.

## ONBUILD

* + Thêm 1 instruction sẽ được trigger khi vài image khác được build bởi việc sử dụng image đó như là base image.
  + Nói cách khác, ONBUILD instruction là 1 instruction the parent Dockerfile ban cho các child Dockerfile.Bất kì build instruction có thể được đăng kí như là trigger và các instruction của chúng sẽ được trigger ngay lập tức sau FROM instruction trong Dockerfile.
  + Syntax:ONBUILD <INSTRUCTION>
  + <INSTRUCTION> là 1 Dockerfile khác, sẽ được trigger khi child image dc build.
  + ONBUILD instruction k cho phép chaining của ONBUILD instruction khác và nó cx k cho phép From và MAINTAINER instruction như là ONBUILD trigger
  + Rất có ích nếu bạn đang xây dựng 1 image để sử dụng làm base để build các image khác.
    - Example: FROM maven:3.3-jdk-8-onbuild

CMD ["java","-jar","/usr/src/app/target/app-1.0-SNAPSHOT-jar-with-dependencies.jar"]

Bên trong docker parent:

FROM maven:3-jdk-8

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

ONBUILD ADD . /usr/src/app

ONBUILD RUN mvn install

* + - Đây là base image mà Java và Maven đều đươc install và 1 list instruction để copy files và chạy Maven
  + ONBUILD instruction add vào image 1 trigger instruction sẽ được thực thi tại thời điểm khi image được sử dụng như là base cho build khác.
  + Trigger sẽ được thực thi trong ngữ cảnh của child build,như thể là nó được chèn ngay sau FROM instruction trong child Dockerfile.
  + Khi Docker gặp 1 lệnh ONBUILD trong quá trình build, builder sẽ thêm một kiểu trigger vào metaData của imăg đang được build.
  + Nhưng đây là cách duy nhất image bị ảnh hưởng.Cuối build, 1 danh sách các trigger sẽ được chứa trong image manifest, dưới key OnBuild.
  + Bạn có thể nhìn thấy nó qua docker inspect command.
  + Sau đó image sẽ được sử dụng như là base cho các build mới, sử dụng From instruction.
  + Như la 1 phần xử lý của FROM instruction,Docker builder tìm kiếm các ONBUILD triggers, và thực thi chúng theo đúng thứ tự chúng đã được đăng kí.
  + Nếu bất kì trigger nào lỗi, FROM instruction sẽ bị bỏ qua và sẽ khiến build bị lỗi.Nếu tất cả trigger thành công, FROM instruction hoàn thành và quá trình build được tiếp tục

## STOPSIGNAL

* + Cụ thể system call signal sẽ được gửi tới container để exit,user sẽ sử dụng STOPSIGNAL instruction
  + Signal có thể cụ thể số math với syscall table.

## HEALTHCHECK

* + - Sử dụng để hướng dẫn Docker các check xem container còn working k.Nó có thể check nếu rest service response to http call hoặc chỉ đơn giản là lắng nghe trên 1 port cụ thể.
    - Container có thể có một vài status có thể lắng nghe sử dụng docker ps command.Chúng có thể là :created,restarting,running, pause, exited,dead.Nhưng thi thoảng nó là k đủ, container vẫn alive từ góc nhìn của Docker nhưng application có thể đã sập.Thêm các checking cho trạng thái ứng dụng có thể hữu ích và HEALTHCHECKS xuất hiện.
    - Trạng thái của HEALTHCHECK được thiết lập chi chạy. Bất cứ khi nào health check pass, nó sẽ trở thành healthy.Sau một số lần check fail, nó sẽ trở thnàh unhealthy
    - Syntax: HEALTHCHECK --interval=<interval> --timeout=<timeout> CMD <command>
      * interval(default 30s),timeout(default 30s) là các giá trị thời gian.
      * command là command được sử dụng để check xem app có đang chạy k.Exit code của command được Docker sử dụng để xác định xem health check thành công hay thất bại.Giá trị đó có thể là 0, tực là OK,1 là NOK
    - Java microservice healthcheck có thể được implement thông qua /ping REST endpoint, trả về bất cứ cái gì thậm chí trả về 1 response rỗng với HTTP status 200
      * Example: HEALTHCHECK --interval=5m --timeout=2s --retries=3 CMD curl -f http://localhost/ping

# Kubernetes

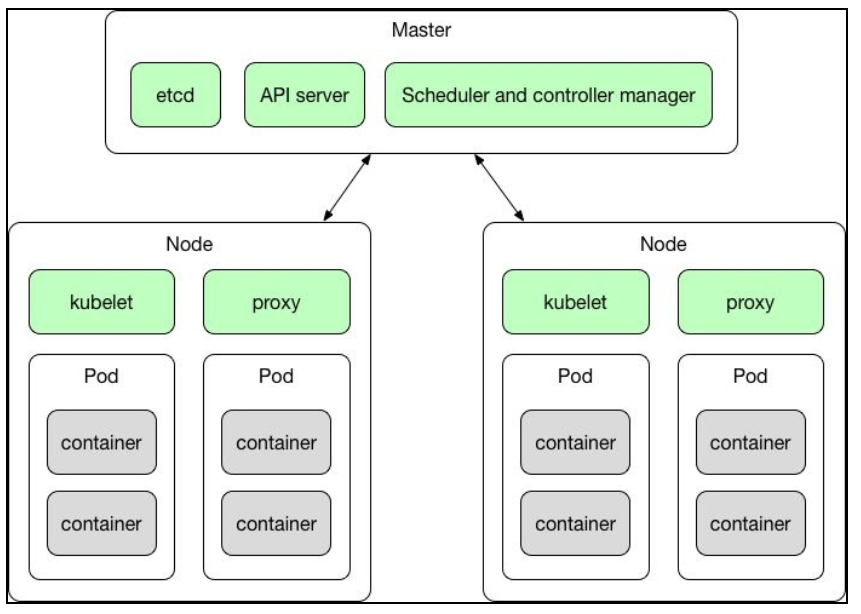
## Tại sao chúng ta cần kubernetes

Như ta đã biết, Docker container cung cấp một cách linh hoạt tuyệt vời cho việc chạy các service được đóng gói nhỏ , các thành phần độc lập của phần mềm. Docker container làm cho các thành phần của ứng dụng trở nên portable- bạn có thể di chuyển các thành phần service dọc các môi trường mà k cần lo lắng về sự phụ thuộc hay OS. Chừng nào OS có thể chạy Docker engine, ứng dụng của bạn có thể chạy trên hệ thống đó. Docker container sử dụng các tài nguyên của hệ điều hành máy host. Ứng dụng của bạn có thể chứa rất nhiều microservice độc lập, số lượng service có thể tang lên theo thời gian. Ngoài ra, nếu ứng dụng của bạn bắt đầu với lượng tải cao, thì bạn cần tăng số lượng container của service đó lên, để điều phối tải. Nó không có nghĩa là bạn chỉ cần sử dụng cở sở hạ tầng server của bạn – các container có thể được đẩy lên cloud. Ngày nay chúng ta có rất nhiều nhà cung cấp, như là Google, Amazon. Bằng khả năng chạy các container trên cloud, nó sẽ có rất nhiều lợi thế. Đầu tiên, bạn không cần quản lý server của bạn. Thứ hai, bạn chỉ cần trả cho nhữn gì bạn dung. Nếu có xảy ra quá tải, giá của dịch vụ cloud sẽ tăng. Nhưng nếu không có tải, bạn sẽ không cần trả gì. Phần mềm quản lý và giám sát container có thể làm giảm đáng kể chi phí phần cứng bằng cách sử dụng tối ưu nhất các thành phần. Kubernetes xử lý việc lập kế hoạch trên các node ở trong một cluster máy tính và quản lý workloads để đảm bảo rằng trạng thái của chúng phù hợp với các khai báo của người dùng. Sử dụng khái niệm về label và pod, Kubernetes nhóm các container của ứng dụng vào các đơn vị logic để dễ dàng quản lý.

Ứng dụng của bạn dưới dạng một tập hợp các container chạy trong môi trường được quản lý cũng làm thay đổi góc nhìn của việc phát triển phần mềm. Bạn có thể làm việc trê nmột phiên bản mới của service và khi nó sẵn sàng, bạn có thể update service đó. Nó cũng có nghĩa là các đội phát triển hoạt động linh hoạt hơn, nhỏ hơn và cá module có thể quản lý. Nó cho phép việc phát triển phần mềm thật sự nhanh. Microservice nhỏ và độc lập, và thời gian build và deploy thấp hơn đáng kể. Ngoài ra, bạn có thể release các thay đổi nhỏ dễ dàng hơn,,thường xuyên hơn, giảm thiểu khả năng xảy ra lỗi lớn.

## Kubernetes concept

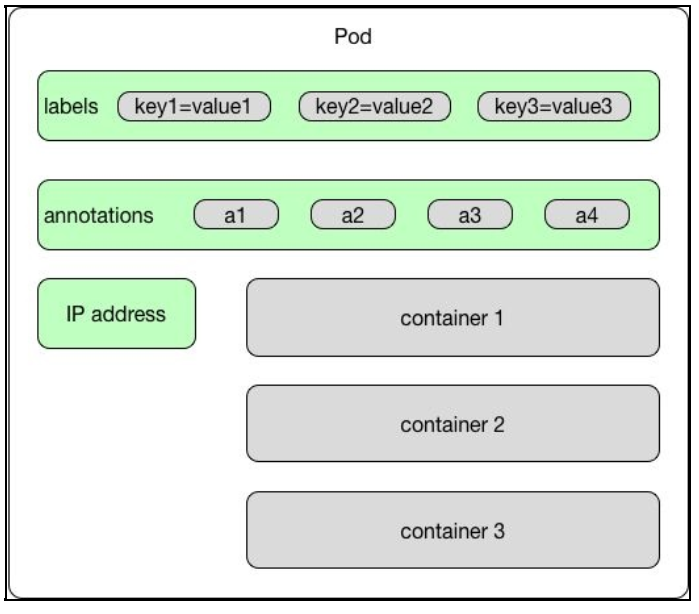
Cluster là một tập hợp các node, chúng có thể là các server vật lý hoặc các máy ảo được cài đặt Kubernetes platform. Kiến trúc Kubernetes như sau:



Kubernetes cluster bao gồm một Master node và các worker node với một vài thành phần trong đó.

### Pods

Pod chứa một hoặc một vài Docker container. Đây là đơn vị cơ bản của Kubernetes platform và là các mảnh thực thi và Kubernetes làm việc cùng.



Các container chạy trên cùng 1 Pod chia sẻ các thành phần chung : network namespace, disk, security context. Trên thực tế, việc giao tiếp thông qua localhost được đề nghị giữa các container đang chạy trên cùng Pod. Mỗi container có thể giao tiếp với bất cứ Pod nào hay service nào bên trong cluster.

Kubernetes hỗ trợ khái niệm về volume. Volume được gắn vào Pod có thể được mount vào một hoặc nhiều container đang chạy trong Pod đó. Kubernetes hỗ trợ rất nhiều kiểu volume: network disk, local hard drives…

Nếu ứng dụng của bạn cần distributed storage và cần xử lý một số lượng lớn dữ liệu, bạn sẽ không bị giới hạn ở local hard drives. Kubernetes hỗ trợ nhiều Volume Provider: GCE,AWS, GlusterFS,OpenStack Cinder, CephRBD,…

Ngoài ra, Pod còn có label và các annotation. Label rất quan trọng trong Kubernetes.Ý tưởng về label là chúng ta có thể sử dụng để định danh các object.

Ví dụ: app=booking-service

layer=backend

Thông qua label selector, client hay user có thể định danh một object hoặc tập object. Một selector, tương tự như một label, là một biểu thức key-value để định danh tài nguyên.Ví dụ với biểu thức selector app=booking-service sẽ chọn tất cả các pod với label app=booking-service. Annoation là một kiểu metadata bạn có thể gắn cùng Pod. Chúng là những thuộc tính có thể được đọcbởi các thư viện hoặc tool. Không có bất cứ quy tắc nào về việc annotation nên chứa.Annotation có thể chứa thông tin như là phiên bản build, release, thời gian, tên nhánh git, số hiệu git pull request, hoặc chỉ đơn giản là số điện thoại.

Label được sử dụng cho việc xác định thông tin về các object Kubernetes như là Pod. Annotation chỉ đơn giản là các metadata được gắn với đối tượng.

Định nghĩa của Pod là 1 file JSON hoặc YAML được gọi là Pod manifest. Ví dụ:

apiVersion: v1

kind: Pod

metadata:

name: rest\_service

spec:

containers:

name: rest\_service

image: rest\_service

ports:

containerPort: 8080

Có một chú ý là life của Pod rất mong manh. Bởi vì Pod là stateless, đơn vị độc lập, nếu một trong số chúng unhealthy hoặc là bị thay thế bởi version mới, Kubernetes Master sẽ hủy nó.

Lifecyle của Pod bao gồm các phase:

* + Pending:Pod đã được chấp nhận bởi hệ thống Kubernetes, nhưng có một hoặc một vài Docker container image chưa được tạo. Pod sẽ dừng ở trạng thái này một khoảng thời gian nếu có image cần tải xuống từ internet chẳng hạn.
  + Running:Pod đã được đặt trên node và tất cả các container trên Pod đã được tạo
  + Succeeded:Tất cả các container trên Pod đã bị kết thúc với trạng thái kết thúc thành công
  + Failed: Tất cả các container trên Pod đã kết thúc, nhưng có ít nhất 1 container kết thúc với lỗi hoặc bị kết thúc bởi hệ thống.
  + Unknown: Trạng thái này cho biết có một vấn đề khi giao tiếp từ Pod với host. Vì lý do nào đó, trạng thái của Pod không thể xác định.

Thường khi lượng tải tăng, chugsn ta cần tang nhiều Pod chạy, khi tải giảm, ta sẽ dispose bớt chúng đi. Tất nhiên, chúng ta có thể làm đó bằng tay, nhưng tốt nhất nên để tự động. Đó chính là về khái niệm ReplicaSets.

### ReplicaSets

Khái niệm này đươcj sử dụng trong việc scale ứng dụng bằng cách sử dụng replication. Thông thường ta muốn tái tạo các container cho một vài lý do:

* + Scaling: Khi tải tang và trở nên áp lực với các instances hiện tại, Kubernetes cho phép bạn dễ dàng scale ứng dụng , bằng các tạo them các instance cần thiết.
  + Load balancing: Chúng ta có thể dễ dàng phân phối tải tới các instance khác nhau để tránh việc quá tải trên 1 instance hoặc 1 node.
  + Reliability and fault tolerance:Bằng việc có nhiều phiên bản của ứng dụng, bạn có thể tránh khỏi các vấn đề nếu một hoặc một vài thành phần bị lỗi.

Replication thích hợp cho nhiều trường hợp sử dụng, bao gồm các ứng dụng dựa trên microservice, các service độc lập cung cấp các chức năng cụ thể, hay các ứng dụng cloud mà các thành phần có thể lỗi bất cứ khi nào. Replication là một giải pháp hoàn hảo cho việc thực thi chúng, như là nhiều instance tự nhiên vừa với kiến trúc.

Một ReplicaSet đảm bảo rằng số lượng các Pod clone, được gọi là replica đang chạy tại bất cứ thời điểm nào.Nếu có quá nhiều, chúng sẽ bị tắt bớt. Nếu cần thêm như trong trường hợp một số bị lỗi, hoặc là do tải lớn, một số Pod sẽ được clone và đẩy lên. ReplicaSets được sử dụng bởi Deployment.

### Deployment

Deployment chịu trách nhiệm cho việc tạo và cập nhật các instance của ứng dụng. Một khi Deployment được tạo, Kubernetes Master lập lịch các instance ứng dụng tới cái node cụ thể trên cluster.Nó quản lý ReplicaSets khi Pod đang được tạo, xóa, cập nhật. Deployment cung cấp thông tin cập nhật tường thuật cho Pod và ReplicaSets. Deployment cho phép dễ dàng cập nhật một Replica Set cũng như khả năng roll back lại deployment trước đó.

Bạn có thể cụ thể số lượng replicas bạn cần và container để chạy bên trong mỗi Pod. Ex:

apiVersion: 1.0

kind: Deployment

metadata:

name: rest\_service-deployment

spec:

replicas: 3

template:

metadata:

labels:

app: rest\_service

spec:

containers:

- name: rest\_service

image: rest\_service

ports:

- containerPort: 8080

Trên ví dụ, Deployment Controller sẽ tạo ra một ReplicaSet chứa 3 pod.

Deployment là một dạng cấu trúc điều khiển sẽ đảm nhiệm việc tăng giảm Pod. Deployment chịu trách nhiệm về trạng thái của Pod hoặc nhóm các Pod bằng cách tạo hoặc tắt replica. Deployment cũng quản lý cập nhật Pod. Deployment tạo ra các tài nguyên ReplicaSets. ReplicaSets đảm bảo việc số lượng replica luôn luôn chạy. Khi bạn muốn cập nhật một Pod, bạn cần thay đổi Deployment manifest. Việc thay đổi này sẽ tạo ra một ReplicaSet mới, nó sẽ được nhân rộng trong khi ReplicaSet trước đó sẽ thu nhỏ lại, do đó sẽ không tốn thời gian deployment lại ứng dụng.

Mục đích chính của Deployment là việc rolling update và rollback. Rolling update là quá trình cập nhật ứng dụng lên phiên bản mới. Bằng việc cập nhật một instance tại một thời điểm, bạn có thể giữ cho ứng dụng chạy. Nếu bạn chỉ cập nhật tất cả instance tại cùng 1 thời điểm, ứng dụng của bạn có thể sẽ ngừng một chút. Bên cạnh đó, việc thực thi một rolling update cho phép bạn tìm ra các lỗi trong quá trình thực thi do đó bạn có thể roll back lại trước đó nếu nó ảnh hưởng tới các user của bạn.

Deployment cho phép ta dễ dàng rollback. Để rollback, chúng ta đơn giản là đặt lại phiên bản chúng ta muốn trở lại. Kubernetes sẽ mở rộng quy mô các ReplicaSet tương ứng và giảm quy mô hiện tại, và kết quả của quá trình này là sẽ rollback về một phiên bản trước đó của service.

### Services

Kubernetes service nhóm một hay nhiều Pod vào một tiến trình internal hoặc tiến trình external cần chạy lâu dài và dễ dàng tiếp cận như là API, database host. Một Serivce tìm các Pod để nhóm chúng bằng cách tìm kiếm một label cụ thể. Chúng ta sử dụng label selector để lựa chọn các Pod với label đặc biệt và áp dụng các dịch vụ hoặc ReplicaSet lên chúng. Các ứng dụng các có thể tìm dịch vụ thông qua Kubernestes service discovery.

Service cung cấp một kết nối mạng tới một hoặc một vài Pod. Theo mặc định, Docker sử dụng mạng host-private, các container có thể giao tiếp với nhau trừ khi chúng chạy trên cùng một máy host. Trên Kubernetes, cluster Pod có thể giao tiếp với Pod khác nhờ service. Mỗi service đưa ra đại chỉ PI của nó và port như là hằng số trong suốt thời gian sống của service. Service có thể được tích hợp cân bằng tải mà sẽ phân phối lưu lượng tới tất cả các Pod. Trong khi life của Pod có thể mong manh, service là một khái niệm hằng hơn.

Mỗi Pod sẽ có được địa chỉ IP riêng của mình, nhưng khi nó chết và một Pod khác được đưa lên, địa chỉ IP có thể khác đi. Điều này gây ra một vấn đề tiềm ẩn: Nếu tập các Pod cung cấp chứng năng tới Pod khác bên trong Kubernetes cluster, ta có thể mất theo dõi của địa chỉ IP của cái khác. Service giải quyết được vấn đề này. Service cho phép decoupling. Giả sử dụng ta có một service Java Rest chạy trên Spring boot. Chúng ta cần một cách để định tuyến HTTP request từ internet tới container. Chúng ta thực hiện điều đó bằng cách thiết lập Kubernetes service sử dụng cân bằng tải để định tuyến các request tới từ địa chỉ IP bên ngoài tới một trong các container. Chúng ta sẽ nhóm các container có Rest service vào một Pod và đặt tên cho nó, Sau đó chúng ta sẽ định nghĩa một Kubernetes service sẽ phục vụ trên port 8080 với bất kì container trong các container ở bên trong Pod kia của chúng ta. Kubernetes sẽ sử dụng cân bằng tải để phân chia lưu lượng giưaũ các container cụ thể.Tóm lại, các tính năng của Kubernetes service là:

* Persistent và permanent.
* Cung cấp discovery.
* Cân bằng tải
* Expose đại chỉ IP mạng ổn định.
* Group Pod bằng label

Kubernetes hỗ trợ 2 chế độ tìm kiếm của service: các biến môi trường và DNS.Service discovery là quá trình tìm ra cách để kết nối tới một service. Kubernetes bao gồm một bult-in DNS server cho mục đích đó: kube-dns.

### Namespace

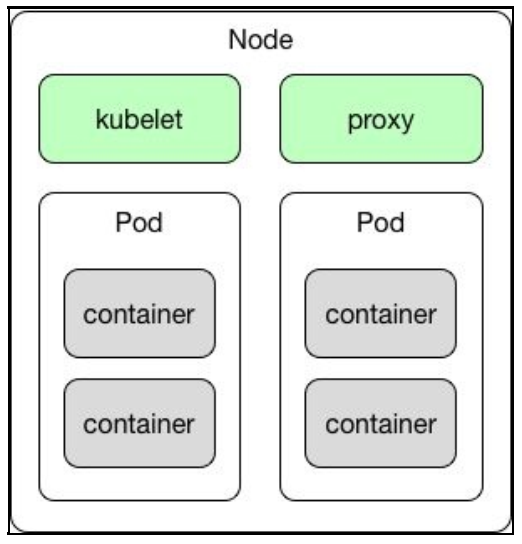
Chức năng namespace như là một cơ chế nhóm bên tỏng Kubernetes. Pod, volume, ReplicaSet và các service có thể dễ dàng hợp tác bên trong namespace, nhưng namespace cung cấp một sự cô lập khởi thành phần khác của cluster.Trường hợp nào ta sẽ sử dụng ? Namespace cho phép bạn quản lý các môi trường khác nhau bên trong cluster. Ví dụ, bạn có thể có môi trường test và môi trường product trên cùng một cluster.

Điều này có thể giúp tiết kiệm tài nguyên bên trong kiến trúc của bnạ, nhưng cũng có thể nguy hiểm; nếu không có namespace, sẽ nguy hiểm nếu ta tung ra một phiên bản mới của ứng dụng tới môi trường test, có một phiên bản trước khi release chạy trên cùng 1 cluster. Nhờ namespace, chúng ta có thể hoạt động trên các môi trường khác nhau , cùng một cluster mà không sợ ảnh hưởng tới các môi trường khác.

Kubernetes sử dụng default namespace,việc sử dụng có thế có hoặc không, nhưng chúng tôi khuyên dùng.

### Nodes

Một node chính là một woker trong kiến trúc Kubernetes. Nó có thể là một máy ảo hoặc máy vật lý, phụ thuộc kiến trúc của bạn. Một woker node chạy các nhiệm vụ được giao từ Master node. Node có thể chạy một hoặc nhiều Pod. Chúng cung cấp một host ảo ứng dụng cụ thể trong môi trường containerized.



#### Kubelet

Là thành phần điều khiển quan trọng. Đây là tiến trình chịu trách nhiệm với các lệnh từ Master node. Mỗi node sẽ có tiến trình lắng nghe. Master gọi nó để quản lý Pod và các container. Kubelet chạy các Pod. Kubelet chịu trách nhiệm về những gì đang chạy trên một máy riêng biệt và nó có một công việc: Cung cấp một tập hợp các container để chạy, đảm bảo tất cả đều chạy. Kubelet cũng có một internal HTTP server lắng nghe các HTTP request và phản hồi một API call để submit một manifest mới.

#### Proxy

Là network proxy, nó tạo ra một địa chỉ IP ảo để các client có thể truy cập. Service cung cấp cách để nhom các Pod vào một kiểu gọi là tiến trình kinh doanh đơn, chúng có thể đạt được thông qua chính sách truy cập chung. Bằng việc có poxy chạy trong node, chúng ta có thể gọi địa chỉ IP service. Nói theo hướng công nghệ, proxy ở đây là tiến trình kube-proxy, xử lý qua các luật trên iptables để bắt csac truy cập tới đại chỉ IP service. Kubernetes network proxy chạy trên mỗi node. Nếu không có nó, chúng ta sẽ không truy cập được service.

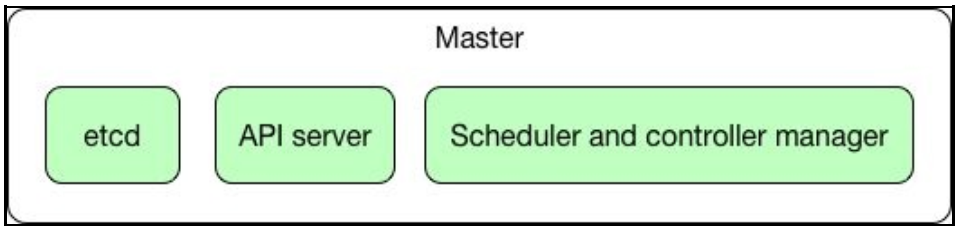
### Docker

Cuối cùng, để mỗi node có thể chạy được là cần có Docker container runtime. Nó chịu trách nhiệm cho việc kéo image về và chạy các container.

Tất cả các node, và bất cứ nhóm của các worker trên thế giới, cần một người quản lý. Trong Kubernetes, vai trò của node quản lý là thực thi bởi một node đặc biệt. Master node.

### Master node

Master node không chạy bất kì container nào, nó chỉ xử lý và quản lý cluster. Master là trung tâm điều khiển cung cấp một cái nhìn thống nhất của cluster. Có một Master node điều khiển các worker node. Master node tự động xử lý việc lập lịch của Pod dọc các worker node trong cluster-bằng cách.Cấu trúc của Master node như sau:



#### Etcd

Chứa các trạng thái của cluster. Etcd là một distributed, dạng key-value cho các dữ liệu quan trọng trong hệ thống phân tán, với trọng tâm vào

* + Simple:Dễ định nghĩa, API
  + Secure: tự động TLS với xác thực client
  + Fast: Lên tới 10 000 writes/sec
  + Reliable: Phân phối đúng cách sử dụng Raft.

Trạng thái đó bao gồm nhữn node nào tồn tại trên cluster, Pod nào nên chạy, node nào đạng chạy, và nhiều thông tin khác nữa. Toàn bộ trạng thái cluster được lưu trong một instance của etcd. Nó cung cấp một cách để lưu trữ các dữ liệu cấu hình đáng tin cậy. Một thành phần quan trọng khác chạy trên Master node chính là API server.

#### API server

Thành phần này là một thành phần quan trọng của Master node. Tên của tiến trình này là kube-apiserver, sẽ xử lý và phản hồi các HTTP REST request sử dụng JSON. Mục đích chính là việc validate và cấu hình dữ liệu cho các đối tượng API là Pod, service, ReplicaSet,… APi server cung cấp một giao diện để chia sẻ trạng thái cluster thông qua đó tất cả các thành phần khác tương tác với nhau. API server là thực thể quản lý trung tâm và là thành phần duy nhất kết nối tới etcd. Các thành phần khách phải thông qua API server đẻ làm việc với trạng thái cluster.

#### Scheduler and controller manager

Như ta đã biết, khi tạo ra một Deployment, Master sẽ lập lịch việc phân phối các instance ứng dụng vào các node cụ thể trên cluster. Một khi các instance đã chạy, Deployment Controller sẽ tiếp tục giám sát chúng. Đây là một cơ chế tự kiểm tra sức khỏe. Nếu một node bị xóa hoặc có vấn đề, Deployment Controller sẽ thay thế nó.

## Một số tool

Chúng ta có thể sử dụng một số tool hỗ trợ: kubectl,dashboard,minikube