Báo cáo thực thực tập công ty an ninh mạng viettel: Docker

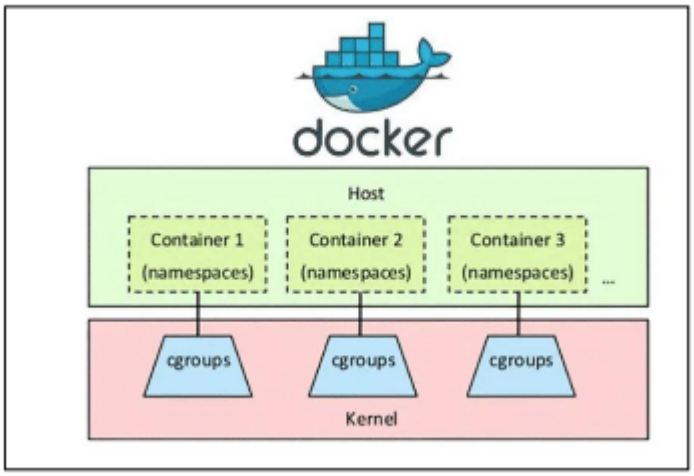
Bùi Hoàng Dũng

February 2024

Mục lục

1 Container

Container được định nghĩa là một thực thể đóng gói các software package. Các container cho phép một ứng dụng có thể được đóng gói với tất cả các thành phần cần thiết (Ví dụ như là tất cả các phần mềm liên quan, cơ sở dữ liệu, và các thành phần phụ thuộc khác). Container là phiên bản thực thi được sinh ra từ những file image, tương tự như ta sử dụng file iso để tạo một máy ảo hay là cài một hệ điều hành. Tất cả container trên cùng một host sẽ cùng chia sẻ và dùng chung hệ điều hành của host. Container cũng có khả năng khởi động nhanh và handle được nhiều dịch vụ.
Linux container: container đầu tiên được tạo ra từ hệ điều hành Linux. Về cơ bản một linux container gồm 2 thành phần chính đó là: linux namespaces và linux c-groups. Trong đó: **Linux namespace** là một chức năng của linux kernel cho phép phân đoạn kernel resources để sao cho một danh sách các processes này sẽ được cấp cho một danh sách các resource này còn danh sách các processes khác sé được cấp cho một danh sách các resource khác. Mỗi process đươc gán với một namespace thì chỉ có thể nhìn hoặc sử dụng resource được cấp cho chính namespace đó. Linux namespaces cung cấp cách để các process có thể nhìn thấy một tập hợp cụ thể hoặc các tài nguyên hệ thống như PID, filesystem, card mạng và các tài nguyên IPC(inter-process Communcation) .Vi dụ một namespace trong linux đó là PID namespace quản lý các processes với một danh sách cách ID khác biệt với các namespace khác. PID namespace sẽ thực hiện việc cấp ID cho một process. Khi môt process mới được tạo nó sẽ có một PID cho mỗi namespace. Còn đối với **linux cgroups** cũng là một tính năng của linux kernel cho phép việc tổ chức các process trong một group với hạn mới sử dụng resource riêng. Cgroups cung cấp cách thức quản lý tài nguyên, cho phép người quản trị có thể allocate resource một cách hiệu quả giữa các processes. Cgroups giúp ngăn chặn sự tranh chấp resources và đảm bảo resource được cung cấp cho các processes một cách công bằng. Nói chung, Linux namespaces tập trung vào việc tạo ra các môi trường tách biệt với hệ thống cho các processes còn linux cgroups tập trung vào việc quản lý resources được cấp phát giữa các nhóm processes khác nhau.



Hình 1: Container

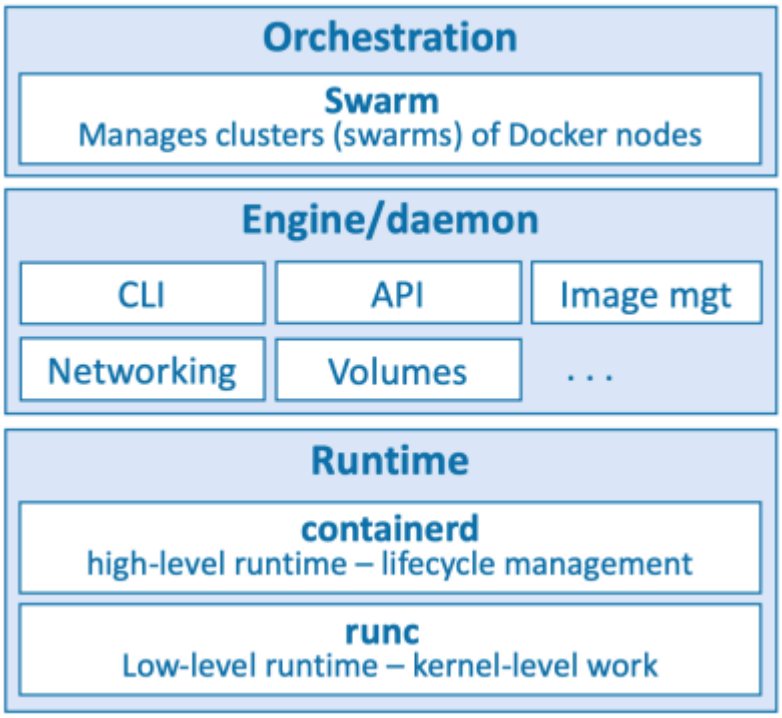
2 Docker

Docker là một phần mềm có thể được chạy trên windows và Linux. Nó thực hiện công viêc tạo, quản lý các container.

Hình 2: Docker logo

Khi nhắc đến Docker ta thường phải chú ý đến ba thứ quan trọng đó là:

- The runtime: thực hiện công việc ở tầng thấp nhất trong hệ thống và chịu trách nhiệm cho việc khởi động cũng như tạm dừng containers (runtime bao gồm cả việc xây dựng tất cả các thứ liên quan đến OS như là namespace và cgroups). Docker triển khai hai kiến trúc runtime với high-level runtime và low-level runtime được hoạt động cùng với nhau. Low-level runtime được gọi là *runc* có nhiệm vụ như là một interface kết nối với tầng OS và thực hiện nhiệm vụ bật tắt các container. High-level runtime được gọi là *containerd* nó quản lý vòng đời của một container bao gồm việc pull images và quản lý các runc instances. Hiểu đơn giản là runc sẽ thực hiện việc chạy các container còn containerd sẽ hướng đẫn runc nên bật hay tắt các container
- Docker demon(dockerd) nằm ở trên containerd thực hiện các tác vụ high-level như là exposing Docker API, quản lý các image, volumes, network,....
- The orchestrator: Docker cũng hỗ trợ việc quản lý các cluster gồm các nodes chạy Docker. Những cluster này được gọi là swarms và công nghệ để quản lý các cluster này gọi là Docker Swarm. Docker Swarm đơn giản và dễ cài đặt hơn Kubernetes những thiếu nhiều các tính năng nâng cao như Kubernetes đã có.



Hình 3: Docker architecture

2.1 Cài đặt Docker trên Linux

Bước 1:thực hiện xóa các tệp gây conflict với docker trước:

```
for pkg in docker.io docker-doc docker-compose docker-compose-v2 podman-docker containerd runc; do sudo
```

Bước 2: Setup docker apt repo:

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docke
  /linux/ubuntu \ $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

Bước 3: Tiến hành cài đặt Docker:

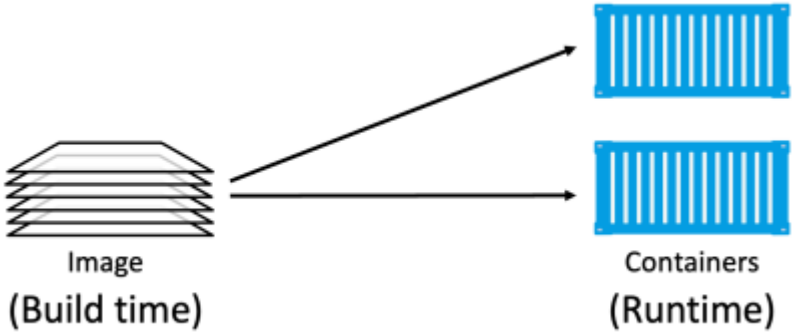
```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

Bước 4: Test xem Docker đã được cài đặt thành công chưa bằng cách run image hello world

```
sudo docker run hello-world
```

2.2 Docker image

Docker image có thể được coi như là một container đã được tắt. Trong thực tế, ta có thể tạm dừng một container và tạo một image mới từ nó. Do đó mà image được coi như là build-time constructs còn container được coi như là một run-time constructs.

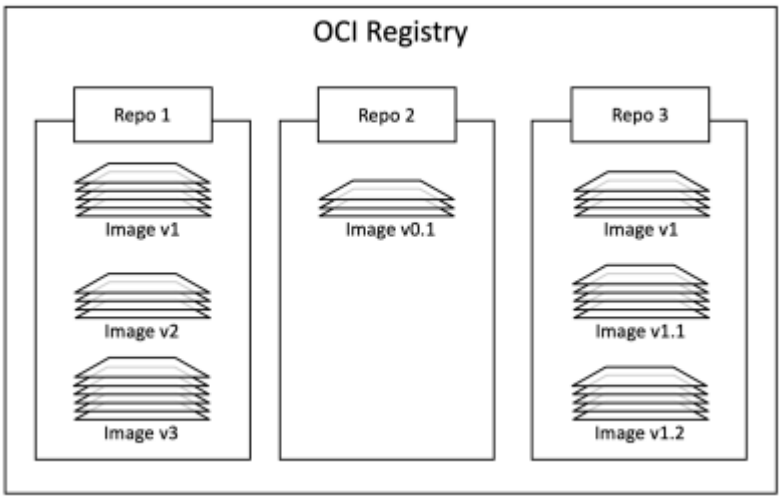


Hình 4: Docker image

Hình trên cho thấy một high-level view mỗi quan hệ giữa images và containers. Khi ta bật một container từ một image, 2 construct trên sẽ phụ thuộc vào nhau. Ta sẽ không thể xóa image cho đến khi container cuối cùng được tạo bởi nó bị dừng lại hoặc bị loại bỏ. Mục đích của một container là để chạy một single application hoặc service. Điều đó có nghĩa là nó chỉ cần code và các dependencies của app. Do đó images thường có kích thước nhỏ và lượt bỏ tất cả các phần không cần thiết. Images sẽ không bao gồm một kernel. Đó là bởi vi containers chia sẻ kernel của host mà chúng được chạy.
Pulling images:Quá trình để có một images ở Docker host gọi là pulling. Image sẽ được tìm trước ở trên local repo rồi mới được tìm ở trên Dockerhub để pull về host. **Image registries**: Các image được lưu ở một nơi tập trung gọi là registries. Registries thực hiện lưu trữ các image container một cách an toàn và khiến chúng có thể kết nối dễ dành với môi trường khác, Registry thông thường phổ biến nhất là Docker Hub, Docker client sẽ được mặc định chọn Dockerhub làm registry. Một registry sẽ bao gồm một hoặc nhiều image repo. Và một image repo sẽ chứa một hoặc nhiều images.

```
$ docker pull ubuntu:latest
latest: Pulling from library/ubuntu
952132ac251a: Pull complete
82659f8f1b76: Pull complete
c19118ca682d: Pull complete
8296858250fe: Pull complete
24e0251a0e2c: Pull complete
Digest: sha256:f4691c96e6bbaa99d...28ae95a60369c506dd6e6f6ab
Status: Downloaded newer image for ubuntu:latest
docker.io/ubuntu:latest
```

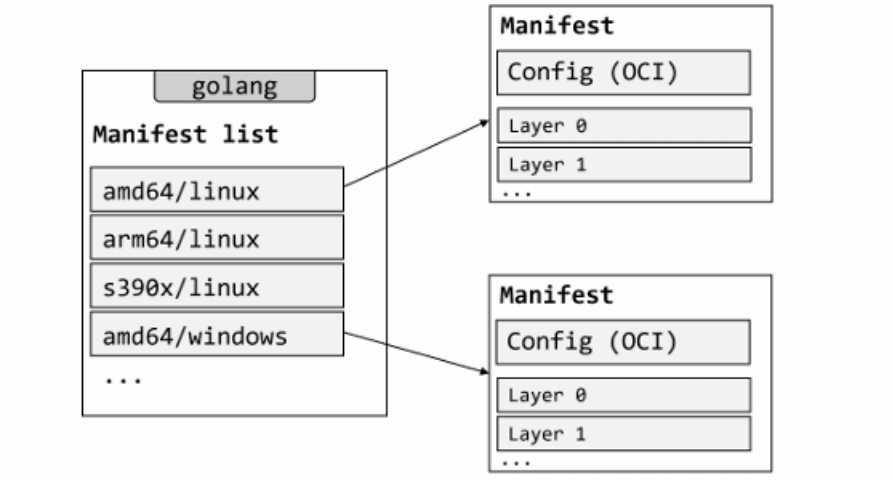
Hình 6: Caption



Hình 5: Registry

Một images có địa chỉ từ một repo chính thức sẽ thường được cung cấp repo name và tag name được ngăn cách bởi dấu :. Một image có thể có nhiều tag. Nếu ta không điền tag name sau repo name thì Docker sẽ giả định là ta đang muốn pull một image với tag name là latest. Nếu repo dó không có image tage là latest thì command sẽ thất bại. Nếu muốn pull một image về từ một nguồn không chính thống, ta cần phải điền tên tổ chức hoặc cá nhân nào đó sở hữu image đó trên Dockerhub. Nếu ta muốn pull image từ một registry thứ 3 thì ta cần phải điền repo name với DNS name của registry đó.
Khi muốn tìm kiếm một image trên Dockerhub ta sử dụng command: *docker search <tên repo>*. Kết quả tìm kiếm sẽ bao gồm cả các image không chính thức. **Images và layers** Một Docker image là một collection các read-only layers được kết nối với nhau. Mỗi layer có thể có một hoặc nhiều files. Docker thực hiện việc stacking các layer và biểu diễn chúng như một single unified object. Như hình trên ta có thể thấy image này có 5 layer và mỗi layer đều có một ID khác nhau. Một cách khác để có thể xem được là image có bao nhiêu layer là ta sử dụng câu lệnh *docker inspect*. Tất cả các images đều được bắt đầu với một base layer, những layer mới được tạo thêm sẽ được xếp lên đầu. Ví dụ khi ta xây dựng một python application thì ta cần phải tuân thủ rằng tất cả ứng dụng đều được chạy trên một hệ điều hành ví dụ như là Ubuntu. Do đó mà Ubuntu có thể làm base layer của container chứa ứng dụng. Một điều quan trọng cần phải hiểu rằng là khi một layer được thêm vào thì image luôn luôn là sự tổng hợp của tất cả các layer được xếp chồng lên nhau theo thứ tự mà chúng được thêm vào.
Nhiều image có thể dùng và chia sẻ các layers chung cho lẫn nhau dẫn đến sự tối ưu về hiệu quả cho bộ

nhớ cũng như performance. Doker hỗ trợ rất nhiều hệ điều hành và các kiến trúc CPU khác nhau. Vậy nên mỗi image đều có một phiên bản riêng để chạy trên các hệ điều hành và kiến trúc CPU khác nhau. Tùy thuộc vào hệ điều hành và kiến trúc chip được sử dụng là gì mà Docker sẽ pull image sao cho phù hợp. Để làm được điều này thì Registry API sẽ hỗ trợ 2 thủ tục là manifest list và manifest. Hiểu đơn giản manifest list là một danh sách các kiến trúc CPU mà image đó được hỗ trợ, mỗi phần tử trong đó gọi là manifest chứa image config và layer data.



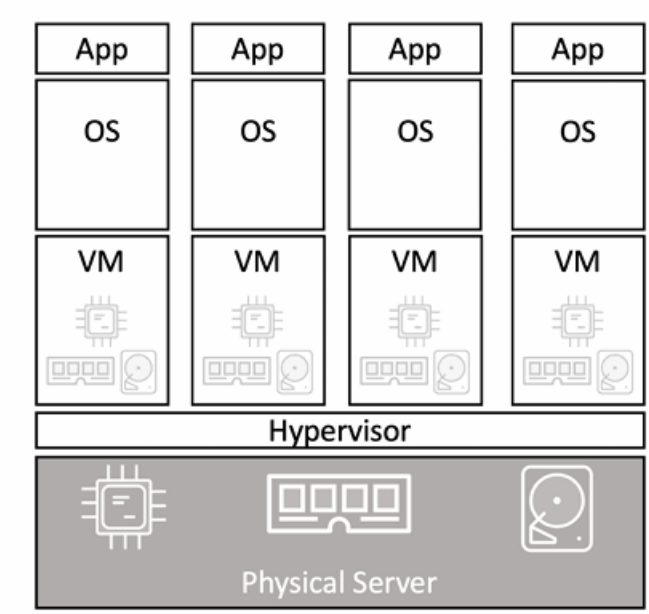
Hình 7: Manifest

Khi muốn xóa một image thì ta sử dụng câu lệnh: **docker rmi**. Lưu ý khi một container đang được chạy thì ta không được xóa image đó. Nếu một image layer được chia sẻ với một image khác thì nó sẽ không xóa tất cả các image sử dụng nó cho đến khi nó được xóa hết.

2.3 Docker container

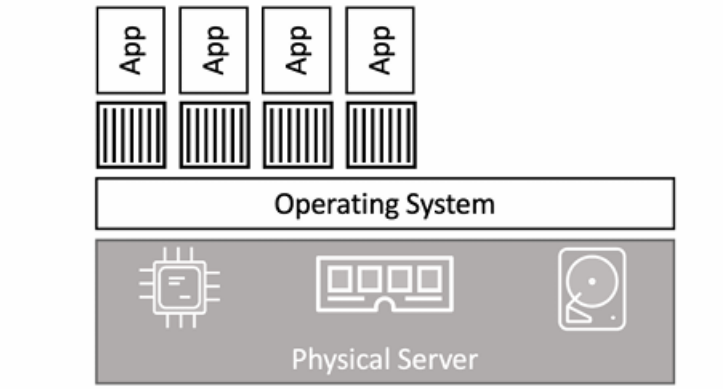
2.3.1 So sách giữa container và VM

Container và VM điều cần một host để chạy trên. Host có thể là bất cứ thứ gì như là laptop, server,..... **Trong mô hình VM**, server vật lý được khởi động và hypervisor thực hiện boots. Trong quá trình boot thì hypervisor sẽ thu thập thông tin về tài nguyên vật lý như là CPU, RAM, storage,...Sau đó thực hiện ảo hóa các tài nguyên đó bằng cách chia nhỏ những resource đó ra và đóng gói lại tạo thành một VM. VM có được muốn sử dụng để host một application thì phải thực hiện cài đặt hệ điều hành.



Hình 8: VM

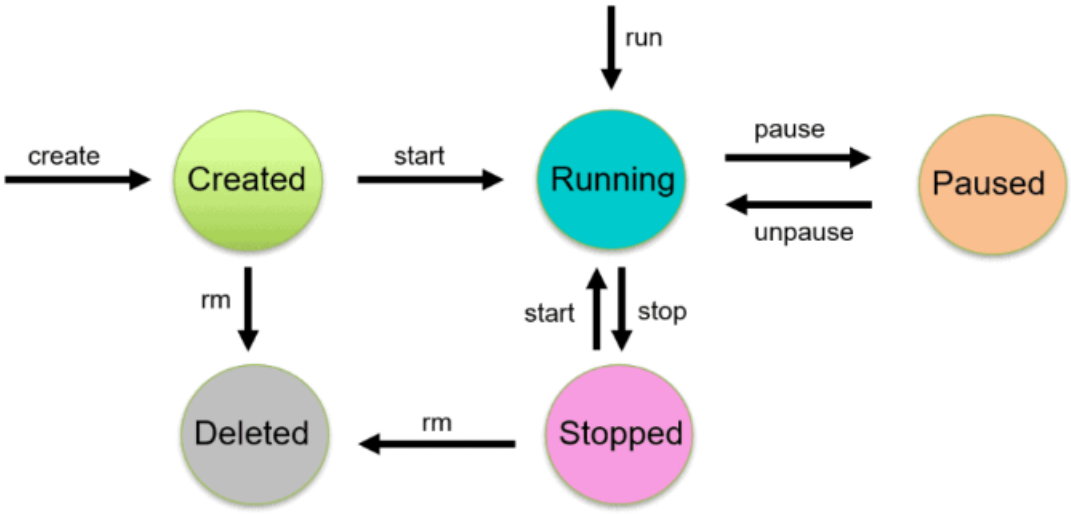
Trong mô hình container, khi server được khởi động và hệ điều hành được boots, Host của hệ điều hành đó sẽ thu thập tất cả thông tin về tài nguyên phần cứng. Container engine (Docker) sẽ thực hiện chia nhỏ và ảo hóa tất cả các OS'resource (filesystem, networkstack, process tree,...) và gói vào một package gọi là container. Mỗi container sẽ được xem như là một hệ điều hành thực sự và có thể chạy ứng dụng.



Hình 9: Container

Nhìn chung thì hypervisor sẽ thực hiện ảo hóa phần cứng-chia tất cả các tài nguyên phần cứng ra thành các phần ảo gọi là các VMs. Còn đối với container thì thực hiện ảo hóa OS - chia tất cả các tài nguyên của hệ điều hành thành các phiên bản ảo hóa. Ở dây ta có thể thấy rằng việc dùng một container để chạy application nhanh hơn rất nhiều khi ta dùng một VM để chạy một ứng dụng. Bởi lẽ, container chỉ thực hiện việc chạy ứng dụng, phần kernel luôn luôn hoạt động và chạy trên host. Trong khi đó mỗi VM cần boot full hết hệ điều hành thì mới có thể chạy được application.

2.3.2 Vòng đời của container



Hình 10: Life cycle of container

Lưu ý rằng khi container được stop thì những dữ liệu được lưu trong container đó vẫn được giữ nguyên mà không bị mất. Tuy nhiên khi host của container đó bị fail thì dữ liệu cũng sẽ bị mất. Container được thiết kế là một object không bị thay đổi do đó ta không nên thực hiện ghi dữ liệu lên chúng dù có thể thực hiện được, Tổng quan lại, ta có thể dừng, khởi động lại một container nhiều lần mà không bị lo mất dữ liệu trừ khi host bị fail.

Chạy một container cơ bản Thực hiện clone từ github: *https://github.com/nigelpoulton/ddd-book.git*. Ở đây em sẽ thực hiện đọc hiểu một file Dockerfile trước:

```
# Test web-app to use with Pluralsight courses and Docker Deep Dive book
FROM alpine

LABEL maintainer="nigelpoulton@hotmail.com"

# Install Node and NPM
RUN apk add --update nodejs npm curl

# Copy app to /src
COPY . /src

WORKDIR /src

# Install dependencies
RUN  npm install

EXPOSE 8080

ENTRYPOINT ["node", "./app.js"]
```

Hình 11: Dockerfile

Thông thường một Dockerfile thường được bắt đầu bằng *FROM* instruction. Dòng này có ý nghĩa là Docker sẽ thực hiện pull image và sử dụng nó như là base layer cho image mà Dockerfile sẽ build. App sẽ được định nghĩa trong Dọckerfile như là một Linux app. Dòng tiếp theo là thêm metadata cho image về người quản trị image. Dòng tiếp theo là sử dụng *apk* package manager để cài đặt nodejs và nodejs-npm vào trong image. Điều này được thực hiện bằng cách app một layer và cài đặt các packages vào layer này. Dòng tiếp theo có ý nghĩa là tạo một layer mới và copy tất cả các file application và file dependency vào trong thư mục */src*. Dòng tiếp theo là định nghĩa thư mục làm việc cho image, dòng này sẽ không tạo thêm layer mà chỉ như là một bước thêm metadata cho image. Dòng tiếp theo thực hiện cài đặt các dependencies được liệt kê trong file *package.json* vào trong một layer mới. Do đó image này sẽ có 4 layer. Dòng cuối cùng miêu tả application này expose một web service ở port 8080, lệnh cho application được chạy khi container được khởi động.
Sau khi xem Dockerfile ta tiến hành build Dockerfile này để tạo thành một image qua câu lệnh **docker build -t ddd=book:ch8.1 .**
Thực hiện chạy container qua câu lệnh **docker run -d –name c1 -p 80:8080 ddd-book:ch8.1**.
Kiểm tra container đã được chạy chưa qua câu lệnh **docker ps**.

```
127 buidung@buidung:~/ddd-book/web-app$ docker ps
CONTAINER ID   IMAGE           COMMAND           CREATED         STATUS
     PORTS                                  NAMES
15564780eaa4   ddd-book:ch8.1   "node ./app.js"   11 seconds ago   Up 11 s
ds   0.0.0.0:80->8080/tcp, :::80->8080/tcp   c1
```

Hình 12: Docker ps

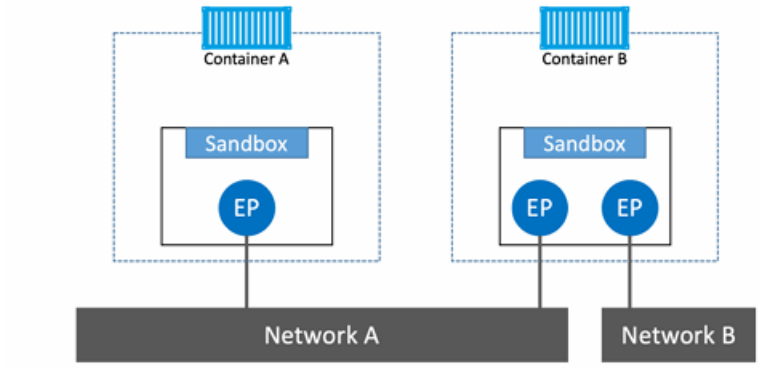
2.4 Docker network

Docker networking có thể được tạo nên bởi 3 thành phần riêng biệt:

- The Container Network Model(CNM) là bản thiết kế của Docker network, định nghĩa cách thức mà các block trong Docker Network được liên kết với nhau như nào
- Libnetwork: là phiên bản triển trong của CNM
- Drivers: Mở rộng model bằng cách triển khai thêm những topo mạng đặc biệt khác

CNM định nghĩa 3 block trong docker network:

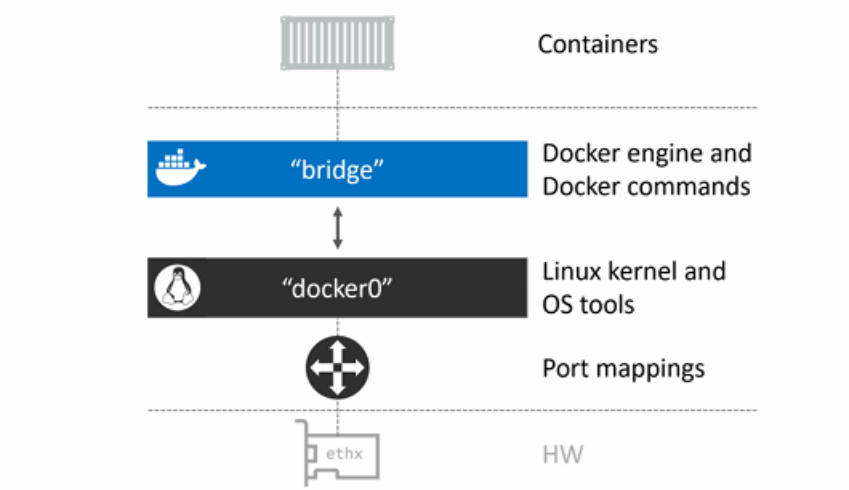
- Sandbox: là một phần tử network tách biệt ở bên trong container. Nó bao gồm Ethernet interfaces, port, routing tables, DNS config.
- Endpoints: là một network interface ảo có nhiệm vụ tạo kết nối.
- Networks là một phần mềm được triển khai của switch, thực hiện việc group và isolate một tập các endpoints cần để kết nối



Hình 13: Docker network

Trong hình trên, container A có một interface (endpoint) và được kết nối đến với network A, container B có 2 interface kết nối đến network A và network B. Hai container này có thể giao tiếp với nhau vì cùng kết nối với một mạng. Tuy nhiên 2 endpoint của container B không thể kết nối được với nhau nếu không có sự hỗ trợ của layer 3 router. Endpoint ở đây hoạt động như một interface trong một mạng bình thường. Network stack của các container được isolated với tầng Ó thông qua sandboxes và chúng chỉ có thể kết nối được với nhau thông qua một mạng.

Single-host bridge networks: Đây là loại topo đơn giản nhất của docker network. Ở trong mạng này các container thuộc cùng một host có thể kết nối được với nhau thông qua một layer 2 switch. Docker trên Linux tạo single-host bridge networks với built-in bridge driver. Mỗi Docker host có một single-host bridge network mặc định. Mặc định các container mới được tạo sẽ được gán vào mạng này nếu không specify cờ –network khi tạo container. Docker network loại này được xây dựng dựa trên module bridge driver của linux kernel.



Hình 14: Single-host bridge network

Để chứng minh thi em sẽ thực hiện tạo 2 container trong cùng một host và test ping giữa 2 container:

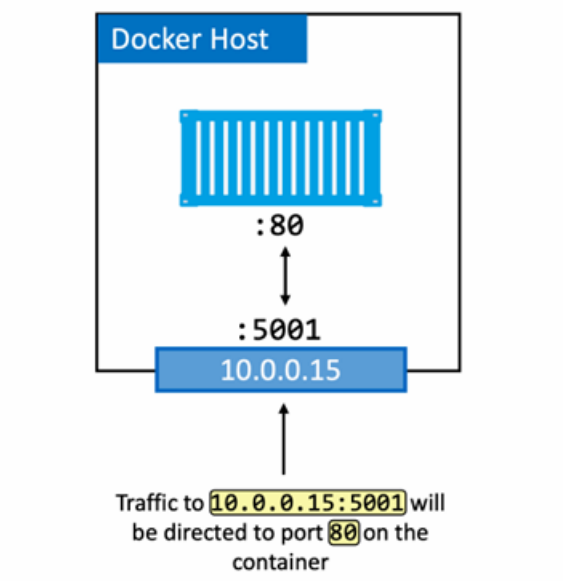
```
#Tạo container d1 và vào terminal của container qua cờ it
docker run -it --name d1 alpine:latest /bin/sh
#Tạo contaier d2 với image alpine và để sleep 1d
docker run --name d2 alpine:latest sleep 1d
```

Thực hiện test ping giữa d1 và d2:

```
125 buidung@buidung:~/ddd-book/multi-contianer$ docker run -it --name d1 alpin
e:latest /bin/sh
/ # ping c1
ping: bad address 'c1'
/ # ping c1
ping: bad address 'c1'
/ # ping d2
ping: bad address 'd2'
/ # ping 172.17.0.4
PING 172.17.0.4 (172.17.0.4): 56 data bytes
64 bytes from 172.17.0.4: seq=0 ttl=64 time=0.397 ms
64 bytes from 172.17.0.4: seq=1 ttl=64 time=0.093 ms
64 bytes from 172.17.0.4: seq=2 ttl=64 time=0.063 ms
64 bytes from 172.17.0.4: seq=3 ttl=64 time=0.090 ms
64 bytes from 172.17.0.4: seq=4 ttl=64 time=0.121 ms
64 bytes from 172.17.0.4: seq=5 ttl=64 time=0.121 ms
^C
--- 172.17.0.4 ping statistics ---
6 packets transmitted, 6 packets received, 0% packet loss
round-trip min/avg/max = 0.063/0.147/0.397 ms
```

Hình 15: Test d1 ping d2

Một khái niệm quan trọng nữa là port mapping. Port mapping thực hiện map một container tới một port trên Docker host. Tất cả các traffic đi vào port của host sẽ được di chuyển đến thẳng container.



Hình 16: Port mapping

Để chứng minh thì em sẽ thực hiện bài lab như sau:

```
##Chạy một Nginx web server container và map port 80 của container tới port 5001 của Docker host
docker run -d --name web --publish 5001:80 nginx
```

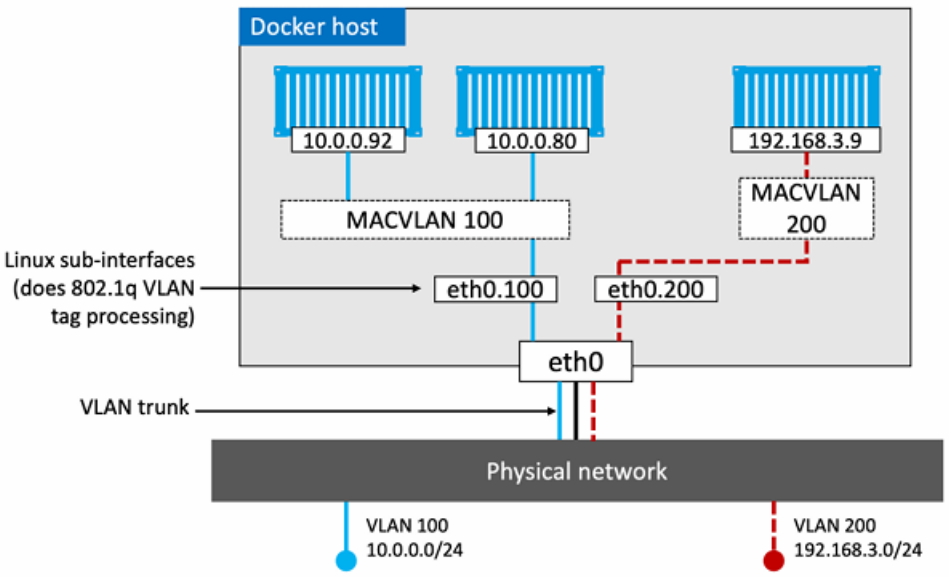
Sau khi chạy thực hiện curl đến localhost:5001 để chứng minh traffic được truyền tới container.

```
buidung@buidung:~$ curl localhost:5001
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully insta
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
```

Hình 17: Curl

Multi-host overlay network: overlay networks là một mạng multi-host đảm bảo việc các containers khác host có thể kết nối được với nhau. Docker cung cấp một driver cho overlay networks bằng cách thêm cờ **-d overlay** khi ta tiến hành tạo một network. Khả năng kết nối của một container app đến

các hệ thống bên ngoài và mạng vật lý bên ngoài là rất quan trọng. Built-in MACVLAN driver được tạo nên với mục đích này. Nó cho mỗi container một địa chỉ IP và địa chỉ MAC riêng của nó trên mạng local bên ngoài hoặc VLAN. Điều này khiến cho container được xem như là một server vật lý hay một VM. MACVLAN không yêu cầu phải port mapping hay additional bridges. Tuy nhiên nó yêu cầu host NIC phải phải ở trong **promiscuous mode** - tức là khi ở chế độ này thì container có thể nghe tất cả các traffic trên mạng bao gồm cả những gói dữ liệu không dành cho nó.



Hình 18: MACVLAN

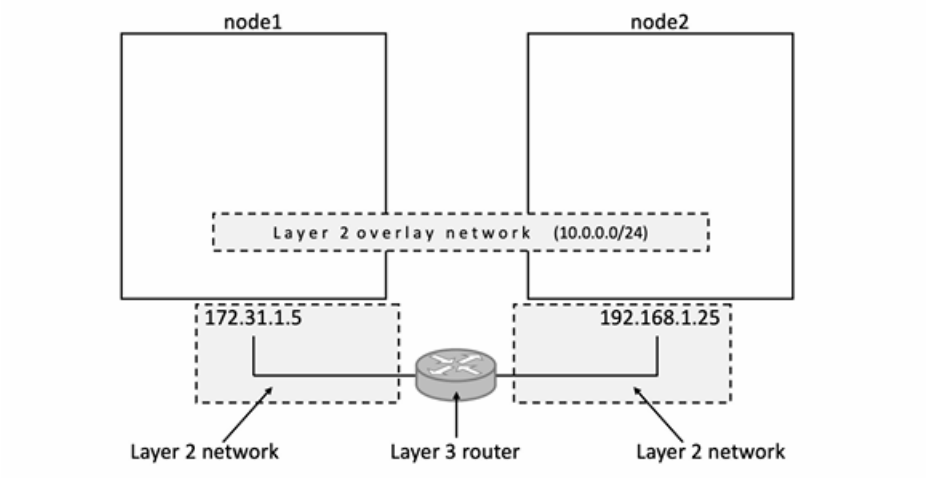
Trong hình bên trên, là ví dụ của một mô hình MACVLAN. Đối với MACVLAN100 khi ta thực hiện tạo một mạng macvlan100 thì Docker sẽ tạo ra một mạng MACVLAN và một interface ảo kèm theo để nối với NIC của host. Lưu ý rằng khi tạo một mạng MACVLAN ta cần phải nêu các thông tin về thông tin subnet, gateway, dải địa chị IP (dải này phải độc nhất cho Docker) mà nó có thể cấp cho các container và interface nào của host được sử dụng. MACVLAN được nối với VLAN100 của mạng vật lý. Ngoài ra docker còn hỗ trợ VLAN trunking điều này có nghĩa ta có thể tạo ra nhiều MACVLAN network trong một Docker host.
Sau đây em sẽ thực hiện tạo một Docker Swarm với 1 manager node và 1 worker node để test kết nối giữa 2 container khác node:

```
#Tạo docker swarm trên manager node:
docker swarm init --advertise-addr=192.168.74.139 --listen-addr=192.168.74.139:2377
#Join docker swarm trên worker node:
docker join --token ..... 192.168.74.139
#Tạo một overlay network trên manager node:
docker network create -d overlay uber-net
#Gán một service đến overlay network vừa tạo
docker service create --name test --network uber-net --replicas 2 ubuntu sleep infinity
#Thực hiện vào terminal của container ở manager node và traceroute đến container ở worker node
docker exec -it 1cc...
```

```
#Traceroute đến container ở worker node
traceroute 10.0.1.3
```

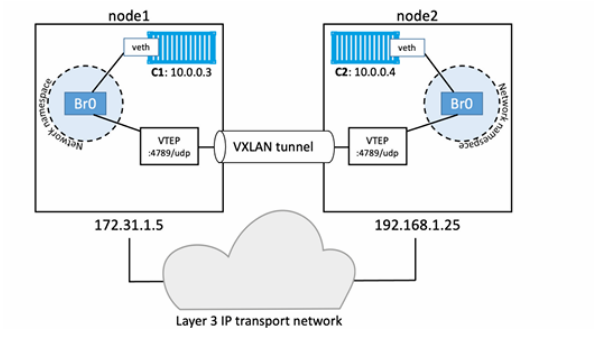
```
buidung@buidung:~/ddd-book/multi-contianer$ docker exec -it 1cc8b3da1047 b
root@1cc8b3da1047:/# traceroute 10.0.1.3
traceroute to 10.0.1.3 (10.0.1.3), 64 hops max
  1   10.0.1.3  0.477ms  0.495ms  0.347ms
```

Hình 19: Traceroute



Hình 20: VXLAN

Hình trên diễn tả 2 container có thể được kết nối thông qua một công nghệ gọi là VXLAN. VXLAN là một giao thức đóng gói router và những thiết bị mạng sẽ nhìn nó như một gói tin IP/UDP bình thường. VXLAN tunnel được tạo thông qua lớp underlay network bên dưới. Ở cuối của mỗi VXLAN tunnel là một VXLAN Tunnel Endpoint(VTEP). VTEP này sẽ thực hiện việc đóng gói và mở gói những traffic đi vào và đi ra khỏi tunnel. Ta sẽ cùng đi vào chi tiết việc gì sẽ xảy ra khi ta tạo một single overlay network cho containers.



Hình 21: VTEP

Đầu tiên một sandbox mới (network namespace) sẽ được tạo trên mỗi host. Một virtual switch được gọi là **Br0** được tạo trong sandbox đó. Một VTEP cũng được tạo và được nối với Br0 và network stack

của host. VTEP này sẽ lấy địa chỉ IP của host và bọc với header layer 4 UDP ở port 4789. Khi đó VXLAN overlay đã được tạo nên và sẵn sáng được sử dụng. Mỗi container trên mỗi host đều có một virtual interface của riêng nó và được cắm vào Br0 switch và từ đó có thể truyền được dữ liệu thông qua Br0 này đến với VXLAN tunnel.

2.5 Docker volume

Như ta đã nói ở phần trước, container được thiết kế với mục đích không bị thay đổi, Điều đó có nghĩa là container chỉ nên được đặt trong chế độ read-only. Nếu có điều gí đó cần thay đổi với container ta nên thự hiện tạo lại một container mới và update những thay đổi vào container mới này để thay thế cho container cũ. Tuy nhiên nhiều ứng dụng yêu cầu việc đọc và ghi dữ liệu để chạy và không thê chạy trên read-only filesystem. Để xử lý việc đó, Docker container đã thêm một read-write layer trên đầu của read-only images mà nó thuộc về.

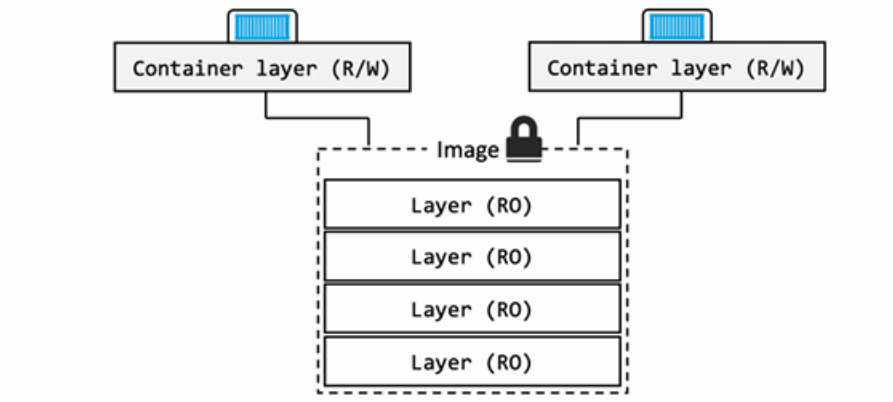


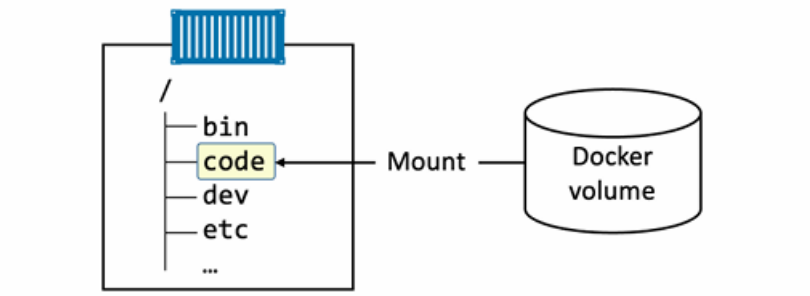
Figure 13.1 Ephemeral container storage

Hình 22: R-W layer

Tuy nhiên nếu như container bị xóa thì layer này sẽ sẽ biến mất theo container và những dữ liệu được application ghi lại cũng mất theo. Vậy nên **docker volume** được sử dụng để lưu dữ liệu một cách mãi mãi. Có 3 lý do khiến docker volume được sử dụng để lưu dữ liệu đó chính là:

- Volumes là một object độc lập và không được gán với vòng đời của một container
- Volumes có thể được ánh xạ đến với các hệ thống lưu trữ bên ngoài
- Volumes có thể được kết nối cho nhiều container khác Docker host và chia sẻ dữ liệu giống nhau

Để sử dụng volume cho container thì sau khi ta tạo một container ta sẽ mount volume vào container vừa được tạo. Volume được mount vào một thư mục trong filesystem của container và tất cả những dữ liệu được lưu vào thư mục này sẽ được lưu vào trong volume. Do đó khi xóa container đi thì dữ liệu sẽ không bị mất đi.



Hình 23: Docker volume

Lab: tạo và quản lý Docker volumes Để tạo một docker volume ta thực hiện câu lệnh:

```
docker volume create myvol
```

```
buidung@buidung:~$ docker volume inspect myvol
[
    {
        "CreatedAt": "2024-03-12T15:10:07Z",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/myvol/_data",
        "Name": "myvol",
        "Options": null,
        "Scope": "local"
    }
]
```

Hình 24: Caption

Ở đây ta có thể thấy *Driver* và *Scope* đều có giá trị là local. Điều này có nghĩa là volume được tạo bởi local driver và chỉ available với những container cùng chung một Docker host, Mountpoint chỉ ra vị trí của volume trên Docker host. Tất cả các volume được tạo bởi local driver đều được lưu ở đường dẫn */var/lib/docker/volumes* trên Linux. Điều này có nghĩa ta có thể tương tác trực tiếp với volume này từ Docker host.

Lab: Deploy microservice bằng docker compose

Docker compose là một file YAML để xây dựng các ứng dụng sử dụng multi-container. Trong bài lab này em sẽ xây dựng một ứng dụng có 2 service là web-frontend và redis để đếm các request đến webpage (lưu trong redis cache).

```
networks:
  counter-net:

volumes:
  counter-vol:

services:
  web-fe:
    build: .
    command: python app.py
    ports:
      - target: 8080
        published: 5001
    networks:
      - counter-net
    volumes:
      - type: volume
        source: counter-vol
        target: /app
  redis:
    image: "redis:alpine"
    networks:
      counter-net:
```

Hình 25: compose.yaml

Ở đây ta quan tâm đến 3 phần của file compose:

- services: nơi mà ta định nghĩa các microservice của application
- networks: nói cho Docker tạo một network mới.
- volumes: Chỉ ra volume sử dụng.

Với webfrontend service, ta cho Docker những chỉ dẫn như sau:

- build: thực hiện build một image mới sử dụng Dockerfile trong thư mục hiện. Image này có thể được sử dụng lần sau để tạo container cho service
- command: python app.py thực hiện chạy file app.py ở mỗi container có service này. File app.py này phải tồn tại trong image và python phải được cài đặt trong image
- ports: thực hiện việc port mapping giữa port của container và port của Docker host
- networks: gán network cho service của container
- volumes: Định nghĩa docker volume được mount vào trong container.

Nói chung, compose sẽ thực hiện nhiệm vụ hướng dẫn Docker deploy từng container một. Nó sẽ dựa vào Dockerfile trong cùng một thư mục. Image này sẽ được khởi chạy như là một container và chạy app.py như là một service.
Đối với redis service thì đơn giản hơn:

- image: thực hiện pull image redis:alpine về
- network: gán cho redis vào mạng counter-net;

Cả hai service trên đều được deploy trên cùng một network và có thể nhìn thấy được lẫn nhau. Sau khi đã hiểu hết ý nghĩa của file compose thực hiện chạy file compose bằng câu lệnh **docker compose up &**. Sau khi chạy xong docker compose ta thực hiện test trang web vừa tạo bằng container.

```
root@buidung:/home/buidung/ddd-book/multi-contianer/app# curl http://0.0.0.0:5001
web-fe-1  | 172.18.0.1 - - [13/Mar/2024 12:35:20] "GET / HTTP/1.1" 200 -
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Docker FTW</title>
        <link rel="stylesheet" href="/static/css/main.css">
    </head>
    <body>
        <img src="/static/images/image.png" alt="NP logo" class="center">
        <header>
        <div class="container">
            <h1 class="logo">Docker FTW</h1>
        </div>
        </header>
    <div class="container">
        <h2>Hit refresh if you think Sunderland AFC are the greatest football team in the world ;-)</h2>
        <h3>You've refreshed <b>1</b> times</h3>
    </div>
  </body>
```

Hình 26: Result

Đối với docker compose để quản lý các app qua compose thì cần phải chú ý các câu lệnh sau:

```
#Để down các application được deploy bằng docker compose thì ta sử dụng câu lệnh:
docker compose down
#Để liệt kê trạng thái của các app sử dụng:
docker compose ps
#Để liệt kê các processes chạy bên trong container của mỗi service sử dụng câu lệnh:
docker compose top
```

Lưu ý khi down một docker compose thì volume được gán ở trong docker compose sẽ không bị xóa. Để thực hiện xóa tất cả các volumes được attach trong file docker compose thì ta sử dụng cờ –volumes ở câu lệnh docker compose down. Đối với những network và volumes được attach ở trong file compose mà chưa tồn tài trước đó. Khi thực hiện build compose thì docker sẽ builds network và volume trước sau đó mới deploy service. Việc thay đổi dữ liệu của volume ở ngoài cũng sẽ ảnh hưởng đến service bên trong container

3 Lab

3.1 Nginx lab

Thực hiện tạo config file default của nginx:

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;

    root /usr/share/nginx/html;
```

```
    index index.html index.htm;

    server_name _;
    location / {
        try_files $uri $uri/ =404;
    }
}
```

Tạo một Dockerfile như sau:

```
FROM ubuntu
RUN apt-get -y update && apt-get -y install nginx
COPY default /etc/nginx/sites-available/default
EXPOSE 80/tcp
CMD ["/usr/sbin/nginx", "-g", "daemon off;"]
```

Tiến hành build image:

```
docker build -t haidar/server:latest .
```

Tiến hành tạo 2 docker volume để ghi log và file config của nginx:

```
docker volume create nginx_log
docker volume create nginx_conf
```

Tiến hành run image vừa được tạo và mount log file của nginx và file config của nginx ra 2 docker volume vừa được tạo:

```
docker run -d -p 80:80 --name nginx --mount source=nginx_log,target=/var/log/nginx
--mount source=nginx_conf,target=/etc/nginx haidar/server:latest
```

Sau khi chạy xong container test kết nối tới webserver bằng lệnh curl ta được kết quả:

```
buidung@buidung:~/Nginx-web/src$ curl http://0.0.0.0:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Hình 27: Curl

Kiểm tra xem log đã được mount ra ngoài chưa:

```
root@buidung:/home/buidung/Nginx-web/src# ls -l /var/lib/docker/volumes/nginx_log/_data/
total 4
-rw-r----- 1 www-data adm 174 Mar 13 11:32 access.log
-rw-r----- 1 www-data adm   0 Mar 13 11:06 error.log
root@buidung:/home/buidung/Nginx-web/src# ls -l /var/lib/docker/volumes/nginx_conf/_data/
total 64
drwxr-xr-x 2 root root 4096 May 30  2023 conf.d
-rw-r--r-- 1 root root 1125 May 30  2023 fastcgi.conf
-rw-r--r-- 1 root root 1055 May 30  2023 fastcgi_params
-rw-r--r-- 1 root root 2837 May 30  2023 koi-utf
-rw-r--r-- 1 root root 2223 May 30  2023 koi-win
-rw-r--r-- 1 root root 3957 May 30  2023 mime.types
drwxr-xr-x 2 root root 4096 May 30  2023 modules-available
drwxr-xr-x 2 root root 4096 Mar 13 11:21 modules-enabled
-rw-r--r-- 1 root root 1447 May 30  2023 nginx.conf
-rw-r--r-- 1 root root  180 May 30  2023 proxy_params
-rw-r--r-- 1 root root  636 May 30  2023 scgi_params
drwxr-xr-x 2 root root 4096 Mar 13 11:21 sites-available
drwxr-xr-x 2 root root 4096 Mar 13 11:21 sites-enabled
drwxr-xr-x 2 root root 4096 Mar 13 11:21 snippets
-rw-r--r-- 1 root root  664 May 30  2023 uwsgi_params
-rw-r--r-- 1 root root 3071 May 30  2023 win-utf
```

Hình 28: Check log

3.2 HAproxy keepalive lab

Config environment:

```
Virtual Server:
    IP: 192.168.74.150/24
    Port H2: 8080
    Port HTTP: 80
Host 1:
    IP: 192.168.74.141/24
    Interface: ens33
    Instances:
        - keepalived-a
        - haproxy-a
        - haproxy-b
        - web-a
        - web-b
Host 2:
    IP: 192.168.74.142/24
    Interface: ens33
    Instances:
        - keepalived-b
        - haproxy-a
        - haproxy-b
        - web-a
        - web-b
```

File config haproxy-a:

```
global_defs {
# UNIQUE #
  router_id LVS_MAIN
# UNIQUE #
}

vrrp_instance VI_1 {
# UNIQUE #
  state MASTER
  priority 150
# UNIQUE #

  advert_int 1
  virtual_router_id 51

# CHANGE TO YOUR NEEDS #
  # real network interface
  interface ens33

  # my ip (on real network)
  unicast_src_ip 192.168.74.141/24

  # peer ip (on real network)
  unicast_peer {
    192.168.74.142/24
  }
# CHANGE TO YOUR NEEDS #

  virtual_ipaddress {
    192.168.74.150/24
  }

  authentication {
    auth_type PASS
    auth_pass d0ck3r
  }
virtual_server 192.168.74.150 80 {
    delay_loop 5
    lb_algo wlc
    lb_kind NAT
    persistence_timeout 600
    protocol TCP

    real_server 172.20.0.50 80 {
        weight 100
```

```
        TCP_CHECK {
            connect_timeout 10
        }
    }
    real_server 172.20.0.60 80 {
        weight 100
        TCP_CHECK {
            connect_timeout 10
        }
    }
}

virtual_server 192.168.74.150 8080 {
    delay_loop 5
    lb_algo wlc
    lb_kind NAT
    persistence_timeout 600
    protocol TCP

    real_server 172.20.0.50 8080 {
        weight 100
        TCP_CHECK {
            connect_timeout 10
        }
    }
    real_server 172.20.0.60 8080 {
        weight 100
        TCP_CHECK {
            connect_timeout 10
        }
    }
}
```

Thực hiện tương tự với file config proxy-b.
File docker-compose:

```
version: "3.7"

x-haproxy-defaults: &haproxy-service
  image: "haproxy:2.0-alpine"
  volumes:
      - "./haproxy/haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg:ro"

x-keepalived-defaults: &keepalived-service
  image: "osixia/keepalived:2.0.19"
  cap_add:
```

```
    - NET_ADMIN
    - NET_BROADCAST
    - NET_RAW
  environment:
    KEEPALIVED_COMMAND_LINE_ARGUMENTS: >-
      --log-detail
#      --dont-release-vrrp
#      --dont-release-ipvs
#      --log-detail
#      --dump-conf

networks:
  ha-stack:
    ipam:
      config:
        - subnet: 172.20.0.0/24

services:

  keepalived-a:
    <<: *keepalived-service
    network_mode: host
    volumes:
      - "./keepalived/proxy-a/keepalived.conf:/usr/local/etc/keepalived/keepalived.conf:ro"
      - "./keepalived/notify.sh:/container/service/keepalived/assets/notify.custom.sh:ro"

  keepalived-b:
    <<: *keepalived-service
    network_mode: host
    volumes:
      - "./keepalived/proxy-b/keepalived.conf:/usr/local/etc/keepalived/keepalived.conf:ro"
      - "./keepalived/notify.sh:/container/service/keepalived/assets/notify.custom.sh:ro"

  haproxy-a:
    <<: *haproxy-service
    networks:
      ha-stack:
        ipv4_address: 172.20.0.50
        aliases:
          - haproxy-a.ha.stack

  haproxy-b:
    <<: *haproxy-service
    networks:
      ha-stack:
```

```
        ipv4_address: 172.20.0.60
        aliases:
          - haproxy-b.ha.stack

  web-b:
    image: "nginx:stable-alpine"
    networks:
      ha-stack:
        aliases:
          - web-b.ha.stack
    volumes:
      - "./web/server-b/index.html:/usr/share/nginx/html/index.html:ro"

  web-a:
    image: "nginx:stable-alpine"
    networks:
      ha-stack:
        aliases:
          - web-a.ha.stack
    volumes:
      - "./web/server-a/index.html:/usr/share/nginx/html/index.html:ro"
```

Thực hiện file config haproxy:

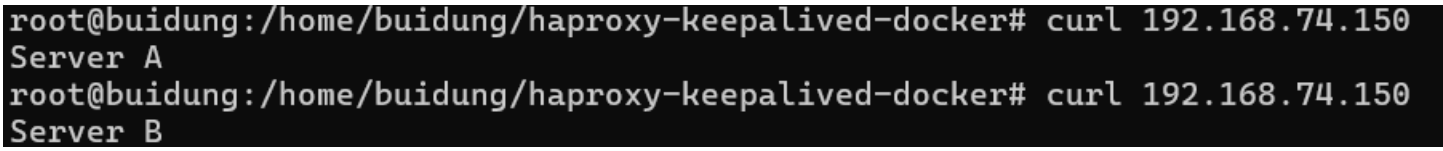
```
global
  maxconn 10000
  log stdout format raw local0

defaults
  log     global
  mode    http
  option  httplog
  option  dontlognull
  option  http-use-htx
  timeout connect 1
  timeout client  5
  timeout server  5

frontend web-in
  bind :80
  bind :8080 proto h2
  default_backend web-upstream

backend web-upstream
  retries 2
  server web-a web-a.ha.stack:80 check
```

```
  server web-b web-b.ha.stack:80 check
```

Thực hiện chạy docker-compose trên 2 host:

```
Host1:
docker-compose up -d keepalived-a haproxy-a haproxy-b web-a web-b
Host2:
docker-compose up -d keepalived-b haproxy-a haproxy-b web-a web-b
```

Thực hiện kiểm tra kết quả:



Hình 29: Result