Docker document

Content

Phần 1: Tổng quan

I. Tổng quan

Docker là 1 platform mở dành cho developing, shipping, và running các ứng dụng. cho phép tách ứng dụng khỏi hạ tầng để có thể triển khai phần mềm nhanh chóng. (tách biệt ở đây nên hiểu là khi app được đóng gói vào 1 container thì ta có thể mang nó đi và chạy trên các host khác mà không cần phải cài đặt thêm các env hay các phần mềm khác để chạy được ứng dụng). với docker ta có thể quản lý hạ tầng như quản lý ứng dụng (PaaS), tức là phần hạ tầng đã được đóng gói luôn vào docker nên việc quản lý docker chính là quản lý hạ tầng. và việc tận dụng ưu điểm shipping, testing, deploying code nhanh chóng sẽ giảm được cái thời gian giữa viết code và triển khai app lên môi trường product.

1. Docker platform

- Docker platform cung cấp khả năng đóng gói và run các ứng dụng trong 1 môi trường riêng được gọi là container. cho phép chạy nhiều container trên 1 host. các container rất nhẹ và có thể chạy trực tiếp trong kernel của host mà không cần thêm bất kỳ phần mềm hay thư viện bổ sung nào khác (đây là lý do có thể chạy nhiều container trên cùng 1 host).

2. Docker engine

Docker engine là 1 ứng dụng client-server với 3 thành phần chính:

- server: Là 1 chương trình chạy liên tục, chịu trách nhiệm nhận và phản hồi các request từ client. còn được gọi là Docker daemon.
- REST API: Là giao diện để giao tiếp giữa client và daemon
- CLI client: CLI sử dụng REST API để điều khiển và tương tác với Docker daemon thông qua scripts hoặc các lệnh CLI trực tiếp.

3. Sử dụng docker cho việc gì?

a. Delivery các ứng dụng nhanh và liên tục

docker tổ chức vòng đời development bằng cách cho phép các developer làm việc trong 1 env chuẩn sử dụng local container cho các ứng dụng và service của mình. Các container rất phù hợp cho CI/CD workflow.

xét kịch bản sau:

- các developer viết code ở local và shared công việc với team sử dụng docker container.
- họ sử dụng docker container để push ứng dụng vào trong môi trường test và thực hiện test tự động và test thủ công
- khi developer nhìn thấy bugs, họ có thể fix chúng trong môi trường development và redeploy chúng tới môi trường test để testing.
- khi quá trình test hoàn thành, đưa bản đã fix lỗi tới khách hàng đơn giản bằng cách push image đã được updated vào môi trường production.

b. Đáp ứng triển khai và mở rộng

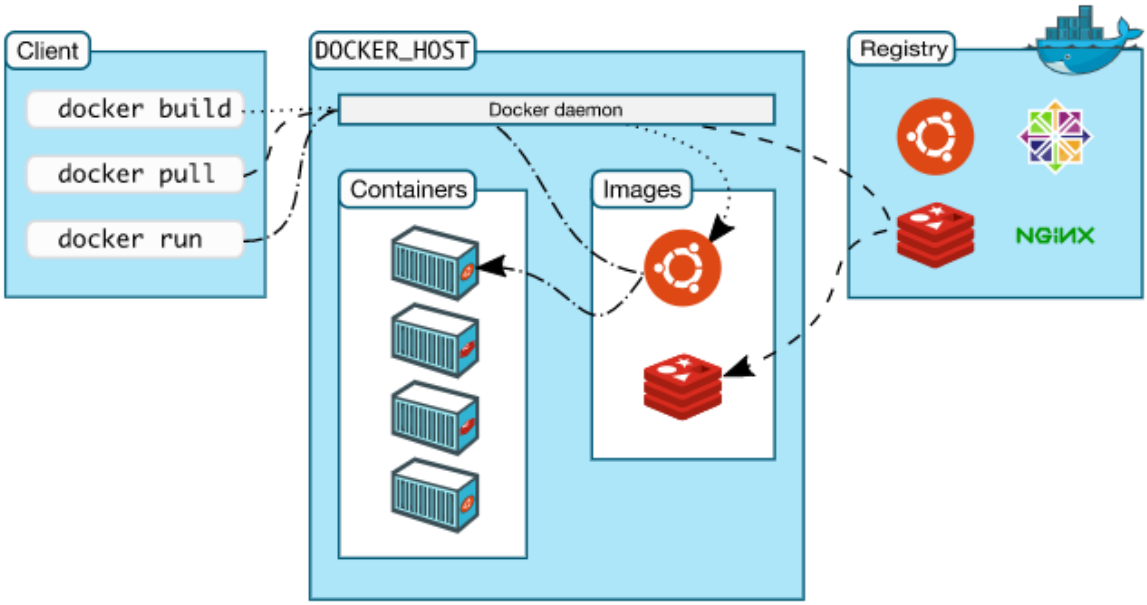
Nó cung cấp sự linh động cao. container có thể chạy trên máy tính cá nhân của developer, các server ảo hoặc server vật lý trong datacenter hoặc cloud hoặc cả 2. sự linh động và nhẹ của docker cho phép quản lý linh động workloads, mở rộng hoặc xóa bỏ các ứng dụng và các service.

c. chạy nhiều ứng dụng trên cùng 1 phần cứng

- docker nhẹ và nhanh, nó cung cấp khả năng tồn tại và hiệu quả về chi phí hơn so với các máy ảo sử dụng công nghệ ảo hóa.
- docker khá lý tưởng khi sử dụng trong các môi trường có mật độ triển khai cao, triển khai các ứng dụng vừa và nhỏ, những nơi cần thực hiện cài đặt nhiều tài nguyên.

II. Kiến trúc docker

- docker sử dụng kiến trúc client-server. docker client giao tiếp với docker daemon để building, running và distributing các docker container. Docker client và docker daemon có thể chạy trên cùng hệ thống hoặc có thể kết nối docker client tới remote docker daemon. Client và daemon giao tiếp với nhau sử dụng REST API, Unix socket hoặc network interface.



a. Docker daemon

Docker Daemon lắng nghe các request docker API và quản lý các Docker Object như là container, image, network, volume. daemon cũng có thể giao tiếp với các daemon khác để quản lý các Docker service.

b. Docker client

Docker client là cách cơ bản nhất mà nhiều người dùng docker sử dụng để tương tác với docker. Khi ta sử dụng lệnh như docker run, client sẽ gửi lệnh này tới dockerd, dockerd sẽ thực thi câu lệnh này. Lệnh docker sử dụng docker API. Docker client có thể giao tiếp với nhiều daemon.

c. Docker registry

Là nơi lưu trữ các image. Docker hub là 1 public registry mà bất kỳ ai cũng có thể sử dụng và mặc định docker được cấu hình để tìm kiếm các image từ docker hub. Ta có thể sử dụng docker registry của riêng mình. Nếu sử dụng Docker Datacenter, nó bao gồm cả Docker trusted

registry. Khi sử dụng lệnh docker pull hoặc docker run, image sẽ được pull từ registry mà ta cấu hình và khi sử dụng lệnh docker push, image sẽ được push lên registry được cấu hình.

d. Docker Object

Khi ta sử dụng docker, ta sẽ tạo và sử dụng các image, container, network, volume, plugins và các object khác. Sau đây sẽ mô tả tổng quan về các Object này:

- **Image**: là 1 template read-only chứa các instructions để tạo 1 Docker container. Thường thì 1 image sẽ được xây dựng dựa trên 1 image khác. Ta có thể tạo image của riêng mình, và chỉ có thể sử dụng image của người khác khi nó được public trên registry. Để build một image ta tạo 1 Dockerfile và định nghĩa các bước cần thiết để tạo image và run nó. Mỗi một instructions trong Dockerfile tạo ra 1 layer trong image. Khi mình thay đổi Dockerfile và build lại image, chỉ những layer nào bị thay đổi sẽ được rebuild. đây là 1 phần lý do khiến cho image nhỏ, nhẹ và nhanh khi so sánh với các công nghệ ảo hóa.
- **Container**: 1 container là 1 instance có khả năng chạy được của 1 image. Ta có thể create, start, stop, delete hoặc move 1 container sử dụng API hoặc CLI. có thể connect 1 container tới 1 hoặc nhiều network, gắn store vào container hoặc tạo 1 image mới dựa vào trạng thái image hiện tại. Mặc định các container tạo ra hoàn toàn độc lập với nhau và độc lập với host machine. ta có thể quản lý cách network, store và subsystem của container độc lập với các container khác hoặc với các host machine khác.

Ví dụ với lệnh docker run: $ docker run -i -t ubuntu /bin/bash

Khi chạy lệnh trên thì quá trình xảy ra như sau (giả sử registry cấu hình mặc định):

+ Docker sẽ tìm image ở local, Nếu image không có ở local, docker sẽ pull image từ registry được cấu hình như khi ta chạy lệnh docker pull ubuntu trên cli.
+ Docker tạo 1 container mới như khi chạy câu lệnh docker container create thủ công.
+ docker Cấp phát filesystem read-only cho container như là layer cuối cùng, việc này cho phép 1 container đang running tạo hoặc chỉnh sửa các file, thư mục trong local filesystem của nó.
+ Docker tạo 1 interface network để kết nối docker với network mặc định của nó khi ta không xác định options network nào khi running container. Việc tạo interface bao gồm cả gán địa chỉ IP cho container. Mặc định container có thể kết nối tới mạng bên ngoài sử dụng kết nối mạng của host.
+ Docker start container và thực thi /bin/bash. bởi vì container đang running ở mode trường tác và được gắn với terminal qua 2 flag (-t và -i). Ta có thể cung cấp đầu vào sử dụng keyboard và đầu ra được logging trên terminal.
+ Khi ta gõ lệnh exit để kết thúc terminal, container sẽ stop nhưng không bị xóa đi và ta có thể restart lại khi cần sử dụng hoặc xóa nó đi.

- **Services**: Các service cho phép ta scale các container qua nhiều Docker daemon, tất cả các service đều làm việc với nhau như 1 swarm với nhiều manager và worker. Mỗi member của 1 swarm là 1 Docker daemon và tất cả các Daemon giao tiếp với nhau sử dụng API. 1 service cho phép ta định nghĩa trạng thái riêng như là số replicas của service phải available ở bất kỳ thời điểm nào. mặc định các service được cân bằng tải trên tất cả các node đang hoạt động. tóm lại là docker là 1 ứng dụng đơn lẻ, và Docker Engine hỗ trợ swarm mode trong docker 1.12 hoặc cao hơn.

III. Một số công nghệ cơ bản

- **NameSpace**: docker sử dụng công nghệ namespace để cung cấp workspace độc lập gọi là container. Khi run docker, docker tạo 1 tập namespace cho container đó. Các namespace cung cấp 1 layer độc lập. Mỗi thành phần của 1 container chạy trong 1 namespace riêng và việc truy cập bị giới hạn tới namespace đó. Docker engine sử dụng các namespace trên linux như sau:
  + **pid namespace**: Process isolation (PID: ID Process)
  + **net namespace:** quản lý network interface
  + **ipc namespace:** quản lý truy cập các tài nguyên ipc (IPC: interprocess communication).
  + **mnt namespace:** Quản lý mount point của filesystem (MNT: Mount)
  + **uts namespace**: tách biệt kernel và version identifier (UTS: unix timesharing system).
- **Control Group**: Docker Engine cũng dựa vào các công nghệ khác gọi là control groups (cgroups). 1 cgroups giới hạn 1 ứng dụng tới 1 số các tài nguyên cụ thể. Control group cho phép docker engine chia sẻ các tài nguyên phần cứng tới các container với các lựa chọn ràng buộc và có hạn chế. ví dụ có thể hạn chế memory với 1 container cụ thể.
- **Union file system**: Union file system (UnionFS) là các file hệ thống hoạt động bằng cách tạo các layer, tạo cho chúng nhẹ và nhanh. Docker engine sử dụng UnionFS để cung cấp các building block cho các container. Docker engine sử dụng nhiều biến UnionFS gồm AUFS, btrfs, vfs và Device mapper.
- **Container format:** Docker engine kết hợp namespace, control group và UnionFS trong 1 wrapper được gọi là docker format. container format mặc định là libcontainer.

Phần 2. Get starting

I. Cài đặt môi trường

1.1 Cài đặt docker

B1. Xóa docker phiên bản cũ đi
phiên bản cũ của docker có tên là docker-engine hay là docker. remove nó và các dependences của nó:

```
$ sudo yum remove docker \
    docker-client \
    docker-client-latest \
    docker-common \
    docker-latest \
    docker-latest-logrotate \
    docker-logrotate \
    docker-engine
```

Gói docker-engine hiện tại là docker-ce

- Cài đặt docker engine community bằng repo
  + Cài đặt các package yêu cầu yum-utils cung cấp công cụ yum-config-manager, evice-mapper-persistent-data và lvm2 được yêu cầu bởi devicemapper store driver.
    $ sudo yum install -y yum-utils device-mapper-persistent-data lvm2
  + dùng lệnh sau để setup repository
    $ sudo yum-config-manager --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo

Cài đặt phiên bản mới nhất của docker-engine community:

$ sudo yum -y install docker-ce docker-ce-cli containerd.io

  + Cài đặt với phiên bản xác định
    tìm kiếm các phiên bản docker-engine hiện có:
    $ yum list docker-ce --showduplicates | sort -r

    docker-ce.x86_64 3:18.09.1-3.el7 docker-ce-stable
    docker-ce.x86_64 3:18.09.0-3.el7 docker-ce-stable
    docker-ce.x86_64 18.06.1.ce-3.el7 docker-ce-stable
    docker-ce.x86_64 18.06.0.ce-3.el7 docker-ce-stable
    cài đặt bằng lệnh:
    $ sudo yum install docker-ce-<VERSION_STRING> /
    docker-ce-cli-<VERSION_STRING> containerd.io
  + Chạy docker
    $ sudo systemctl start docker
  + Kiểm tra hoạt động của docker bằng cách tạo 1 container hello-world
    docker run -d --name test hello-world

- Cấu hình sau cài đặt
- Quản lý docker bằng user non-root

mặc định docker daemon bind tới Unix socket thay vì TCP port. Unix socket mặc định được quản lý bởi user root, các user khác chỉ có thể truy cập thông qua sudo do đó docker daemon luôn luôn chạy bằng root user. Nếu không muốn sử dụng sudo thì tạo thêm group docker và add user vào group này. và khi docker daemon start, socket của nó có thể truy cập bằng user trong group docker.

Tạo group:

$ sudo groupadd docker

Add user:

$ sudo usermod -aG docker $USER

- Cấu hình docker start khi boot

$ sudo systemctl enable docker

- Sử dụng storage engine khác
- Cấu hình sử dụng default logging driver
- Cấu hình nơi docker daemon lắng nghe các connection.
- Cấu hình truy cập từ xa với systemd

+ sử dụng lênh sudo systemctl edit docker.service để mởi file docker.service và chỉnh sửa.
+ Thêm dòng sau và chỉnh sửa các giá trị mình muốn:

[Service]

ExecStart=
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://127.0.0.1:2375

+ save file và reaload systemctl.

$ sudo systemctl daemon-reload

+ restart docker

$ sudo systemctl restart docker.service

+ Check docker daemon hoạt động đúng như mình đã cấu hình

$ sudo netstat -lntp | grep dockerd
tcp 0 0 127.0.0.1:2375 0.0.0.0:* LISTEN

- cấu hình truy cập từ xa bằng daemon.json

Set hosts trong file /etc/docker/daemon.json như sau:

```
{
"hosts": ["unix:///var/run/docker.sock", "tcp://127.0.0.1:2375"]
}
```

Restart docker và check như trên.

1.2. Container

- Định nghĩa container bằng Dockerfile

Tạo 1 thư mục mới, cd vào thư mục mới tạo, tạo file gọi là Dockerfile, paste đoạn code sau vào file Dockerfile

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim
# Set the working directory to /app
WORKDIR /app
# Copy the current directory contents into the container at /app
COPY . /app
# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt
# Make port 80 available to the world outside this container
EXPOSE 80
# Define environment variable
ENV NAME World
# Run app.py when the container launches
CMD ["python", "app.py"]
```

- tạo 2 file requirements.txt và app.py cùng trong thư mục file Dockerfile với nội dung như sau:

requirements.txt: chứa 2 lib python cần cài đặt

```
Flask
Redis
```

app.py: app để run

```
from flask import Flask
from redis import Redis, RedisError
```

```
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2,
socket_timeout=2)

app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

    html = "<h3>Hello {name}!</h3>" \
           "<b>Hostname:</b> {hostname}<br/>" \
           "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"),
hostname=socket.gethostname(), visits=visits)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

- Build App

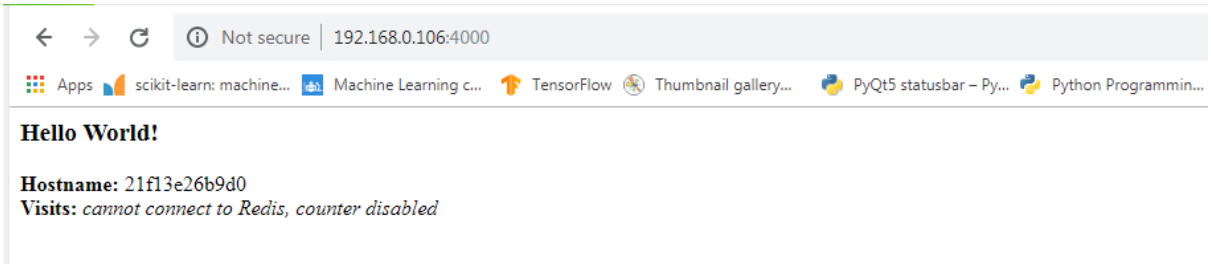
sử dụng lệnh:

```
docker build --tag=friendlyhello .
```

- Run app

```
docker run -p 4000:80 friendlyhello
```

truy cập web brower để kiểm tra



- chạy app ở mode background

```
docker run -d -p 4000:80 friendlyhello
```

- **Share Image**

Mục đích: upload image ta đã build lên registry để có thể sử dụng ở mọi nơi

B1. Login vào docker từ local machine

```
$ docker login
```

B2. Tag image

Ký hiệu để liên kết image ở local và repository trên registry là: `username/repository:tag .`

Tag là tùy chọn nhưng khuyến nghị nên có.

Giờ sử dụng lệnh `docker tag image với username, tag và repository để tag image:`

```
docker tag image username/repository:tag
```

B3. push image lên registry

Upload image đã tag lên repository lên registry:

```
docker push username/repository:tag
```

B4. kiểm tra bằng cách pull và run image từ remote repository

```
docker run -p 4000:80 username/repository:tag
```

1.3 Service

Chúng ta sẽ scale ứng dụng và load-balancing. Với các ứng dụng phân tán hay microservice ta có rất nhiều service. Các service này là các container trong môi trường production. Mỗi service chỉ chạy 1 image nhưng sẽ có các cấu hình khác nhau như là port sử dụng cho service, có bao nhiêu replicas của container nên run để service có được hiệu xuất nó cần, …. Scaling 1 service là thay đổi số instance container đang running 1 service, cấp phát thêm resource cho service. Để dễ dàng định nghĩa, run và scale các service ta sử dụng file là: docker-compose.yml

Save file này ở bất kỳ đâu bạn muốn, add nội dung sau vào file:

```
version: "3"
services:
  web:
    # replace username/repo:tag with your name and image details
    image: username/repo:tag
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
      restart_policy:
        condition: on-failure
    ports:
      - "4000:80"
    networks:
      - webnet
networks:
  webnet:
```

File này sẽ y/c docker thực hiện những công việc sau:

- Pull image mà ta đã upload lên registry từ trên
- chạy 5 instance của image đó như 1 service gọi là web, hạn chế sử dụng tài nguyên với từng instance (CPU, RAM)
- restart container ngay lập tức nếu có lỗi xảy ra
- Map port 4000 trên host với port 80 của web.

- Cài đặt để các container web chia sẻ port 80 thông qua load-balance network gọi là webnet.
- Định nghĩa webnet network với các thiết lập mặc định.

1.3.1 Run app load-balanced mới

trước khi chạy lệnh `docker stack deploy` ta chạy lệnh lệnh:

```
docker swarm init
```

bây giờ ta chạy câu lệnh trên, và đặt tên cho app

```
docker stack deploy -c docker-compose.yml getstartedlab
```

kiểm tra thấy 5 container đang running trên host

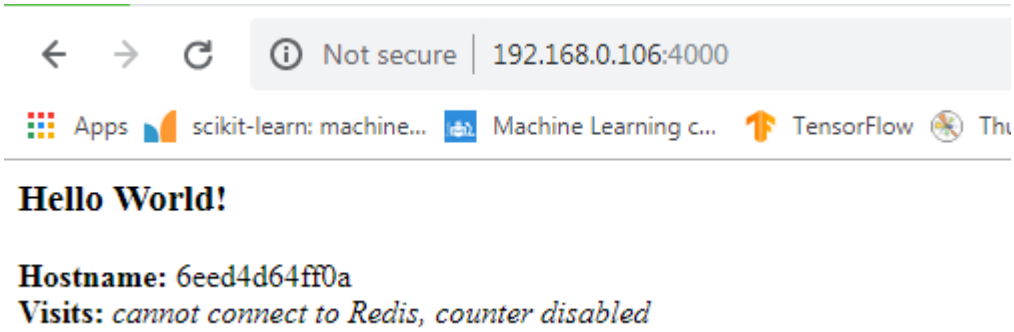
```
[root@LPI01 app_flask]# docker ps
CONTAINER ID        IMAGE                 COMMAND             CREATED             STATUS
 PORTS               NAMES
62d9e28bd168        huylt27/get-start:p2   "python app.py"     39 seconds ago      Up 33 seconds
 80/tcp              getstartedlab_web.1.k6uin86exn98lgax8wmegei5r
6eed4d64ff0a        huylt27/get-start:p2   "python app.py"     39 seconds ago      Up 33 seconds
 80/tcp              getstartedlab_web.2.hhpuodpasy4lwi7orpe2jqwpu
d790837407af        huylt27/get-start:p2   "python app.py"     39 seconds ago      Up 33 seconds
 80/tcp              getstartedlab_web.4.xmkkcxmfem9v42njt8hju6818
ec3e1b2f1d08        huylt27/get-start:p2   "python app.py"     39 seconds ago      Up 34 seconds
 80/tcp              getstartedlab_web.5.j10hc47hx6yp57fzl8q1tnnzp
16a983cad9bf        huylt27/get-start:p2   "python app.py"     39 seconds ago      Up 33 seconds
 80/tcp              getstartedlab_web.3.oksefy782hh5z75c4qzm1ulep
```

kiểm Service ID của 1 service trong ứng dụng của mình:

```
docker service ls
```

- Kiểm tra app đã hoạt động

trên trình duyệt gõ lệnh http://ip server:4000



1.3.2 scale app

ta scale app bằng cách thay đổi giá trị replicas trong file docker-compose.yml. save và rerun lại `docker stack deploy -c docker-compose.yml getstartedlab`

chạy lại câu lệnh `docker container ls -q` ta thấy các instance đã deployed đã được cấu hình lại. giảm replicas gọi là scale down, và tăng giá trị replicas gọi là scale up.

- off app và swarm

off app:

```
docker stack rm getstartedlab
```

off swarm:

```
docker swarm leave --force
```

1.4. Swarm

- **Introduce**

Phần này ta sẽ deploy ứng dụng ở phần 3 trong 1 cluster, chạy ứng dụng trên nhiều machine. các ứng dụng Multi-container, Multi-machine có thể được tạo ra bằng cách join nhiều machine vào trong cluster đã được container hóa được gọi là swarm.

- **Hiểu swarm cluster**

1 Swarm là 1 group các machine chạy docker và được join vào trong 1 cluster. sau khi các docker đang running join vào cluster, Trong 1 cluster các lệnh docker chỉ có thể thực thi trên swarm manager node. các machine trong 1 swarm là các service vật lý hay VPS, sau khi join cluster chúng được gọi 1 node.

Swarm manager có thể sử dụng nhiều cách để run container như là "empties node", tức là chạy container trên machine có ít container nhất. hoặc "Global", để đảm bảo mỗi machine chạy 1 instance của container cụ thể. Ta chỉ thị swarm manager sử dụng phương pháp này trong compose file.

Swarm manager là machine duy nhất có thể thực thi các lệnh , hoặc xác thực với các machine khác để join swarm như là các worker. các worker chỉ có thể cung cấp capacity và không có khả năng xác thực để nói với bất kỳ machine khác những gì nó có thể làm và không thể làm.

- **Setup swarm**

1 swarm được sinh ra từ nhiều node, các node này tốt nhất là server vật lý hoặc các máy ảo. chạy lệnh `docker swarm init` để enable swarm mode và biến machine này thành swarm manager, và sau đó chạy lệnh `docker swarm join` để join các machine khác để chúng join swarm như các worker.

+ Tạo cluster: Tạo thêm 2 VM, cấp IP cho 2 VM mới. Cài đặt docker và start docker
+ Khởi tạo swarm và add node: machine đầu tiên mà swarm manager chúng thực thi các lệnh quản lý và xác thực worker để join swarm. Machine thứ 2 join vào swarm được gọi là worker.

```
docker swarm init --advertise-addr <myvm1 ip>"
Swarm initialized: current node <node ID> is now a manager.

To add a worker to this swarm, run the following command:

  docker swarm join \
  --token <token> \
  <myvm ip>:<port>

To add a manager to this swarm, run 'docker swarm join-token manager'
and follow the instructions.
```

sử dụng lệnh : `sudo docker node ls` để check node trong swarm

chạy lệnh `docker swarm leave` để bỏ node ra khỏi swarm.

- Deploy app trên swarm cluster: Bây giờ ta có thể sử dụng các lệnh như ở mục 3 để deploy swarm mới. nên nhớ chỉ swarm manager mới thực thi các lệnh docker.
- Deploy App trên swarm manager:

chạy lệnh `docker stack deploy -c docker-compose.yml getstartedlab` để deploy app trên myvm1 như phần 3.

Note: Nếu image được lưu trữ ở private registry mà không phải docker hub, ta cần logged in sử dụng lệnh docker login <your-repository> và sau đó add --with-registry-auth flag vào lệnh trên: Ví dụ:

```
docker login registry.example.com
```

```
docker stack deploy --with-registry-auth -c docker-compose.yml getstartedlab
```

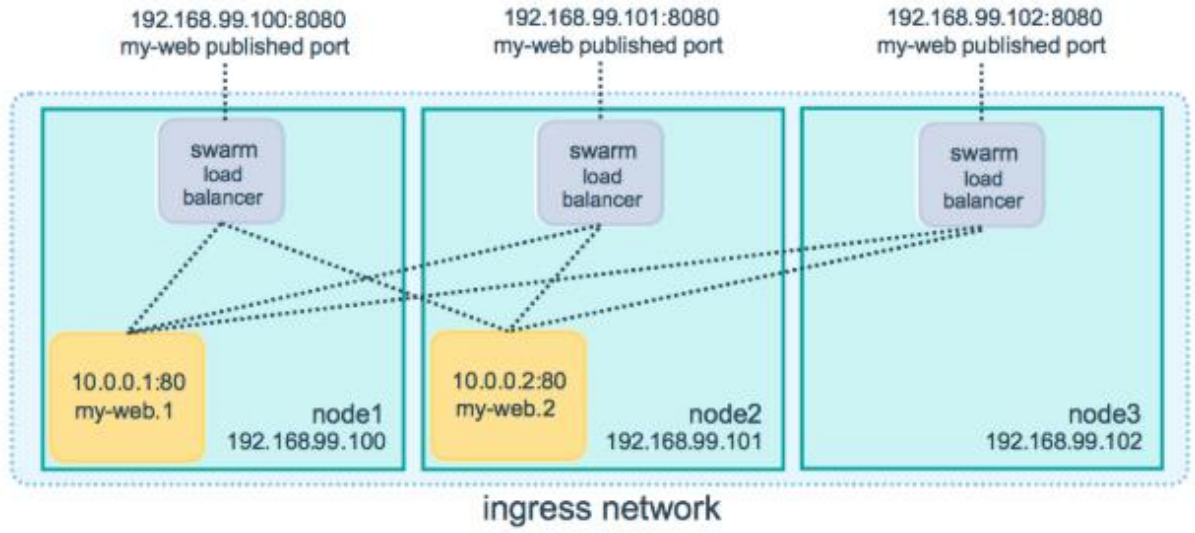
Bây giờ ta có thể sử dụng các lệnh docker như phần 3. Chỉ có 1 điều cần chú ý là các service và các container liên quan được phân tán trên cả myvm1 và myvm2.

```
$ docker stack ps getstartedlab
```

```
ID             NAME                  IMAGE                    NODE    DESIRED
STATE
jq2g3qp8nzwx   getstartedlab_web.1   gordon/get-started:part2  myvm1   Running
88wgshobzoxl   getstartedlab_web.2   gordon/get-started:part2  myvm2   Running
vbb1qbkb0o2z   getstartedlab_web.3   gordon/get-started:part2  myvm2   Running
ghii74p9budx   getstartedlab_web.4   gordon/get-started:part2  myvm1   Running
0prmarhavs87   getstartedlab_web.5   gordon/get-started:part2  myvm2   Running
```

- Truy cập cluster: ta có thể truy cập vào app sử dụng địa chỉ IP của myvm1 hoặc myvm2. network mà ta tạo ra và chia sẻ giữa các node và load-balance

dưới đây là sơ đồ network của swarm cluster:



Note: chú ý khi sử dụng ingress network trong swarm cần mở các port giữa các node trong swarm trước khi enable swarm mode.

  + Port 7946 TCP/UDP for container network discovery.
  + Port 4789 UDP for the container ingress network.

- Interating và scaling APP:

Từ đây ta có thể làm mọi thứ như phần 2, 3.

- Clean up và reboot

+ Stack và swarms: ta có thể xóa stack bằng lệnh docker stack rm:

```
docker stack rm getstartedlab
```

Remove swarm bằng lệnh `docker swarm leave` trên worker.

Remove swarm bằng lệnh: `docker swarm leave --force` trên manager.

1.4.1 Stack

Phần này chúng ta sẽ tìm hiểu stack, stack là 1 group các service có liên quan với nhau chia sẻ các dependence, có thể được sắp đặt (bố trí) và scaled với nhau. 1 Single stack có khả năng xác định hoặc kết hợp các chức năng của toàn bộ ứng dụng.

Chúng ta đã làm việc với stack trong phần 3 khi tạo compose file và chạy lệnh `docker stack deploy` . tuy nhiên đây là 1 single stack chạy trên single host và kịch bản này thường sử dụng trên môi trường production. phần này ta sẽ sử dụng những gì đã biết tạo nhiều service liên quan với nhau và chạy chúng trên nhiều machine.

a. Thêm 1 service mới và redeploy.

Mở file docker-compose thay thế nội dung sau: chú ý thay thế username/repo:tag

```
version: "3"
services:
  web:
    # replace username/repo:tag with your name and image details
    image: username/repo:tag
    deploy:
      replicas: 5
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
    ports:
      - "80:80"
    networks:
      - webnet
  visualizer:
    image: dockersamples/visualizer:stable
    ports:
      - "8080:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      placement:
        constraints: [node.role == manager]
    networks:
      - webnet
networks:
  webnet:
```

Có 1 service web mới là visualizer và 2 key mới: volume key cho phép docker truy cập file socket của host và placement key đảm bảo service này chỉ run trên swarm manager mà không chạy trên worker.
chạy lại câu lệnh sau trên manager để update service

```
$ docker stack deploy -c docker-compose.yml getstartedlab
```

Nhìn vào virtualizer: ta nhìn thấy trong file docker-compose, visualizer chạy ở port 8080, lấy địa chỉ IP là địa chỉ IP của 1 trong các node cluster. để thấy visualizer running, vào brower gõ địa chỉ http://ip server:8080. Ta thấy virtualizer đang running trên swarm manager, còn 5 instance web được phân bố trên 3 node của swarm. ta có thể kiểm tra việc ảo hóa này bằng lệnh: `docker stack ps <stack>`:

```
docker stack ps getstartedlab
```

Virtualizer là 1 service độc lập có thể chạy trên bất kỳ ứng dụng nào chứa nó trong stack, nó không phụ thuộc vào bất kỳ thứ gì.

b. Persist data

Bây giờ ta add Redis database để lưu trữ data của app sử dụng file docker-compose sau:

```
version: "3"
services:
  web:
    # replace username/repo:tag with your name and image details
    image: username/repo:tag
    deploy:
      replicas: 5
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
    ports:
      - "80:80"
    networks:
      - webnet
  visualizer:
    image: dockersamples/visualizer:stable
    ports:
      - "8080:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      placement:
        constraints: [node.role == manager]
    networks:
      - webnet
  redis:
    image: redis
    ports:
      - "6379:6379"
    volumes:
```

```
      - "/home/docker/data:/data"
    deploy:
      placement:
        constraints: [node.role == manager]
    command: redis-server --appendonly yes
    networks:
      - webnet
networks:
  webnet:
```

Redis là 1 image trong thư viện của docker có tên là redis, chạy trên port 6379 và được cấu hình mởi port từ container đến host.

có 2 thông số quan trọng nhất để cho data không đổi giữa các deployment của stack này:

+ redis luôn chạy trên manager, vì vậy nó luôn sử dụng cùng filesystem.
+ redis truy cập thư mục con trong filesystem của host là /data trong container, đây là nơi redis lưu trữ dữ liệu.

điều này sẽ tạo ra 1 nguồn tin cậy trong filesystem vật lý của host cho dữ liệu của redis. Nếu không có điều này , redis sẽ lưu trữ dữ liệu trong thư mục /data trong filesystem của container, điều này sẽ khiến cho data bị xóa khi redeploy.

Nguồn tin cậy này có 2 thành phần:

+ Sự ràng buộc về placement ta đặt trên service redis, đảm bảo rằng nó luôn sử dụng cùng 1 host.
+ volume ta tạo ra cho phép container truy cập vào ./data (trên các host) như là /data bên trong Redis container. Trong khi các container được tạo và xóa, các file được lưu trữ trong ./data trên các host xác định, cho phép lưu trữ liên tục.

Bây giờ ta sẽ build 1 redis container sử dụng stack

- Tạo thư mục ./data trên manager
- Đảm bảo các node đã join swarm
- chạy lệnh deploy

```
$ docker stack deploy -c docker-compose.yml getstartedlab
```

- chạy lệnh docker service ls để kiểm tra có 3 service đang running như mong muốn.
- kiểm tra bằng cách truy cập vào Brower và đồng thời kiểm tra trên visualizer

1.5 Develop with docker

1.5.1 develop images

https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

a. Tạo base image
b. Các khuyến nghị khi build image

- Sử dụng multi-stage builds

  Multi-stage Build cho phép giảm kích thước của image cuối cùng trong quá trình build mà không cần phải làm giảm số file hay các layer trung gian. Bởi vì image được build

trong giai đoạn cuối cùng của tiến trình build, do đó ta có thể giảm thiểu số layer của image bằng cách sử dụng build cache.

- không cài đặt các package không cần thiết:
  Để giảm độ phức tạp, dependences, kích thước file, thời gian build, và tránh cài đặt thêm hoặc không cần thiết các package
- Tách riêng các ứng dụng
  Mỗi container chỉ nên có chạy 1 ứng dụng, Việc tách riêng các ứng dụng thành nhiều container giúp cho dễ dàng scale và tái sử dụng nó.
- Giảm tối thiểu các layers
  Cần chú ý những instructions sau sẽ thêm hoặc giảm số layer của image
  + chỉ các chỉ thị RUN, COPY, ADD sẽ tạo ra layer mới, những instruction còn lại chỉ tạo ra các image trung gian tạm thời và chúng không làm tăng size của image.
  + Bất kỳ khi nào có thể hãy sử dụng multi-stage build
- Sắp xếp các arguments trên nhiều dòng

II. Quản lý dữ liệu trong Docker

https://docs.docker.com/storage/

Mặc định tất cả các file được tạo ra bên trong container được lưu trữ ở writable layer của container. Tức là:

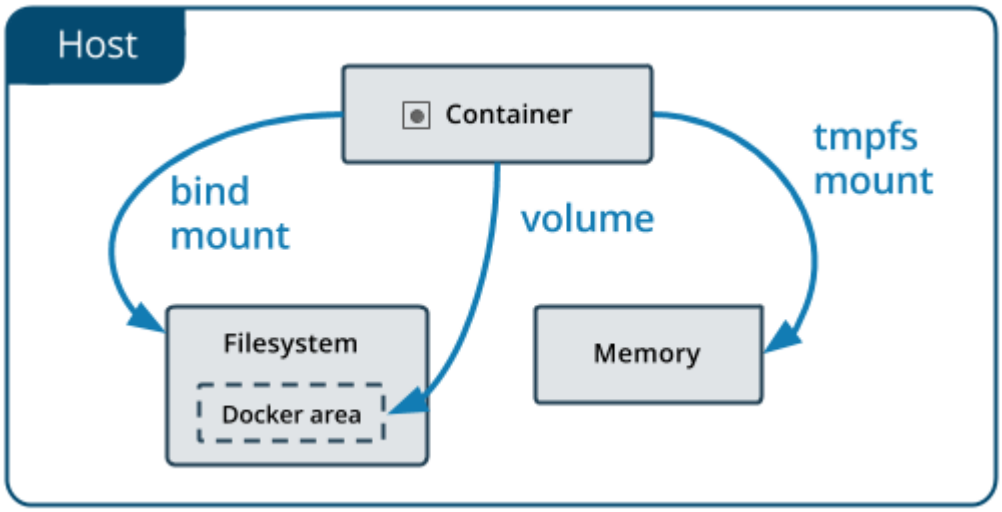
- Dữ liệu không còn tồn tại khi container không tồn tại, ta không thể sử dụng data của container đó nếu các process khác cần.
- Một writable layer của container được liên kết với host machine mà container running. ta không thể di chuyển dữ liệu tới những nơi khác.
- việc ghi vào writable layer của container yêu cầu 1 driver store để quản lý filesystem. Storage driver cung cấp Union filesystem sử dụng linux kernel. việc này sẽ làm giảm hiệu suất so với sử dụng volume data ghi trực tiếp lên host filesystem.

Docker có 2 lựa chọn cho các container để lưu trữ file trên host machine sao cho dữ liệu sẽ được giữ lại khi các container stop: Volume, bind mounts. nếu sử dụng linux ta có thể sử dụng *tmpfs mount*, nếu docker running trên windows có thể sử dụng *named pipe*.

2.1 Tổng quan về storage

2.1.1 Lựa chọn kiểu mount phù hợp

Cho dù ta lựa chọn kiểu mount nào thì dữ liệu trong container đều giống nhau. nó được hiển thị trong 1 thư mục hoặc 1 file riêng trong filesystem của container.
Cách dễ dàng nhất để hình dung sự khác biệt giữa volume, bind mounts và tmpfs đó là nghĩ về nơi data tồn tại trên docker host.



+ Volume: data được lưu trữ trong 1 phần của host filesystem và được quản lý bởi docker (/var/lib/docker/volume trên linux). Các chương trình non-docker không thể chỉnh sửa phần này của filesystem. volumes là cách tốt nhất để lưu trữ dữ liệu trong docker.
+ Bind mounts: data có thể lưu trữ ở bất kỳ đâu trên host machine. Chúng là những file và thư mục hệ thống quan trọng. Các chương trình none-docker trên Docker host hoặc 1 docker container có thể chỉnh sửa bất kỳ thời gian nào
+ tmpfs Mounts: Dữ liệu chỉ được lưu trữ trên memory hệ thống, không được lưu trữ trong filesystem của hệ thống.

● Chi tiết về các loại mount point

- Volumes: được tạo và quản lý bởi docker (nên các process khác không thể truy cập vào vùng này). tạo bằng cách sử dụng câu lệnh docker create volume hoặc docker có thể tạo volume khi tạo container hoặc service.

Khi ta tạo 1 volumes, nó được lưu trữ trong 1 thư mục trên host. Khi ta mount volume vào trong container thì thư mục này chứa tất cả những gì ta mount vào container.

1 Volume có thể được mount cho nhiều container đồng thời. Khi không có container nào đang running sử dụng volume, volume vẫn available trên docker và không bị remove. Ta có thể remove volume không sử dụng bằng cách sử dụng lệnh docker volume prune

Khi ta tạo 1 volume nó có thể được đặt tên (named) hoặc không (anomynous). anomynous volume không có tên khi chúng được mount vào container lần đầu tiên nên docker sẽ tạo cho nó 1 cái tên là 1 chuỗi ngẫu nhiên và đảm bảo tên này là duy nhất trong docker host. Ngoài sự khác nhau về đặt tên thì chúng không có sự khác biệt gì khác.

volume cũng hỗ trợ sử dụng volume driver, chúng cho phép lưu trữ dữ liệu ở remote host hoặc cloud.

- Bind mounts: bind mounts hạn chế một số chức năng so với volumes, khi sử dụng bind mount, các file hoặc thư mục trên host machine được mount vào trong 1 container. file hoặc thư mục sẽ được chỉ định bằng đường dẫn trên host machine và file hay thư mục này không cần thiết phải nằm trên host docker. Bind mount khá là hiệu suất tuy nhiên chúng lại dựa vào filesystem của host machine có một cấu trúc thư mục xác định. Nếu chúng ta phát triển một ứng dụng mới trên docker nên sử dụng named volume. Ta không thể sử dụng các lệnh docker CLI để quản lý bind mounts.

NOTE: Bind mount cho phép truy cập các file nhạy cảm. Ta có thể thay đổi filesystem của hệ thống thông qua các process đang chạy trong container gồm: creating, modifying, deleting các file hoặc thư mục hệ thống quan trọng. việc này gây mất bảo mật hệ thống và ảnh hưởng tới các chương trình non-docker.

- tmpfs mount: 1 tmpfs mount không được duy trì trên disk của host docker hoặc container. Nó có thể được sử dụng bởi container trong vòng đời của container để lưu trữ trạng thái non-persistent hoặc các thông tin nhạy cảm. Ví dụ các service swarm sử dụng tmpfs mount để mount secrets vào trong container của service.
- named pipe: 1 npipe mount có thể được sử dụng để giao tiếp giữa docker host và container, thường sử dụng để chạy các công cụ cung cấp bởi bên thứ 3 trong 1 container và kết nối tới docker engine API sử dụng named pipe.

Cả volume và bind mount đều được mount vào trong container sử dụng option -v (--volume) tag nhưng cú pháp câu lệnh cho từng loại lại khác nhau. với tmpfs mount sử dụng option --tmpfs flag. Tuy nhiên ở docker 17.06 hoặc mới hơn khuyến nghị nên sử dụng option --mount flag cho cả container và service đối với Bind mounts, volume và tmpfs vì cú pháp của nó rõ ràng hơn.

2.1.2 Các trường hợp sử dụng volume tốt nhất.

- chia sẻ dữ liệu giữa nhiều container đang running. Nếu ta không tạo nó, 1 volume sẽ được tạo ra ngay lần đầu tiên nó được mount vào trong container. khi container đó dừng hoặc bị xóa, volume vẫn tồn tại. Nhiều container có thể mount cùng volume đồng thời, có thể là read-write hoặc read-only. Volume chỉ bị remove khi ta remove chúng.
- Khi Docker host không đảm bảo có 1 thư mục được tạo trước hoặc cấu trúc file. Volume sẽ giúp ta tách cấu hình của Docker host từ container runtime.
- Khi ta muốn lưu trữ dữ liệu của container của mình trên remote host hoặc cloud.
- khi ta cần backup, restore hoặc migrate dữ liệu từ Docker host này tới 1 Docker host khác. Ta có thể stop container đang sử dụng volume sau đó backup thư mục của volume (linux mặc định là: `/var/lib/docker/volumes/<volume-name>`).

2.1.3 Các trường hợp sử dụng Bind mount.

- Chia sẻ các file cấu hình từ host tới container. Điều này lý giải việc mặc định docker cung cấp phân giải DNS cho container như thế nào, bằng cách mount /etc/resolv.conf từ host tới từng container.
- Chia sẻ source code hoặc build các artifact giữa môi trường phát triển trên Docker host và container. ví dụ ta có thể mount thư mục target/ của Maven vào trong container và mỗi lần ta build project Maven trên Docker host thì container sẽ truy cập tới artifact đã rebuilt. Nếu sử dụng Docker để phát triển theo cách này thì Dockerfile cho production có thể copy các artifact production-ready trực tiếp vào image hơn là dựa vào Bind mount.
- Khi các file hoặc cấu trúc thư mục của Docker host đảm bảo tính liên tục với Bind mount mà container yêu cầu.

2.1.4 Các trường hợp sử dụng tmpfs Mount.

tmpfs mount được sử dụng tốt nhất trong các trường hợp mà ta không muốn dữ liệu được lưu lại trên host machine hoặc trong container. có thể là lý do bảo mật hoặc để đảm bảo hiệu suất của container khi ứng dụng cần ghi vào 1 volume có dung lượng lớn dữ liệu ở trạng thái không cần lưu trữ.

2.1.5 Lời khuyên khi sử dụng Bind mount hoặc Volume.

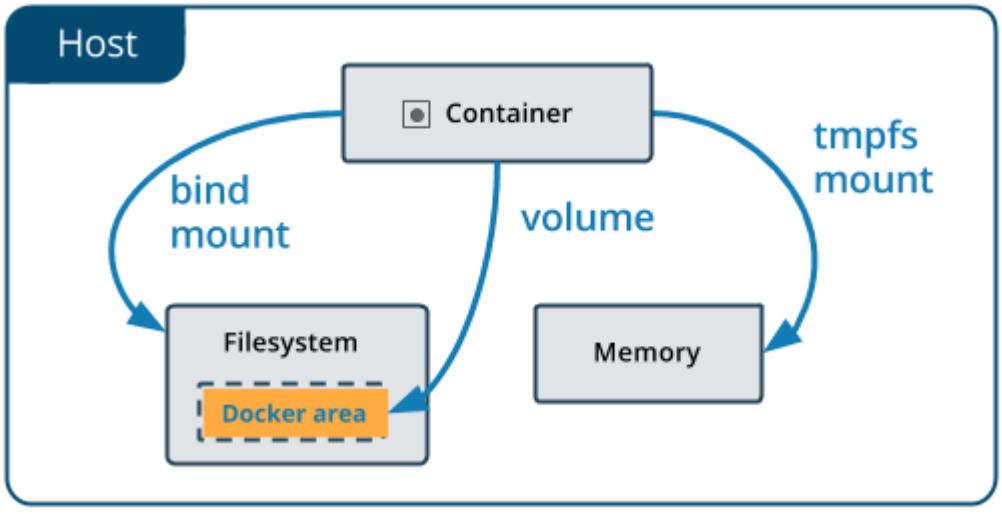
- Nếu ta sử dụng 1 volume rỗng trong 1 thư mục trong container mà thư mục này có các file hoặc thư mục đã tồn tại thì chúng sẽ được copy vào trong volume. Tưng tự nếu ta start 1 container và xác định 1 volume mà nó chưa tồn tại thì 1 volume rỗng sẽ được tạo ra. Đây là 1 cách để copy dữ liệu mà 1 container khác cần.
- Nếu ta mount một Bind mount hoặc non-empty volume vào trong 1 thư mục trong container mà thư mục này đã tồn tại các file hoặc thư mục thì những file và thư mục này sẽ bị che đi bởi mount, chỉ khi ta lưu các file vào trong /mnt trên linux host và sau đó mounted 1 USB Driver vào trong /mnt. Nội dung của /mnt sẽ bị che bởi nội dung của USB Driver cho đến khi USB drive được unmount. Những file bị che sẽ không bị xóa hay thay đổi nhưng chúng sẽ không thể truy cập tới trong khi Bind mount hoặc volume đang được mounted.

2.2 Sử dụng Volumes

Volume có một số ưu điểm so với Bind mount:

+ dễ dàng backup và migrate so với bind mount
+ có thể quản lý volume sử dụng các lệnh Docker CLI hoặc Docker API
+ Volume làm việc trên cả linux và windows container.
+ Volume được chia sẻ an toàn hơn giữa nhiều container.
+ Volume Driver cho phép ta lưu trữ volume trên các remote host hoặc cloud, mã hóa nội dung của volume và thêm nhiều chức năng khác.
+ 1 volume mới có thể có dữ liệu được copy từ container khác.

Ngoài ra volume thường là 1 sự lựa chọn tốt hơn với dữ liệu đang tồn tại trong writable layer của container bởi vì volume không làm tăng kích thước của container sử dụng nó và những nội dung của volumes tồn tại ngoài vòng đời của container.



Nếu container sinh ra dữ liệu ở trạng thái không lưu trữ nên sử dụng tmpfs mount.
Volume sử dụng rprivate bind propagation và bind propagation không thể cấu hình cho volume

- **Chọn -v flag hay --mount flag?**

Ban đầu -v flag được sử dụng cho các container độc lập và --mount flag được sử dụng cho các swarm service. Tuy nhiên bắt đầu từ docker 17.06 ta có thể sử dụng --mount flag cho các container riêng. Khác biệt lớn nhất đó là cú pháp -v kết hợp tất cả các option với nhau trong 1 field, trong khi --mount tách biệt các option.
Nếu cần xác định volume driver, ta phải sử dụng --mount option.

+ -v (--volume): gồm có 3 trường, được tách nhau bởi dấu 2 chấm (:), các trường này phải theo thứ tự chính xác.
  - trong trường hợp volume được đặt tên (named volume) thì trường đầu tiên là tên của volume và nó phải là duy nhất trên host machine. với anomynous volume trường này được bỏ qua.
  - Trường thứ 2 là đường dẫn nơi mà file hoặc thư mục được mount trong container.
  - Trường thứ 3 là tùy chọn, có nhiều option có thể lựa chọn và ta sẽ nói tới các option này phía dưới.
+ --mount: Gồm nhiều cặp key-value, cách nhau bằng dấu phẩy. Cú pháp --mount thì dài hơn so với -v tuy nhiên thứ tự các trường không quan trọng và giá trị của các flag dễ hiểu hơn.
  - type của mount có thể là bind, volume hoặc tmpfs
  - source của mount. Với named volume đây là tên của volume, với anomynous volume bỏ qua trường này, được xác định bằng source hoặc src
  - destination là đường dẫn nơi file hoặc thư mục được mount vào container. được xác định bằng destination, dst hoặc target
  - readonly option - nếu sử dụng thì bind mount sẽ được mount vào trong container dạng read-only.
  - volume-opt option - xác định nhiều hơn 1 option sử dụng cặp key-value gồm tên của option và giá trị của nó.

Ví dụ:

```
$ docker service create \
      --mount   'type=volume,src=<VOLUME-NAME>,dst=<CONTAINER-PATH>,volume-driver=local,volume-opt=type=nfs,volume-opt=device=<nfs-server>:<nfs-path>,"volume-opt=o=addr=<nfs-address>,vers=4,soft,timeo=180,bg,tcp,rw"'
      --name myservice <IMAGE>
```

+ So sánh -v và --mount: ngược lại với bind mount, tất cả các option của volume đều có thể sử dụng cho -v hoặc --mount flag.
+ Tạo và quản lý volume.

```
$ docker volume create my-vol
```

+ Liệt kê danh sách các volume

```
docker volume ls
```

+ kiểm tra 1 volume

```
$ docker volume inspect my-vol
```

```
kết quả:
[
    {
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/my-vol/_data",
        "Name": "my-vol",
        "Options": {},
        "Scope": "local"
    }
]
```

+ **xóa 1 volume.**

```
$ docker volume rm my-vol
```

- **Start container với volume**: ví dụ sau sẽ mount volume my-volume vào trong /app/ trong container:
  + sử dụng --mount

```
$ docker run -d \
  --name devtest \
  --mount source=myvol2,target=/app \
  nginx:latest
```

  + Sử dụng -v

```
$ docker run -d \
  --name devtest \
  -v myvol2:/app \
  nginx:latest
```

sử dụng câu lệnh docker inspect devtest để kiểm tra.

- **Start service với volume:** khi ta start service và định nghĩa 1 volume, mỗi service container sử dụng volume local của nó. không container nào có thể chia sẻ dữ liệu này nếu ta sử dụng local volume driver. Tuy nhiên có 1 số volume driver hỗ trợ chia sẻ storage. Docker for AWS hoặc Docker for Azure đều hỗ trợ store sử dụng Plugin Cloudstor. Ví dụ sau sẽ start service nginx với 4 replicas mỗi một replicas sử dụng 1 local volume có tên là myvol2.

```
$ docker service create -d \
  --replicas=4 \
  --name devtest-service \
  --mount source=myvol2,target=/app \
  nginx:latest
```

+ Sự khác biệt về cú pháp với service: lệnh docker service create không hỗ trợ -v flag.

- **Gắn volume đang sử dụng cho container**: Nếu ta start 1 container mà tạo 1 volume mới, container sẽ chứa các file và thư mục trong thư mục được mount. Nội dung trong thư mục được copy vào trong volume. Khi đó container sẽ mount và sử dụng volume này, các container khác sử dụng volume này cũng có thể truy cập vào nội dung đã copy trước đó.

Ví dụ start 1 container nginx và gắn volume mới nginx-vol với các nội dung của thư mục /usr/share/nginx/html của container, đây là thư mục mặc định chứa source của nginx.

+ Sử dụng option --mount

```
$ docker run -d \
  --name=nginxtest \
  --mount source=nginx-vol,destination=/usr/share/nginx/html \
  nginx:latest
```

+ Sử dụng option -v

```
$ docker run -d \
  --name=nginxtest \
  -v nginx-vol:/usr/share/nginx/html \
  nginx:latest
```

- **Sử dụng read-only volume:**

Với 1 số ứng dụng, container cần ghi vào bind mount để cho những thay đổi được copy tới Docker host. Đôi khi container chỉ cần truy cập để đọc data. Ta nên nhớ nhiều container có thể mount cùng volume, và có thể mount read-write đối với 1 số container và một vài container chỉ đọc cùng thời điểm. ví dụ sau sẽ mount thư mục read-only bằng cách thêm ro option vào trong danh sách các option khi tạo container sau mount point trong container:

+ Sử dụng --mount option:

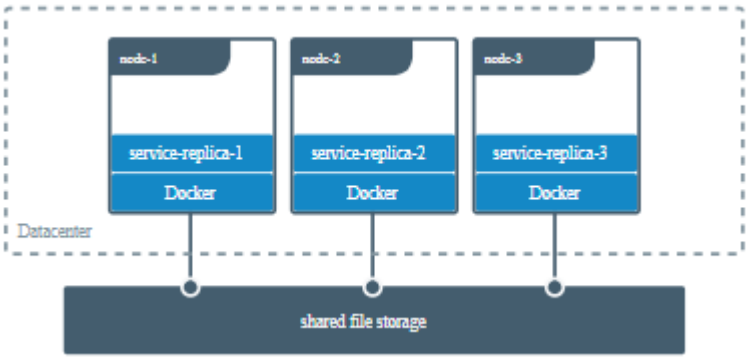
```
$ docker run -d \
  --name=nginxtest \
  --mount source=nginx-vol,destination=/usr/share/nginx/html,readonly
\
  nginx:latest
```

+ Sử dụng -v option:

```
$ docker run -d \
  --name=nginxtest \
  -v nginx-vol:/usr/share/nginx/html:ro \
  nginx:latest
```

- **Chia sẻ dữ liệu giữa các machine**.

Khi ta build 1 ứng dụng có khả năng chịu đựng được lỗi, ta cần cấu hình nhiều replicas cho 1 service truy cập cùng 1 file hoặc thư mục.



có 1 số cách để thực hiện việc này khi phát triển ứng dụng. Cách thứ nhất là thêm logic vào ứng dụng để lưu trữ các file trên hệ thống lưu trữ cloud. Cách khác đó là tạo volume bằng driver có hỗ trợ ghi file tới 1 hệ thống lưu trữ bên ngoài như NFS hay AWS S3.

Volume driver cho phép ta tách hệ thống lưu trữ với logic của ứng dụng. ví dụ nếu service sử dụng volume là NFS driver thì ta có thể update các service để sử dụng driver khác. ví dụ như lưu trữ trên cloud mà không thay đổi logic của ứng dụng.

- **Sử dụng volume Driver**

Khi ta tạo volume bằng lệnh docker volume create hoặc khi start 1 container sử dụng 1 volume chưa được tạo ta cần xác định volume driver. ví dụ sau sử dụng vieux/sshfs volume driver, trước tiên khi tạo 1 volume độc lập, sau đó là khi start 1 container cùng với tạo 1 volume mới.

+ **thiết lập ban đầu:** giả sử ta có 2 node, node đầu tiên là Docker host và có thể kết nối tới host thứ 2 thông qua ssh. Trên docker host cài đặt plugin `vieux/sshfs`:

```
$ docker plugin install --grant-all-permissions vieux/sshfs
```

+ **Tạo volume sử dụng volume driver.**

Ví dụ này xác định password SSH, nếu 2 node xác thực bằng key thì bỏ qua option này. Mỗi volume driver có thể không có hoặc có nhiều hơn 1 option cấu hình và chúng được chỉ định bởi -o flag.

```
$ docker volume create --driver vieux/sshfs \
```

```
  -o sshcmd=test@node2:/home/test \
  -o password=testpassword \
  sshvolume
```

+ **Start container kết hợp tạo volume sử dụng volume driver**

```
$ docker run -d \
  --name sshfs-container \
  --volume-driver vieux/sshfs \
             --mount                          src=sshvolume,target=/app,volume-
      opt=sshcmd=test@node2:/home/test,volume-opt=password=testpassword \
  nginx:latest
```

+ **Tạo service cùng với tạo NFS volume**

Ví dụ này ta tạo NFS volume khi tạo service. Ở đây ví dụ server 10.0.0.10 là server NFS và /var/docker-nfs là thư mục được chia sẻ trên NFS server. Chú ý là volume driver ở đây là local.

Với NFSv3:

```
$ docker service create -d \
  --name nfs-service \
      --mount              'type=volume,source=nfsvolume,target=/app,volume-
driver=local,volume-opt=type=nfs,volume-opt=device=:/var/docker-nfs,volume-
opt=o=addr=10.0.0.10' \
  nginx:latest
```

Với NFSV4:

```
docker service create -d \
    --name nfs-service \
      --mount              'type=volume,source=nfsvolume,target=/app,volume-
driver=local,volume-opt=type=nfs,volume-opt=device=:/,"volume-
opt=o=10.0.0.10,rw,nfsvers=4,async"' \
    nginx:latest
```

- **Backup, restore, migrate dữ liệu từ các volume.**

Sử dụng --volume-from flag để tạo 1 container mới mount từ volume đó.

+ **Backup container:**

  ```
  $ docker run --rm --volumes-from dbstore -v $(pwd):/backup ubuntu tar
  cvf /backup/backup.tar /dbdata
  ```

  Lệnh trên thực hiện công việc:

  - chạy 1 container mới và mount volume từ container dbstore

- mount 1 thư mục local host là backup
- sử dụng tar để đóng gói nội dung của dbdata volume thành 1 file backup.tar bên trong thư mục /backup

+ **Restore container từ backup:**

  ví tạo tạo 1 container mới dbstore2

  ```
  $ docker run -v /dbdata --name dbstore2 ubuntu /bin/bash
  ```

  untar file backup vào trong volume data của container.

  ```
  docker run --rm --volumes-from dbstore2 -v $(pwd):/backup ubuntu bash -c "cd /dbdata && tar xvf /backup/backup.tar --strip 1"
  ```

+ **Remove volume:**
  - Remove anonymous volume: để tự động xóa volume ta sử dụng rm option. Ví dụ sau sẽ tạo volume /foo anonymous, khi container bị xóa, docker engine sẽ xóa /foo volume mà không xóa /awesome volume.

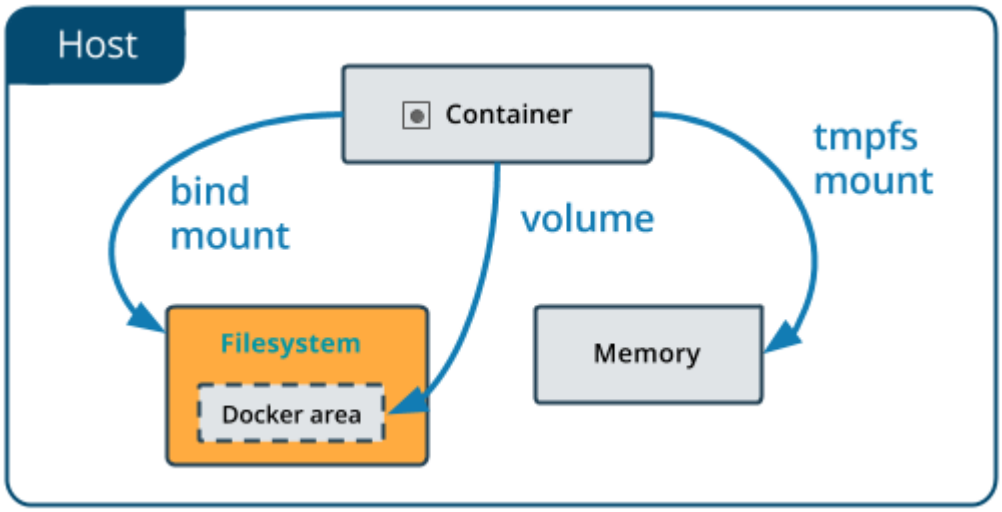
    ```
    $ docker run --rm -v /foo -v awesome:/bar busybox top
    ```

  - remove all volume:

    ```
    $ docker volume prune
    ```

2.3 Sử dụng Bind Mount.

Bind mount hạn chế một số chức năng so với volume, khi ta sử dụng Bind mount, file và thư mục trên host machine được mount vào trong container. Ngược lại khi sử dụng volume, 1 thư mục mới được tạo trong thư mục store của docker trên host machine và docker sẽ quản lý nội dung của thư mục này. file hoặc thư mục không cần thiết đã tồn tại trên Docker host, nó sẽ được tạo ra theo yêu cầu nếu không tồn tại. Bind mount khá là hiệu suất, tuy nhiên nó lại dựa vào filesystem của host machine có cấu trúc thư mục xác định. Nếu bạn đang build một ứng dụng docker mới nên sử dụng named volume. không thể sử dụng các lệnh docker CLI để quản lý các Bind mount.



a. Start container với bind mount

Xem xét trường hợp trong đó ta có thư mục source và khi ta build source code, các artifact được lưu vào trong 1 thư mục khác /source/target. Ta muốn các artifact available trong container ở thư mục /app/, và muốn container có thể truy cập new build mỗi lúc ta build source trên development host. Sử dụng lệnh sau để Bind mount thư mục /target vào trong container tại thư mục /app/. Chạy lệnh từ trong thư mục source:

- Sử dụng option: --mount

```
$ docker run -d \
  -it \
  --name devtest \
  --mount type=bind,source="$(pwd)"/target,target=/app \
  nginx:latest
```

- Sử dụng option -v

```
$ docker run -d \
  -it \
  --name devtest \
  -v "$(pwd)"/target:/app \
  nginx:latest
```

b. Mount vào thư mục rỗng trong container

Nếu ta Bind mount vào 1 thư mục non-empty trong container, những nội dung tồn tại trong thư mục sẽ bị che đi bởi bind mount. Việc này khá hữu ích khi ta muốn test 1 version mới của ứng dụng mà không cần build 1 image mới.

ví dụ sau sẽ tạo 1 container, thay thế nội dung ở thư mục /usr/share/nginx/html bằng thư mục /tmp trên host machine. Kết quả tạo ra 1 container non-functioning.

- Sử dụng --mount option:

```
$ docker run -d \
  -it \
  --name broken-container \
  -v /tmp:/usr \
  nginx:latest

docker: Error response from daemon: oci runtime error:
container_linux.go:262:
starting container process caused "exec: \"nginx\": executable file
not found in $PATH".
```

- Sử dụng -v option

```
$ docker run -d \
  -it \
  --name broken-container \
  -v /tmp:/usr \
  nginx:latest

docker: Error response from daemon: oci runtime error:
container_linux.go:262:
starting container process caused "exec: \"nginx\": executable file
not found in $PATH".
```

c. Sử dụng bind mount read-only

- Sử dụng --mount option:

```
$ docker run -d \
  -it \
  --name devtest \
  --mount type=bind,source="$(pwd)"/target,target=/app,readonly \
  nginx:latest
```

- Sử dụng -v option:

```
$ docker run -d \
  -it \
  --name devtest \
  -v "$(pwd)"/target:/app:ro \
  nginx:latest
```

d. Cấu hình gắn bind

mặc định Bind propagation là rprivate cho cả bind mount và volume. Nó chỉ có thể cấu hình cho bind mount và chỉ trên linux machine. đây là 1 chủ đề nâng cao và nhiều người không không bao giờ cần cấu hình nó.

Bind propagation đề cập tới có mount được tạo ra trong 1 bind mount cho trước hoặc named volume có thể được gắn cho các replicas của mount đó. xét 1 mount point /mnt, nó cũng được mount vào /tmp. Các setting về propagation điều khiển 1 mount trên /tmp/a có thể available trên /mnt/a hay không. Mỗi một setting propagation sẽ có 1 đối trọng đệ quy, trong trường hợp đệ quy , coi /tmp/a cũng được mount như /foo.

propagation settings	Mô tả
shared	sub-mount của mount gốc được hiển thị cho replicas mounts và sub-mount của replica mounts cũng được gắn vào mount gốc.
slave	similar to a shared mount, but only in one direction. If the original mount exposes a sub-mount, the replica mount can see it. However, if the replica mount exposes a sub-mount, the original mount cannot see it.
rshared	The same as shared, but the propagation also extends to and from mount points nested within any of the original or replica mount points.
rslave	The same as slave, but the propagation also extends to and from mount points nested within any of the original or replica mount points.
rprivate	The default. The same as private, meaning that no mount points anywhere within the original or replica mount points propagate in either direction.

Trước khi set bind propagation trên 1 mount point, host filesystem cần hỗ trợ bind propagation. tham khảo link: https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt

ví dụ sau sẽ mount thư mục /target vào 2 container và mount thứ 2 thiết lập option ro và bind propagation option là rslave.

- Sử dụng --mount option:

```
$ docker run -d \
  -it \
  --name devtest \
  --mount type=bind,source="$(pwd)"/target,target=/app \
```

```
  --mount
type=bind,source="$(pwd)"/target,target=/app2,readonly,bind-
propagation=rslave \
  nginx:latest
```

- Sử dụng -v option

```
$ docker run -d \
  -it \
  --name devtest \
  -v "$(pwd)"/target:/app \
  -v "$(pwd)"/target:/app2:ro,rslave \
  nginx:latest
```

e. Cấu hình selinux label

Nếu sử dụng selinux có thể thêm z hoặc Z option để chỉnh sửa label selinux của file host hoặc thư mục được mount vào trong container.

- z option chỉ định nội dung trong bind mount được sử dụng giữa nhiều container
- Z option chỉ định nội dung trong bind mount là privated và không được chia sẻ

ví dụ:

```
$ docker run -d \
  -it \
  --name devtest \
  -v "$(pwd)"/target:/app:z \
  nginx:latest
```

Nếu không thể chỉnh sửa selinux label thì sử dụng option --mount.

f. Cấu hình mount liên tục trên MacOS

Docker cho Mac sử dụng osxfs để gắn thư mục hoặc file được chia sẻ từ Mac cho linux VM. việc gắn này cho phép các thư mục và các file available với các Docker container đang running trên Docker Desktop của Mac.

Mặc định những chia sẻ này là hoàn toàn phù hợp, tức là mỗi khi có sự kiện ghi trên macOS host hoặc qua mount trong 1 container, các thay đổi sẽ được đẩy tới disk để tất cả các thành phần tham dự trong share có thể view chính xác. Từ docker 17.05 có một số option để điều chỉnh các cài đặt chính xác dựa trên từng mount, từng container. có 1 số option sau:

- consistent or default: các cài đặt mặc định hoàn toàn chính xác
- delegated: Các view runtime của container của mount là có thẩm quyền, có thể là trễ trước khi update được thực hiện trên host có thể hiển thị trong 1 host.

- cached: view của host MacOS của mount là có thẩm quyền. có thể là trễ trước khi update được thực hiện trên host có thể hiển thị trong 1 container.

Ví dụ:

+ sử dụng option --mount

```
$ docker run -d \
  -it \
  --name devtest \
  --mount
type=bind,source="$(pwd)"/target,destination=/app,consistency=cached
\
  nginx:latest
```

+ sử dụng -v

```
$ docker run -d \
  -it \
  --name devtest \
  -v "$(pwd)"/target:/app:cached \
  nginx:latest
```

2.4 tmpfs Mount

Không giống Bind mount và volume, tmpfs mount là tạm thời, chỉ tồn tại trên host memory. Khi container remove, dữ liệu ghi vào đây sẽ bị xóa.

a. hạn chế của tmpfs mount

- không thể chia sẻ giữa các container
- Nó chỉ available trên linux Docker

b. Chọn --tmpfs hay --mount flag.

Sự khác biệt giữa lớn nhất đó là --tmpfs flag không hỗ trợ bất kỳ option cấu hình nào.

- --tmpfs: Mounts a tmpfs mount without allowing you to specify any configurable options, and can only be used with standalone containers.
- --mount: Consists of multiple key-value pairs, separated by commas and each consisting of a <key>=<value> tuple. The --mount syntax is more verbose than --tmpfs:

+ The type of the mount, which can be bind, volume, or tmpfs. This topic discusses tmpfs, so the type is always tmpfs.
+ The destination takes as its value the path where the tmpfs mount is mounted in the container. May be specified as destination, dst, or target.

+ The tmpfs-type and tmpfs-mode options. See tmpfs options (https://docs.docker.com/storage/tmpfs/#tmpfs-options).

c. Sự khác biệt giữa --tmpfs và --mount

- --tmpfs không cho phép bạn xác định bất kỳ option cấu hình nào
- --tmpfs không được sử dụng trong swarm service, phải sử dụng --mount.

d. Sử dụng tmpfs mount trong 1 container.

để sử dụng tmpfs mount trong 1 container, sử dụng --tmpfs flag hoặc sử dụng --mount flag với các option type=tmpfs và destination. không có source cho --tmpfs mount. ví dụ sau tạo 1 tmpfs mount tại /app trong Nginx container.

+ Sử dụng --mount:

```
$ docker run -d \
  -it \
  --name tmptest \
  --mount type=tmpfs,destination=/app \
  nginx:latest
```

+ Sử dụng --tmpfs

```
$ docker run -d \
  -it \
  --name tmptest \
  --tmpfs /app \
  nginx:latest
```

e. Xác định các tmpfs options.

có 2 option, muốn sử dụng 2 option này thì phải sử dụng --mount flag.

Option	mô tả
tmpfs-size	kích thước tmpfs mount bằng byte, mặc định là unlimited
tmpfs-mode	file mode của tmpfs bằng octal, ví dụ 700, 0770, mặc định là 1777 hoặc world-writable

Ví dụ sau set tmpfs-mode là 1777

```
docker run -d \
  -it \
  --name tmptest \
  --mount type=tmpfs,destination=/app,tmpfs-mode=1770 \
  nginx:latest
```

2.5 Troubleshoot các vấn đề về volume

2.6 Lưu trữ dữ liệu trong container

III. Cấu hình networking

- **Tổng quan:** Một trong những lý do Docker và các service của nó trở nên mạnh mẽ đó là ta có thể kết nối chúng với nhau, hoặc kết nối chúng tới các workload non-Docker.
- **Network driver:** networking subsystem của docker sử dụng các driver, mặc định có một số driver có sẵn và cung cấp một số tính năng lõi:

+ bridge: là driver mặc định. Nếu ta không xác định 1 driver đây là loại network sẽ được tạo ra . Bridge network thường được sử dụng khi các ứng dụng chạy trong các container độc lập cần liên lạc với nhau.
+ host: Dành cho container chạy độc lập, sẽ sử dụng trực tiếp networking của host. host chỉ available đối với swarm service trên Docker 17.06 hoặc cao hơn.
+ overlay: overlay network kết nối nhiều Docker daemon với nhau và cho phép các swarm service giao tiếp với nhau. Ta cũng có thể sử dụng overlay network để thuận tiện giao tiếp giữa một swarm service và 1 container độc lập trên các Docker daemon khác nhau. với cách này có thể xóa nhu cầu cần thực hiện định tuyến ở mức OS giữa những container này.
+ macvlan: macvlan network cho phép ta gán địa chỉ MAC cho một container, cho phép nó như 1 thiết bị vật lý trong network. Docker daemon sẽ định tuyến lưu lượng tới container bằng địa chỉ MAC. Đôi khi việc sử dụng macvlan network là sự lựa chọn tốt nhất khi liên kết với các ứng dụng kế thừa có mong muốn kết nối trực tiếp tới network vật lý, hơn nữa được định tuyến thông qua network stack của docker host.
+ none: với container này, disable tất cả các networking. Thường được sử dụng kết hợp với custom network driver. none không được sử dụng trong swarm service
+ Network plugin: ta có thể cài đặt và sử dụng network plugin của bên thứ 3. các plugin này có sẵn trên Docker hub hoặc các vendor bên thứ 3

- **Tổng kết network driver:**

+ **User-defined bridge networks**: là tốt nhất khi ta cần nhiều container giao tiếp trên cùng Docker host
+ **Host networks** : tốt nhất khi network stack không nên tách biệt với Docker host, nhưng vẫn muốn những muốn những mặt khác của container tách riêng.
+ **Overlay networks**: tốt nhất khi cần các container chạy trên các docker host khác nhau có thể giao tiếp với nhau, hoặc khi nhiều ứng dụng làm việc với nhau sử dụng swarm service.

+ **Macvlan networks:** tốt nhất khi migrating từ một VM setup hoặc cần container trông giống như một host vật lý trong network. Mỗi container có 1 địa chỉ MAC duy nhất.
+ **Third-party network plugins:** cho phép ta tích hợp docker với các network stack đặc biệt.

3.1 Sử dụng network bridge

a. Sự khác biệt giữa bridge network do user định nghĩa và bridge network mặc định.

- User-defined Bridge cung cấp sự độc lập và khả năng tương tác tốt hơn giữa các ứng dụng được container hóa.

Những container kết nối với nhau bởi bridge network do user định nghĩa sẽ mở tất cả các port với nhau, nhưng không mở port nào ra ngoài. Điều này cho phép các ứng dụng được container hóa dễ dàng kết nối với nhau mà không cần mở cho phép truy cập từ bên ngoài.

ví dụ 1 ứng dụng web frontend và backend là database, sử dụng bridge network do user-defined chỉ cần mở port 80 để truy cập từ ngoài vào mà không cần mở port mysql backend.

Nếu ta chạy cùng stack ứng dụng trên bridge network mặc định thì cần phải mở cả port cho web và port cho mysql sử dụng -p flag (--public) cho mỗi port cần mở.

- User-defined bridge cung cấp phân giải DNS tự động giữa các container.

Các container sử dụng bridge network mặc định chỉ có thể truy cập tới nhau sử dụng địa chỉ IP trừ khi ta sử dụng --link option. Với User-defined bridge các container có thể phân giải với nhau dựa vào name hoặc alias.

Nếu ta chạy cùng application stack trên bridge network mặc định, ta cần tạo link thủ công giữa các container (sử dụng --link flag). Link này cần được tạo trên cả 2 phía. Một cách khác ta có thể sử dụng file /etc/hosts trong các container, tuy nhiên việc này tạo ra nhiều vấn đề gây khó khăn cho việc Debug.

- Mỗi User-defined bridge network tạo ra một bridge có thể cấu hình

Nếu container sử dụng bridge network mặc định, ta có thể cấu hình nó nhưng tất cả các container sẽ sử dụng chung cấu hình như MTU, các rule iptable. ngoài ra việc cấu hình bridge network mặc định được thực hiện bên ngoài docker và yêu cầu phải restart docker.

User-defined bridge network được tạo ra và cấu hình sử dụng lệnh docker network create. Nếu các group ứng dụng khác nhau có các yêu cầu về network khác nhau ta có thể cấu hình từng User-defined bridge network riêng.

- Các container có thể được gắn hoặc không từ User-defined network nhanh chóng

Trong khoảng thời gian tồn tại của container, ta có thể kết nối hoặc ngắt kết nối nó từ User-defined network nhanh chóng. để xóa một container từ bridge network mặc định cần stop container và tạo lại nó với các network option khác.

- Các container được liên kết với nhau trên bridge network mặc định chia sẻ các biến môi trường

Ban đầu, cách duy nhất để chia sẻ các biến môi trường giữ 2 container là liên kết chúng sử dụng --link flag. kiểu chia sẻ là không thể với User-defined network, tuy nhiên có một số cách để chia sẻ các biến môi trường:

+ Nhiều container có thể mount 1 file hoặc thư mục chứa chưa thông tin được chia sẻ sử dụng Docker volume
+ Nhiều container có thể start cùng lúc sử dụng docker-compose và file compose định nghĩa các biến chia sẻ.
+ Có thể sử dụng swarm service thay thế các container độc lập, tận dụng ưu điểm secrets và config chia sẻ.

b. Quản lý User-defined network

Sử dụng lệnh docker network create để tạo User-defined bridge network

```
$ docker network create my-net
```

ta có thể xác định subnet, dải địa chỉ IP gateway và các option khác, sử dụng lệnh docker network create --help để xem các chi tiết. để xóa 1 User-defined network sử dụng lệnh:

```
$ docker network rm my-net
```

c. Kết nối container tới User-defined bridge network.

Khi ta tạo 1 container cần xác định 1 hoặc nhiều --network flag . Ví dụ kết nối nginx container tới my-net network, đồng thời map public port 80 trong container với port 8088 của Docker host.

```
$ docker create --name my-nginx \
  --network my-net \
  --publish 8080:80 \
  nginx:latest
```

để kết nối container đang running tới User-defined bridge đã có sử dụng lệnh docker network connect:

```
$ docker network connect my-net my-nginx
```

d. Cho phép forwarding từ Docker container ra ngoài

mặc định, traffic từ container kết nối tới bridge network mặc định sẽ không được forward ra bên ngoài. Để cho phép forwarding, ta cần thay đổi 2 tham số

- Cấu hình kernel linux cho phép IP forwarding

```
$ sysctl net.ipv4.conf.all.forwarding=1
```

- thay đổi policy của iptables, FORWARD policy từ DROP thành ACCEPT

```
$ sudo iptables -P FORWARD ACCEPT
```

f. Disconnect container từ User-defined bridge network

Sử dụng lệnh sau:

```
$ docker network disconnect my-net my-nginx
```

g. Sử dụng Bridge network mặc định

bridge network mặc định khuyến nghị không nên sử dụng trong môi trường production.

- Kết nối container với bridge network mặc định

Nếu không chọn network sử dụng --network flag và chọn network driver thì mặc định container sẽ kết nối tới bridge network mặc định, các container kết nối với bridge network mặc định có thể giao tiếp với nhau nhưng chỉ bằng địa chỉ IP trừ khi chúng được liên kết sử dụng --link flag.

- Cấu hình bridge network mặc định

Để cấu hình default bridge network, ta cần xác định các option trong file daemon.js. dưới đây là 1 ví dụ:

```
{
  "bip": "192.168.1.5/24",
  "fixed-cidr": "192.168.1.5/25",
  "fixed-cidr-v6": "2001:db8::/64",
  "mtu": 1500,
  "default-gateway": "10.20.1.1",
  "default-gateway-v6": "2001:db8:abcd::89",
  "dns": ["10.20.1.2","10.20.1.3"]
}
```

restart Docker để những thay đổi có hiệu lực

3.2 Sử dụng overlay network

overlay network driver tạo 1 network phân bố giữa các Docker deamon. network này sẽ nằm trên cùng (overlay) các network host-specific, cho phép các container kết nối đến nó (gồm cả các container của swarm service) để giao tiếp an toàn. Docker sẽ định tuyến từng gói tin đến và đi từ Docker daemon đến đích.

Khi khởi tạo 1 swarm hoặc join các Docker daemon vào swarm, sẽ có 2 network được tạo ra:

- Trên overlay network được gọi là ingress, chúng sẽ xử lý các traffic dữ liệu và điều khiển liên quan tới swarm service. Khi ta tạo 1 swarm service và không kết nối nó với User-defined network, mặc định nó sẽ kết nối tới ingress network.
- bridge network được gọi là docker_gwbridge, chúng sẽ kết nối các Docker daemon riêng lẻ với các Docker daemon trong swarm.

ta có thể tạo User-defined overlay network sử dụng lệnh docker network create tương tự như bridge network.

a. Vận hành overlay network.

- Tạo 1 overlay network

Yêu cầu: Mở firewall đối với các port sau: TCP/2377 (Quản lý cluster), TCP/UDP/7946 cho giao tiếp giữa các node và UDP/4789 cho overlay network traffic. Trước khi tạo overlay network cần khởi tạo swarm và join các node vào swarm.

Để tạo overlay network sử dụng cho swarm service dùng lệnh sau:

```
$ docker network create -d overlay my-overlay
```

Để tạo overlay network có thể sử dụng bởi các swarm service hoặc các container độc lập để giao tiếp với các container độc lập khác đang chạy trên các Docker daemon khác sử dụng --attachable flag.

```
$ docker network create -d overlay --attachable my-attachable-overlay
```

ta cũng có thể xác định subnet, gateway, địa chỉ IP như bridge network.

- Mã hóa lưu lượng trên overlay network

Tất cả các traffic quản lý swarm service mặc định được mã hóa sử dụng thuật toán AES ở mode GCM. để mã hóa dữ liệu của ứng dụng ta sử dụng 1 option: --otp encrypted khi tạo overlay network

Note: Không gắn các Windows nodes vào overlay network đã mã hóa

- Overlay network mode swarm và các container độc lập

ta có thể sử dụng cả 2 thuộc tính overlay network --opt encrypted --attachable và gắn các container không được quản lý vào mạng đó.

```
$ docker network create --opt encrypted --driver overlay --attachable my-attachable-multi-host-network
```

- Điều chỉnh default ingress network

Hầu hết các user không cần cấu hình ingress network, nhưng với Docker 17.05 hoặc cao hơn cho phép ta có thể cấu hình.

Việc chỉnh sửa ingress network có 2 việc đó là xóa vào tạo lại nó. việc này thường được thực hiện trước khi tạo các service trong swarm. Nếu đã tồn tại các service và có các port đã public thì cần remove service trước khi remove ingress network.

- Kiểm tra ingress network sử dụng lệnh docker network inspect ingress
- remove ingress network sử dụng lệnh : các service cần stop trước khi remove.

  $ docker network rm ingress

- tạo ingress network sử dụng --ingress flag.

  ```
  $ docker network create \
    --driver overlay \
    --ingress \
    --subnet=10.11.0.0/16 \
    --gateway=10.11.0.2 \
    --opt com.docker.network.driver.mtu=1200 \
    my-ingress
  ```

  Note: có thể đặt tên cho network bằng tên khác ingress tuy nhiên chỉ có thể tạo được 1 ingress network.

- restart lại các service đã stop

- Điều chỉnh docker_gwbridge

docker_gwbridge là một bridge ảo kết nối overlay network tới physical network của Docker daemon. Docker tạo nó tự động khi khởi tạo swarm hoặc join Docker host vào swarm nhưng nó không phải là Docker device. nó tồn tại trong kernel của docker host. Nếu ta chỉnh sửa các cấu hình này ta cần thực hiện một số bước trước khi join Docker host vào swarm, hoặc sau khi tạm thời remove host khỏi swarm.

- Stop Docker

- xóa interface docker_gwbridge đang tồn tại

```
$ sudo ip link set docker_gwbridge down
$ sudo ip link del dev docker_gwbridge
```

- Start docker và không khởi tạo hoặc join swarm
- tạo hoặc tạo lại docker_gwbridge network với các cấu hình của mình sử dụng lệnh docker network create.

```
$ docker network create \
--subnet 10.11.0.0/16 \
--opt com.docker.network.bridge.name=docker_gwbridge \
--opt com.docker.network.bridge.enable_icc=false \
--opt com.docker.network.bridge.enable_ip_masquerade=true \
docker_gwbridge
```

- Khởi tạo lại swarm hoặc join các node vào swarm

b. Vận hành swarm network

- Public port trên overlay network

để public port sử dụng thêm -p hoặc --public flag trên câu lệnh docker network create hoặc docker network update. Dưới đây là 1 số ví dụ với -p flag.

Flag value	Mô tả
-p 8080:80 hoặc -p published=8080,target=80	Map port 80/TCP của service với port 8080 trên routing mesh
-p 8080:80/udp hoặc -p published=8080,target=80, protocol=udp	Map port 80/UDP của service với port 8080 trên routing mesh
-p 8080:80/tcp -p 8080:80/udp hoặc -p published=8080,target=80,protocol=tcp -p published=8080,target=80,protocol=udp	Map port 80/TCP của service với port 8080 trên routing mesh và Map port 80/UDP của service với port 8080 trên routing mesh

- bypass routing mesh cho swarm service

mặc định swarm service public port sử dụng routing mesh. khi kết nối tới port public trên bất kỳ node swarm nào thì không thể redirect đến worker đang chạy service đó. Docker đóng vai trò là 1 load-balancer cho các swarm service. Các service đang sử dụng routing mesh running ở Virtual IP mode (VIP). Khi sử dụng routing mesh, không đảm bảo rằng client gửi request tới service trên Docker host nào.

Để bypass routing mesh, ta start service sử dụng mode DNS round robin (DNSRR) bằng cách thiết lập --endpoint-mode flag là dnsrr. Ta phải chạy load-balance riêng ở trước service. 1 query đến service name trên Docker host sẽ trả về 1 danh sách các IP tương ứng với các node đang chạy service. Ta cần cấu hình load-balance với các IP trong danh sách này và cân bằng traffic trên các node.

- Tách lưu lượng control và lưu lượng data

Mặc định control traffic liên quan tới quản lý swarm và traffic đến và từ các ứng dụng chạy trên cùng network, tuy nhiên swarm control traffic được mã hóa. ta có thể cấu hình Docker tách riêng các interface để xử lý riêng 2 loại traffic này. Khi khởi tạo hoặc join swarm, cần chỉ định --advertise-addr và --datapath-addr riêng, và cần thực hiện trên từng node join swarm

c. Vận hành overlay network với các container độc lập

- Gắn container độc lập vào overlay network

ingress network được tạo ra mà không có --attachable flag có nghĩa là chỉ swarm service mới có thể sử dụng nó. ta có thể kết nối các container độc lập tới User-defined overlay network sử dụng --attachable flag. Việc này cho phép các container độc lập chạy trên các Docker daemon khác nhau có khả năng giao tiếp với nhau mà không cần thiết lập routing trên từng Docker daemon host.

- Public port

Flag value	mô tả
-p 8080:80	Map port 80/TCP trong container với port 8080 trên overlay network
-p 8080:80/udp	Map port 80/UDP trong container với port 8080 trên overlay network
-p 8080:80/sctp	Map port 80/SCTP trong container với port 8080 trên overlay network
-p 8080:80/tcp -p 8080:80/udp	Map port 80/TCP trong container với port 8080 trên overlay network và Map port 80/UDP trong container với port 8080 trên overlay network

3.3 Sử dụng host networking

Nếu sử dụng host network cho container, network stack của container đó sẽ không độc lập với docker host (container chia sẻ networking namespace của host) và container sẽ không nhận được địa chỉ IP riêng của nó.Ví dụ nếu ta chạy 1 container bind port 80 và ta sử dụng host networking thì ứng dụng của container available trên port 80 của địa chỉ IP của host.

Note: container không có địa chỉ IP riêng khi sử dụng host network, nên port map cũng không còn tác dụng do đó các -p, --public, -P hoặc --public-all flag sẽ bị bỏ qua.

Host networking hữu ích khi tối ưu hiệu suất và trong nhiều trường hợp khi 1 container cần xử lý một dải lớn các port không yêu cầu NAT. Host networking chỉ hoạt động trên linux.

ta cũng có thể sử dụng host network cho swarm service sử dụng --network host option trong câu lệnh tạo network. Trong trường hợp này, control traffic vẫn trên overlay network, nhưng các container swarm service truyền dữ liệu sử dụng port và host network của Docker daemon.

3.4 Sử dụng macvlan network

Một số ứng dụng, đặc biệt là những ứng dụng có tính kế thừa hoặc những ứng dụng giám sát network traffic, muốn kết nối trực tiếp tới physical network. Với những trường hợp như này ta có thể sử dụng network driver macvlan để gán địa chỉ MAC cho từng network interface ảo của container, cho phép nó như 1 physical network interface kết nối trực tiếp với physical network. Trong trường hợp này ta cần chỉ định một interface vật lý trên Docker host sử dụng macvlan cũng như subnet và gateway của macvlan. ta có thể tách các macvlan network sử dụng các network interface vật lý khác nhau. với macvlan network ta cần ghi nhớ:

- dễ gây tổn hại không mong muốn tới network do hết địa chỉ IP hoặc mở rộng VLAN, đây là tình huống mà ta sẽ có 1 lượng lớn địa chỉ MAC không phù hợp trong network.
- các thiết bị networking cần có khả năng xử lý “promiscuous mode”, ở đây 1 interface vật lý có thể gán nhiều địa chỉ MAC.
- Nếu ứng dụng sử dụng bridge (Trên single docker host), hoặc overlay (giao tiếp giữa nhiều docker host), giải pháp này rất khả thi.

- Tạo macvlan network

Khi tạo 1 macvlan network, nó có thể ở 1 trong 2 mode: bridge mode hoặc 802.1q trunk bridge mode

+ trong bridge mode, macvlan traffic sẽ đi qua physical device trên host
+ 802.1q trunk mode, traffic đi qua 1 sub-interface 802.1q mà Docker có thể tạo ra dễ dàng. điều này cho phép ta control routing và filtering ở mức cụ thể hơn.

- Bridge mode

Để tạo macvlan network nối với network interface vật lý sử dụng --driver macvlan option trong câu lệnh tạo network. ta cũng có thể xác định parent, đây là interface mà traffic sẽ đi qua trên Docker host.

```
$ docker network create -d macvlan \
  --subnet=172.16.86.0/24 \
  --gateway=172.16.86.1 \
  -o parent=eth0 pub_net
```

Nếu không muốn sử dụng địa chỉ IP nào trong macvlan network sử dụng --aux-address:

```
$ docker network create -d macvlan \
  --subnet=192.168.32.0/24 \
  --ip-range=192.168.32.128/25 \
  --gateway=192.168.32.254 \
  --aux-address="my-router=192.168.32.129" \
  -o parent=eth0 macnet32
```

- 802.1q trunk bridge mode

Nếu ta xác định tên parent interface không có dấu chấm (.), ví dụ: eth0.50 thì docker hiểu rằng đó là một sub-interface của eth0 và tạo sub-interface tự động.

```
$ docker network create -d macvlan \
    --subnet=192.168.50.0/24 \
    --gateway=192.168.50.1 \
    -o parent=eth0.50 macvlan50
```

- Sử dụng ipvlan thay thế macvlan

Trong các ví dụ trên ta đang sử dụng bridge layer 3, ta có thể sử dụng ipvlan thay thế và tạo 1 bridge layer 2 bằng cách sử dụng option -o ipvlan_mode=l2

```
$ docker network create -d ipvlan \
    --subnet=192.168.210.0/24 \
    --subnet=192.168.212.0/24 \
    --gateway=192.168.210.254 \
    --gateway=192.168.212.254 \
     -o ipvlan_mode=l2 ipvlan210
```

3.5 Disable networking cho container

Nếu muốn disable hoàn toàn network stack trên 1 container ta sử dụng --network none flag khi start container. khi này trong container chỉ loopback device được tạo ra. như ví dụ sau:

- Tạo container

```
$ docker run --rm -dit \
  --network none \
  --name no-net-alpine \
  alpine:latest \
  ash
```

- check network stack của container

```
$ docker exec no-net-alpine ip link show
```

Lệnh sau để check routing

```
$ docker exec no-net-alpine ip route
```

Khi disable networking thì bảng định tuyến sẽ trống.

- Stop container, nó tự động xóa do nó được tạo với --rm flag.

```
$ docker container rm no-net-alpine
```

3.5 Hướng dẫn sử dụng chi tiết networking

https://docs.docker.com/network/network-tutorial-standalone/

a. Bridge network
b. Host networking
c. Overlay networking
d. Macvlan network

3.6 Cấu hình daemon và container

3.7 Kế thừa nội dung networking

IV. Chạy ứng dụng trong môi trường production

https://docs.docker.com/config/labels-custom-metadata/

4.1 Cấu hình tất cả các object

a. sử dụng metadata cho các object

- Đánh nhãn Docker object

Đánh nhãn là 1 cơ chế sử dụng metadata cho các Docker object, gồm:

+ Images

    + Containers
    + Local daemons
    + Volumes
    + Networks
    + Swarm nodes
    + Swarm services
- Key và value của nhãn

Nhãn là 1 cặp key-value được lưu trữ dưới dạng string, có thể sử dụng nhiều nhãn cho 1 object nhưng mỗi cặp key-value phải là duy nhất với object đó. Nếu cùng một key cung cấp nhiều value thì giá trị ghi gần nhất sẽ ghi đè lên giá trị trước đó.

- Các khuyến nghị định dạng key

Key nằm phía bên trái của cặp key-value, là chuỗi các chữ cái có thể chứa các ký tự (.) hoặc (-)

b. Xóa những object không sử dụng

Docker thực hiện thận trọng clearning các object không còn sử dụng như image, volume, container, network. Những object này thường không bị remove trừ khi ta yêu cầu Docker xóa. Với mỗi object docker cung cấp lệnh prune. Ngoài ra ta có thể sử dụng lệnh docker system prune để clean up nhiều object một lần.

- Prune image: lệnh docker image prune cho phép clean up các image không sử dụng. Mặc định lệnh này chỉ xóa các dangling image, dangling image là image không có tag. và để xóa tất cả các image mà container đang tồn tại không sử dụng sử dụng option -a trong câu lệnh trên: docker image prune -a
- Prune container: Khi stop 1 container, nó sẽ không tự động removed, trừ khi ta chạy câu lệnh docker run với option --rm. Để clean sử dụng lệnh docker container prune, mặc định nó sẽ xóa tất cả các container stop. Ta có thể giới hạn phạm vi container bị xóa sử dụng --filter flag. Ví dụ sau chỉ remove những container stop trong khoảng 24h.

```
$ docker container prune --filter "until=24h"
```

- Prune volume: Volume có thể được sử dụng bởi 1 hoặc nhiều container và chiếm giữ không gian lưu trữ của Docker host. Volume không bao giờ bị xóa tự động vì nó sẽ khiến mất data. sử dụng lệnh docker volume prune để xóa volume, khi chạy lệnh này sẽ có cảnh báo có chắc muốn xóa không, để bỏ qua cảnh báo sử dụng -f (--force) flag. Mặc định lệnh này sẽ xóa

những volume không sử dụng. và ta cũng có thể hạn chế việc xóa sử dụng --filter flag. ví dụ sau xóa volume có nhãn không phải là keep

```
$ docker volume prune --filter "label!=keep"
```

- Prune network

Docker network không chiếm dung lượng server nhưng chúng tạo ra các iptable rule, bridge network device, và bảng định tuyến. Để clean các network không được sử dụng bởi container nào sử dụng lệnh: docker network prune. Mặc định tất cả các network không sử dụng sẽ bị xóa. Có thể dùng --filter flag để giới hạn network cần xóa.

```
$ docker network prune --filter "until=24h"
```

- Prune everything

Lệnh docker system prune là shortcut của prun image, prun container và prun network, với các bản Docker 17.06.0 trở về trước lệnh này bao gồm cả prune volume, tuy nhiên từ Docker 17.06.1 trở đi phải thêm --volume flag để prun volume.

```
$ docker system prune
```

```
$ docker system prune --volumes
```

c. Định dạng đầu ra của câu lệnh và log

Docker sử dụng go template (https://golang.org/pkg/text/template/), ta có thể sử dụng để chỉnh sửa định dạng đầu ra của các lệnh và log driver.

Docker cung cấp 1 tập các chức năng cơ bản để điều chỉnh các thành phần của template. Tất cả những ví dụ sau sử dụng lệnh docker inspect, còn những lệnh CLI khác có --format flag.

- **Join**: join sẽ liên kết danh sách các chuỗi thành 1 chuỗi duy nhất, nó chứa 1 ký hiệu để phân tách các chuỗi trong list.

```
docker inspect --format '{{join .Args " , "}}' container
```

- **json**: json mã hóa 1 phần tử như là 1 chuỗi json

```
docker inspect --format '{{json .Mounts}}' container
```

- **lower**: chuyển 1 string thành 1 string chỉ gồm chữ thường

```
docker inspect --format "{{lower .Name}}" container
```

- **split**: split cắt string thành một list các string tách nhau bởi dấu phân cách

```
docker inspect --format '{{split .Image ":"}}'
```

- **title**: Chuyển ký tự đầu tiên thành in hoa.

```
docker inspect --format "{{title .Name}}" container
```

- **upper**: Chuyển string thành tất cả các ký tự in hoa

```
docker inspect --format "{{upper .Name}}" container
```

- **println**: in từng giá trị trên 1 dòng mới

```
docker inspect --format='{{range .NetworkSettings.Networks}}{{println
.IPAddress}}{{end}}' container
```

- **Hint**: Tìm data có thể in ra, show tất cả nội dung dạng json

```
docker container ls --format='{{json .}}'
```

4.2 Cấu hình daemon

a. Cấu hình và chạy Docker

- Start docker sử dụng công cụ hệ thống
- Start Docker daemon thủ công: chỉ cần chạy lệnh dockerd bằng root user hoặc sử dụng sudo. khi khởi động docker theo cách này nó sẽ chạy ở foreground, và output sẽ in ra terminal.
- Cấu hình docker daemon: có 2 cách
  - + sử dụng file cấu hình json, cách này được khuyến nghị vì nó lưu tất cả cấu hình ở 1 nơi.
  - + Sử dụng flag khi start dockerd

Để cấu hình Docker daemon sử dụng file json, tạo 1 file /etc/docker/daemon.json trên linux, C:\ProgramData\docker\config\daemon.json trên windows và taskbar > Preferences > Daemon > Advanced trên MacOS.

file cấu hình sẽ giống như sau:

```
{
  "debug": true,
  "tls": true,
  "tlscert": "/var/docker/server.pem",
  "tlskey": "/var/docker/serverkey.pem",
  "hosts": ["tcp://192.168.59.3:2376"]
}
```

Với cấu hình này docker daemon chạy ở debug mode, sử dụng TLS, và lắng nghe traffic được định tuyến 192.168.59.3 trên port 2376. Để biết chi tiết hơn tham khảo

(https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file)

Ta có thể Start Docker daemon thủ công và cấu hình nó sử dụng các flag. Cách này rất hữu ích khi troubleshoot. dưới đây là 1 ví dụ cấu hình thủ công với các flag.

```
dockerd --debug \
  --tls=true \
  --tlscert=/var/docker/server.pem \
  --tlskey=/var/docker/serverkey.pem \
  --host tcp://192.168.59.3:2376
```

- Thư mục Docker daemon

Docker daemon lưu trữ tất cả data chỉ trong 1 thư mục, mặc định đó là /var/lib/docker trong linux và C:\ProgramData\docker trong windows, ta có thể cấu hình sử dụng thư mục khác sử dụng --data-root option. Cần đảm bảo mỗi Docker daemon sử dụng 1 thư mục riêng, nếu 2 Docker daemon dùng chung thư mục sẽ gây ra lỗi rất khó xử lý.

- Troubleshoot Daemon

Ta có thể bật debugging trên daemon để troubleshoot.

- troubleshoot trùng cấu hình giữa daemon.json và startup scripts: Nếu sử dụng file daemon.json và cùng sử dụng các option trong lệnh dockerd hoặc sử dụng scripts start-up những option này có thể trùng nhau. Khi này cần thay đổi flag hoặc chỉnh sửa daemon.json file để loại bỏ sự trùng nhau này.
- Sử dụng host key trong daemon.json với systemd: Một ví dụ đáng chú ý trong việc trùng cấu hình khó troubleshoot khi ta muốn xác định một địa chỉ daemon khác với mặc định. Mặc định Docker listen trên 1 socket. trên các hệ thống sử dụng systemd -H flag (host flag) luôn luôn được sử dụng khi start dockerd, nếu ta xác định 1 hosts trong daemon.json sẽ gây ra trùng cấu hình và Docker start bị lỗi. Để xử lý vấn đề này ta cần tạo 1 file mới /etc/systemd/system/docker.service.d/docker.conf với những nội dung sau để remove -H option được sử dụng mặc định khi start daemon.

  ```
  [Service]
  ExecStart=
  ExecStart=/usr/bin/dockerd
  ```

  chạy lệnh `sudo systemctl daemon-reload` trước khi start docker. Nếu docker start thành công, nó sẽ listen trên địa chỉ IP trong hosts key ở file cấu hình daemon.json thay vì socket.

NOTE: setting hosts chỉ dành cho Docker chạy trên linux

- Out of memory exception (OOME): Nếu Docker daemon sử dụng nhiều memory hơn memory hệ thống có thì có thể xảy ra lỗi OOME, và 1 container hoặc 1 Docker daemon sẽ bị kill bởi kernel killer. để điều này không xảy ra cần đảm bảo rằng ứng dụng đang chạy trên các host có memory thích hợp (tìm hiểu thêm https://docs.docker.com/engine/admin/resource_constraints/#understand-the-risks-of-running-out-of-memory)
- Đọc log
    + trên linux: /var/log/messages hoặc journalctl -u docker.service
    + Trên ubuntu: /var/log/upstart/docker.log
    + Trên windows: `AppData\Local`
    + `TrênMacOS: ~/Library/Containers/com.docker.docker/Data/vms/0/console-ring`
- enable debug

  có 2 cách để enable debugging, khuyến nghị thiết lập debug key là true trong file cấu hình daemon.json

    + Edit file daemon.json
    + add nội dung sau vào file

      ```
      {
        "debug": true
      }
      ```

      Nếu file json đã tồn tại chỉ cần add key "debug": true và nếu key log-level được thiết lập thì đặt nó là info hoặc debug. Mặc định là info. Các giá trị khác gồm: debug, info, warn, error, fatal
    + Send HUB signal tới daemon để reload cấu hình, trên linux sử dụng câu lệnh:

      ```
      $ sudo kill -SIGHUP $(pidof dockerd)
      ```

      trên windows, restart Docker

  Thay vì thực hiện các bước trên ta có thể stop docker và start nó với -D flag
- Force a stack trace to be logged

  Nếu Docker daemon không có phản hồi, ta cần bắt buộc 1 full stack track logged bằng cách gửi SIGUSR1 signal tới daemon. trên linux dùng lệnh

```
$ sudo kill -SIGUSR1 $(pidof dockerd)
```

Trên windows server: dowload docker signal (https://github.com/jhowardmsft/docker-signal) → tìm IP process của dockerd (Get-Process dockerd) → thực thi với flag --pid=<PID of daemon>

- Xem stack traces

  Docker daemon log có thể được xem sử dụng cách sau:

  + sử dụng lệnh journalctl -u docker.service trên hệ thống linux sử dụng systemctl
  + /var/log/messages, /var/log/daemon.log, or /var/log/docker.log trên hệ thống linux.
  + Get-EventLog -LogName Application -Source Docker -After (Get-Date).AddMinutes(-5) | Sort-Object trên windows server.

- kiểm tra docker có đang chạy hay không

Cách độc lập với OS để kiểm tra docker có đang chạy hay không là hỏi docker sử dụng lệnh docker info. Ngoài ra cũng có thể sử dụng các lệnh khác của hệ thống như sudo systemctl is-active docker hoặc sudo status docker hoặc sudo service docker status

b. Cấu hình docker với systemd

- Start docker daemon
  + systemctl: `$ sudo systemctl start docker`
  + servcie: `$ sudo service docker start`
- Start tự động khi hệ thống reboot
  + systemctl: `$ sudo systemctl enable docker`
  + servcie: chkconfig docker on

4.4 Cấu hình container

a. Cấu hình container start tự động

Docker có 1 số policy để kiểm soát việc container start tự động khi chúng exit hoặc khi Docker restart. các policy restart phải đảm bảo các container được liên kết được start theo đúng trình tự. Docker khuyến nghị sử dụng các restart policy, tránh sử dụng công cụ quản lý tiến trình để start container (service, systemctl).

Các restart policy là khác với --live-store flag của lệnh dockerd, sử dụng --live-store cho phép giữ các container running trong khi Docker đang upgrade dù networking hay đầu vào của user bị ngắt.

- Sử dụng restart policy: Để cấu hình restart policy cho container, sử dụng --restart flag cùng lệnh docker run, và giá trị của --restart flag là:

Flag	Mô tả
no	Không tự động restart container (default)
on-failure	Restart khi container bị lỗi
always	Restart container nếu nó dừng, nếu stop container thủ công thì nó chỉ restart khi Docker daemon restart hoặc container được restart thủ công
unless-stopped	Tương tự always, ngoại trừ khi container stop (stop thủ công hay cách nào đó), nó cũng không restart ngày cả sau khi Docker daemon restart.

- Chi tiết về restart policy:
  + 1 restart policy chỉ có hiệu lực khi container start thành công. Start thành công có nghĩa là container run được ít nhất 10s và Docker bắt đầu giám sát nó. Điều này sẽ ngăn container bị loop restart
  + Nếu stop container thủ công, thì restart policy bị bỏ qua cho đến khi Docker daemon restart hoặc container được restart thủ công. Điều này cũng là để ngăn việc restart loop.
  + Restart policy chỉ áp dụng đối với container, restart policy cho swarm service được cấu hình khác
- Sử dụng công cụ quản lý tiến trình

Nếu các policy không đáp ứng được nhu cầu của bạn như trường hợp các process bên ngoài Docker phụ thuộc Docker container, ta có thể sử dụng các tiến trình quản lý upstart, systemd, supervisor.

Note: Không nên sử dụng kết hợp giữa các policy restart của docker với công cụ quản lý tiến trình của host bởi vì nó sẽ tạo ra trùng lặp.

  + Để sử dụng công cụ quản lý tiến trình , cấu hình nó để start cùng container hoặc service.
  + Sử dụng công cụ quản lý tiến trình trong container

    Các công cụ quản lý tiến trình cũng có thể run trong container để kiểm tra 1 process có chạy hay không và start/stop nó.

b. Duy trì container running khi daemon down

Mặc định, khi Docker daemon stop, nó sẽ shutdown các container đang running, bắt đầu từ docker 1.12 ta có thể cấu hình daemon để các container vẫn running nếu daemon unavailable. Chức năng này được gọi là live restore. Lưu ý chức năng live restore không có trên windows container nhưng nó lại làm việc trên linux container chạy trên Docker desktop for windows.

- enable live restore

có 2 cách để enable live restore setting , chỉ thực hiện 1 trong 2 cách:

- add cấu hình vào trong file daemon.json, sử dụng chuỗi json sau để enable live-restore

```
{
  "live-restore": true
}
```

```
restart hoặc reload cấu hình để cấu hình có hiệu lực
```

- Ta có thể start tiến trình dockerd kết hợp với --live-restore flag. Phương pháp này không được khuyến nghị

- live store khi upgrade

live restore hỗ trợ duy trì các container running trong khi Docker daemon upgrade, mặc dù chức năng này bị hạn chế đối với các bản vá (patch release), và không hỗ trợ đối với minor và major damon upgrade.

Nếu ta bỏ qua các release trong khi upgrade, daemon không thể khôi phục lại được kết nối của nó với các container, và nếu daemon không khôi phục được kết nối nó không thể quản lý được các container đang running và ta phải stop nó thủ công.

- live restore khi restart

Live restore chỉ thực hiện khôi phục các container nếu như các option của daemon như, địa chỉ IP của bridge, graph driver không thay đổi. Nếu có bất kỳ option cấu hình mức daemon nào thay đổi thì live restore sẽ không làm việc và ta phải stop các container thủ công.

- ảnh hưởng của live restore

Nếu daemon down một khoảng thời gian dài, các container đang running sẽ lấp đầy FIFO log mà daemon thường đọc, khi log full nó sẽ block các container ghi thêm dữ liệu logging. mặc định buffer size là 64k, nếu bộ nhớ này đầy ta phải restart docker daemon để giải phóng nó.

Trên linux ta có thể điều chỉnh buffer-size của kernel bằng cách thay đổi file /proc/sys/fs/pipe-max-size. Trên windows hay MacOS không hỗ trợ việc này.

- live restore và swarm mode

live restore chỉ thích hợp với các container độc lập, không phù hợp với swarm service. swarm service được quản lý bởi swarm managers. Nếu swarm manager không available, các swarm service vẫn running trên các node worker nhưng không được quản lý cho đến khi manager available.

c. run nhiều service trong 1 container

https://docs.docker.com/config/containers/multi-service_container/

Khuyến nghị mỗi service chạy 1 container, anh em muốn tìm hiểu thì tham khải ở link trên

d. các metrics runtime của container

https://docs.docker.com/config/containers/runmetrics/

- Docker stats

Sử dụng lệnh docker stats để live stream các tham số runtime của container. ví dụ

```
$ docker stats redis1 redis2
```

- Control groups

linux container dựa vào Control groups không chỉ track groups các process mà còn biểu diễn các tham số về CPU, memory, việc sử dụng block IO.

Control groups nằm ở thư mục /sys/fs/cgroup. Để tìm control groups được mount ở đâu sử dụng lệnh:

```
$ grep cgroup /proc/mounts
```

- Enumerate cgroups
- Find the cgroup for a given container
- Metrics from cgroups: memory, CPU, block I/O
- Network metrics
- Tips for high-performance metric collection
- Collect metrics when a container exits

e. Hạn chế tài nguyên của container

Mặc định các container không bị hạn chế tài nguyên. Container có thể sử dụng toàn bộ tài nguyên mà kernel được cung cấp.

- Memory

+ Rủi ro khi out of memory: trên linux nếu hệ thống phát hiện không đủ memory để thực hiện các chức năng quan trọng của hệ thống, nó sẽ đưa ra lỗi OOME, và bắt đầu kill các process để giải phóng memory. Bất kỳ tiến trình nào bị kill,có thể khiến hệ thống down nếu như tiến trình bị kill gặp lỗi.

  Docker cho phép giảm thiểu nguy cơ này bằng cách điều chỉnh mức ưu tiên OOM trong Docker daemon để nó ít khả năng bị kill hơn so với các tiến trình khác trên hệ thống. Mức ưu tiên OOM trên container không thể điều chỉnh được. Mức ưu tiên này cho phép kill các container riêng lẻ hoặc các tiến trình khác của hệ thống thay vì kill docker daemon. Ta có thể giảm thiểu rủi ro mất ổn định hệ thống do OOM bằng cách:

  - thực hiện test để hiểu các yêu cầu về memory cho ứng dụng của bạn trước khi đưa lên production
  - đảm bảo ứng dụng của bạn chạy trên 1 host có đủ tài nguyên
  - giới hạn lượng memory container có thể sử dụng
  - nên cấu hình swap trên Docker host
  - nên chuyển container thành service, và sử dụng các hạn ngạch mức service và nhãn của node để đảm bảo ứng dụng chỉ chạy trên các host có đủ memory.
+ Giới hạn truy cập của container tới memory: Docker cung cấp các hạn ngạch memory cứng để các container không sử dụng quá memory được cấp, hoặc giới hạn mềm cho phép các container có thể sử dụng memory vượt quá giới hạn cho phép nếu đáp ứng được điều kiện nào đó. Hầu hết các option đều là 1 số nguyên, theo sau là ký tự biểu thị đơn vị của memory: b, k ,m ,g tương ứng byte, kilobyte, megabyte và gigabyte.

option	Mô tả
`-m` or `--memory=`	Lượng memory lớn nhất mà container có thể sử dụng, giá trị tối thiểu là 4m
`--memory-swap`*	Lượng memory container này được phép swap to disk (cho phép container sử dụng bộ nhớ swap khi memory hệ thống hết)
`--memory-swappiness`	Mặc định, host kernel có thể swap tỉ lệ pages ẩn danh được sử dụng bởi container. Có thể thiết lập giá trị này từ 0-100.
`--memory-reservation`	cho phép xác định giới hạn mềm nhỏ hơn --memory được actived khi docker nhận thấy sự tranh chấp hoặc memory thấp trên host machine
`--kernel-memory`	Lượng memory của kernel lớn nhất mà container có thể sử

	dụng, giá trị tối thiểu là 4m,
`--oom-kill-disable`	Mặc định, khi xảy ra OOM, kernel sẽ kill hết các tiến trình của trong container. để thay đổi cấu hình này sử dụng option `--oom-kill-disable`

- CPU

Mặc định, mỗi container truy cập vào CPU của host machine không giới hạn. Ta có thể cấu hình hạn ngạch để hạn chế việc này. Hầu hết user sử dụng và cấu hình CSF schedule mặc định. từ Docker 1.13 trở đi ta có thể cấu hình realtime schedule.

+ Cấu hình CSF schedule mặc định: CSF là 1 CPU schedule của nhân linux cho các tiến trình linux thông thường. có một số runtime flag cho phép ta cấu hình số lượng truy cập tới tài nguyên CPU mà container có. khi ta sử dụng cấu hình này, Docker chỉnh sửa các cài đặt với cgroups của container trên host machine.

option	mô tả
`--cpus=<value>`	Xác định lượng tài nguyên cpu available mà container có thể sử dụng. Ví dụ nếu host có 2 CPU ta có thể thiết lập --cpus="1.5"
`--cpu-period=<value>`	Xác định chu kỳ CSF schedule của CPU, được sử dụng cùng với --cpu-quota. Mặc định là 100 micro giây. nếu sử dụng docker 1.13 hoặc cao hơn, sử dụng --cpus thay thế.
`--cpuset-cpus`	Giới hạn cpu hoặc core container sử dụng, giá trị của nó list các CPU và tách nhau bằng dấu phẩy (1,3,4) hoặc 1 dải CPU (1-3).
`--cpu-shares`	Thiết lập giá trị này thành lớn hơn hoặc nhỏ hơn 1024 để tăng hoặc giảm weight của container và cho phép nó truy cập với tỉ lệ lớn hơn hoặc nhỏ hơn chu trình CPU của machine.
`--cpu-quota=<value>`	Tận dụng hạn ngạch CSF CPU trên container. là số microseconds trên --cpu-period mà container bị giới hạn trước khi bị nghẽn. Với docker 1.13 hoặc cao hơn sử dụng --cpus thay thế.

Nếu ta có 1 CPU, sử dụng lệnh sau sẽ đảm bảo container sử dụng 50% CPU mỗi giây

Với docker 1.13 hoặc cao hơn:

```
docker run -it --cpus=".5" ubuntu /bin/bash
```

Với docker dưới 1.13

```
$ docker run -it --cpu-period=100000 --cpu-quota=50000 ubuntu /bin/bash
```

+ Cấu hình realtime schedule:

Cảnh báo: CPU scheduling và mức độ ưu tiên là 2 thuộc tính cao cấp mức kernel. Hầu hết người sử dụng không cần thay đổi giá trị này so với giá trị mặc định. Nếu thiết lập giá trị này không chính xác có thể khiến cho host system không ổn định.

Cấu hình machine kernel của host : kiểm tra CONFIG_RT_GROUP_SCHED đã được enabled trong linux kernel sử dụng lệnh zcat /proc/config.gz | grep CONFIG_RT_GROUP_SCHED hoặc check trong file /sys/fs/cgroup/cpu.rt_runtime_us.

Cấu hình Docker daemon: để run container sử dụng realtime scheduler, chạy docker daemon với --cpu-rt-runtime flag thiết lập số microsecond lớn nhất dành cho những nhiệm vụ realtime cho tường chu kỳ runtime. ví dụ giá trị mặc định là 1000000 microsecond (1s), thiết lập --cpu-rt-runtime=950000 đảm bảo rằng các container sử dụng realtime scheduler có thể run khoảng 950000 microsecond cho mỗi chu kỳ 1000000 microseconds, dư ít nhất 50000 microseconds cho những nhiệm vụ non-realtime.

Cấu hình container riêng: ta có thể sử dụng một số flag để kiểm soát mức ưu tiên CPU của container khi ta start container.

option	Mô tả
`--cap-add=sys_nice`	Cấp cho container khả năng CAP_SYS_NICE, chúng cho phép container tăng giá trị nice của process, thiết lập các policy realtime scheduling, ...
`--cpu-rt-runtime=<value>`	giá trị lớn nhất tính bằng microseconds container có thể run ở mức ưu tiên real-time trong chu kỳ real-time scheduler của Docker daemon. Ta cũng cần `--cap-add=sys_nice flag.`
`--ulimit rtprio=<value>`	Mức độ ưu tiên real-time lớn nhất cho phép đối với container. Ta cũng cần `--cap-add=sys_nice flag.`

ví dụ:

```
$ docker run -it --cpu-rt-runtime=950000 \
                 --ulimit rtprio=99 \
                 --cap-add=sys_nice \
                 debian:jessie
```

f. logging

https://docs.docker.com/config/containers/logging/configure/

Docker cung cấp nhiều cơ chế logging cho phép ta lấy thông tin từ các container hoặc service đang running. những cơ chế này được gọi là logging drivers. Mỗi container đều có logging driver mặc định và container sẽ sử dụng nó nếu như ta không cấu hình logging driver khác.

- Cấu hình logging driver mặc định: Để cấu hình logging driver mặc định ta thiết lập giá trị log-drive với tên của logging driver trong file cấu hình daemon.json. ví dụ ta muốn cấu hình logging driver là syslog ta thiết lập file cấu hình như sau:

```
{
  "log-driver": "syslog"
}
```

Ngoài ra ta có thể thiết lập thêm các option cho logging driver với key log-opts:

```
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3",
    "labels": "production_status",
    "env": "os,customer"
  }
}
```

Để kiểm tra logging driver mặc định hiện tại của docker daemon sử dụng lệnh sau:

```
$ docker info --format '{{.LoggingDriver}}'
```

- Cấu hình logging driver cho container

Khi ta start container, ta có thể cấu hình logging driver khác với mặc định sử dụng --log-driver flag và --log-opt <name>=<value> flag nếu có các option kèm theo.
Và để kiểm tra cấu hình logging driver của container cụ thể sử dụng lệnh:

```
$ docker inspect -f '{{.HostConfig.LogConfig.Type}}' <CONTAINER>
```

- Cấu hình delivery mode các log message từ container đến logging driver

Docker cung cấp 2 cách để chuyển các bản tin logs từ container đến logging driver:

+ Blocking delivery, chuyển trực tiếp (mặc định) từ container đến logging driver
+ non-blocking delivery, tức là lưu trữ log message trong 1 bộ nhớ đệm trung gian và driver sẽ sử dụng.

Note: Khi bộ nhớ đệm bị tràn hoặc các message mới không nằm trong queue thì các message cũ sẽ bị drop.

mode log option sẽ quyết định việc sử dụng cách chuyển message từ container đến logging driver là blocking hay non-blocking.
max-buffer-size log option điều khiển size của ring buffer được sử dụng để lưu trữ message khi mode được thiết lập là non-blocking. Mặc định giá trị này là 1 MB.
ví dụ: thiết lập cơ chế non-blocking và thiết lập max-buffer-size là 4 MB

```
$ docker run -it --log-opt mode=non-blocking --log-opt max-buffer-size=4m
alpine ping 127.0.0.1
```

- Sử dụng biến môi trường hoặc labels với logging driver

Một số logging driver thêm giá trị `--env|-e` Hoặc `--label` flag của container vào logging driver. ví dụ sau cấu hình logging driver mặc định nhưng sử dụng biến môi trường là `os=ubuntu.`

```
$ docker run -dit --label production_status=testing -e os=ubuntu alpine sh
```

- Các logging driver được hỗ trợ

Driver	Mô tả
`none`	Không lưu logs của container, đầu ra của lệnh docker logs là none
`local`	logs được lưu trữ trong 1 định dạng tùy chỉnh
`json-file`	Log được định dạng là JSON (mặc định)
`syslog`	ghi log message tới syslog, syslog cần được cài đặt trên host
`journald`	ghi logs ra journald, cài đặt journald trên host machine
`gelf`	Ghi logs ra Graylog hoặc Logstash
`fluentd`	Ghi logs ra `fluentd`, `fluentd` daemon phải running trên host
`awslogs`	Ghi logs ra Amazon CloudWatch Logs
`splunk`	Ghi logs ra splunk sử dụng HTTP Event Collector
`etwlogs`	Ghi logs ra Event Tracing for Windows (ETW), chỉ trên windows
`gcplogs`	Ghi logs ra Google Cloud Platform (GCP) Logging
`logentries`	Ghi logs ra Rapid7 Logentries

Với docker community engine, lệnh docker logs chỉ available với các logging driver sau:

+ `local`
+ `json-file`
+ `journald`

- xem logs của container hoặc service

sử dụng lệnh docker logs hoặc docker service logs

- sử dụng docker logs với logging driver
- Tùy chỉnh output của log driver

`tag` log option chỉ định cách định dạng một tag để nhận biết log message của container. Mặc định hệ thống sử dụng 12 ký tự đầu tiên của container ID. Để ghi đè giá trị này sử dụng tag option:

```
$ docker run --log-driver=fluentd --log-opt fluentd-address=myhost.local:24224 --log-opt tag="mailer"
```

Docker hỗ trợ một số thẻ template có thể sử dụng khi xác định giá trị của tag:

Thẻ	Mô tả
`{{.ID}}`	12 ký tự đầu tiên của contaienr
`{{.FullID}}`	full container ID
`{{.Name}}`	Tên container
`{{.ImageID}}`	12 ký tự đầu tiên image ID của container
`{{.ImageFullID}}`	full image ID của container
`{{.ImageName}}`	Tên image mà container sử dụng
`{{.DaemonName}}`	Tên của docker program

Ví dụ cấu hình tag là `--log-opt tag="{{.ImageName}}/{{.Name}}/{{.ID}}"` output trên sysload là

```
Aug 7 18:33:19 HOSTNAME hello-world/foobar/5790672ab6a0[9103]: Hello from Docker.
```

Khi hệ thống đã startup và ta đã cấu hình 2 trường container_name và {{.Name}} cho tag thì khi ta rename của container thì tên mới sẽ không có trong logs.

- chi tiết về logging driver

g. security

4. 5. Scale app

https://docs.docker.com/engine/swarm/

4.5.1. Tổng quan về swarm mode

Các thuộc tính nổi bật:

- Quản lý cluster được tích hợp với docker engine: Có thể sử dụng docker engine CLI để tạo và quản lý swarm mà không cần cài đặt công cụ khác.
- Thiết kế phân cấp: thay vì xử lý sự khác nhau về vai trò giữa các nodes ở thời điểm deployment thì docker engine xử lý bất kỳ tại thời điểm runtime. Ta có thể deploy trên cả node manager và worker sử dụng docker engine. Tức là có thể build toàn bộ swarm với 1 image.
- Xác định mô hình của service: Docker sử dụng một phương pháp tiếp cận để định nghĩa trạng thái mong muốn của nhiều service trong stack ứng dụng của mình . Ví dụ ta có thể mô tả ứng dụng của mình gồm có 1 service frontend web với các service message queueing và 1 database backend.
- scaling: Mỗi service ta có thể khai báo một vài nhiệm vụ mà mình muốn chạy. khi ta scale up hoặc down, swarm manager sẽ tự động tương thích bằng cách thêm hoặc xóa các task để duy trì trạng thái mong muốn.
- Cân bằng trạng thái mong muốn. node quản lý swarm giám sát liên tục trạng thái cluster và cân bằng bất kỳ sự khác biệt nào giữa trạng thái thực thế và trạng thái mong muốn. ví dụ ta thiết lập service chạy trên 10 replicas của 1 container, và có 2 replicas trên worker crash thì manager sẽ tạo thêm 2 replicas mới để thay thế replicas bị crash, swarm manager sẽ gán cho 2 replicas mới thành worker để running và available.
- multi-host networking: ta có thể xác định 1 overlay network của service. swarm manager sẽ tự động gán địa chỉ tới container trên overlay network khi nó khởi động hoặc update ứng dụng.
- service discovery: swarm manager node sẽ gán cho mỗi service trong swarm 1 name DNS duy nhất và cân bằng tải cho các container đang running. Ta có thể truy vấn từng container đang running trong swarm thông qua DNS server trong swarm.
- Cân bằng tải: Ta có thể mở nhiều port cho các service để cân bằng tải ngoài. bên trong, swarm cho phép bạn xác định cách phân bổ các service container giữa các node.
- bảo mật: Mỗi node trong swarm bắt buộc xác thực lẫn nhau bằng thuật toán TLS, và mã hóa thông tin giao tiếp giữa nó với các node khác. Ta có thể lựa chọn sử dụng chứng thực root tự mình tạo hoặc từ bên cung cấp thứ 3.
- rolling update: Tại thời điểm triển khai ta có thể sử dụng service update để tăng số node. swarm manager cho phép kiểm soát trễ giữa việc triển khai service với 1 tập các node khác nhau. Nếu có bất kỳ thứ gì đó bị sai , ta có thể roll-back 1 task về phiên bản trước đó.

4.5.2. Các khái niệm chính swarm mode.

a. Swarm là gì?

Là những thuộc tính quản lý và điều phối cluster được nhúng trong Docker engine và được xây dựng bằng cách sử dụng swarmkit.

một swarm bao gồm nhiều Docker host chạy ở swarm mode và hoạt động như manager hoặc worker. Một docker host có thể đóng vai là manager, worker hoặc cả hai. khi ta tạo 1 service ta có thể định nghĩa trạng thái tối ưu của nó (số replicas, network, tài nguyên storage available cho nó, các port được mở public, …). Docker thực hiện duy trì trạng thái mong muốn cho các service. chẳng hạn như nếu 1 worker bị chết, Docker sẽ chuyển nhiệm vụ (task) của node đó cho node khác. một task là một container đang running và là 1 phần của swarm và được quản lý bởi swarm manager.

Một trong những ưu thế của swarm so với các container độc lập là nó có thể điều chỉnh được cấu hình của service, bao gồm network, volume nó kết nối tới mà không cần restart service. Docker sẽ update cấu hình, stop các service task có cấu hình hết hạn (out-of-date) và tạo ra một service task với cấu hình mong muốn.

Khi Docker running ở swarm mode, ta vẫn có thể chạy các container độc lập trên bất kỳ Docker host nào tham gia swarm mode. Một sự khác biệt chính giữa các container chạy độc lập và các swam service đó là chỉ swarm manager có thể quản lý swarm trong khi đó các container chạy độc lập có thể được started trên bất kỳ node nào.

b. Nodes

Là 1 instance của docker engine tham gia swarm mode.Có thể xem nó như 1 Docker node. ta có thể chạy nhiều node trên 1 machine host. Nhưng việc triển khai swarm ở môi trường production thường gồm những node được xây dựng trên nhiều machine host. 1 swarm có 1 node là manager, các node còn lại là worker.

c. Các service và task

service là việc định nghĩa của các task để thực thi trên manager hoặc các worker node. đó là kiến trúc của hệ thống swarm.

Khi ta tạo 1 service, cần xác định image nào được sử dụng và lệnh nào sẽ thực thi trong các container đang running.

Trong mô hình replicated service, swarm manager sẽ phân bổ một số các replicas task cụ thể giữa các node và dựa vào scale mà ta thiết lập để có trạng thái mong muốn.

Đối với global service, swarm chạy 1 task cho service ở node luôn available trong cluster.

d. Cân bằng tải

swarm manager sử dụng cân bằng tải ingress để mở cho các service muốn đi ra ngoài. swarm manager sẽ tự động gán các **Publishedport** hoặc ta có thể cấu hình **Publishedport** cho service. Ta có thể chọn bất kỳ port nào chưa được sử dụng. Nếu ta không chọn port thì mặc định swarm sẽ gán các port ở dải 30000-32767.

Các ưng dụng bên ngoài có thể truy cập vào service qua public port của bất kỳ node nào trong swarm đang running task của service. tất cả các node trong swarm sẽ định tuyến các kết nối ingress tới các instance có task đang running.

Swarm mode có 1 thành phần DNS nội bộ, nó sẽ tự động gán cho từng service trong swarm một entry. Swarm manager sẽ sử dụng cân bằng tải nội bộ để phân bổ các request giữa các service trong cluster dựa vào tên DNS của service.

4.5.3 Start với swarm mode.

https://docs.docker.com/engine/swarm/swarm-tutorial/

Gồm các công việc từ khởi tạo đến remove service:

- Chuẩn bị cấu hình: cài đặt 3 node Docker engine 1.12 hoặc cao hơn, mở các port cần thiết
- tạo swarm
- Add các node vào swarm
- Deploy service
- Kiểm tra service
- scale service: số container chạy trong 1 service được gọi là “tasks”:

```
$ docker service scale <SERVICE-ID>=<NUMBER-OF-TASKS>
```

- delete service
- rolling updates
- Drain 1 node trong swarm: Đôi khi những thời điểm cần bảo trì hệ thống, ta cần thiết lập 1 node thành DRAIN, drain sẽ ngăn 1 node nhận các task mới từ swarm manager. nó cũng có nghĩa là manager sẽ stop các task đang running trên node đó và chuyển các replicas task cho node khác ACTIVE.

```
$ docker node update --availability drain worker1
```

```
Sau khi hoàn thành update thì active lại:
```

```
$ docker node update --availability active worker1
```

- Sử dụng routing mesh ở swarm mode:
  + Public port cho service

```
$ docker service create \
  --name <SERVICE-NAME> \
  --publish published=<PUBLISHED-PORT>,target=<CONTAINER-PORT> \
  <IMAGE>
```

  + Public port cho service đang running

```
$ docker service update \
  --publish-add published=<PUBLISHED-PORT>,target=<CONTAINER-PORT> \
  <SERVICE>
```

hoặc:

$ docker service inspect --format="{{json .Endpoint.Spec.Ports}}" my-web

[{"Protocol":"tcp","TargetPort":80,"PublishedPort":8080}]

  + public port chỉ tcp/udp

tcp:

```
$ docker service create --name dns-cache \
  -p 53:53 \
  dns-cache
```

udp:

```
$ docker service create --name dns-cache \
  -p 53:53/udp \
  dns-cache
```

cả tcp/udp

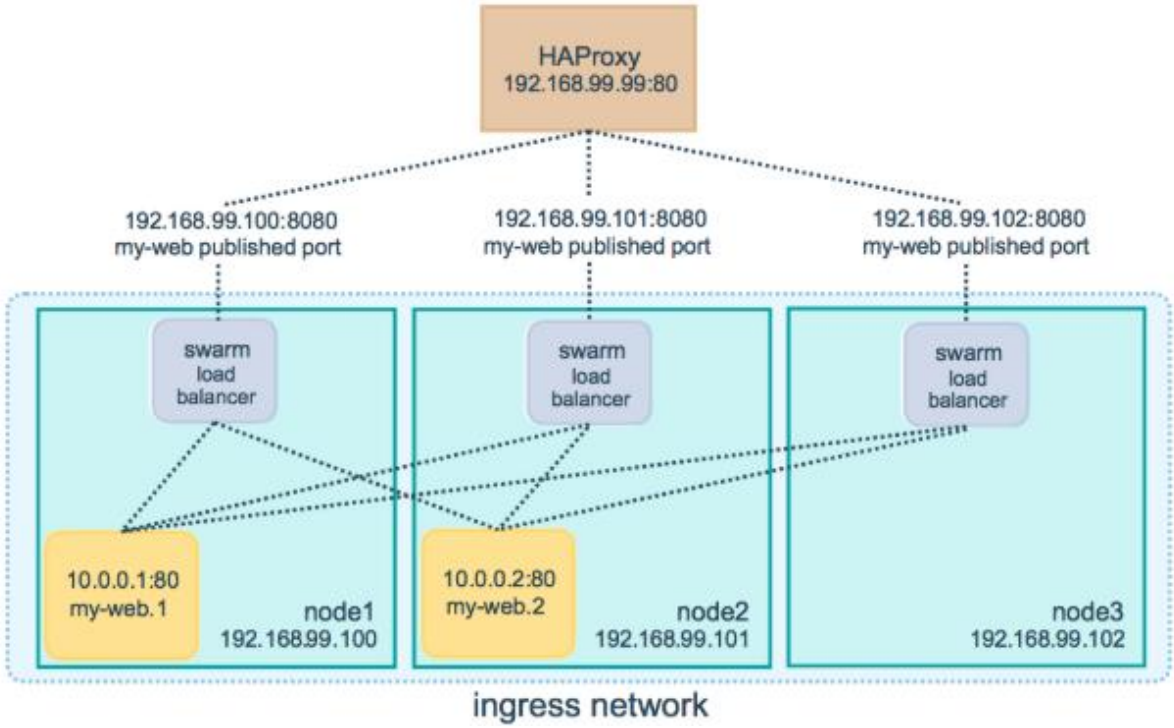
```
$ docker service create --name dns-cache \
  -p 53:53 \
  -p 53:53/udp \
  dns-cache
```

  + Bypass routing mesh

```
$ docker service create --name dns-cache \
  --publish published=53,target=53,protocol=udp,mode=host \
  --mode global \
  dns-cache
```

  + Cấu hình cân bằng tải ngoài

Sử dụng routing mesh: Ta có thể cấu hình sử dụng phần mềm cân bằng tải của bên thứ 3 để định tuyến các request đến các service. ví dụ có thể cấu hình haproxy để cân bằng các request tới các service nginx public port 8080.



Nội dung file cấu hình haproxy:

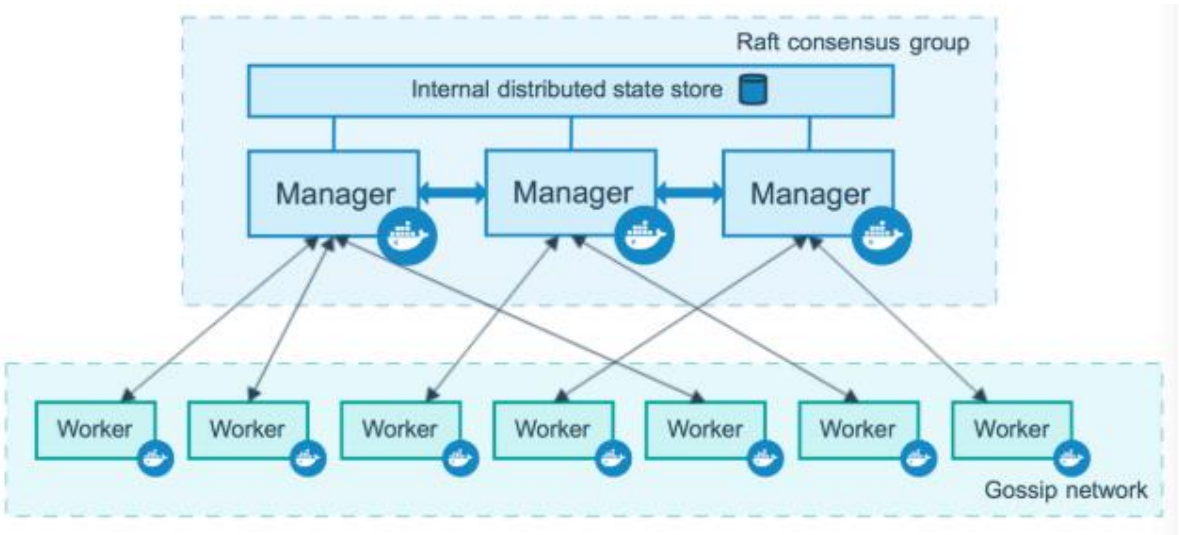
```
global
        log /dev/log    local0
        log /dev/log    local1 notice
...snip...

# Configure HAProxy to listen on port 80
frontend http_front
   bind *:80
   stats uri /haproxy?stats
   default_backend http_back

# Configure HAProxy to route requests to swarm nodes on port 8080
backend http_back
   balance roundrobin
   server node1 192.168.99.100:8080 check
   server node2 192.168.99.101:8080 check
   server node3 192.168.99.102:8080 check
```

Không sử dụng routing mesh: ta set --endpoint-mode thành dnsrr thay vì giá trị vip mặc định.

4.5.4 Cách các mode trong swarm làm việc.

1. Cách các node làm việc

Có 2 loại node là manager và worker



a. Manager node:

Để tận dụng ưu điểm chịu lỗi của swarm mode, Docker khuyến nghị nên sử dụng số node là số lẻ. Khi có nhiều manager, ta có thể khôi phục sự số của node manager mà không làm downtime.

- 1 swarm có 2 node manager chịu được lỗi khi mất tối đa 1 manager
- 5 manager node sẽ chịu được lỗi khi mất đồng thời 2 node manager
- N manager node sẽ chịu được lỗi khi mất (N-1)/2 node manager
- Docker khuyến nghị nên dùng 7 manager trong 1 swarm.

Note: add nhiều manager không có nghĩa là tăng scale hoặc hiệu suất.

b. Worker node

c. thay đổi vai trò

Ta có thể chuyển 1 worker thành 1 manager sử dụng câu lệnh: `docker node promote`

2. Cách các service làm việc

Để triển khai 1 image ứng dụng trong swarm mode, ta tạo 1 service. Thường 1 service là 1 image đối với microservice chứa nội dung một số ứng dụng lớn.

Khi ta tạo 1 service, cần xác định image cho container và những lệnh được thực thi bên trong container, cũng có thể định nghĩa một số option trong service:

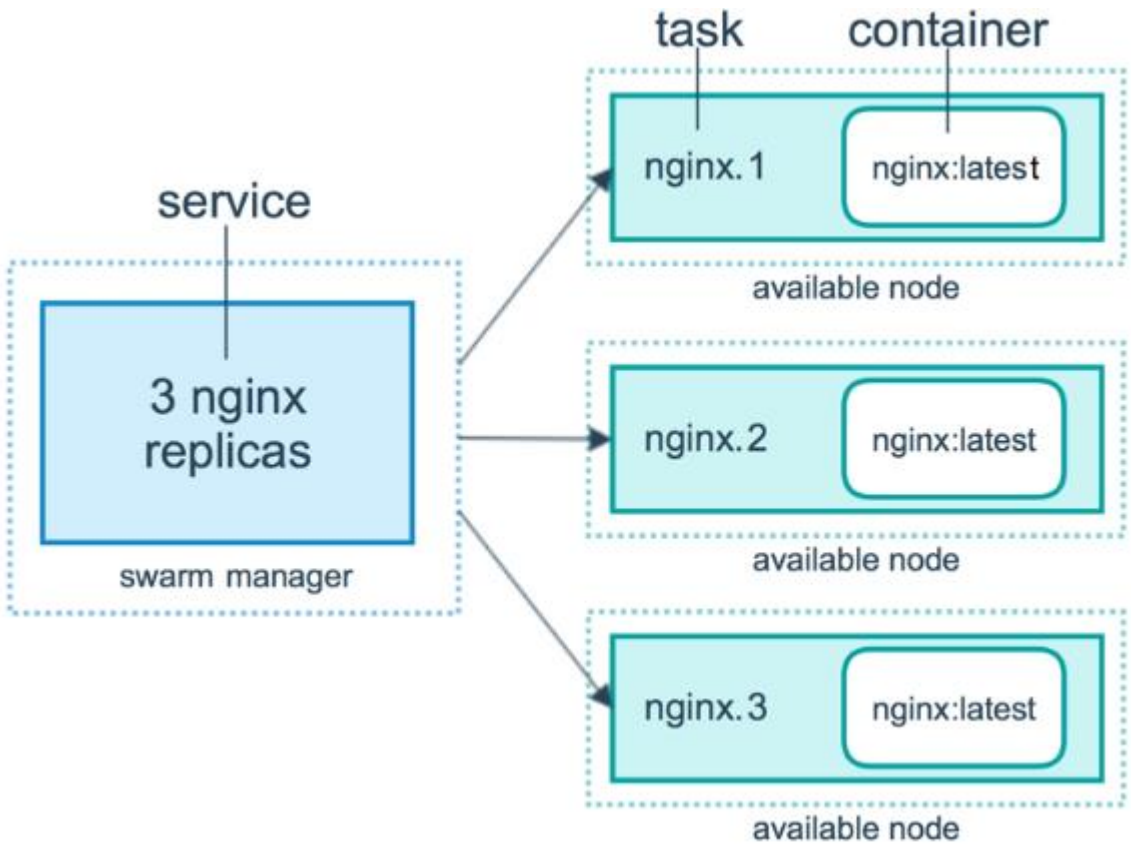
- port

- overlay network
- giới hạn cpu, memory
- số replicas

a. Service, task và container

Khi ta triển khai 1 service trong swarm, swarm manager sẽ chấp nhận các định nghĩa service và trạng thái mong muốn của service. Sau đó nó lên lịch cho service trên các node trong swarm là một hay nhiều replicas task. Các task sẽ run độc lập với các node khác trong swarm.

Ví dụ ta muốn cân bằng tải giữa 3 instance của một http listen. Như hình dưới cho thấy 1 service http listener với 3 replicas, các instance của listener là 1 task trong swarm:



1 container là 2 process độc lập, trong swarm mode, mỗi task gọi chính xác 1 container. 1 task tương tự là 1 "slot" trong đó scheduler đặt một container. Khi container sống, scheduler nhận thấy task đó đang ở trạng thái running . Nếu container lỗi health check hoặc terminal thì task cũng terminal.

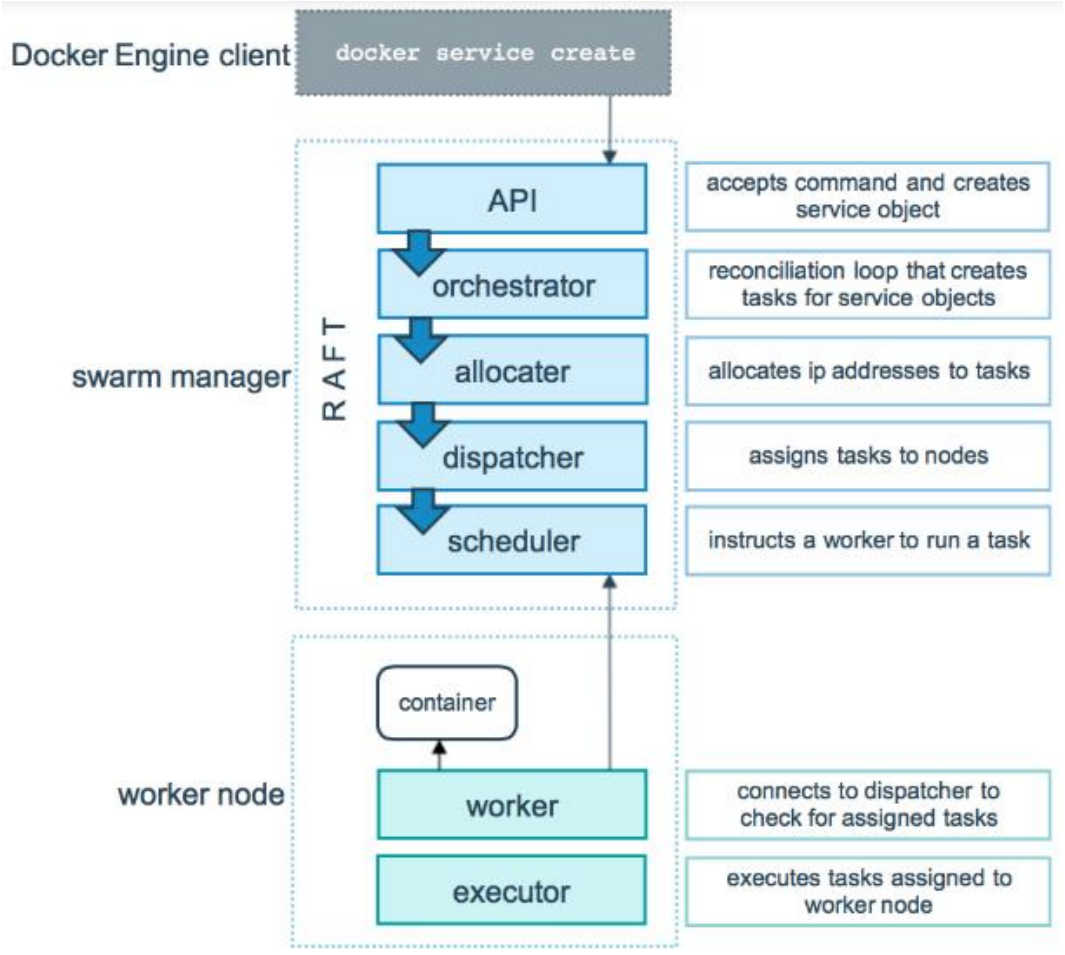
b. Task và scheduling

1 task là 1 đơn vị scheduling của swarm. Khi ta khai báo trạng thái service mong muốn bằng cách tạo hoặc update service, orchestrator nhận thấy trạng thái mong muốn nhờ các scheduling task. ví dụ ta định nghĩa một service chỉ thị cho orchestrator duy trì 3 instance của 1 HTTP

listener đang running ở bất kỳ thời gian nào. orchestrator phản hồi bằng cách tạo 3 task. Mỗi task là 1 slot mà scheduler lấp đầy bằng cách sinh ra 1 container. Nếu một HTTP listener task liên tiếp bị lỗi health check hoặc crash thì orchestrator sẽ tạo 1 replicas task mới để sinh ra container mới.

Task là cơ chế 1 chiều, nó thực thi thông qua 1 loạt các trạng thái: assigned, prepared, running, … Nếu task fail, orchestrator xóa task và container của nó, sau đó tạo 1 task mới để thay thế nó tương ứng với trạng thái mong muốn được xác định bởi service.

Sơ đồ dưới đây sẽ cho thấy cách swarm mode chấp nhận service tạo các task request và schedule task cho các worker như thế nào.



* Pending service

Một service được cấu hình ở trạng thái pending có nghĩa là hiện tại không có node nào trong swarm có thể chạy được task của nó. Một số ví dụ khi nào một service ở trạng thái pending.

Chú ý: Nếu mục đích là ngăn service deployed, ta scale service thành 0 thay vì cấu hình nó thành pending.

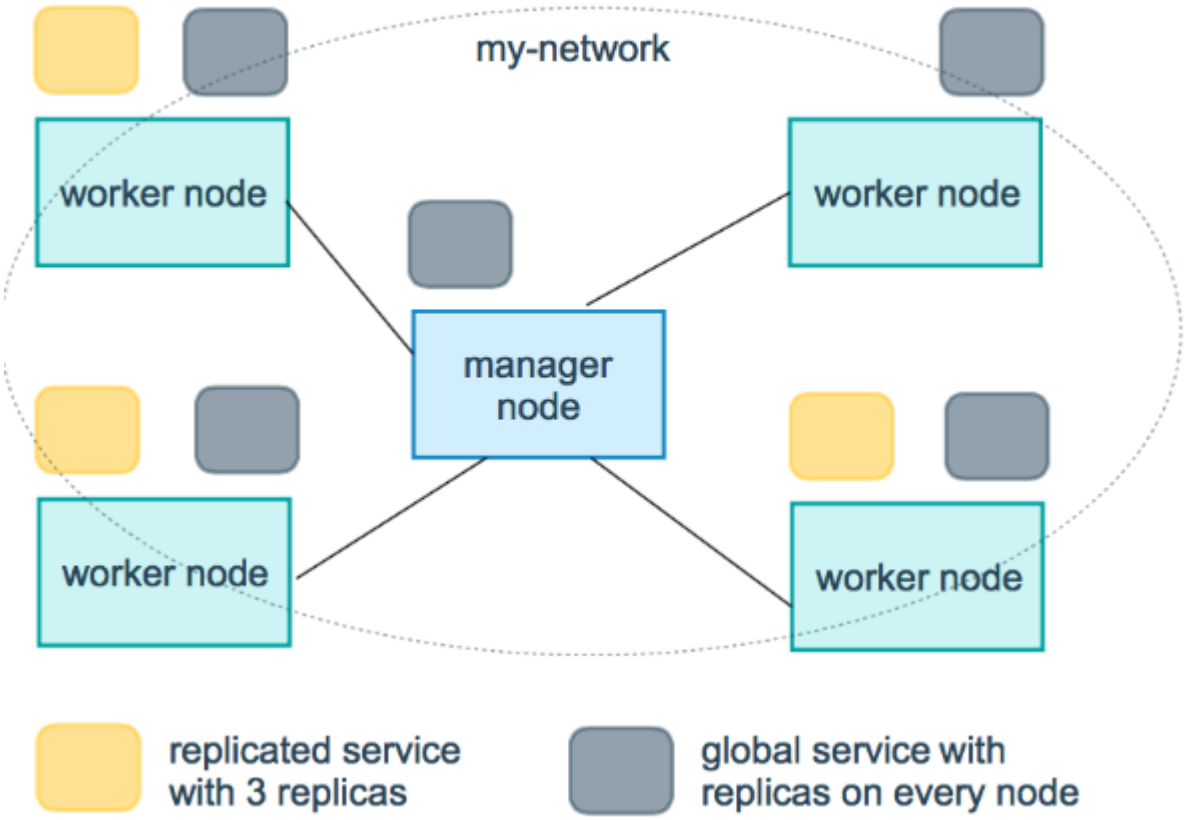
- Nếu tất cả các node paused hoặc drained, và ta tạo 1 service, nó sẽ pending cho đến khi node available. Trong thực tế, node đầu tiên available sẽ nhận tất cả các task, đây là một điều tốt trên môi trường production.
- ta có thể cấp một lượng memory nhất định cho 1 service. Nếu không node nào trong swarm yêu cầu memory, service sẽ duy trì ở trạng thái pending cho đến khi node available và có thể run task của nó. Nếu bạn xác định memory rất lớn (500G), task sẽ ở trạng thái pending mãi mãi, trừ khi có 1 node đáp ứng được yêu cầu này.
- ta có thể áp đặt các ràng buộc về vị trí cho service và những ràng buộc này có thể không có khả năng được thực hiện tại 1 thời điểm nhất định.

c. Deplicated global services.

Có 2 kiểu deployment service là replicated và global.

Đối với replicated service, ta cần phải xác định số task mà mình muốn run. Chẳng hạn ta xác định deploy một HTTP service với 3 replicas, mỗi replicas sẽ phục vụ cùng nội dung.

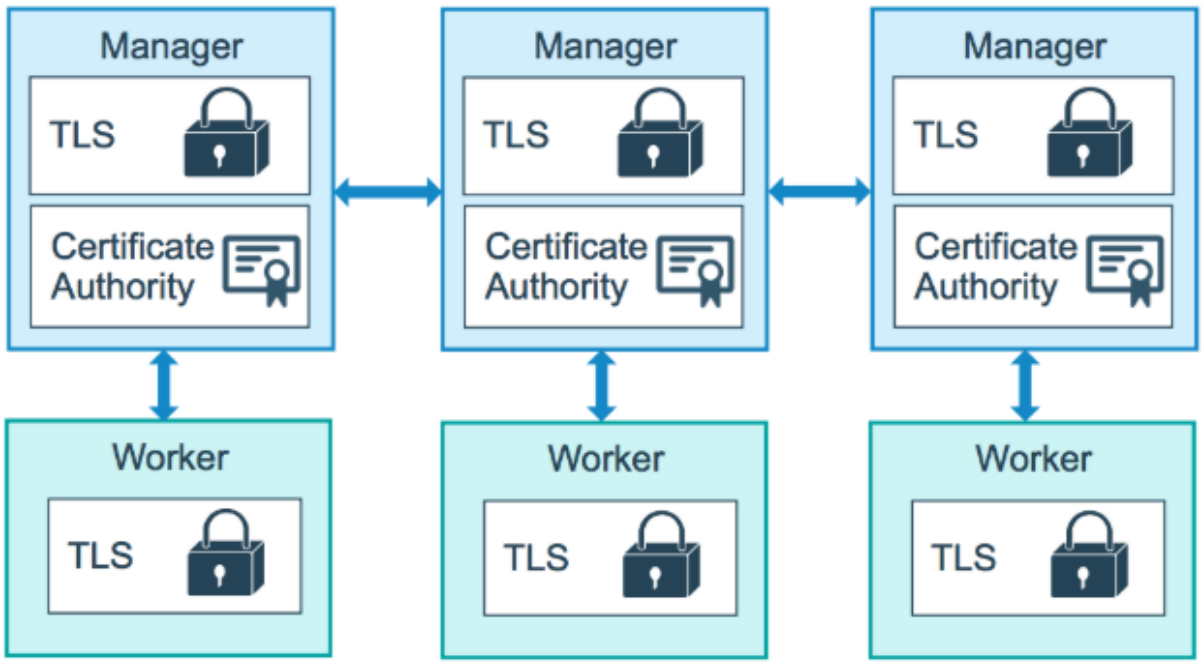
Global service là 1 service sẽ run mỗi task trên tất cả các node. không có số task cụ thể. Mỗi thời điểm add một node vào swarm, orchestrator tạo 1 task và scheduler gán task đó cho node mới. các ứng dụng có thể triển khai kiểu global service như là monitoring, anti-virus, … Sơ đồ dưới đây cho thấy 1 replicas 3 service bằng màu vàng và 1 service global bằng màu xám.



3. Quản lý an toàn swarm bằng hạ tầng public key (KPI)

hệ thống KPI swarm mode được build trong docker để đảm bảo an toàn khi deploy 1 hệ thống điều hướng container. swarm node sử dụng TLS để xác thực, chứng thực và bảo mật thông tin trao đổi giữa các node. Mặc định manager node tạo CA hoặc có thể sử dụng key ngoài bằng cách sử dụng --external-ca flag trong câu lệnh docker swarm init.

Manager node cũng tạo ra 2 token để sử dụng khi ta join 1 node vào swarm, 1 token cho worker, 1 token cho manager. Mỗi khi 1 node join swarm, manager sẽ cấp certificate cho node đó. Hình sau đây mô tả cách manager và worker mã hóa thông tin trao đổi giữa các node sử dụng TLS1.2



Mặc định, mỗi node trong swarm sẽ gia hạn certificate 3 tháng 1 lần. Ta có thể cấu hình thời gian này bằng cách sử dụng lệnh `docker swarm update --cert-expiry <TIME PERIOD>`.

4. Các trạng thái swarm task

Docker cho phép tạo các service chạy các task. 1 service là một mô tả trạng thái mong muốn và task thực hiện công việc. Công việc được lên lịch trên các node của swarm theo thứ tự sau:

- tạo service sử dụng lệnh `docker service create` hoặc UCP web UI hoặc CLI.
- Gửi các request tới manager node
- manager node lên lịch cho các service chạy trên các node cụ thể
- mỗi service có thể run nhiều task
- mỗi task có 1 vòng đời với các trạng thái: New, pending hoặc complete

Các task là các đơn vị thực thi chạy 1 lần cho tới khi hoàn thành, khi task dừng không thể chạy lại task đó nhưng ta có thể tạo 1 task mới thực thi thay thế cho task đó. 1 task sẽ đi qua các trạng thái theo thứ tự sau:

Task state	Mô tả
NEW	task được khởi tạo
PENDING	Các tài nguyên của task được cấp phát
ASSIGNED	Docker gán task cho các node

ACCEPTED	task được chấp nhận bởi worker, nếu worker từ chối task, trạng thái của task là reject
PREPARING	Docker chuẩn bị cho task
STARTING	Docker start task
RUNNING	Docker thực thi task
COMPLETE	task kết thúc không có lỗi
FAILED	task kết thúc bị lỗi
SHUTDOWN	Docker được request task là shutdown
REJECT	worker từ chối task
ORPHANED	Node bị down quá lâu
REMOVE	task chưa kết thúc nhưng service liên quan task bị remove hoặc scaled down.

- Xem trạng thái của task

  Sử dụng câu lệnh sau để xem trạng thái của task:

  ```
  docker service ps <service-name>
  ```

5. Chạy docker ở swarm mode.

a. Tạo swarm

Khi chạy câu lệnh docker swarm init để tạo single-node swarm ở node hiện tại, Engine thiết lập swarm theo trình tự sau:

- switch node hiện tại thành swarm mode
- tạo swarm với tên mặc định
- gán node hiện tại thành manager của swarm
- đặt tên node là hostname của machine
- cấu hình manager listen trên 1 interface network active với port 2377
- Cấu hình node hiện tại thành Active, tức là nó có thể nhận các task từ scheduler
- start một nơi lưu trữ data ở local cho các node tham gia swarm
- mặc định, tạo 1 self-signed root CA cho swarm
- mặc định nó sẽ tạo các tokens cho các manager và worker để join swarm
- tạo 1 overlay network tên là ingress để public các port ra ngoài swarm
- tạo các địa chỉ IP và subnet mask cho network

b. Cấu hình Pool địa chỉ IP mặc định

Mặc định swarm sẽ Pool địa chỉ mặc định là 10.0.0.0/8 cho global scope network (overlay network), không có subnet cụ thể. Để cấu hình Pool mặc định ta phải định nghĩa pool khi khởi tạo swarm sử dụng option `--default-addr-pool`

```
$ docker swarm init --default-address-pool <IP range in CIDR> [--default-address-pool <IP range in CIDR> --default-addr-pool-mask-length <CIDR value>]
```

c. Cấu hình địa chỉ IP cho quảng bá

manager sử dụng địa chỉ quảng bá để cho phép các node khác trong swarm truy cập swarmket API và overlay network. Nếu ta không xác định 1 địa chỉ IP quảng bá, docker kiểm tra nếu hệ thống có 1 địa chỉ IP thì nó sẽ sử dụng địa chỉ IP đó và listen trên port 2377. Nếu hệ thống có nhiều địa chỉ IP, ta phải xác định địa chỉ chính xác sử dụng `--advertise-addr` flag.

```
$ docker swarm init --advertise-addr <MANAGER-IP>
```

d. Xem lệnh join và update swarm join token

để xem lệnh join và token cho các node worker sử dụng lệnh:

```
$ docker swarm join-token worker
```

để xem lệnh join và token cho các node manager sử dụng lệnh:

```
$ docker swarm join-token manager
```

Sử dụng option --quiet để chỉ xem token

```
$ docker swarm join-token --quiet worker
```

Hãy cẩn thận với join tokens bởi vì chúng là các mã bảo mật cần để join swarm. Trên thực tế việc check mã bảo mật này trên phần mềm version control là điều không nên bởi vì nó cho phép bất kỳ ai truy cập vào source code của ứng dụng đều có thể thêm 1 node mới vào swarm đặc biệt là manager tokens. Vì thế khuyến nghị chúng ta nên xoay vòng join tokens và nên thực hiện việc xoay vòng này 6 tháng 1 lần.

Chạy lệnh `swarm join-token --rotate` để vô hiệu hóa các token cũ và tạo các token mới. và xác định loại token mà ta muốn xoay vòng là worker hay manager:

vd: `$ docker swarm join-token  --rotate worker`

6. join node vào swarm

7. deploy service trong swarm

8. lưu trữ data cấu hình service

9. quản lý dữ liệu nhạy cảm bằng Docker secrets

10. Lock swarm

Từ docker 1.13 và cao hơn, mặc định Raft logs được sử dụng bởi swarm manager được mã hóa trên disk. Khi docker restart cả TLS key được sử dụng để mã hóa thông tin trao đổi giữa các node trong swarm và key được sử dụng để mã hóa và giải mã Raft log trên disk được load vào trong memory của từng manager node. từ docker 1.13 giới thiệu khả năng bảo vệ key mã hóa TLS chung, và key được sử dụng để mã hóa và giải mã Raft log còn lại bằng cách cho phép ta sở hữu key này và yêu cầu unlocking manager thủ công. thuộc tính này được gọi là autolock.

Khi restart swarm, trước tiên ta cần unlock swarm trước, bằng cách sử dụng key dùng để mã hóa key được sinh ra bởi docker khi swarm bị lock.

- Khởi tạo swarm và enable autolock: sử dụng lệnh sau:

```
docker swarm init --autolock
```

Lệnh trên sẽ sinh ra 1 key để ta unlock, lưu lại key này. khi ta restart swarm, ta cần unlock swarm. swarm bị lock sẽ sinh ra lỗi sau:

```
$ sudo service docker restart

$ docker service ls
Error response from daemon: Swarm is encrypted and needs to be unlocked
before it can be used. Use "docker swarm unlock" to unlock it.
```

- enable hoặc disable autolock trên swarm đang tồn tại.

  dùng lệnh sau: enable

  ```
  $ docker swarm update --autolock=true
  ```

  disable:

  ```
  $ docker swarm update --autolock=false
  ```

- unlock swarm: dùng lệnh sau:

  ```
  docker swarm unlock
  ```

- Xem key unlock hiện tại của swarm đang running: sử dụng lệnh sau:

  ```
  $ docker swarm unlock-key
  ```

- rotate unlock key:

```
$ docker swarm unlock-key --rotate
```

11. Hướng dẫn quản trị swarm

- Thêm các manager node để tăng khả năng chịu lỗi

Swarm Size	Majority	Fault Tolerance
1	1	0
2	2	0
3	2	**1**
4	3	1
5	3	**2**
6	4	2
7	4	**3**
8	5	3
9	5	**4**

- Phân bổ các node manager

Ngoài việc duy trì số node manager là 1 số lẻ ta cần chú ý tới kiến trúc datacenter khi đặt manager. để khả năng chịu lỗi tối ưu nhất, cần phân bổ manager tối thiểu trên 3 zones. Nếu bạn bị lỗi trên bất kỳ zone nào swarm nên duy trì hạn mức số các node manager available để xử lý các request và tái cân bằng tải workloads.

Swarm manager nodes	Repartition (on 3 Availability zones)
3	1-1-1
5	2-2-1
7	3-2-2
9	3-3-3

- Chạy node chỉ làm chức năng manager

sử dụng lênh sau:

```
docker node update --availability drain <NODE>
```

- add worker để cân bằng tải
- Moniter swarm: sử dụng một số lệnh sau để check status, reachable, …

```
docker node inspect manager1 --format "{{ .ManagerStatus.Reachability }}"

reachable

docker node inspect manager1 --format "{{ .Status.State }}"

ready
```

Nếu trạng thái là unreachable ta cần thực hiện các action cần thiết để khôi phục trạng thái available

  + restart daemon
  + reboot machine
  + nếu cả 2 cách trên không hiệu quả thì ta cần add thêm manager hoặc nâng cấp 1 worker thành manager, đồng thời remove node bị lỗi.

- Troubleshoot manager: Không bao giờ restart 1 manager node bằng cách copy thư mục raft từ 1 node khác vì thư mục data có nodeID là duy nhất.

Để re-join node manager vào swarm:

  + chuyển node thành worker
  + remove node khỏi swarm
  + re-join node vào swarm

- Remove cưỡng bức 1 node khởi swarm

dùng lệnh:

```
$ docker node rm --force node9
```

- Backup swarm

Manager lưu trữ logs trạng thái và manager trong thư mục /var/lib/docker/swarm. dữ liệu này gồm các keys được sử dụng để mã hóa Raft log. không có những key này không thể khôi phục swarm. Backup swarm có thể thực hiện ở bất kỳ manager nào theo quy trình sau:

  + nếu swarm bật tính năng auto-lock, cần key unlock để khôi phục swarm từ backup.

+ stop Docker trên manager trước khi backup data để không cho docker ghi thêm dữ liệu vào hệ thống. Khi manager down, các node khác sẽ tiếp tục sinh ra swarm data không nằm trong data backup này.
+ backup toàn bộ thư mục /var/lib/docker/swarm
+ restart manager

- Khôi phục sau sự cố
  + Khôi phục từ bản backup: Quy trình như sau:
    - shutdown docker trên target host cần khôi phục swarm
    - remove thư mục /var/lib/docker/swarm trên swarm mới
    - Khôi phục thư mục /var/lib/docker/swarm với nội dung của file backup (chú ý: node mới cần sử dụng cùng key mã hóa cho on-disk storage như node cũ. Trong trường hợp auto-lock cũng cần key unlock trên node cũ.)
    - start Docker trên node mới, unlock swarm nếu cần, khởi tạo lại swarm sử dụng lệnh sau để node không kết nối tới các node của swarm cũ

      ```
      $ docker swarm init --force-new-cluster
      ```

    - Kiểm tra trạng thái của swarm mới đã như mong muốn: docker service ls
    - Nếu sử dụng auto-lock nên rotate unlock key
    - add các node manager và worker
    - Khôi phục lại chế độ sao lưu trước đó trên swarm mới.
  + Khôi phục khi mất quorum

    Swarm có khả năng khôi phục lỗi và swarm cũng có khả năng khôi phục từ bất kỳ node bị lỗi tạm thời nào (machine reboot hoặc crash khi restart) nhưng không thể tự động khôi phục nếu như mất quorum. Cách tốt nhất để khôi phục là đêm node manager bị lỗi online trở lại, nếu không thể cách duy nhất để khôi phục là sử dụng action `--force-new-cluster` trên node manager.

    ```
    # From the node to recover

    docker swarm init --force-new-cluster --advertise-addr node01:2377
    ```

  + Yêu cầu swarm tái cân bằng tải lại: Nhìn chung là không cần bắt swarm cân bằng tải lại task của nó bởi vì khi add 1 node mới vào swarm, hoặc node sẽ kết nối lại swarm sau khoảng thời gian unavailable, swarm sẽ không tự động cung cấp workload cho node rỗi. từ docker 1.13 ta có thể sử dụng -f (--force) flag với câu lệnh docker service update để bắt service phân bổ lại task của nó trên các node worker available.

a. mở rộng docker

https://docs.docker.com/engine/extend/

1. Hệ thống plugin quản lý
2. plugin xác thực truy cập
3. mở rộng docker bằng các plugin
4. Driver network plugin
5. Volume plugin
6. Plugin cấu hình
7. Plugin API

V. Hướng dẫn viết Dockerfile

https://docs.docker.com/engine/reference/builder/

Dockerfile có thể build image tự động bằng cách đọc các hướng dẫn từ Dockerfile. Dockerfile là 1 file text chứa tất cả các câu lệnh user sử dụng trên CLI để xây dựng 1 image.

Hầu hết các Dockerfile đều start từ 1 image cha. Nếu ta muốn kiểm soát hoàn toàn nội dung trong image của mình ta cần tạo 1 image base.

- Parent image là image mà các image ta tạo đều dựa vào nó.Trong Dockerfile nó được xác định bởi chỉ thị FROM. và hầu hết Dockerfile đều bắt đầu bằng parent image mà không phải là base image.
- Base image sẽ không có dòng FROM trên Dockerfile hoặc có FROM là scratch.

Nếu không có khả năng tự build được base image thì ta nên download các bản official base image tại:
https://hub.docker.com/search/?q=&type=image&operating_system=linux

5.1 Sử dụng

Lệnh Docker build sẽ build 1 image từ dockerfile. Nội dung build là tập các file ở vị trí xác định bởi PATH hoặc URL.

NOTE: Không thư mục root , /, là Path bởi vì nó sẽ khiến cho thư mục root chứa toàn bộ nội dung của Docker daemon. Thông thường Dockerfile file được đặt tên là Dockerfile và nằm tại thư mục root chứa context. Sử dụng -f flag để chỉ định Dockerfile nằm ở bất kỳ đâu trên filesystem.

```
$ docker build -f /path/to/a/Dockerfile .
```

- Có thể xác định repository và tag để lưu image mới khi build thành công:

```
$ docker build -t shykes/myapp .
```

- để tag image lên nhiều repository sau khi build, thêm tham số -t khi chạy lệnh build.

```
$ docker build -t shykes/myapp:1.0.2 -t shykes/myapp:latest .
```

Từ phiên bản 18.09, Docker hỗ trợ backend mới để thực thi công việc build là buildkit. Backend buildkit mang lại nhiều lợi ích hơn so với cách triển khai cũ. Chẳng hạn Buildkit có khả năng:

+ phát hiện và bỏ qua thực thi các tag không sử dụng
+ Thực hiện song song các stage build độc lập building
+ chỉ tranfer những file có thay đổi trong context được build.
+ Sử dụng cách triển khai Dockerfile ngoài có nhiều thuộc tính mới
+ Tránh ảnh hưởng với phần còn lại của API
+ ưu tiên build cache để pruning tự động.

Để sử dụng buildkit backend ta cần enable biến môi trường `DOCKER_BUILDKIT=1` trước khi thực thi lệnh docker build.

5.2 Định dạng Dockerfile

Đây là định dạng của Dockerfile:

```
# Comment

INSTRUCTION arguments
```

instruction không phân biệt hoa hay thường, tuy nhiên ở đây ta tự quy ước nó là chữ hoa để phân biệt với các arguments. Nội dung sau ký tự # là ghi chú.

a. Sử dụng biến môi trường

Biến môi trường được ký hiệu là $variable-name hoặc ${variable-name}. cú pháp `${variable_name}` hỗ trợ 1 số chuẩn như sau:

- `${variable:-word}` chỉ thị nếu variable được thiết lập thì kết quả là giá trị đó, nếu variable không được thiết lập thì word là kết quả.
- `${variable:+word}` Chỉ thị nếu variable được thiết lập thì word là kết quả, ngược lại thì kết quả là 1 string rỗng.

Trong tất cả các trường hợp trên thì word có thể là 1 string hoặc có thể là 1 biến khác.

Các biến môi trường chỉ được sử dụng bởi các chỉ thị sau trong Dockerfile:

- ADD
- COPY

- ENV
- EXPOSE
- FROM
- LABEL
- STOPSIGNAL
- USER
- VOLUME
- WORKDIR
- ONBUILD

b. file .Dockerignore

Trước khi Docker CLI gửi context đến docker daemon nó sẽ tìm file có tên là .dockerignore trong thư mục root chứa context. Nếu file này tồn tại nó sẽ loại bỏ các file hoặc thư mục không match với pattern của nó.

c. Các instructions

- FROM
- RUN
- CMD
- LABEL
- MAINTAINER
- EXPOSE
- ENV
- ADD
- COPY
- ENTRYPOINT
- VOLUME
- USER
- WORKDIR
- ARG
- ONBUILD
- STOPSIGNAL
- HEALTHCHECK
- SHELL

- **FROM**

cú pháp:

`FROM <image> [AS <name>]`

hoặc

`FROM <image>[:<tag>] [AS <name>]`

hoặc `FROM <image>[@<digest>] [AS <name>]`

Chỉ thị này khởi tạo 1 trạng thái build mới và thiết lập Base image cho các chỉ thị tiếp theo. 1 Dockerfile phải bắt đầu bằng chỉ thị FROM.

+ FROM sử dụng nhiều lần để build nhiều image
+ tag hoặc digest: là giá trị tùy chọn. Nếu bỏ qua chúng thì builder mặc định lấy giá trị latest.

- **RUN**

cú pháp: có 2 dạng

+ `RUN <command>` (shell form,lệnh chạy trong shell)
+ `RUN ["executable", "param1", "param2"]` (exec form)

Chỉ thị RUN sẽ thực thi bất kỳ lệnh nào trên một layer mới ở trên đầu image hiện tại và commit kết quả. ở dạng shell có thể sử dụng ký tự \ (backslash) để viết nhiều lệnh trên nhiều dòng.

```
RUN /bin/bash -c 'source $HOME/.bashrc; \
echo $HOME'
```

Note:

+ Để sử dụng shell khác /bin/sh, sử dụng exec form ở shell mong muốn. ví dụ: `RUN ["/bin/bash", "-c", "echo hello"]`
+ exec form có cú pháp dạng JSON, tức là ta phải sử dụng dấu nháy kép “ bao quanh word thay vì dấu nháy đơn ‘.

- **CMD**

có 3 form:

+ `CMD ["executable","param1","param2"]` (exec form, form này được khuyến nghị)
+ `CMD ["param1","param2"]` (Như các tham số mặc định của entrypoint)
+ `CMD command param1 param2` (dạng shell)

Chỉ có duy nhất 1 chỉ thị CMD trong Dockerfile. Nếu có nhiều hơn 1 chỉ thị CMD thì chỉ thị cuối cùng được thực hiện. Mục đích chính của CMD là cung cấp các giá trị default cho 1 container thực thi. Giá trị mặc định này có thể là giá trị m khả năng thực thi hoặc không.

Note:

+ Nếu CMD được sử dụng để cung cấp các default argument cho chỉ thị ENTRYPOINT thì cả CMD và ENTRYPOINT nên xác định ở mạng JSON array format.
+ exec form được truyền vào như là array JSON, tức là phải sử dụng dấu nháy kép thay vì dấu nháy đơn
+ không giống shell form, exec form không gọi command shell. tức là sẽ không thực thi các câu lệnh ở dạng shell. ví dụ `CMD [ "echo", "$HOME" ]` sẽ không in ra tham số biến $HOME.

Khi sử dụng shell hoặc exec format, chỉ thị CMD sẽ thiết lập lệnh sẽ được thực thi khi running image.

Nếu sử dụng shell form khi đó <command> sẽ được thực thi trong `/bin/sh -c`.

```
FROM ubuntu
CMD echo "This is a test." | wc -
```

Nếu muốn chạy <command> không sử dụng shell ta cần biểu diễn lệnh như 1 mảng JSON và cung cấp đầy đủ đường dẫn tới lệnh thực thi. Dạng array này là định dạng được ưu tiên của CMD.

```
FROM ubuntu
CMD ["/usr/bin/wc","--help"]
```

NOTE: Đừng nhầm lẫn giữa lệnh RUN và CMD. RUN thực tế là chạy lệnh và commit kết quả còn CMD không thực thi gì trong khi building nhưng xác định lệnh được sử dụng cho image.

- **LABEL**

Cú pháp:

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

Label add metadata vào image. Label là 1 cặp key-value. Nếu muốn sử dụng space, giá trị nên đặt trong dấu ngoặc kép. một số ví dụ:

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

Có thể viết nhiều Label trong 1 image. để viết label trên 1 dòng sử dụng 1 trong 2 cách sau:

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

Để view label của image sử dụng lệnh : `docker inspect`

hoặc:

```
LABEL multi.label1="value1" \
      multi.label2="value2" \
      other="value3"
```

- **MAINTAINER (Dedicated)**

Cú pháp:

```
MAINTAINER <name>
```

Thiết lập author cho image đã được tạo. ta có thể sử dụng label thay thế cho chỉ thị này để có thể kiểm tra thông tin dễ dàng:

```
LABEL maintainer="SvenDowideit@home.org.au"
```

- **EXPOSE**

Cú pháp:

```
EXPOSE <port> [<port>/<protocol>...]
```

Chỉ thị này thông báo cho Docker biết container sẽ lắng nghe trên 1 port mạng cụ thể lúc runtime.

```
docker run -p 80:80/tcp -p 80:80/udp ...
```

- **ENV**

Cú pháp:

```
ENV <key> <value>
ENV <key>=<value> ...
```

Chỉ thị này thiết lập biến môi trường key với giá trị là value. Có 2 dạng như trên.dạng đầu tiên sẽ thiết lập 1 biến với 1 giá trị. toàn bộ string sau dấu cách (space) đầu tiên là value bao gồm cả khoảng trắng. Dạng thứ 2 cho phép nhiều biến được thiết lập 1 lần.

ví dụ:

```
ENV myName="John Doe" myDog=Rex\ The\ Dog \
    myCat=fluffy
```

và:

```
ENV myName John Doe
ENV myDog Rex The Dog
ENV myCat fluffy
```

Note: Việc tồn tại biến môi trường có thể gây ra một số ảnh hưởng không mong muốn ví dụ như khi thiết lập `ENV DEBIAN_FRONTEND noninteractive` có thể gây nhầm lẫn với apt-get user trên debian-based image. để thiết lập giá trị cho 1 lệnh, sử dụng `RUN <key>=<value> <command>`.

- **ADD**

Cú pháp: có 2 dạng

  + `ADD [--chown=<user>:<group>] <src>... <dest>`
  + `ADD [--chown=<user>:<group>] ["<src>",... "<dest>"]` (dạng này yêu cầu đường dẫn chứa khoảng trắng).

Chỉ thị ADD copy các file, thư mục mới hoặc xóa URL file từ <src> và add chúng tới filesystem của image ở path <dest>. Có thể xác định nhiều <src> nhưng nếu chúng là những file hoặc thư mục, đường dẫn của chúng được hiểu là liên quan tới source của context của build.

Mỗi <src> có thể chứa Wildcard, ví dụ:

```
ADD hom* /mydir/        # adds all files starting with "hom"
ADD hom?.txt /mydir/    # ? is replaced with any single character, e.g., "home.txt"
```

<dst> là đường dẫn tuyệt đối hoặc đường dẫn liên quan tới WORKDIR. Trong đó source sẽ được copy vào trong container đích.

```
ADD test relativeDir/          # adds "test" to `WORKDIR`/relativeDir/
ADD test /absoluteDir/         # adds "test" to /absoluteDir/
```

Khi add file hoặc thư mục chứa các ký tự đặc biệt như (như là [hoặc]) cần thoát đường dẫn này theo các quy tắc Golang để ngăn chặn chúng được xử lý như matching pattern. ví dụ muốn add file có tên là `arr[0].txt`, sử dụng như sau:

```
ADD arr[[]0].txt /mydir/    # copy a file named "arr[0].txt" to /mydir/
```

1 thư mục hoặc 1 file mới được tạo ra với UID và GID bằng 0 trừ khi sử dụng --chown flag để xác định, user, group. Ví dụ:

```
ADD --chown=55:mygroup files* /somedir/
ADD --chown=bin files* /somedir/
ADD --chown=1 files* /somedir/
ADD --chown=10:11 files* /somedir/
```

Trong trường hợp <src> là remote file URL, destination sẽ có quyền là 600. ADD tuân theo một số quy tắc sau:

+ đường dẫn <src> phải nằm trong context của build, ta không thể `ADD ../something /something`, bởi vì bước đầu tiên của docker build là send thư mục chứa context (thư mục con) tới docker daemon.
+ Nếu <src> là URL và <dest> không kết thúc bằng dấu gạch chéo "/" thì 1 file được download từ URL và được copy tới <dest>.
+ Nếu <src> là URL và <dest> kết thúc bằng dấu gạch chéo thì filename được suy ra từ URL và file này được download tới `<dest>/<filename>`. ví dụ `ADD http://example.com/foobar /` sẽ tạo file /foobar.
+ Nếu <src> là thư mục, toàn bộ context trong thư mục được copy bao gồm cả filesystem metadata.

  Note: Thư mục không được copy mà chỉ copy context trong nó.

+ Nếu <src> là định dạng file nén ở local (zip, tar, ..) thì được unpacked như 1 thư mục. Các tài nguyên từ remote URL không được giải nén. Khi thư mục được copy hay unpacked, nó được giống như tar -x.
+ Nếu <src> là bất kỳ loại file nào khác, nó được copy riêng cùng với metadata của nó. Trong trường hợp này nếu <dest> kết thúc bởi dấu gạch chéo thì nó được coi như 1 thư mục và các content của <src> sẽ được ghi ở `<dest>/base(<src>)` .
+ Nếu có nhiều <src> thì <dest> phải là 1 thư mục và kết thúc bằng dấu gạch chéo.
+ Nếu <dest> Không kết thúc bằng dấu gạch chéo thì nó sẽ được coi như file thường và các content của <src> sẽ được ghi vào <dest>
+ Nếu <dest> không tồn tại nó sẽ tạo tất cả các thư mục bị thiếu trong đường dẫn của nó.

- **COPY**

Cú pháp: Có 2 form:

+ `COPY [--chown=<user>:<group>] <src>... <dest>`
+ `COPY [--chown=<user>:<group>] ["<src>",... "<dest>"]`(form này Y/c đường dẫn chứa các khoảng trắng whitespace)

Chỉ thị COPY sẽ copy file hoặc thư mục mới từ <src> và add chúng vào filesystem của container ở đường dẫn <dest>. Có thể xác định nhiều <src> nhưng đường dẫn của các file hay thư mục sẽ được hiểu là liên quan tới source của context build.

<src> có thể chứa các ký tự đặc biệt. ví dụ:

```
COPY hom* /mydir/        # adds all files starting with "hom"
COPY hom?.txt /mydir/    # ? is replaced with any single character, e.g., "home.txt"
```

<dest> là đường dẫn tuyệt đối hoặc liên quan tới WORKDIR mà source sẽ được copy vào trong container đích.

```
COPY test relativeDir/   # adds "test" to `WORKDIR`/relativeDir/
COPY test /absoluteDir/  # adds "test" to /absoluteDir/
```

Khi add file hoặc thư mục chứa các ký tự đặc biệt như (như là [hoặc]) cần thoát đường dẫn này theo các quy tắc Golang để ngăn chặn chúng được xử lý như matching pattern. ví dụ muốn add file có tên là `arr[0].txt`, sử dụng như sau:

```
COPY arr[[]0].txt /mydir/    # copy a file named "arr[0].txt" to /mydir/
```

1 thư mục hoặc 1 file mới được tạo ra với UID và GID bằng 0 trừ khi sử dụng --chown flag để xác định, user, group. Ví dụ:

```
COPY --chown=55:mygroup files* /somedir/
COPY --chown=bin files* /somedir/
COPY --chown=1 files* /somedir/
COPY --chown=10:11 files* /somedir/
```

giống như ADD, COPY cũng tuân theo các quy tắc như ADD.

- **ENTRYPOINT**

cú pháp: Có 2 form:

+ `ENTRYPOINT ["executable", "param1", "param2"]` (exec form: Được ưu tiên dùng)
+ `ENTRYPOINT command param1 param2` (shell form)

Entrypoint cho phép cấu hình một container sẽ run như là một container có khả năng thực thi. ví dụ ta sẽ start nginx với context mặc định của nó và listen port 80.

```
docker run -i -t --rm -p 80:80 nginx
```

Tất cả các tham số dòng lệnh docker run <image> sẽ được thêm vào sau tất cả các phần tử ở entrypoint exec form .

shell form ngăn ngừa các tham số dòng lệnh CMD và RUN được sử dụng, nhưng có nhược điểm đó là entrypoint sẽ được start như 1 lệnh con của `/bin/sh -c`, Chỉ ENTRYPOINT cuối cùng trong Dockerfile mới có hiệu lực.

Ví dụ exec form entrypoint:

```
FROM ubuntu
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
```

Khi ta chạy container, ta thấy rằng top là process duy nhất.

```
$ docker run -it --rm --name test  top -H
top - 08:25:00 up  7:27,  0 users,  load average: 0.00, 0.01, 0.05
Threads:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   2056668 total,  1616832 used,   439836 free,    99352 buffers
KiB Swap:  1441840 total,        0 used,  1441840 free.  1324440 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
    1 root      20   0   19744   2336   2080 R  0.0  0.1   0:00.04 top
```

để kiểm tra kết quả cụ thể sử dụng lệnh sau:

```
$ docker exec -it test ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  2.6  0.1  19752  2352 ?        Ss+  08:24   0:00 top -b -H
root         7  0.0  0.1  15572  2164 ?        R+   08:25   0:00 ps aux
```

+ Dockerfile sau sẽ sử dụng ENTRYPOINT để chạy apache ở foreground

```
FROM debian:stable
RUN apt-get update && apt-get install -y --force-yes apache2
EXPOSE 80 443
VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Note:

+ Có thể ghi đề entrypoint bằng cách sử dụng option --entrypoint khi chạy run container. nhưng chỉ áp dụng cho exec form.
+ exec form có cú pháp là JSON array và không gọi lệnh shell.

Ví dụ shell form entrypoint:

```
FROM ubuntu
ENTRYPOINT exec top -b
```

Khi run image này ta sẽ thấy như sau:

```
$ docker run -it --rm --name test top
Mem: 1704520K used, 352148K free, 0K shrd, 0K buff, 140368121167873K cached
CPU:   5% usr   0% sys   0% nic  94% idle   0% io   0% irq   0% sirq
Load average: 0.08 0.03 0.05 2/98 6
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
    1     0 root     R     3164   0%   0% top -b
```

Hiểu cách hoạt động của CMD và ENTRYPOINT:

+ Dockerfile nên xác định ít nhất 1 trong các lệnh CMD hoặc ENTRYPOINT.

+ ENTRYPOINT nên được định nghĩa khi sử dụng container dạng thực thi
+ CMD nên được sử dụng theo cách định nghĩa các argument mặc định cho lệnh ENTRYPOINT hoặc thực thi 1 lệnh ad-hoc trong 1 container.
+ CMD sẽ được ghi đè khi running container với các aguments khác.

Bảng sau sẽ show lệnh nào sẽ được thực thi khi kết hợp CMD và ENTRYPOINT

	No ENTRYPOINT	ENTRYPOINT exec_entry p1_entry	ENTRYPOINT ["exec_entry", "p1_entry"]
No CMD	*error, not allowed*	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD ["exec_cmd", "p1_cmd"]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD ["p1_cmd", "p2_cmd"]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

- **VOLUME**

Cú pháp:

```
VOLUME ["/data"]
```

Nó sẽ tạo 1 mount point với tên xác định và đánh dấu nó như một volume được mount từ ngoài. Giá trị có thể là JSON array, `VOLUME ["/var/log/"]`, hoặc 1 string với nhiều arguments như: `VOLUME /var/log`, hoặc `VOLUME /var/log /var/db`.

Một số lưu ý với VOLUME:

+ Khi sử dụng windows container, destination của volume trong container phải là 1 thư mục rỗng hoặc chưa tồn tại, hoặc là 1 ổ đĩa khác ổ C.
+ Thay đổi VOLUME từ Dockerfile: Nếu có bước build nào thay đổi dữ liệu trong volume sau khi nó được khai báo thì những thay đổi này sẽ bị loại bỏ
+ JSON formating: List được truyền vào như 1 JSON array, do đó phải bao quanh word bằng dấu nháy kép.

+ Thư mục của host được khai báo tại thời điểm container runtime: Thư mục host (mount point) về bản chất nó phụ thuộc vào host. chỉ thị volume không hỗ trợ tham số host-dir, vì vậy phải xác định mount point trước khi tạo và chạy container.

- **USER**

Cú pháp:

```
USER <user>[:<group>] or
USER <UID>[:<GID>]
```

Chỉ thị user thiết lập user name và group để sử dụng khi run image và cho các chỉ trị CMD, RUN, ENTRYPOINT

Warning: Khi user không có primary group thì image (hoặc chỉ thị kế tiếp) chạy bằng root group.

- **WORKDIR**

cú pháp:

```
WORKDIR /path/to/workdir
```

Thiết lập thư mục working cho các chỉ thị `RUN`, `CMD`, `ENTRYPOINT`, `COPY` và ADD. Nếu workdir không tồn tại nó sẽ được tạo ngay cả khi chỉ thị sau đó trong Dockerfile không dùng tới.

workdir có thể được sử dụng nhiều lần trong Dockerfile:

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

- **ARG**

Cú pháp:

```
ARG <name>[=<default value>]
```

Định nghĩa biến mà user có thể thông qua tại thời điểm build để builder cùng lệnh docker build sử dụng `--build-arg <varname>=<value>` flag.

Dockerfile có thể có 1 hoặc nhiều chỉ thị ARG:

```
FROM busybox
ARG user1
ARG buildno
...
```

Sử dụng ARG:

Có thể sử dụng ARG hoặc ENV để định nghĩa các biến cho chỉ thị RUN. Biến được định nghĩa bởi ENV luôn luôn ghi đè biến định nghĩa bởi ARG nếu chúng trùng tên.

- **ONBUILD**

Cú pháp:

```
ONBUILD [INSTRUCTION]
```

Add vào image 1 trigger để thực thi tại thời điểm mới nhất khi image được sử dụng như base image để build image khác.

Hoạt động của onbuild:

+ Khi nó gặp 1 chỉ thị onbuild, builder sẽ add 1 trigger vào metadata của image đang build.
+ Cuối quá trình build, một danh sách các trigger sẽ được lưu trong image manifest dưới Onbuild key.
+ Sau đó image sẽ được sử dụng như 1 base cho build image mới bằng cách sử dụng chỉ thị FROM.
+ trigger được clear từ image cuối cùng sau khi đã được thực thi.

- **STOPSIGNAL**

Cú pháp:

```
STOPSIGNAL signal
```

Được sử dụng để gọi hệ thống gọi signal send tới container để exit.

- **HEALTHCHECK**

Cú pháp: có 2 form:

+ `HEALTHCHECK [OPTIONS] CMD command` (check health container bằng cách chạy các lệnh trong container)
+ `HEALTHCHECK NONE` (disable health check kế thừa từ base image)

healthcheck cho docker biết cách để test 1 container có làm việc hay không. việc này có thể phát hiện các trường hợp như web server bị loop và không thể xử lý các kết nối mới mặc dù tiến trình của server vẫn đang running.

Một số option có thể đặt trước CMD:

+ `--interval=DURATION` (default: `30s`)
+ `--timeout=DURATION` (default: `30s`)

+ `--start-period=DURATION` (default: `0s`)
+ `--retries=N` (default: `3`)

Lệnh sau CMD keyword có thể là 1 lệnh shell (`HEALTHCHECK CMD /bin/check-running`) hoặc một exec array. Trạng thái kết thúc của lệnh chỉ thị trạng thái health của container:

+ 0: success - container health và sẵn sàng để sử dụng
+ 1: unhealthy
+ 2: Không sử dụng mã exit này.

ví dụ:

```
HEALTHCHECK --interval=5m --timeout=3s \
  CMD curl -f http://localhost/ || exit 1
```

- **SHELL**

cú pháp:

```
SHELL ["executable", "parameters"]
```

Cho phép shell mặc định sử dụng cho shell form. trên linux mặc định là `["/bin/sh", "-c"]`.

- Một số ví dụ Dockerfile:

```
# Nginx
#
# VERSION               0.0.1

FROM      ubuntu
LABEL Description="This image is used to start the foobar executable"
Vendor="ACME Products" Version="1.0"
RUN apt-get update && apt-get install -y inotify-tools nginx apache2 openssh-
server
```

Ví dụ 2: build image postgresql trên ubuntu.

```
#example                              Dockerfile                              for
https://docs.docker.com/engine/examples/postgresql_service/
#
FROM ubuntu:16.04
# Add the PostgreSQL PGP key to verify their Debian packages.
#        It        should        be        the        same        key        as
https://www.postgresql.org/media/keys/ACCC4CF8.asc
RUN apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys
B97B0AFCAA1A47F044F244A07FCC7D46ACCC4CF8
```

```
# Add PostgreSQL's repository. It contains the most recent stable release
#     of PostgreSQL, ``9.3``.
RUN echo "deb http://apt.postgresql.org/pub/repos/apt/ precise-pgdg main" >
/etc/apt/sources.list.d/pgdg.list
# Install ``python-software-properties``, ``software-properties-common`` and
PostgreSQL 9.3
#  There are some warnings (in red) that show up during the build. You can
hide
#         them     by     prefixing     each     apt-get     statement     with
DEBIAN_FRONTEND=noninteractive
RUN apt-get update && apt-get install -y python-software-properties software-
properties-common  postgresql-9.3  postgresql-client-9.3  postgresql-contrib-
9.3
# Note: The official Debian and Ubuntu images automatically ``apt-get clean``
# after each ``apt-get``
# Run the rest of the commands as the ``postgres`` user created by the
``postgres-9.3`` package when it was ``apt-get installed``
USER postgres
# Create a PostgreSQL role named ``docker`` with ``docker`` as the password
and
# then create a database `docker` owned by the ``docker`` role.
# Note: here we use ``&&\`` to run commands one after the other - the ``\``
#       allows the RUN command to span multiple lines.
RUN    /etc/init.d/postgresql start &&\
    psql --command "CREATE USER docker WITH SUPERUSER PASSWORD 'docker';"
&&\
    createdb -O docker docker

# Adjust PostgreSQL configuration so that remote connections to the
# database are possible.
RUN    echo    "host    all       all              0.0.0.0/0       md5"    >>
/etc/postgresql/9.3/main/pg_hba.conf
#            And            add            ``listen_addresses``            to
``/etc/postgresql/9.3/main/postgresql.conf``
RUN echo "listen_addresses='*'" >> /etc/postgresql/9.3/main/postgresql.conf
# Expose the PostgreSQL port
EXPOSE 5432
# Add VOLUMEs to allow backup of config, logs and databases
VOLUME  ["/etc/postgresql", "/var/log/postgresql", "/var/lib/postgresql"]
# Set the default command to run when starting the container
CMD              ["/usr/lib/postgresql/9.3/bin/postgres",              "-D",
"/var/lib/postgresql/9.3/main",                                        "-c",
"config_file=/etc/postgresql/9.3/main/postgresql.conf"]
```

https://docs.docker.com/engine/reference/builder/

VI. Hướng dẫn viết docker-compose

https://docs.docker.com/compose/

6.1 Tổng quan docker-compose

Là 1 công cụ cho phép định nghĩa và running ứng dụng với Docker multi-container. compose làm việc trên tất cả các môi trường : production, staging, developing, testing cũng như CI workflow.

Sử dụng compose cơ bản có 3 bước:

- Định nghĩa biến môi trường bằng Dockerfile
- Định nghĩa các service trong docker-compose.yml để chúng nó thể run cùng môi trường độc lập.
- chạy docker-compose up, và Compose start và run toàn bộ app

Ví dụ về 1 file docker-compose.yml

```
version: '3'
services:
  web:
    build: .
    ports:
    - "5000:5000"
    volumes:
    - .:/code
    - logvolume01:/var/log
    links:
    - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

a. Cài đặt Docker compose

Trên linux:

Chạy lệnh sau để download phiên bản ổn định ở hiện tại của Docker compose:

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.24.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

Cấp quyền thực thi cho docker-compose file:

```
sudo chmod +x /usr/local/bin/docker-compose
```

sử dụng lệnh `$ docker-compose --version` để check version.

b. Upgrade compose

Nếu upgrade compose từ bản 1.2 hoặc cũ hơn thì cần remove hoặc migrate các container đang tồn tại sau khi upgrade composer. Lý do là từ bản 1.3, composer sử dụng Docker label để duy trì track các container, và container cần được tạo lại và thêm label.

Nếu composer phát hiện container được tạo ra không có label, nó sẽ từ chối chạy nó. Có thể sử dụng composer bản 1.5.x để migrate sử dụng lệnh sau:

```
docker-compose migrate-to-labels
```

6.2 mở đầu

Phần này sẽ build 1 ứng dụng python bằng docker-compose.

steps 1: thiết lập môi trường

1. tạo thư mục composetest

   ```
   $ mkdir composetest
   $ cd composetest
   ```

2. tạo file named app.py nội dung như sau:

   ```
   import time
   import redis
   from flask import Flask

   app = Flask(__name__)
   cache = redis.Redis(host='redis', port=6379)
   def get_hit_count():
       retries = 5
       while True:
           try:
               return cache.incr('hits')
           except redis.exceptions.ConnectionError as exc:
               if retries == 0:
   ```

```
                raise exc
            retries -= 1
            time.sleep(0.5)
@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello World! I have been seen {} times.\n'.format(count)
```

3. tạo file requirements.txt với dụng dung sau:

   Flask

   Redis

steps2: Tạo Dockerfile

Trong thư mục của project tạo file named Dockerfile và add nội dung sau vào file:

```
FROM python:3.7-alpine
WORKDIR /code
ENV FLASK_APP app.py
ENV FLASK_RUN_HOST 0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
COPY . .
CMD ["flask", "run"]
```

Step 3: Định nghĩa service bằng compose file:

Tạo file docker-compose.yml và add nội dung sau vào file:

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
  redis:
    image: "redis:alpine"
```

file compose định nghĩa 2 service: web và redis

web service: sử dụng image được build từ Dockerfile ở thư mục hiện tại, public port 5000.

redis service: Sử dụng image redis public kéo về từ docker hub.

Step 4: build app từ Composer.

1. Từ thư mục của project chạy câu lệnh: `docker-compose up`.

2. Truy cập http://localhost:5000 từ browser để kiểm tra ứng dụng đang chạy
3. Stop ứng dụng chạy lệnh `docker-compose down` trong thư mục project

Step 5: Chỉnh sửa file docker-compose.yml để add bind mount:

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_ENV: development
  redis:
    image: "redis:alpine"
```

volume key mới mount thư mục project hiện tại trên host vào thư mục /code trong container. Việc này cho phép chỉnh sửa code nhanh chóng mà không cần rebuild lại image.

Step 6: Rebuid và chạy app bằng compose

Từ thư mục project, gõ lệnh docker-compose up để build app từ file docker-compose.yml đã được update.

Step 7: Update ứng dụng

1. Bởi vì code của ứng dụng đã được mount tới thư mục trên host nên ta có thể thay đổi ứng dụng nhanh chóng mà không cần rebuild image. Thay đổi bản tin hello-world trong file app.py bằng nội dung sau:

   ```
   return 'Hello from Docker! I have been seen {} times.\n'.format(count)
   ```

2. refresh trình duyệt để xem thay đổi

Step 8: Sử dụng 1 số option khác

Nếu muốn chạy service như là background service sử dụng -d flag (cho detached mode).

và để stop service dùng câu lệnh: `docker-compose stop`

6.3 Docker CLI

- Sử dụng -f option để xác định đường dẫn của 1 hoặc nhiều docker-compose file.

- Sử dụng -p để xác định tên project. Nếu không sử dụng composer sử dụng tên thư mục là tên project

6.4 Composer file

Composer version 3 tương thích với Docker engine từ bản 1.13 trở lên.

Cấu trúc file composer và ví dụ:

```
version: "3.7"
services:
  redis:
    image: redis:alpine
    ports:
      - "6379"
    networks:
      - frontend
    deploy:
      replicas: 2
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure
  db:
    image: postgres:9.4
    volumes:
      - db-data:/var/lib/postgresql/data
    networks:
      - backend
    deploy:
      placement:
        constraints: [node.role == manager]
  vote:
    image: dockersamples/examplevotingapp_vote:before
    ports:
      - "5000:80"
    networks:
      - frontend
    depends_on:
      - redis
    deploy:
      replicas: 2
      update_config:
```

```
        parallelism: 2
      restart_policy:
        condition: on-failure
  result:
    image: dockersamples/examplevotingapp_result:before
    ports:
      - "5001:80"
    networks:
      - backend
    depends_on:
      - db
    deploy:
      replicas: 1
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure
  worker:
    image: dockersamples/examplevotingapp_worker
    networks:
      - frontend
      - backend
    deploy:
      mode: replicated
      replicas: 1
      labels: [APP=VOTING]
      restart_policy:
        condition: on-failure
        delay: 10s
        max_attempts: 3
        window: 120s
      placement:
        constraints: [node.role == manager]
  visualizer:
    image: dockersamples/visualizer:stable
    ports:
      - "8080:8080"
    stop_grace_period: 1m30s
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      placement:
```

```
        constraints: [node.role == manager]
networks:
  frontend:
  backend:
volumes:
  db-data:
```

Phần này chứa một loạt các option cấu hình được hỗ trợ để định nghĩa service trong V3.

build: xác định đường dẫn build context.

```
version: "3.7"
services:
  webapp:
    build: ./dir
```

Hoặc một object có đường dẫn được định nghĩa trong context và Dockerfile và args tùy chọn:

```
version: "3.7"
services:
  webapp:
    build:
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        buildno: 1
```

CONTEXT: hoặc là đường dẫn tới thư mục chứa Dockerfile hoặc là URL của git repository.

CACHE_FROM: list các image sử dụng cache

```
build:
  context: .
  cache_from:
    - alpine:latest
    - corp/web_app:3.14
```

6.5 Docker stack và các gói ứng dụng

a. Tổng quan

1 Dockerfile có thể được build thành image và các container có thể được build từ image đó. Tương tự docker-compose.yml có thể được build thành một gói (gồm nhiều ứng dụng) ứng dụng phân tán và stack có thể được tạo thành từ gói đó. Trong trường hợp đó, một gói là một định dạng image có khả năng phân bổ nhiều service.

Tuy nhiên trong swarm mode, các ứng dụng multi-service và các stack file được hỗ trợ hoàn toàn. 1 stack file là 1 loại đặc biệt của file composer version 3.

b. Viết 1 gói (Bundle)

Cách dễ nhất để tạo 1 bundle là sử dụng docker-compose từ file docker-compose.yml đang tồn tại. Từ docker-compose:

```
$ docker-compose bundle
WARNING: Unsupported key 'network_mode' in services.nsqd - ignoring
WARNING: Unsupported key 'links' in services.nsqd - ignoring
WARNING: Unsupported key 'volumes' in services.nsqd - ignoring
[...]
Wrote bundle to vossibility-stack.dab
```

c. Tạo 1 stack từ 1 bundle

1 stack được tạo thành sử dụng lệnh docker deploy:

```
# docker deploy --help
Usage:  docker deploy [OPTIONS] STACK
Create and update a stack
Options:
      --file    string         Path to a Distributed Application Bundle file
(Default: STACK.dab)
      --help                   Print usage
      --with-registry-auth     Send registry authentication details to Swarm
agents
```

d. Quản lý stack

stack được quản lý sử dụng lệnh: docker stack . để xem các option thêm --help flag.

e. Bundle file format

https://docs.docker.com/compose/bundles/

6.6 Sử dụng compose với swarm

Docker compose và Docker swarm nhằm tới việc tích hợp hoàn toàn, tức là ta có thể đặt 1 Composer app vào trong Swarm cluster và nó chỉ hoạt động nếu chúng ta đang sử dụng docker host đơn lẻ.

a. Các hạn chế

- Build image

Swarm có thể build image từ một Dockerfile giống như 1 Docker instance single-host, nhưng image sinh ra chỉ tồn tại trên 1 node và không thể phân bổ cho các node khác.

Nếu muốn sử dụng composer để scale service cho nhiều node thì build image và push image lên repository, tham khảo docker-compose.yml sau:

```
$ docker build -t myusername/web .
$ docker push myusername/web
$ cat docker-compose.yml
web:
  image: myusername/web
$ docker-compose up -d
$ docker-compose scale web=3
```

- multi dependences:

Nếu service có nhiều dependencies, có thể swarm lập lịch dependence trên các node khác, khiến cho dependence service không thể lập lịch. ví dụ sau service foo cần cùng lập lịch với bar và baz.

```
ersion: "2"
services:
  foo:
    image: foo
    volumes_from: ["bar"]
    network_mode: "service:baz"
  bar:
    image: bar
  baz:
    image: baz
```

Vấn đề ở đây là Swarm có thể lập lịch cho bar và baz trước trên các node khác, khiến cho nó không thể nhận 1 node phù hợp cho foo.

Để khắc phục vấn đề này, ta sẽ sử dụng cơ chế scheduling thủ công, để đảm bảo rằng tất cả các service đều kết thúc trên cùng 1 node.

```
version: "2"
services:
  foo:
    image: foo
    volumes_from: ["bar"]
    network_mode: "service:baz"
    environment:
      - "constraint:node==node-1"
  bar:
    image: bar
    environment:
```

```
      - "constraint:node==node-1"
  baz:
    image: baz
    environment:
      - "constraint:node==node-1"
```

- host port và tạo lại container

Nếu service map port từ host, khi đó có thể gặp lỗi khi chạy lệnh docker-compose up

```
docker: Error response from daemon: unable to find a node that satisfies
container==6ab2dfe36615ae786ef3fc35d641a260e3ea9663d6e69c5b70ce0ca6cb373c02
.
```

Nguyên nhân là container có 1 volume (do định nghĩa trong image hoặc docker-compose) không được map rõ ràng, vì vậy để bảo vệ dữ liệu composer sẽ chuyển hướng Swarm để schedule container mới trên cùng node container cũ.

có 2 cách khắc phục vấn đề này:

+ xác định named volume và sử dụng 1 volume driver có khả năng mouting volume vào trong container
+ remove container cũ trước khi tạo container mới. và data trong volume sẽ bị mất.

b. Scheduling container

- Scheduling tự động:

Một số option cấu hình sinh ra trong container được tự động scheduled trên cùng swarm node để đảm bảo chúng làm việc chính xác, đó là các option sau:

+ `network_mode: "service:..."` and `network_mode: "container:..."` (and`net: "container:..."` in the version 1 file format).
+ `volumes_from`
+ `links`

- Manual scheduling

Swarm cung cấp một số scheduling và các gợi ý có liên quan, cho phép ta kiểm soát nơi container được đặt. Chúng được xác định thông qua các biến môi trường trong container. Ta có thể sử dụng option environment của composer để thiết lập chúng

```
# Schedule containers on a specific node
environment:
  - "constraint:node==node-1"
# Schedule containers on a node that has the 'storage' label set to 'ssd'
environment:
```

```
  - "constraint:storage==ssd"
# Schedule containers where the 'redis' image is already pulled
environment:
  - "affinity:image==redis"
```

6.7 Environment file

Composer hỗ trợ khai báo các biến môi trường mặc định trong file environment tên là .env được đặt trong thư mục nơi lệnh docker-compose được thực thi (thư mục làm việc hiện tại).

Các quy tắc cú pháp:

- compose muốn mỗi dòng trong file env có định dạng VAR=VAL
- dòng bắt đầu bằng # là comment và bỏ qua
- Blank line được bỏ qua
- không xử lý các dấu trích dẫn đặc biệt (chúng là 1 phần của VAL)

6.8 Biến môi trường trong composer

Có rất nhiều phần của compose liên quan tới biến môi trường.

a. Thay thế biến môi trường trong file compose

Có thể sử dụng các biến môi trường để trong shell để đưa các value vào trong file Compose.

```
web:
  image: "webapp:${TAG}"
```

b. thiết lập các biến môi trường trong các container

Ta có thể thiết lập biến môi trường trong các container của service với environment key:

```
web:
  environment:
    - DEBUG=1
```

c. Dẫn các biến môi trường trong container

Ta có thể dẫn (pass) các biến môi trường từ shell thẳng tới các container của service với 'environment' key mà không cần cung cấp value

```
web:
  environment:
    - DEBUG
```

d. Các option cấu hình của env_file

Ta có thể pass nhiều biến môi trường từ file ngoài tới container của service với 'env_file' option, giống như run `docker run --env-file=FILE …:`

```
web:
  env_file:
    - web-variables.env
```

e. thiết lập biến môi trường với docker-compose run

cùng với docker run -e, ta có thể thiết lập các biến môi trường trên 1 container one-off (run 1 lần)

```
docker-compose run -e DEBUG=1 web python console.py
```

ta cũng có thể pass biến từ shell không cung cấp cho nó value

```
docker-compose run -e DEBUG web python console.py
```

Giá trị của biến DEBUG trong container được lấy từ giá trị của cùng biến trong shell mà Composer đang running.

f. file .env

Ta có thể thiết lập các giá trị mặc định cho các biến môi trường từ compose file hoặc được sử dụng để cấu hình compose trong environment file tên là .env

```
$ cat .env
TAG=v1.5
$ cat docker-compose.yml
version: '3'
services:
  web:
    image: "webapp:${TAG}"
```

ta có thể kiểm tra bằng lệnh docker-compose config.

- Khi ta thiết lập cùng biến môi trường trong nhiều file thì compose sử dụng mức ưu tiên để lựa chọn giá trị sử dụng.
    1. Compose file
    2. Shell environment variables
    3. Environment file
    4. Dockerfile
    5. Variable is not defined

g. Cấu hình compose sử dụng biến môi trường

Một số biến môi trường available cho phép ta cấu hình trạng thái CLI của Docker compose. Chúng bắt đầu bằng COMPOSE_ hoặc DOCKER_.

6.9 Mở rộng service trong composer

Compose hỗ trợ 2 2 phương thức chia sẻ cấu hình chung.

+ mở rộng toàn bộ file compose bằng cách sử dụng nhiều file compose
+ mở rộng từng service bằng cách sử dụng trường extends (chỉ dùng cho compose bản 2.1 về trước).

a. Sử dụng nhiều file

mặc định compose sẽ đọc 2 file docker-compose.yml và 1 file tùy chọn là docker-compose.override.yml. Nếu service được định nghĩa trên cả 2 file thì compose sẽ kết hợp các cấu hình sử dụng các rule trong phần c (thêm và ghi đè cấu hình). để sử dụng nhiều file override hoặc sử dụng tên file override khác sử dụng thêm -f option.

Ví dụ một số case sử dụng: thay đổi compose app cho environment khác, và chạy các task quản lý

TH1: Trường hợp sử dụng multi-file phổ biến nhất là thay đổi development compose app cho một môi trường giống production. Để thực hiện ta có thể chia cấu hình compose thành nhiều file khác nhau.

Bắt đầu là 1 base file (docker-compose.yml) định nghĩa cấu hình cơ bản cho service

```
web:
  image: example/my_web_app:latest
  links:
    - db
    - cache
db:
  image: postgres:latest
cache:
  image: redis:latest
```

Cấu hình mở development sẽ mở một số port public, mount code vào volume, và build web image.

docker-compose.override.yml

```
web:
  build: .
  volumes:
    - '.:/code'
  ports:
    - 8883:80
  environment:
```

```
    DEBUG: 'true'
db:
  command: '-d'
  ports:
    - 5432:5432
cache:
  ports:
    - 6379:6379
```

khi chạy lệnh docker-compose up nó sẽ tự động đọc override.

Sử dụng compose app này trong môi trường production, tạo 1 file override khác (file này có thể lưu trữ trên git repo hoặc được quản lý bởi 1 team khác) named: docker-compose.prod.yml.

```
web:
  ports:
    - 80:80
  environment:
    PRODUCTION: 'true'
cache:
  environment:
    TTL: '500'
```

Để deploy file compose production này sử dụng lệnh:

```
docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d
```

TH2: các task quản lý

Ví dụ chạy backup database. Start bằng docker-compose.yml

```
web:
  image: example/my_web_app:latest
  links:
    - db
db:
  image: postgres:latest
```

Trong docker-compose.admin.yml add một service mới để chạy database export hoặc backup

```
dbadmin:
  build: database_admin/
  links:
    - db
```

để run backup chạy lệnh:

```
docker-compose -f docker-compose.yml -f docker-compose.admin.yml \
    run dbadmin db-backup
```

b. Mở rộng từng service

Khi định nghĩa các service trong docker-compose.yml, ta có thể khai báo mở rộng 1 service khác như sau:

```
web:
  extends:
    file: common-services.yml
    service: webapp
```

nó sẽ chỉ thị cho compose tái sử dụng cấu hình cho service webapp được định nghĩa trong file common-service.yml. file common-service.yml có nội dung:

```
webapp:
  build: .
  ports:
    - "8000:8000"
  volumes:
    - "/data"
```

c. thêm và ghi đè cấu hình

compose copy cấu hình từ service ban đầu đến local service. Nếu cấu hình được định nghĩa trên cả original service và local service thì giá trị local sẽ được thay thế giá trị original với option chỉ có 1 giá trị. Với option có nhiều giá trị như `ports`, `expose`, `external_links`, `dns`, `dns_search`, and `tmpfs` thì compose sẽ kết hợp cả 2 tập giá trị, trong trường hợp là `environment`, `labels`, `volumes`, and `devices` thì compose đồng nhất các entries cùng với các giá trị local có quyền cao hơn.

6.10 Networking trong composer

Mặc định compose thiết lập một single network cho app.

ví dụ app nằm trong thư mục /myapp và nội dung file docker-compose.yml như sau:

```
version: "3"
services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
    ports:
```

```
      - "8001:5432"
```

Khi chạy docker-compose up, nó sẽ thực hiện các bước sau:

+ tạo 1 network mặc định myapp_default
+ 1 container được tạo ra sử dụng cấu hình web , và join vào network trên với tên là web.
+ tạo 1 container khác named db, container này cũng join vào network bằng tên db

a. Update container

Nếu thay đổi cấu hình của service và chạy lệnh docker-compose up để update nó thì container cũ sẽ bị xóa đi và container mới sẽ join vào network bằng địa chỉ IP khác nhưng cùng tên.

b. links

Cho phép định nghĩa thêm các aliase mà service có thể tìm kiếm từ 1 service khác. Mặc định chúng không yêu cầu cho phép các service giao tiếp với nhau, bất kỳ service nào cũng có thể tìm các service khác bằng tên của nó

c. multi-host networking

Nội dung phần này đề cập tới các hoạt động của legacy Docker Swarm. và chỉ hoạt động khi targeting là 1 legacy swarm cluster.

Khi deploy một ứng dụng bằng compose trong một swarm cluster, ta có thể sử dụng built-in `overlay` driver để enable giao tiếp multi-host giữa các container mà không cần thay đổi Compose file hoặc code ứng dụng.

d. Chỉ định custom network

Thay vì sử dụng default network, ta có thể định nghĩa network riêng sử dụng key network. Ta cũng có thể sử dụng nó để kết nối service với network tạo ngoài không được quản lý bởi compose.

ví dụ compose file định nghĩa 2 network:

```
version: "3"
services:
  proxy:
    build: ./proxy
    networks:
      - frontend
  app:
    build: ./app
    networks:
      - frontend
```

```
      - backend
  db:
    image: postgres
    networks:
      - backend
networks:
  frontend:
    # Use a custom driver
    driver: custom-driver-1
  backend:
    # Use a custom driver which takes special options
    driver: custom-driver-2
    driver_opts:
      foo: "1"
      bar: "2"
```

e. Cấu hình network mặc định

Ta có thể thay đổi cấu hình default mặc định bằng cách định nghĩa các entries dưới networks named default:

```
version: "3"
services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
networks:
  default:
    # Use a custom driver
    driver: custom-driver-1
```

f. Sử dụng network đang có

Sử dụng option external:

```
networks:
  default:
    external:
      name: my-pre-existing-network
```

6.11 Sử dụng composer trong môi trường production

Chỉnh sửa compose file cho môi trường production:

    + xóa bất kỳ volume bindings nào trong code ứng dụng, để code không thể bị thay đổi từ bên ngoài
    + Binding 1 port khác trên host
    + thiết lập các biến môi trường khác
    + chỉ định restart policy để tránh downtime: `restart: always`.
    + Bổ sung thêm các service khác như logging.
- deploy khi có thay đổi

Khi có thay đổi code của app, hãy rebuild lại image và tạo lại container của app.

- Chạy compose trên single host: ta có thể sử dụng compose để depoy app tới remote docker host bằng cách thiết lập các biến môi trường `DOCKER_HOST`, `DOCKER_TLS_VERIFY`, and `DOCKER_CERT_PATH`.

6.12 link các biến môi trường

https://docs.docker.com/compose/link-env-deprecated/

6.13 Kiểm soát thứ tự startup

https://docs.docker.com/compose/startup-order/

6.14 Ví dụ với composer

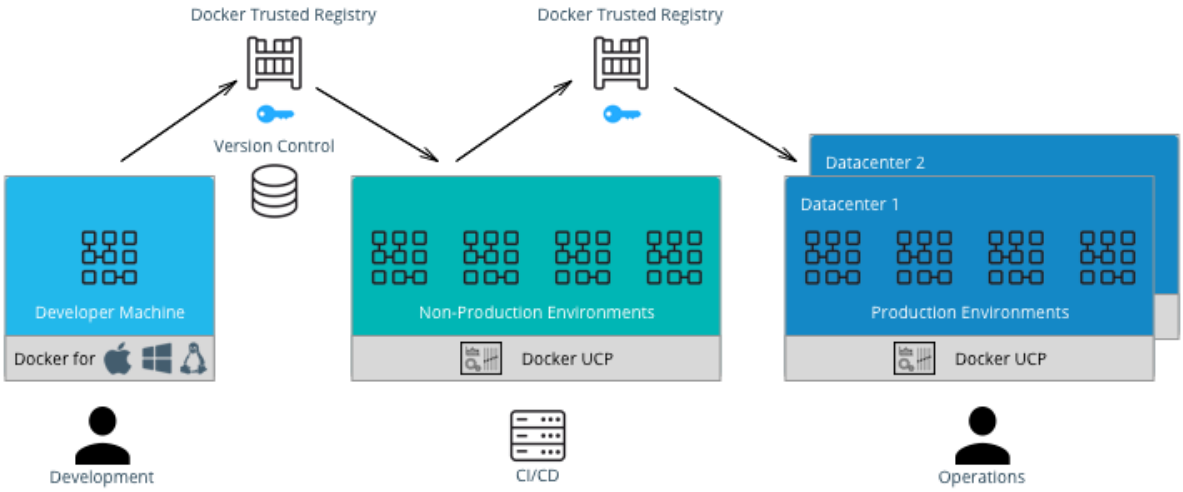
https://docs.docker.com/compose/samples-for-compose/

VIII. Workflow

https://success.docker.com/article/dev-pipeline#developerworkflow

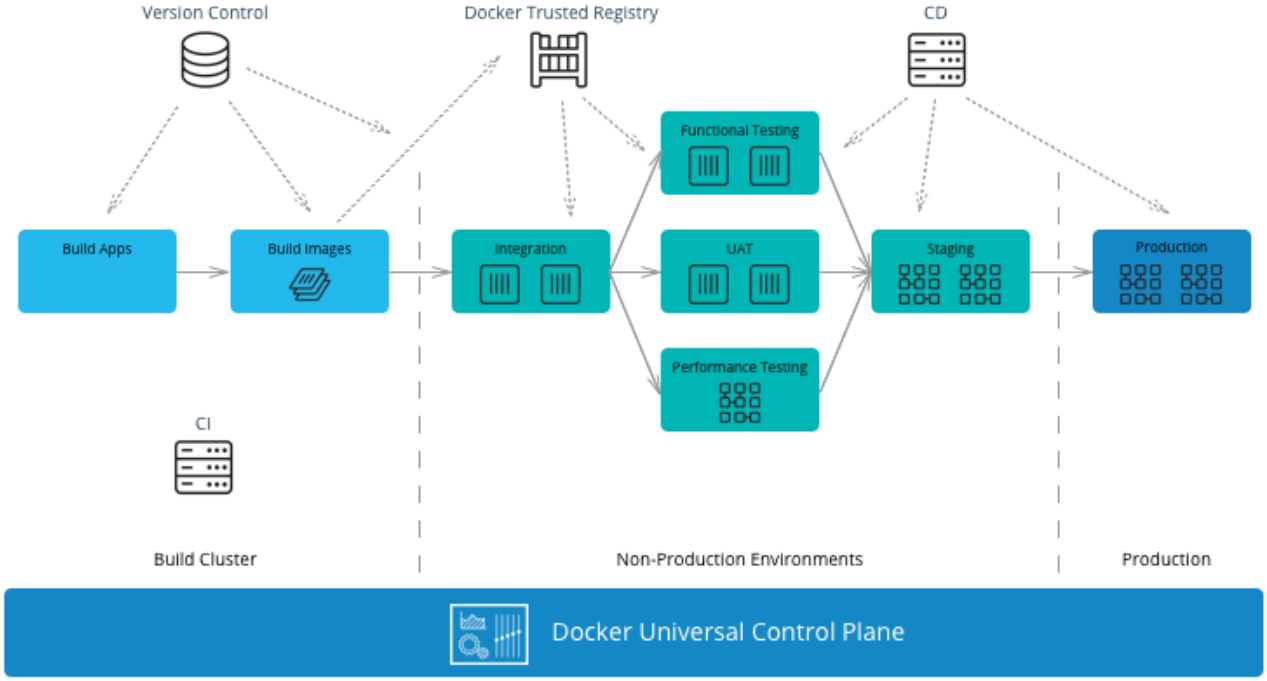
8.1 Kiến trúc chung

Team vận hành chịu trách nhiệm delivering và hỗ trợ xây dựng hạ tầng từ OS đến các thành phần middleware. Team developer chịu trách nhiệm build và maintain các ứng dụng. Đồng thời có 1 số kiểu CI để build và test tự động cũng như CD để triển khai các phiên bản lên các môi trường khác nhau.



8.2 Workflow chung

CI/CD workflow được thể hiện ở hình dưới đây



Bắt đầu từ bên trái, team developer building các ứng dụng, hệ thống CI/CD sau đó chạy unit test, sau đó đóng gói ứng dụng, build image trên Docker Universal Control Plane (UCP). Nếu quá trình test không có lỗi, image sẽ được chuyển tới Docker trusted registry (DTR). Image có thể được run trên môi trường non-production cho việc test. Nếu việc test pass thì image có thể được đăng ký lại và sau đó được deploy bởi team vận hành tới môi trường production.

8.3 UCP cluster (cluster môi trường test)

Thông thường các doanh nghiệp đòi hỏi môi trường production có tình bảo mật cao hơn, hạn chế truy cập từ người vận hành, hạ tầng tốt hơn, cấu hình có tính sẵn sàng cao, và có khả năng khôi phục sự cố hoàn toàn với nhiều datacenter. Môi trường non-production có các yêu cầu

khác với mục tiêu chính là để testing và đánh giá ứng dụng cho production. Interface giữa production và non-production là DTR

Trong môi trường doanh nghiệp, có hàng trăm team building và run các ứng dụng, cách tốt nhất là tách việc build từ các tài nguyên đang running. khi này thì quá trình build image sẽ không ảnh hưởng chất lượng và available của các service/container đang running.

có 2 phương pháp build image phổ biến sử dụng Docker EE:

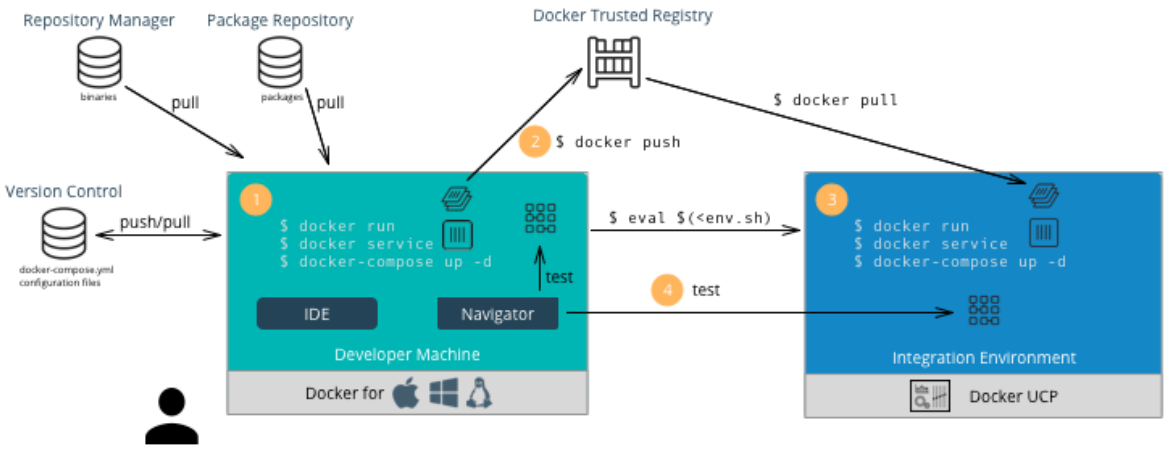
- Developer build image của mình và push chúng lên DTR: Phương pháp này phù hợp nếu không có hệ thống CI/CD và không có dedicated build cluster.
- Tiến trình CI/CD build image trên cluster rồi đẩy chúng lên DTR. Phương pháp này phù hợp với các doanh nghiệp muốn kiểm soát chất lượng các image được đẩy lên DTR. Các developer commit Dockerfile với version control. Sau đó chúng sẽ được phân tích và được kiểm soát việc tuân thủ các tiêu chuẩn của doanh nghiệp trước khi hệ thống CI/CD build image, test chúng và đẩy chúng lên DTR. Trong trường hợp này CI/CD agent nên chạy trực tiếp trên các dedicated build nodes.

8.4 DTR Clusters

Các doanh nghiệp thường có một master DTR cluster để cho phép doanh nghiệp thực hiện các tiến trình dư rà soát an toàn (Scaning security). Nếu image được kéo về từ một nơi global sẽ mất nhiều thời gian, khi đó ta cần sử dụng DTR Content Cache để tạo local caches.

8.5 Developer Workflow

Developer và các team ứng dụng thường duy trì các repository khác nhau trong nội bộ công ty để develop, deploy và test các ứng dụng. Dưới đây là sơ đồ workflow phổ biến của các developer sử dụng Docker EE



Workflow của developer có các bước sau:

1. **Develop ở local: t**rên machine của developer, hoặc môi trường developer build image, run container, và test các container ở local. có một số loại file và các repository tương ứng của chúng được sử dụng để build docker image.
   - Control version: là file text-based như Dockerfile, docker-compose.yml và các file cấu hình

- Repository manager: lưu giữ các binaries file lớn hơn như: Maven/Java, npm, NuGet, and RubyGems
- Package repository: lưu trữ các ứng dụng được đóng gói tương ứng với từng OS.

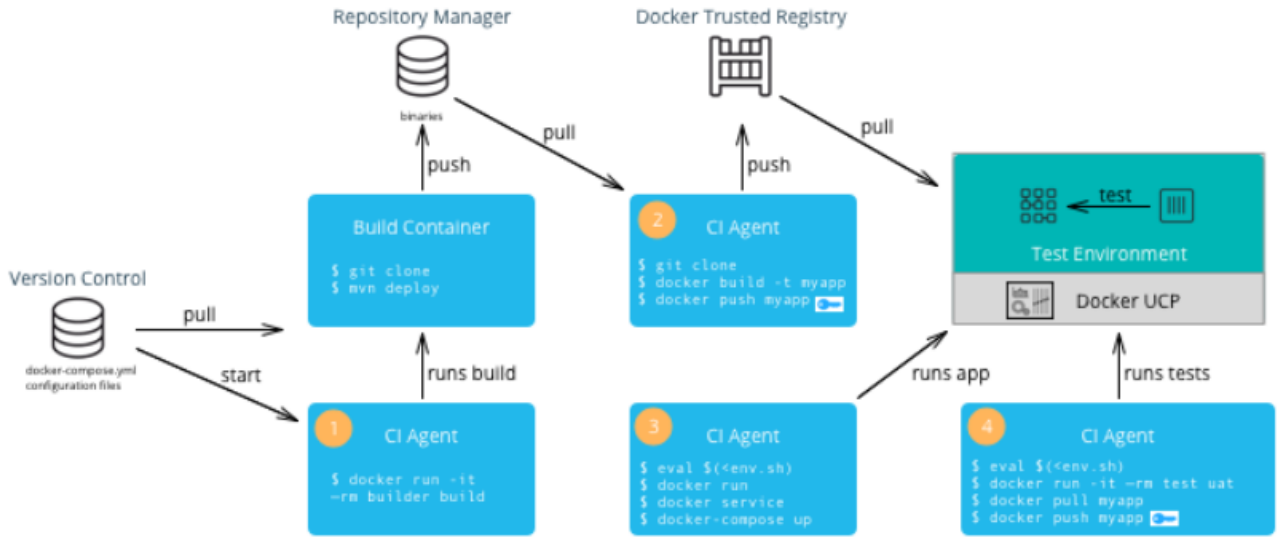
2. **Push image:** Khi image đã được test ở local, nó có thể được push lên DTR, các developer phải có tài khoản trên DTR và có thể push lên registry bằng tài khoản của mình để testing trên UCP.
3. **Deploy trên UCP**: developer muốn thực hiện deploy test trên môi trường tích hợp trong UCP trong trường hợp development machine không thể truy cập tới tất cả các tài nguyên để chạy toàn bộ ứng dụng. Đồng thời test ứng dụng đã scale phù hợp chưa nếu nó được deploy như 1 service. Trong trường hợp developer muốn truy cập bằng CLI để deploy ứng dụng trên UCP, sử dụng `$ eval $(<env.sh)` để trỏ Docker client tới UCP. Khi đó chạy các lệnh sau:

```
$ docker run -dit --name apache2 dtr.example.com/kathy.seaweed/apache2:1.0
$ docker-compose --project-name wordpress up -d $ docker service create --name apache2 dtr.example.com/kathy.seaweed/apache2:1.0
```

4. **Test ứng dụng**: Developer có thể test ứng dụng đã được deploy trên UCP từ local machine để đánh giá cấu hình của môi trường test
5. **Commit với Version Control**: khi ứng dụng đã được test trên UCP, developer có thể commit các file đã được sử dụng để tạo ứng dụng , image của nó và các ứng dụng của nó lên Version control. Commit này sẽ kích hoạt CI/CD workflow.

8.6 CI/CD Workflow

CI/CD platform sử dụng nhiều hệ thống khác nhau để tự động build, deploy và test các ứng dụng. Hình sau mô tả CI/CD workflow



CI/CD workflow gồm có những bước sau:

1. **Build ứng dụng**: Khi có thay đổi trên version control của ứng dụng sẽ kích hoạt build bởi CI agent. một build container start và lấy các tham số cụ thể cho ứng dụng.Container có thể run trên Docker host riêng hoặc UCP. Container có thể nhận source code từ

version control, chạy các lệnh từ công cụ build của ứng dụng và cuối cùng push kết quả lên **repository manager.**

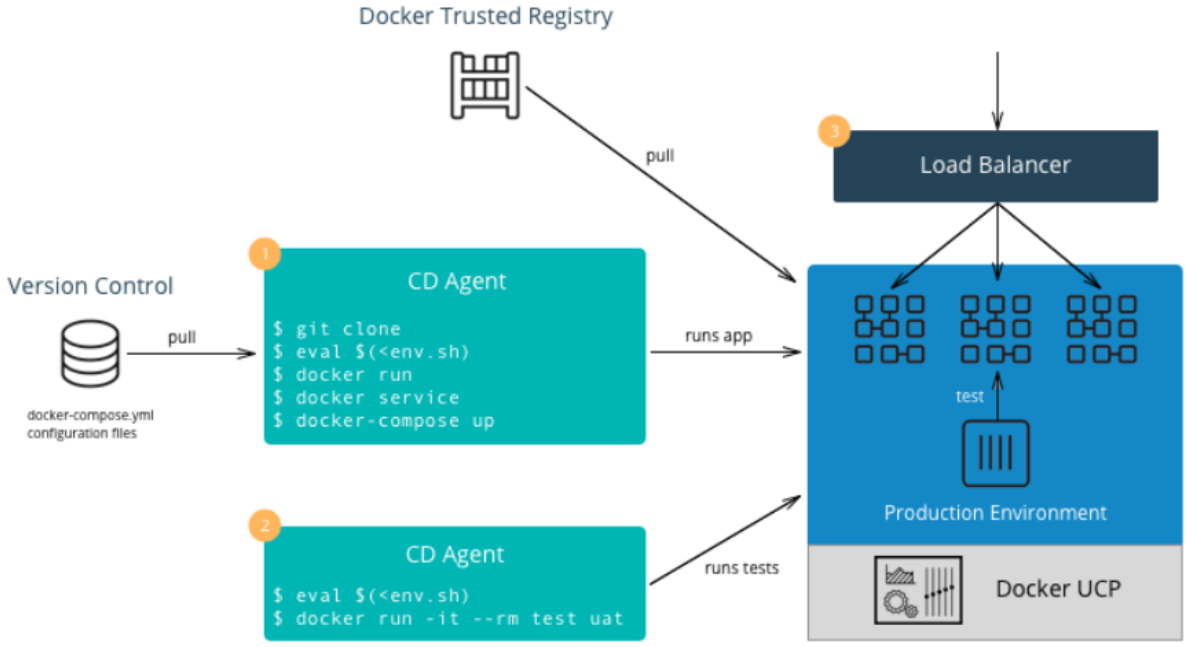
2. **Build image**: CI agent kéo Dockerfile và các file liên quan về để build image từ version control. Dockerfile được thiết lập để các artifact build ở phần trước được copy vào trong image, image sinh ra được đẩy lên DTR. Nếu Docker Content Trust đã được enable và Notary đã được cài đặt, khi đó image được đăng ký với CI/CD signature.
3. **Deploy ứng dụng**: CI agent có thể kéo cấu hình run-time từ version control và sử dụng chúng để deploy ứng dụng trên UCP thông qua truy cập CLI.
4. **Test ứng dụng**: CI agent deploy một test container để test ứng dụng được deploy ở phần trước. Nếu tất cả việc test đều tốt, image có thể được đăng ký với QA signature bằng cách kéo và đẩy image lên DTR. việc Push này sẽ kích hoạt Workflow của team vận hành.

TIP: CI agent cũng có thể được docker hóa. Tuy nhiên, khi chạy các lệnh Docker, cần truy cập tới engine của Docker host.

TIP: Sử dụng Docker image khi build cache. Khi ứng dụng được build trong 1 container, thường cần download các dependences nên cần cache hoặc proxy để tăng tốc độ.

8.7 Operations Workflow

Workflow vận hành có 2 phần: bắt đầu tại điểm khởi đầu của toàn bộ pipeline developing là tạo base image cho team development sử dụng. và kết thúc bằng việc pulling và deploying production bằng image từ developer team. Workflow tạo image giống như workflow của development. Hình sau sẽ mô tả workflow deploy image trong môi trường production.



Workflow của vận hành gồm các bước:

1. **Deploy ứng dụng:** deployment lên production có thể được kích hoạt tự động thông qua một sự thay đổi trên version control của ứng dụng hoặc có thể được kích hoạt từ team vận hành. Nó cũng có thể thực thi thủ công hoặc được thực hiện bởi CI/CD agent. 1 tag các file cấu hình deployment xác định môi trường production được pull từ version

control. Những file này gồm có 1 file docker-compose.yml hoặc các scripts để deploy service cũng như các file cấu hình. bảo mật bằng password hoặc certificate nên được sử dụng trong môi trường production . tức Docker 17.03 (Docker engine 1.13) Docker cung cấp quản lý bảo mật. Khi đó CD agent có thể deploy production trong UCP.

2. **Test ứng dụng:** CD agent deploy một test container để test ứng dụng được deployed ở trên, nếu tất cả các bước test đều pass thì ứng dụng sẵn sàng xử lý tải ứng dụng.
3. **Cân bằng tải:** Tùy thuộc vào mô hình deployment (Big Bang, Rolling, Canary, Blue Green, etc.) Một service cần bằng tải ngoài, DNS server hoặc router sẽ được cấu hình lại để send tất cả hoặc 1 phần các request tới ứng dụng được deploy mới. Version cũ của ứng dụng có thể được giữ lại cho roll-back, khi ứng dụng mới build ổn định thì có thể remove Version cũ đi.

https://success.docker.com/article/dev-pipeline#developerworkflow