Công ty An ninh mạng Viettel

Báo cáo về docker container

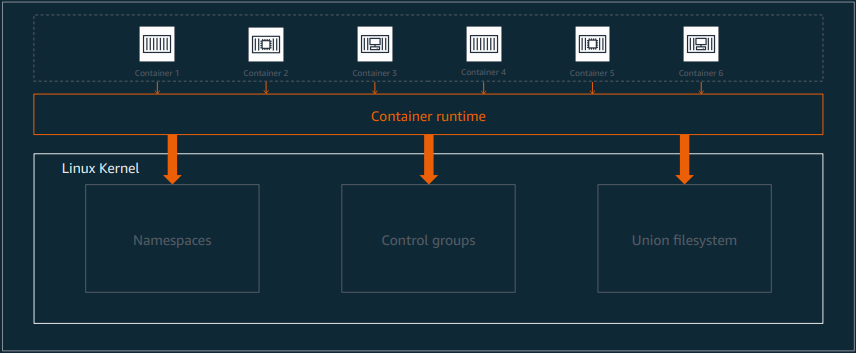
Sinh viên: Nguyễn Đan Trường

**MỤC LỤC**

[1. Linux Container Primitives 3](#_Toc157078437)

[2. Docker 13](#_Toc157078438)

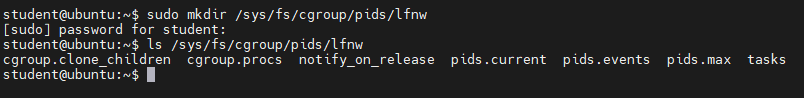
## Linux Container Primitives



Linux Container Primitives là các tính năng của nhân Linux được sử dụng để tạo và chạy các container. Các tính năng này bao gồm:

* Namespaces: Namespaces giúp cô lập các container với nhau và với phần còn lại của hệ thống. Có 7 loại namespaces:
  + Cgroup namespace: Cô lập các resource của container, chẳng hạn như CPU, memory, và I/O.
  + IPC namespace: Cô lập các IPC object, chẳng hạn như semaphores và shared memory.
  + Mount namespace: Cô lập các mount point của container.
  + Network namespace: Cô lập mạng của container.
  + PID namespace: Cô lập PID của container.
  + User namespace: Cô lập người dùng và nhóm của container.
* Control groups (cgroups): Control groups giúp hạn chế và giám sát các resource của container. Cgroups có thể được sử dụng để giới hạn CPU, memory, I/O, và các resource khác của container.
* Union filesystems: Union filesystems giúp tạo ra một filesystem riêng cho container. Union filesystems có thể được sử dụng để chia sẻ các file giữa container và hệ thống host, hoặc để tạo ra các filesystem độc lập cho container.

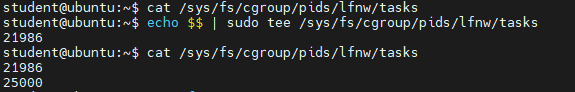
**Cgroup**

****

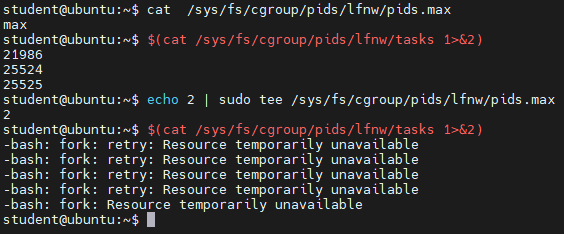
Thư mục */sys/fs/cgroup/pids/lfnw* và các tệp trong đó liên quan đến cgroups với loại tài nguyên là "pids" (số lượng tiến trình). Dưới đây là giải thích về vai trò của từng tệp:

* cgroup.clone\_children:
  + Được sử dụng để xác định xem khi một tiến trình được thêm vào cgroup này, liệu các tiến trình con của nó có nên được thêm vào cùng không.
* Cgroup.procs:
  + Chứa danh sách các Process ID (PID) của các tiến trình thuộc cgroup. Khi bạn thêm một PID vào cgroup, nó sẽ được liệt kê trong tệp này.
* Notify\_on\_realease:
  + Liên quan đến sự kiện thông báo khi cgroup này được giải phóng (released). Nếu bạn đặt giá trị là 1, có thể kích hoạt các thông báo khi cgroup được giải phóng.
* Pids.current:
  + Hiển thị số lượng PID hiện tại đang thuộc cgroup này.
* Pids.events
  + Liên quan đến các sự kiện liên quan đến số lượng PID, chẳng hạn như khi đạt đến giới hạn pids.max.
* Pids.max
  + Chứa giới hạn tối đa cho số lượng PID mà cgroup này có thể chứa.
* Tasks:
  + Tương tự như cgroup.procs, tasks là một phiên bản khác của danh sách PID thuộc cgroup, chỉ làm một biệt danh khác.

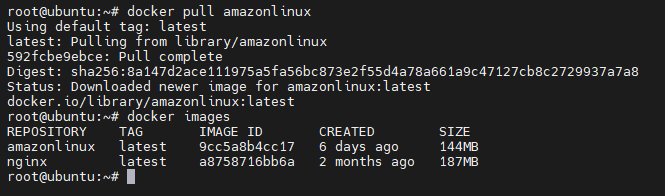
Ta có thể tạo ra một cgroup mới và thêm tiến trình vào trong cgroup đó

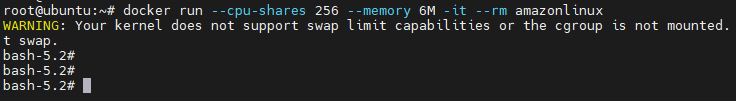


Giới hạn số lượng tiến trình trong một cgroup:

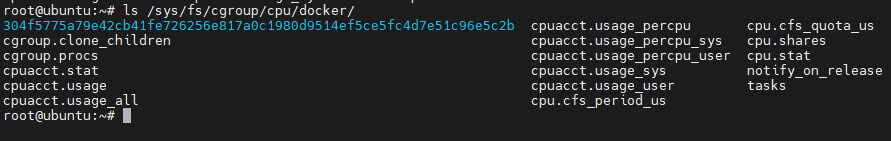


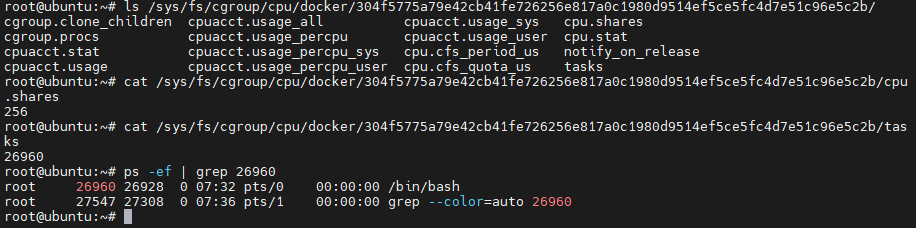
Thử run một container có set thông số cpu-share và limit memory

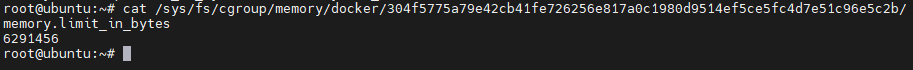




Ta thấy được với từng cgroup sẽ tạo ra 1 file liên quan tới container được tạo:

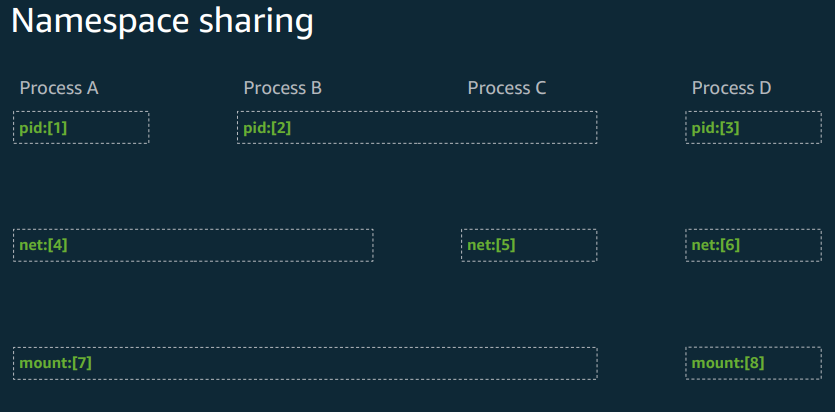






**Namespace**

* Trong Linux, namespace được sử dụng như một cơ chế cô lập tài nguyên, tạo ra một môi trường cô lập trong đó các tài nguyên có thể tồn tại mà không ảnh hưởng đến tài nguyên trong môi trường khác.
* Nếu có một tiến trình thay đổi tài nguyên trong một namespace cụ thể thì thông thường những thay đổi này sẽ không ảnh hưởng đến các tài nguyên tương tự bên ngoài namspace đó.



Hình ảnh cho thấy sự chia sẻ namespace giữa các process.

* Process A là process gốc, không chia sẻ namespace nào với các process khác.
* Process B chia sẻ namespace PID với Process A. Điều này có nghĩa là Process B có thể truy cập các PID được tạo bởi Process A.
* Process C chia sẻ namespace mạng với Process B. Điều này có nghĩa là Process C có thể truy cập các network interface và route được sử dụng bởi Process B.
* Process D chia sẻ namespace mount với Process A. Điều này có nghĩa là Process D có thể truy cập các filesystem được mount bởi Process A.

Trong thực tế, các container thường chia sẻ một số namespace với hệ thống host. Điều này giúp các container có thể truy cập các resource của hệ thống host, chẳng hạn như mạng và filesystem.

**Network namespace**

Là một tính năng của Linux Kernel cho phép tạo ra các môi trường mạng riêng biệt.

Khi bạn chạy một container bằng lệnh docker run, docker sẽ tự động tạo ra một network namspace riêng cho container đó.

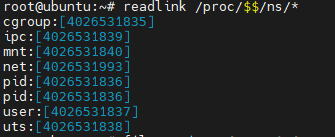
Mặc dù Docker mặc định sử dụng một network namespace riêng cho mỗi container, nhưng bạn có thể cấu hình để nhiều container chia sẻ chung một network namespace.

**Mount namespace**

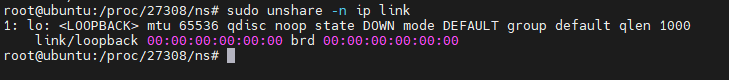
Mount namespace là một tính năng của kernel Linux cho phép tạo ra các môi trường hệ thống tệp riêng biệt. Giống như các namespace khác, mount namespace giúp cô lập môi trường của container, đảm bảo an toàn và quản lý tài nguyên hiệu quả hơn.

Khi bạn khởi chạy một container bằng lệnh *docker run*, Docker sẽ tự động gắn image container dưới dạng root filesystem của container đó. Điều này đảm bảo container chỉ truy cập được vào các file và thư mục được cung cấp trong ảnh đó.

Các namespace được hiển thị trong thi mục /proc, các thông tin này được biểu diễn dưới dạng các tệp tin symblic links

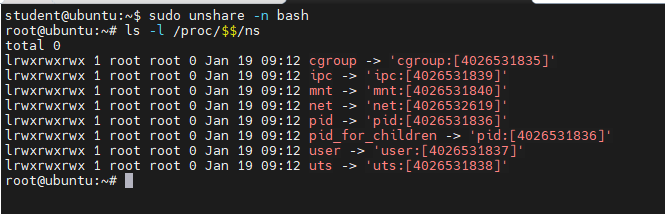


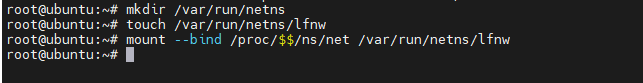
Tạo một network namspace mới:

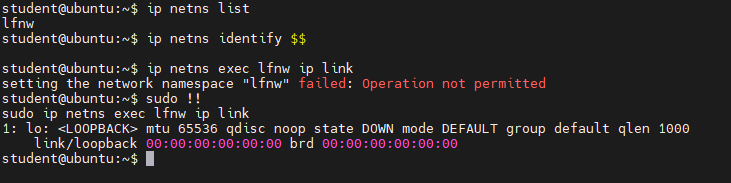


Lệnh trên sẽ tạo một network namespace mới và sau đó mở một shell bên trong namespace đó. Sau khi bạn nhập lệnh trên, bạn sẽ thấy prompt của shell đã thay đổi, thể hiện rằng bạn đang ở trong một network namespace độc lập.

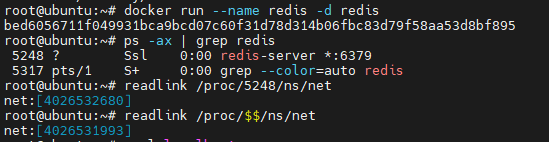
Tạo một namspace mới:

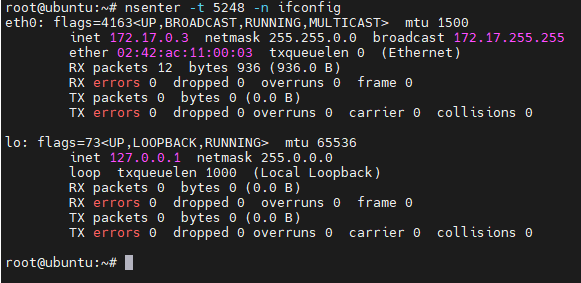


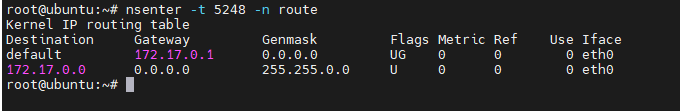


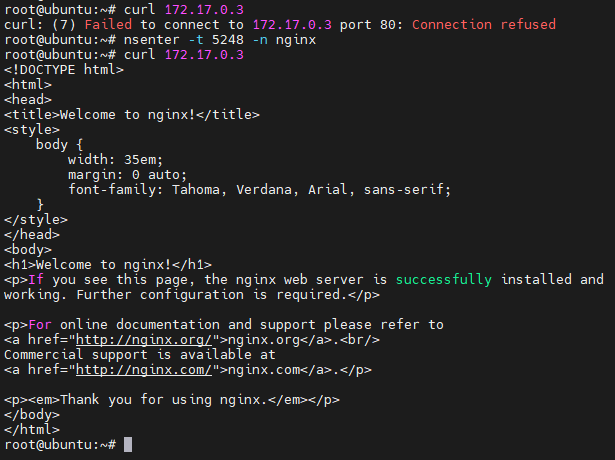


Run một container và kiểm tra network namspace









**Filesystem images**

Images là một định nghĩa về hệ thống tệp, thư viện, và các ứng dụng cần thiết để chạy một ứng dụng cụ thể. Images được sử dụng để tạo ra các container.

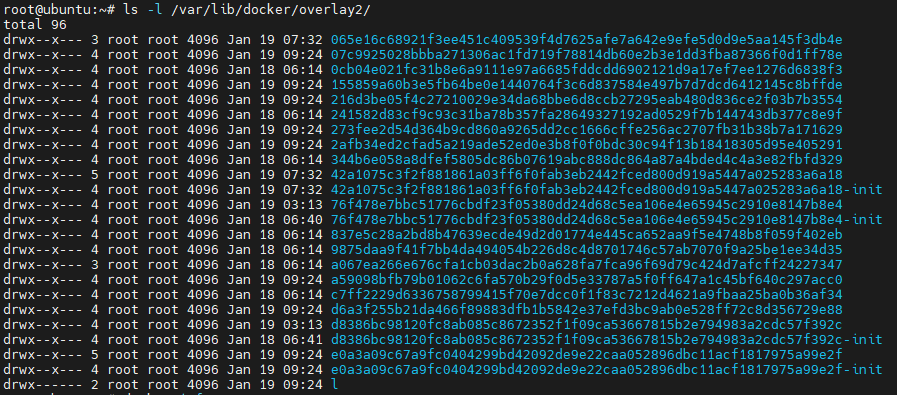
Layers là các thành phần nhỏ của một image. Mỗi layer đại diện cho một thay đổi đối với hệ thống tệp của image.

Union filesystems là một loại hệ thống tệp cho phép kết hợp nhiều filesystems thành một. Union filesystems được sử dụng để tạo ra các container từ các images.

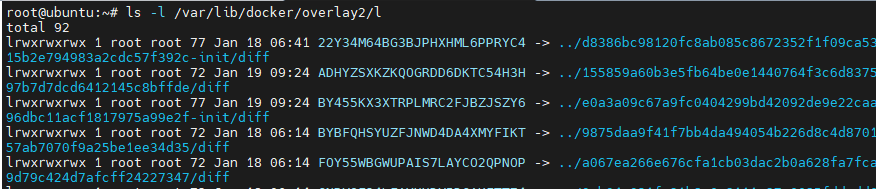
Files thuộc nhiều hệ thống file khác nhau có thể cùng nằm trong một folder, tuy nhiên nếu 2 file có chung path, thì file được mounted sau cùng sẽ giấu đi những file được mount trước đó.

**Overlay2 driver và layer**

Đối với **overlay2 driver** các layer sẽ được lưu trữ lại **/var/lib/docker/overlay2/** có dạng:

****

Thư mục **l** chứa Symbolic Links là tên rút gọn ta thấy trong **docker history** link đến thư mục diff của các layer.

****

Thư mục **/link** chứa tên rút gọn của layer:

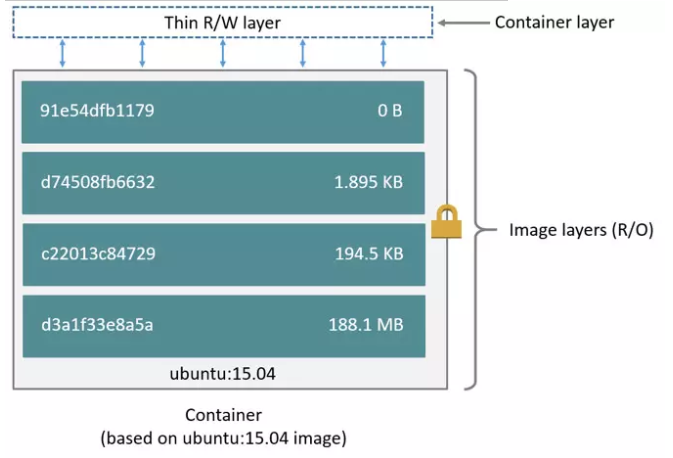


Thư mục **/diff** chứa system file của layer:

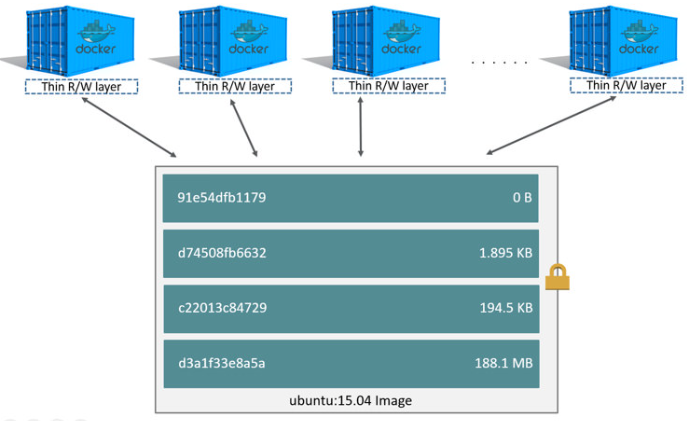


**Container, Image và layer**

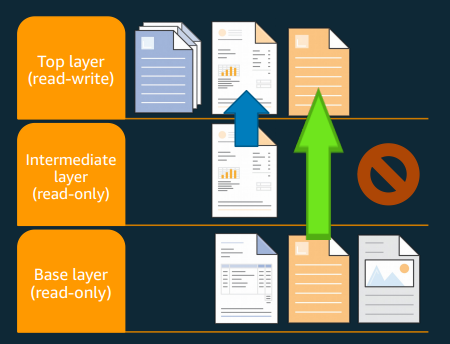
Một Image được tạo thành từ một stack các read only layer

****

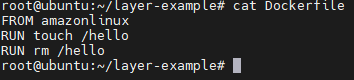
Khi tạo một container từ image, một read/write layer sẽ được thêm vào trên cũng của stack. Khi có nhiều container được khởi tạo từ image, chúng sẽ sử dụng chung các layer phía dưới của image. Việc này giúp tiết kiệm tài nguyên cho hệ thống, khi container bị remove chỉ top layer bị loại bỏ, các layer của image sẽ không thay đổi.

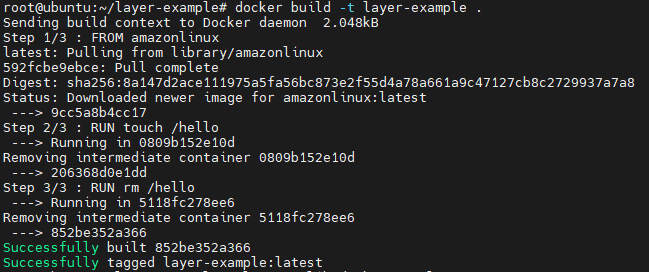
****

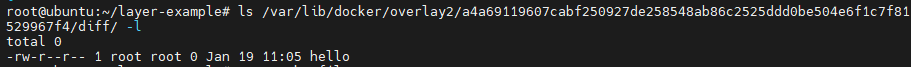
**How docker layer work**

****

* "A copy-on-write view of your files": Thay vì trực tiếp chỉnh sửa các file trong image, mỗi layer mới chứa một bản sao của những file được thay đổi từ layer trước. Điều này đảm bảo các layer bên dưới luôn nguyên vẹn.
* "New files exist only in the top layer": Mọi file mới được thêm vào chỉ nằm trong layer trên cùng. Các layer dưới không chứa file mới đó.
* "When a file is modified, it is copied up to the top layer": Khi một file được chỉnh sửa, thay vì cập nhật file gốc trong layer cũ, một bản sao được tạo ra và đặt trong layer trên cùng. File gốc ở layer cũ vẫn giữ nguyên.
* "Unmodified files exist in whatever layer they were added/modified": Các file không bị thay đổi sẽ vẫn nằm trong layer mà chúng được thêm vào hoặc sửa đổi lần cuối.
* "Deleted files are hidden, but still exist": Khi xóa một file, Docker không thực sự xóa file đó ngay lập tức. Thay vào đó, file được đánh dấu là "bị xóa" và ẩn đi. File này vẫn tồn tại trong layer cho đến khi tất cả các layer phụ thuộc đến nó đều bị xóa.







## Docker

Docker là nền tảng ảo hóa phục vụ cho quá trình triển khai và quản lý ứng dụng trên container.

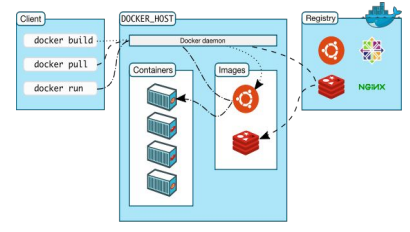
**Docker image** là một gói chứa tất cả các thứ cần thiết để một ứng dụng hoạt động, bao gồm mã nguồn, thư viện, biến môi trường, tập tin cấu hình …

**Docker container** là một thực thể chạy từ một image. Nó sẽ sử dụng chung tài nguyên của hệ thống (RAM, disk, Network…) chính nhờ vậy những container sẽ nhẹ và việc khởi động sẽ nhanh gọn hơn.

**Docker Registry** là nơi lưu trữ các image (có thể là docker hub)

**Docker Client** là công cụ dòng lệnh hoặc giao diện người dùng để tương tác với docker daemon

Cơ bản, quá trình làm việc của Docker như sau:



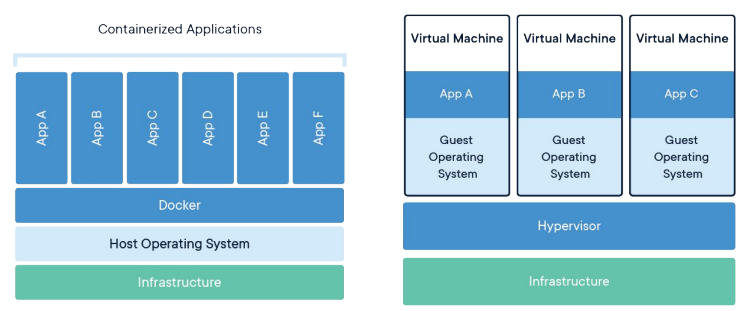
**Xây dựng Image**: Tạo Docker Image chứa ứng dụng và môi trường cần thiết.

**Lưu trữ Image**: Image được lưu trữ trên một Registry như Docker Hub.

**Chạy Container**: Sử dụng Docker Client để tạo và chạy Container từ Image đã xây dựng.

**Quản lý Containers**: Sử dụng Docker Client để quản lý các Container, bao gồm khởi động, dừng, xóa, và theo dõi trạng thái.

**Docker container vs Virtual Machine**

****

|  |  |  |
| --- | --- | --- |
|  | **VM** | **Container** |
| Khả năng tích hợp | Mỗi VM bao gồm một hệ điều hành đầy đủ và một bản sao của tài nguyên vật lý (RAM, CPU, Disk). | Containers chia sẻ hệ điều hành với host và chỉ cần các thư viện và tài nguyên cần thiết. |
| Hiệu suất | Tích hợp một hệ điều hành đầy đủ, do đó tốn nhiều tài nguyên hơn và có thể có overhead hiệu suất. | Do chia sẻ hạ tầng với host và các containers khác, containers thường có hiệu suất cao hơn và thời gian khởi động nhanh. |
|  | Cung cấp độ cách ly hoàn toàn giữa các máy ảo. Mỗi VM chạy một hệ điều hành riêng biệt. | Các container chia sẻ hạ tầng và kernel với host, có nghĩa là độ cách ly thấp hơn so với VM |
| Khả năng di động | Yêu cầu cài đặt lại chương trình và các phụ thuộc khi chuyển đổi giữa các máy tính. | Di động giữa các môi trường máy chủ một cách dễ dàng vì chúng đóng gói tất cả các phụ thuộc cần thiết. |

**Lab 1: docker hub**

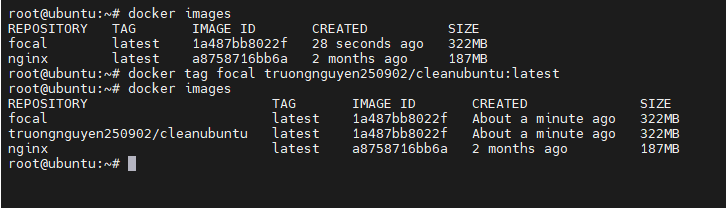
Để push docker images lên doker hub, thì tên docker image phải có định dạng:  
**<dockerhub-user>/<repo-name>:<tag>**

Kiểm tra lại tên docker của mình vừa tạo:

**Docker images**

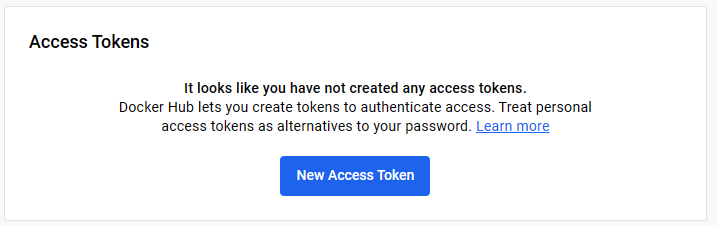
Để đổi tên docker images:

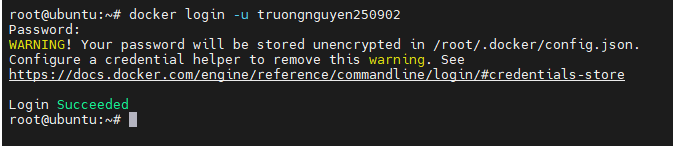
**docker tag focal truongnguyen250902/cleanubuntu:latest**

****

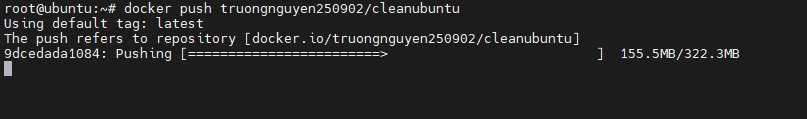
**docker images**

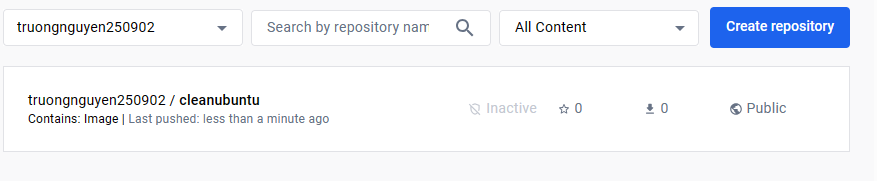
Login vào docker hub (sử dụng token tạo trên docker hub)

****

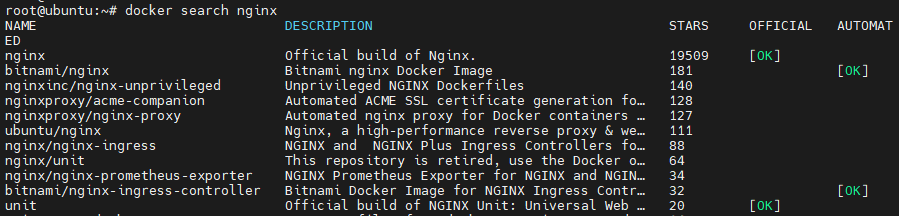
****

Push image lên docker hub

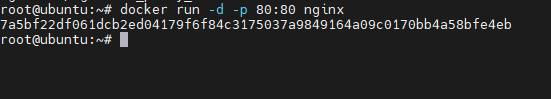
****

****

Tìm kiếm các docker image trên docker hub bằng lệnh: **docker search nginx** sẽ thấy có nhiểu bản image của nginx có sẵn, có bản official



Tạo thử một container nginx:

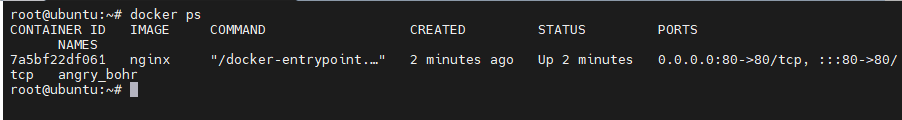


Trước tiên docker sẽ tìm image ở local nếu không thấy nó sẽ download image từ docker hub. Sau đó tiến hành tạo container từ image lấy về.

-d là detach, tức là chạy ở background

-p là port forward cổng 80 ra port 80 của máy host.

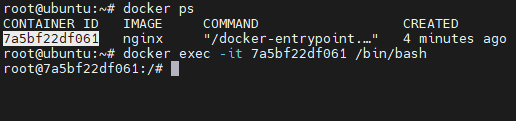
Gõ lệnh: **docker ps –a** để xem container id và names tự sinh ra của container nginx



Đổi tên container: **docker rename 7a5bf22df061 cleanNginx**

Trong đó 7a5bf22df061 là ContainerID và cleanNginx là tên mới.

Truy cập vào bash shell của Container:



Thoát khỏi bash shell bằng lệnh exit

Attach container đang chạy output thông tin ra terminal:

**docker attach nginx**

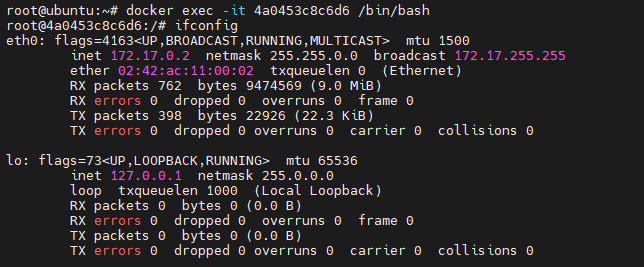
Giữ nguyên và vào trình duyệt web truy cập trang web nginx (IP của Host) sẽ thấy log truy cập của container



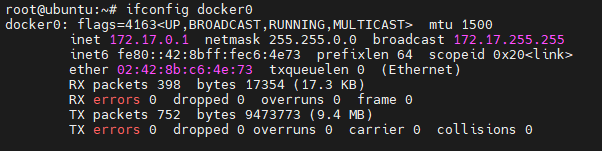
Muốn xóa các Container đã exit thì gõ lệnh: **docker container prune** chọn **y**

**Lab 2: Docker Network**

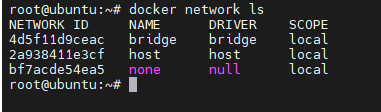
Kiểm tra IP của container



Ta thaays ip card docker0 cuar host và ip của container có cùng một subnet



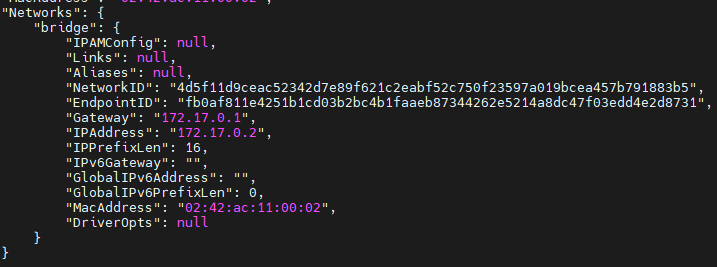
Trong docker có 3 kiểu driver network mà container có thể được cài đặt:



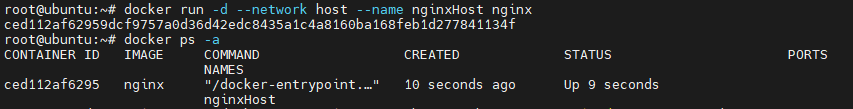
Kiểu bridge là kiểu mặc địch khi tạo container, khác với card mạng Bridge trong vmware kiểu bridge trong container lại giống với NAT trong vmware

Kiểu none thì container không đấu nối vào đâu

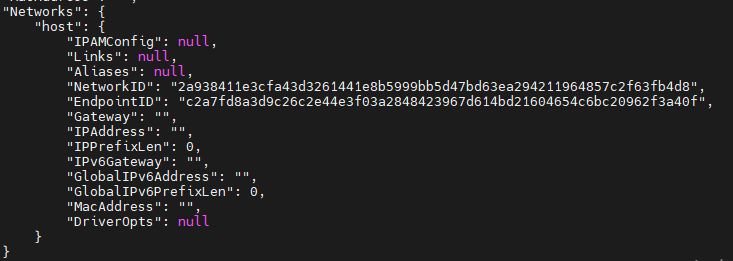
**Docker inspect nginx**

****

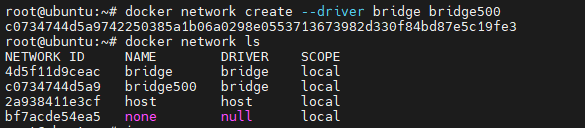
Chạy thêm một container nginx với kết nối mạng kiểu host:



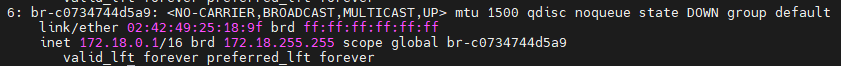
Kiểm tra kết nối mạng của nginxHost sẽ thấy kiểu network là host và không có IP, do dùng chung IP với máy host.

****

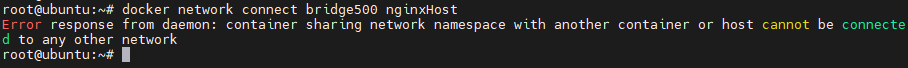
Tạo thử một docker network mới:



Thấy có thêm card mạng trên máy host:

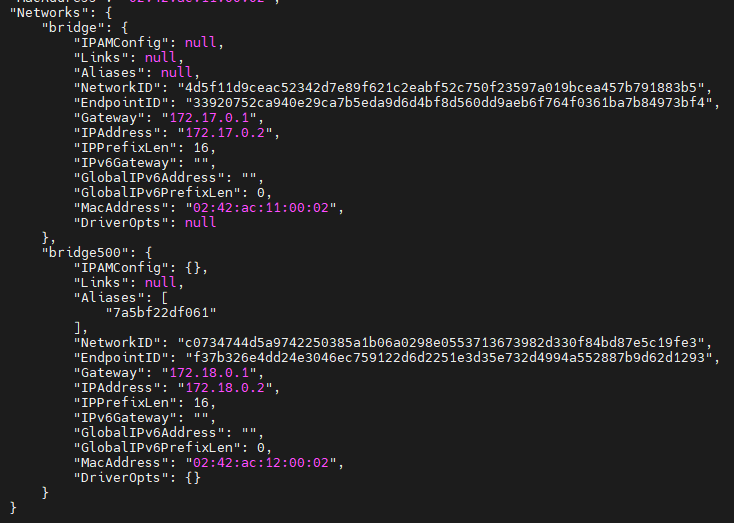


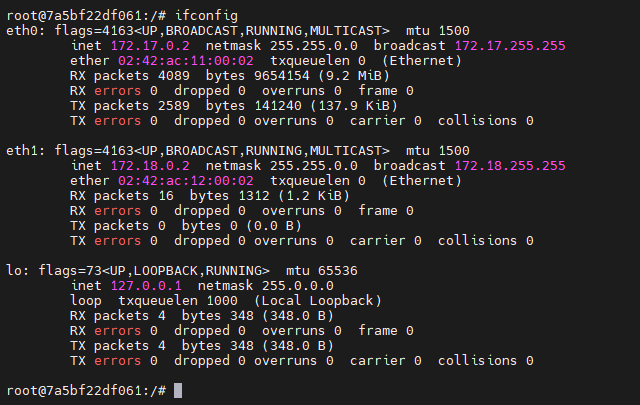
Thử thêm một network cho container nginxHost nhưng báo lỗi vì vì kiểu host không thể chuyển đổi sang kiểu khác được.





Ta thấy được container đã có thêm một card mạng.

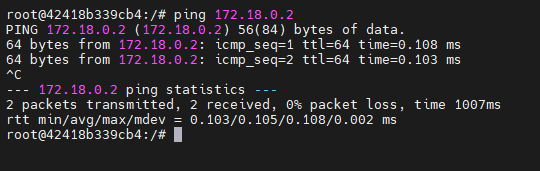


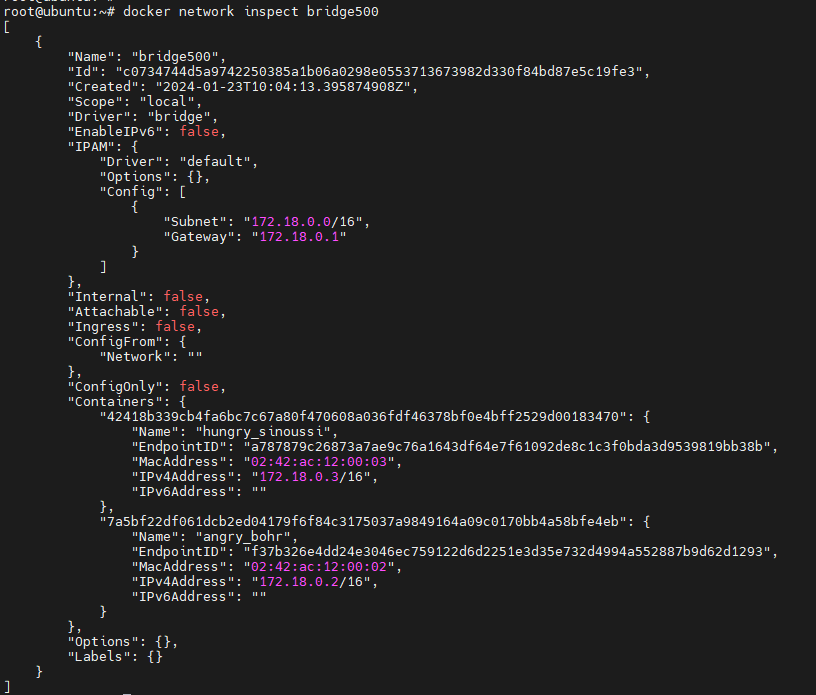


Tạo thêm một container ngĩn mới map port 86 của host:



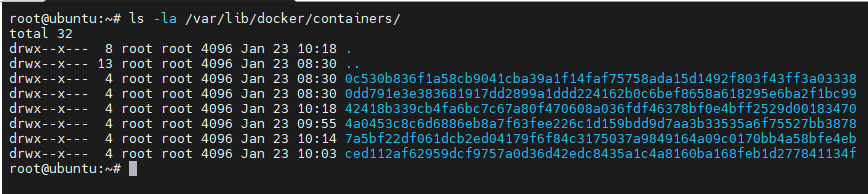
Các container được connect vào cùng một card mạng sẽ có thể giao tiếp với nhau



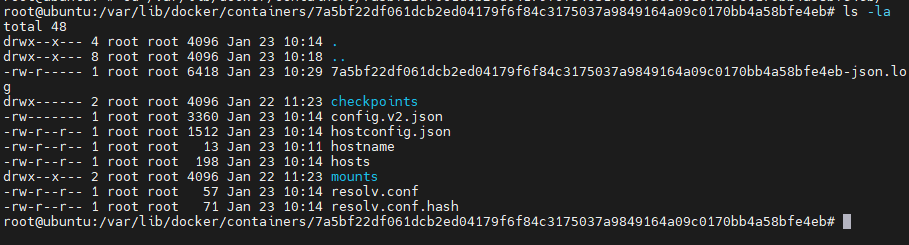


**Lab 3: Docker log**

Kiểm tra logs của container chạy nginx xem các truy cập, trên máy host:



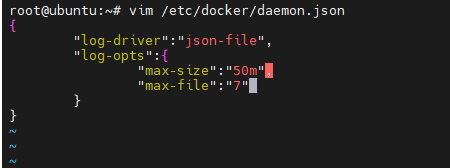
Truy cập thử một thư mục và xem bên trong



Xem log ai đã truy cập nginx. File log có Idcontainer và đuôi –json.log

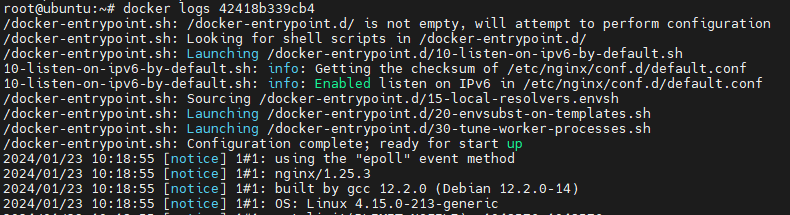
Như vậy log của container được lưu trên máy host và không mất khi tắt container

Giới hạn kích thước file log bằng cách cấu hình trong file: /etc/docker/daemon.json:

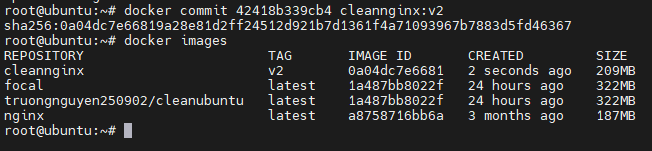


Sau đó restart lại docker, các container mới sẽ chị cấu hình mặc định này

Để xem logs của một container có thể dùng lệnh:



Sau khi cài đặt thêm vào container đang chạy, có thể tạo ra image mới bằng lệnh commit (snapshot lại trạng thai hiện tại của container để tạo):



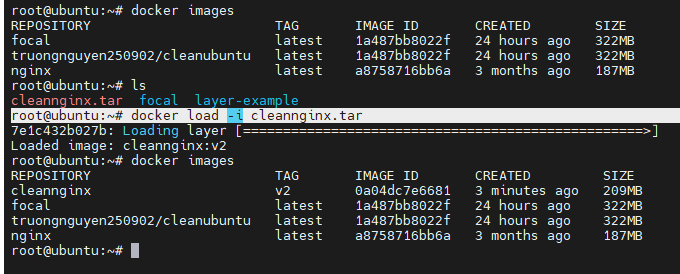
Docker lưu repository chứa các image và các layer ở đường dẫn:

/var/lob/docker

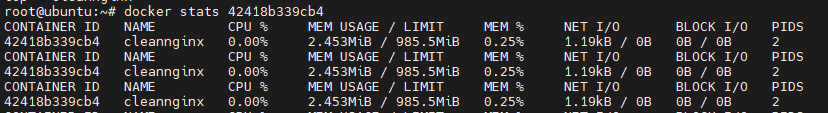
Nếu muốn export một image tử repository trở thành file tar để copy và import vào máy khác thì dùng lệnh



Sau đó copy sang máy khác, để sử dụng lại import file tar vào repository:



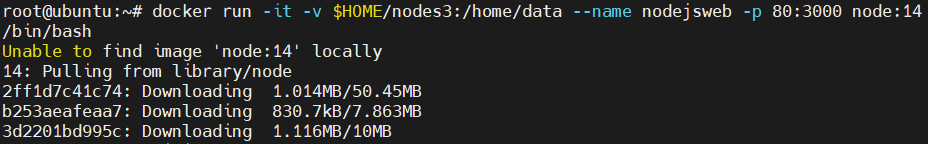
Xem container sử dụng tài nguyên trên máy host bằng lệnh:



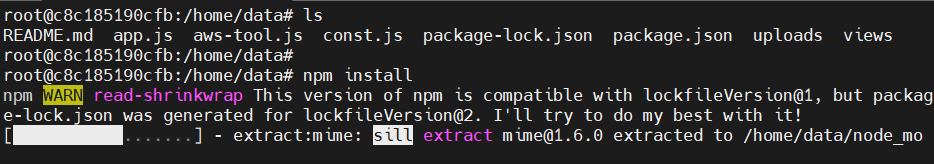
**Lab 5: Docker Volume – Backup Data:**

Yêu cầu: Map thư mục chứa code trên máy Host vào container để chạy và test ứng dụng.

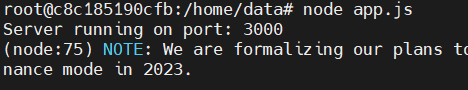
Khởi chạy container nodejs và map volume từ thư mục $HOME/nodes3 hiện tại trên máy host vào thư mục /home/data trên container



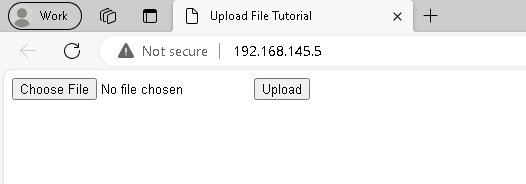
Sau khi vào container rồi thì chạy lệnh cài đặt các gói thư viện:



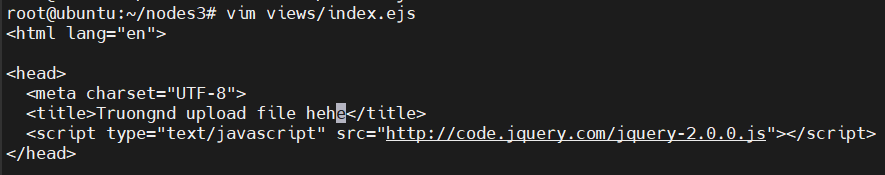
Và chạy app bằng lệnh:



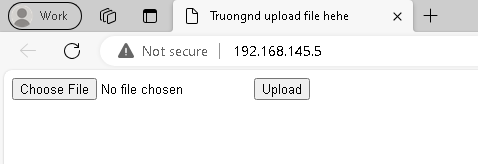
Test truy cập từ Internet với IP máy host thấy output ra màn hình log có người dùng truy cập đến.



Thoát ra ngoài máy host và thử edit file views để xem nội dung trang web trong container:



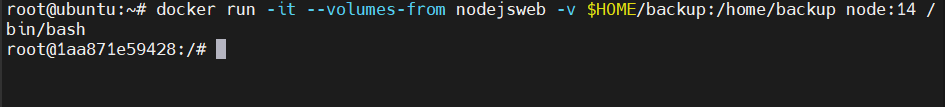
Truy cập lại web thấy nội dung trang web đã được thay đổi:



Việc map trực tiếp thư mục máy host vào container để sửa code thường được sử dụng trong quá trình viết app. Khi viết hoàn chỉnh muốn đưa thực tes sử dụng thì sẽ copy từ máy host vào trong container luôn, chỉ có phần lưu DB thì thường sẽ được map ra ngoài.

Để backup dữ liệu trong volume của container, thông thường sẽ khởi động một container khác cùng sử dụng volume đó với container thứ nhất, nhưng chỉ chạy chức năng backup dữ liệu volume đó sang một nơi lưu trữ khác.

Volume đã map vào một container có thể chia sẻ cho container khác. VD tạo container thứ 2 cũng sử dụng cùng volume như container đã tạo ở trên:

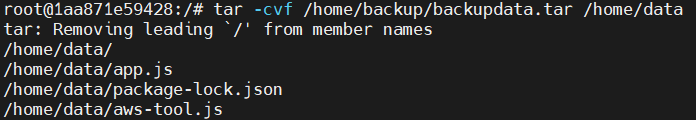


Container thứ 2 này không mở port mà mount thêm thư mục backup trên máy host vào container thứ 2 ngoài các volume mount chia sẻ với container thứ 1

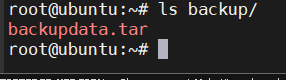
Trên container thứ 2 kiểm tra các mount point: **findmnt**



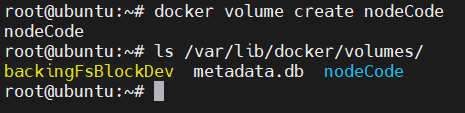
Tiến hành backup dữ liệu:

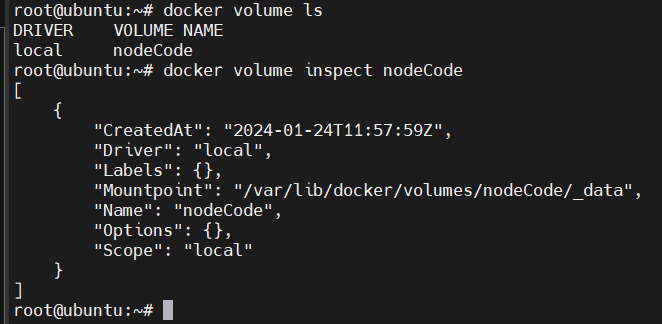


Thoát khỏi container 2 và kiểm tra file backup trên máy host:



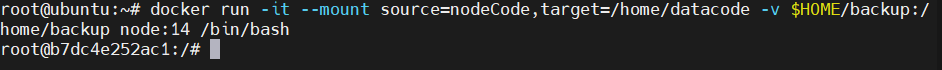
Nếu không mount thư mục trên máy host vào container, có thể tạo docker volume rồi mới mount vào container:



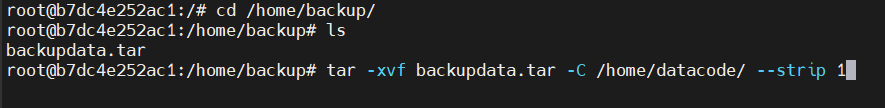


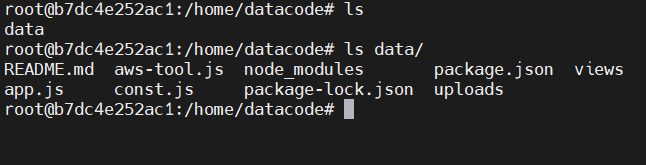
Thực hiện restore dữ liệu backup vào container mới.

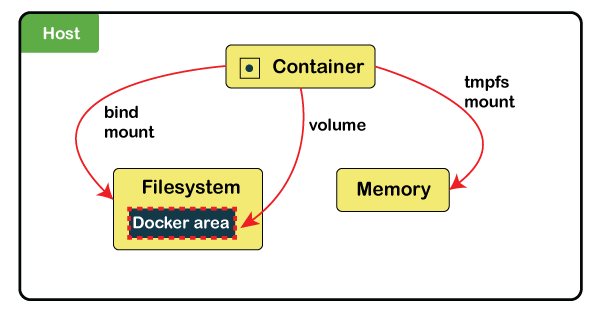
Mount volume đã tạo vào container mới, đồng thời mount cả thư mục /backup trên máy host:



Sau đó giải nén để restore file tar sang volume mới:

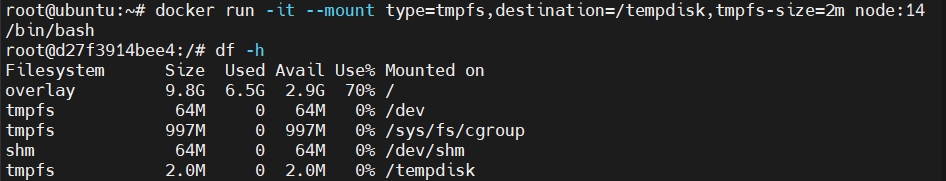






* Bind mount: sẽ mount một folder trên máy host vào container, bind mount đôi khi hoạt động không chính xác nếu máy host không có thư mục tương tự
* Volume được quản lý bởi docker và máy host sẽ không thể truy cập docker volume như một folder thông thường. Docker volume được sử dụng khi tạo dockerfile
* Tmpfs mount: Dữ liệu tồn tại trong bộ nhớ và không lưu trữ trên host, thích hợp cho các trường hợp cần tốc độ cao và dữ liệu tạm thời.

Sử dụng 2MB RAM làm ổ đĩa tmpfs cho container:



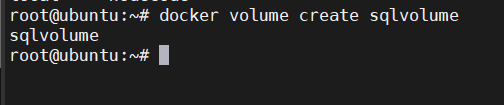
ổ đĩa RAM tmpfs được sử dụng để lưu các file thông số đầu vào, ví dụ thay đổi 1s/1 lần cho các ứng dụng realtime.

Nếu có một container đang chạy tốt mà muốn thêm volume với vào thì phải commit container đó thành docker image mới và tạo container từ image đó nhưng mount thêm volume.

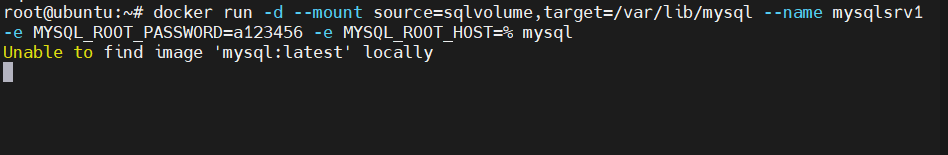
**Lab 6; Docker Mysql**

Tạo một cụm docker container gồm 2 containers: Nodejs App và Mysql

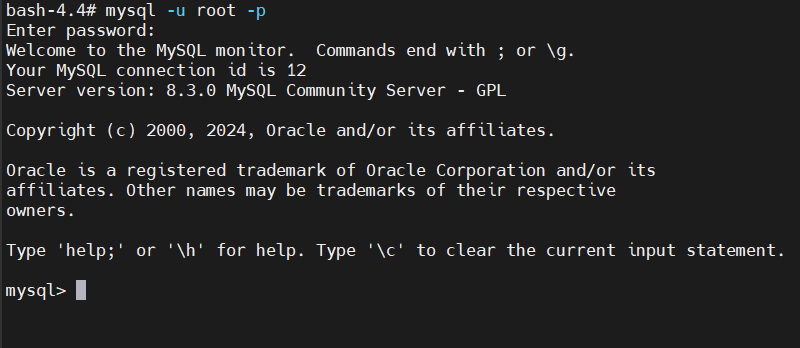
Tạo một docker volume tên là **sqlvolume**

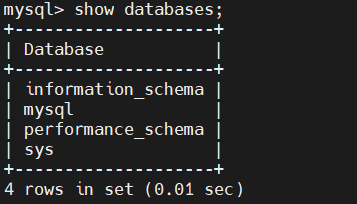
****

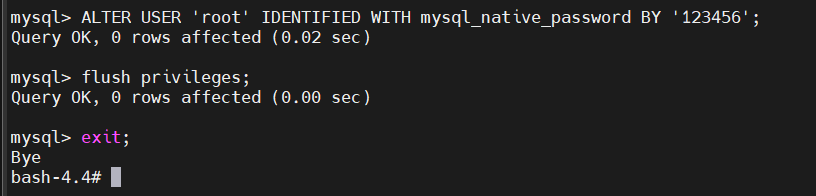
Khởi chạy một container có mount volume sqlvolume vào container mysql với root password=123456



Truy cập vào container và chạy lệnh mysql sau đó chạy các lệnh sau:



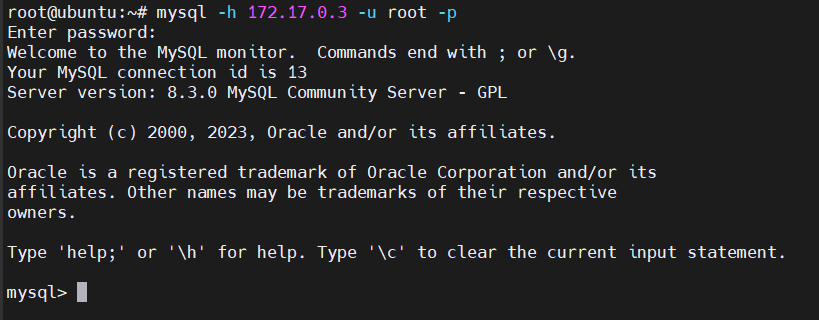




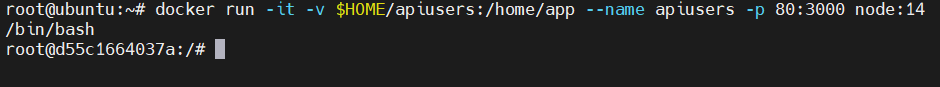
Kiểm tra IP container mysql: 172.17.0.3

Cài đặt lệnh mysql sau đó truy cập thử vào mysql trên container bằng lệnh:

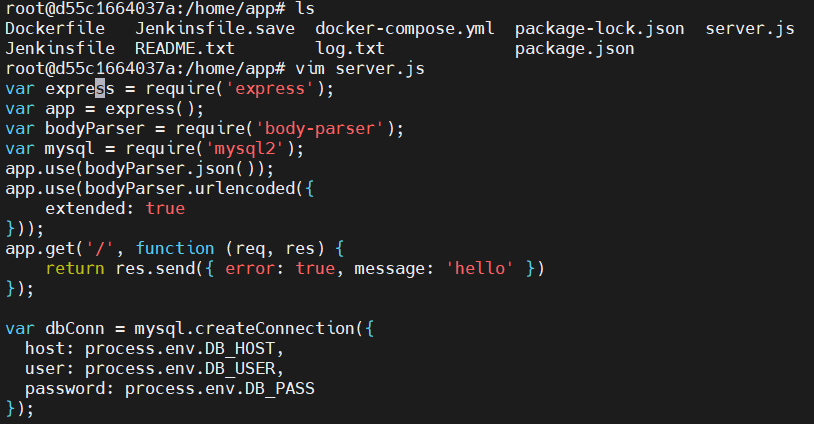
Mysql -h 172.17.0.3 -u root -p



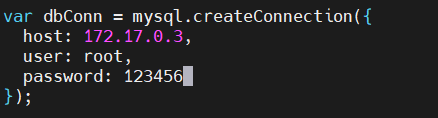
Cài đặt Container chạy app: API users. Khởi chạy một container có mount thư mục $HOME/apiusers vào thư mục /home/app trong container sử dụng image node:14, truy cập vào /bin/bash, mapp port 80:3000



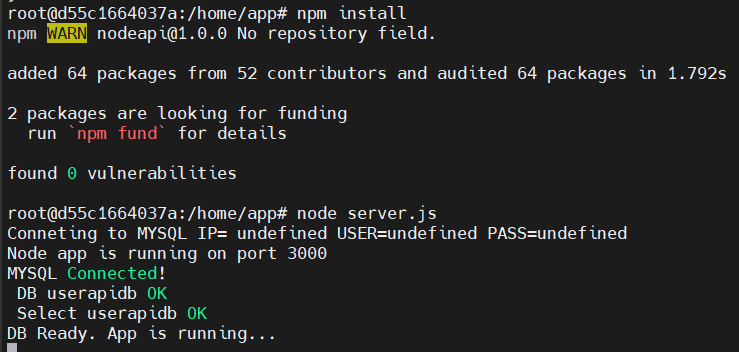
Chuyển đến thư mục: /home/app trong container và edit code:



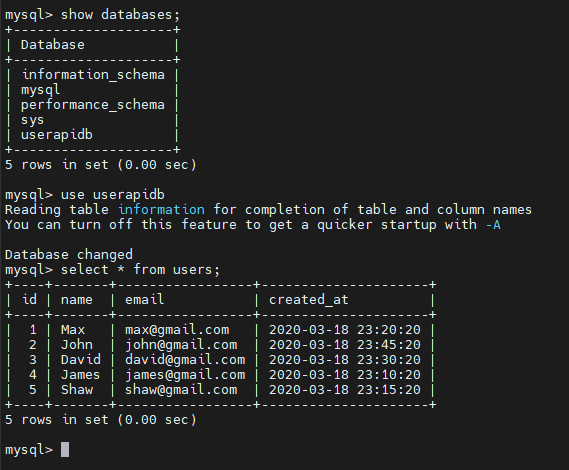
Khai báo lại địa chỉ IP của container chạy Mysql

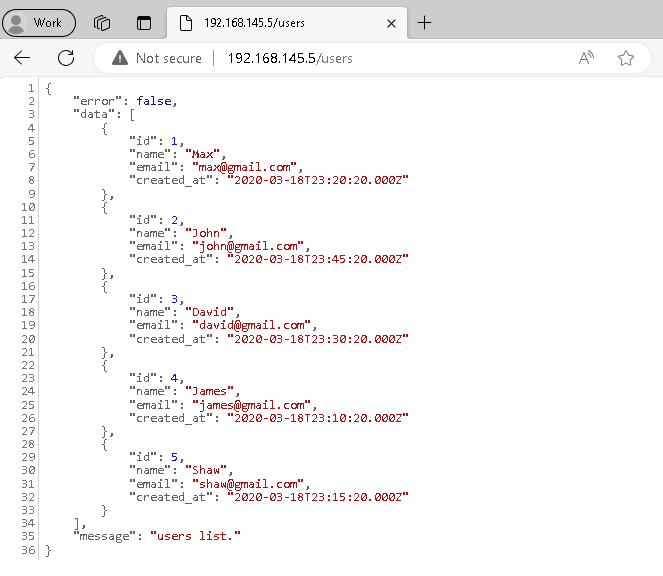


Chạy code bằng lệnh

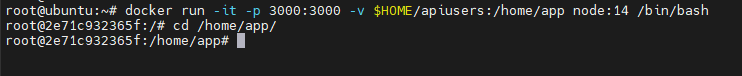


Truy cập mysql kiểm tra dữ liệu thấy đã có:

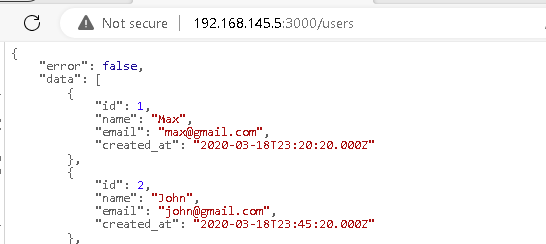




Stop container chạy node app và khởi tạo một container khác chạy cổng 3000



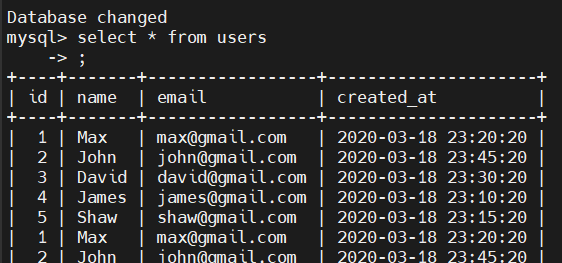
Truy cập bằng trình duyệt vào IP host, port 3000 thấy dữ liệu cũ vẫn còn



Stop container mysql đang chạy và chạy container mysql khác xem có mấ dữ liệu hay không



Sau đó kiểm tra IP container mysqlsrv2 (172.17.0.3):



Như vậy container mysql mới không bị mất dữ liệu

Nhưng cần phải thay đổi lại IP trong file server.js, thực tế không dùng IP container để kết nối DB do dễ bị thay đổi liên tục. Có thể dùng IP của host thay thế do container đã được map port 3306 ra ngoài.

*Docker run -d –mount source=sqlvolume,target=/var/lib/mysql –name mysqlsrv1 -e MYSQL\_ROOT\_PASSWORD=a123456 -e MYSQL\_ROOT\_HOST=% -p 3306:3306 mysql*

**Lab 7: Dockerfile**

Để tự động hóa quá trình tạo ra docker images, người ta tạo ra Dockerfile gồm các chỉ thị, docker sẽ làm từng bước theo hướng dẫn trong file để build ra image có chức năng theo yêu cầu.

Gọi lệnh **docker build** để chạy file Dockerfile đã tạo.

Các chỉ thỉ phổ biến trong Dockerfile gồm có:

**FROM**: chỉ ra image gốc để xây dựng nên docker image mới

**COPY (or ADD):** sap chép dữ liệu từ máy host vào container

**ENV:** thiết lập biến môi trường sau này container sẽ dùng

**ARG:** biến môi trường sử dụng trong quá trình tạo ra Image

**RUN:** Chạy lệnh trong quá trình tạo Image, chứ không phải hạy trong container sau này.

**VOLUME:** gắn ổ đĩa, thư mục vào container

**WORKDIR:** tạo thư mục làm việc trong container

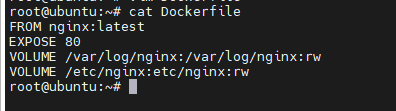
**EXPOSE:** mở cổng chạy ứng dụng trên container

**CMD:** lệnh khởi động mặc định khi chạy container nếu không chỉ định lệnh khác khi tạo container.

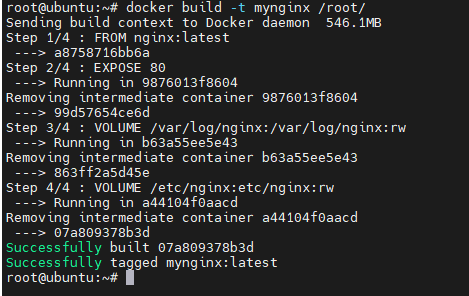
**ENTRYPOINT:** Tạo lệnh luôn chạy khi khởi động container

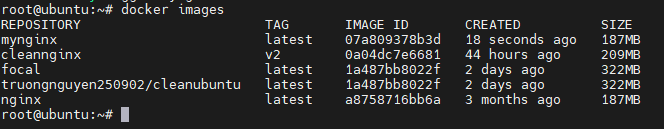
**USER:** tạo user và login với user này khi chạy container

Thử tạo một image nginx bằng Dockerfile:

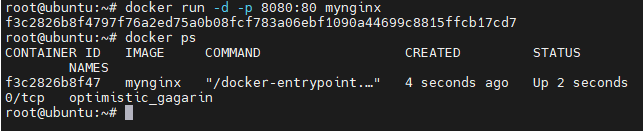


Để build image chạy lệnh sau:



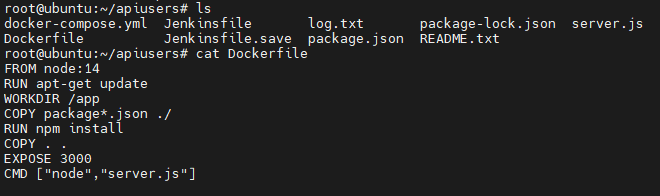


Chạy container bằng images mynginx vừa tạo

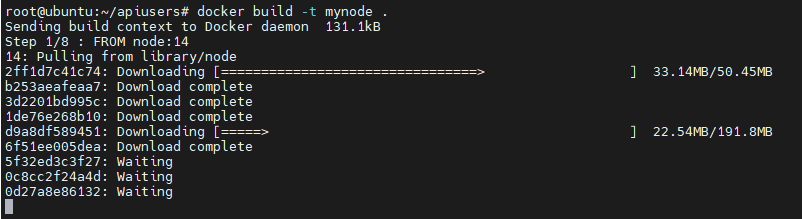




Thử tạo một container khác:

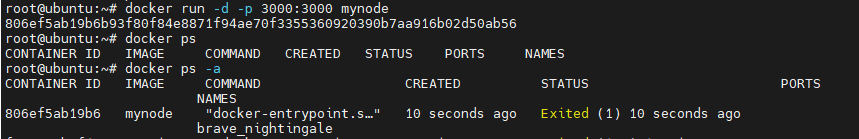


Sau đó lưu lại và chạy lệnh sau để tạo docker image

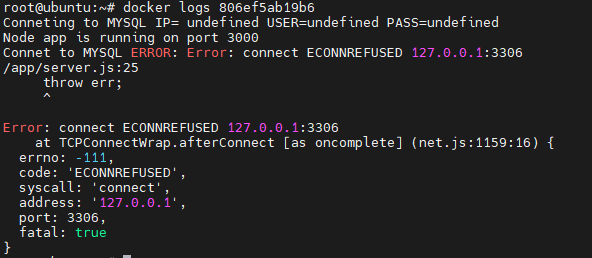


****

Chạy thử container vừa tạo:

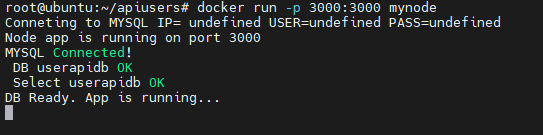


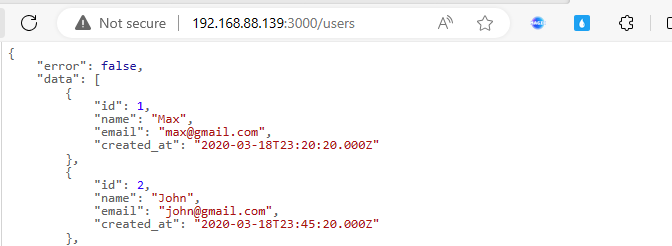
Ta thấy có vẻ bị lỗi gì đó, xem log của container:



Ta thấy container cố gắng connect tới DB server nhưng không được nên bị lỗi

Tiến hành tạo một DB bằng container và sửa lại IP

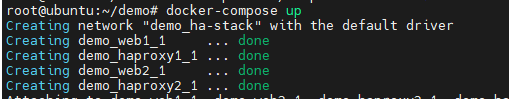




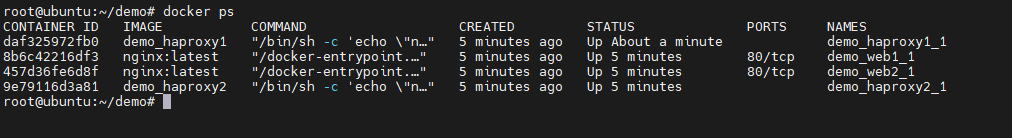
**Lab 8: Docker-compose**

Để tự động hóa quá trình tạo ra containers, sử dụng docker-compose. Docker-compose có thể khởi tạo container từ docker image hoặc khởi tạo container từ dockerfile

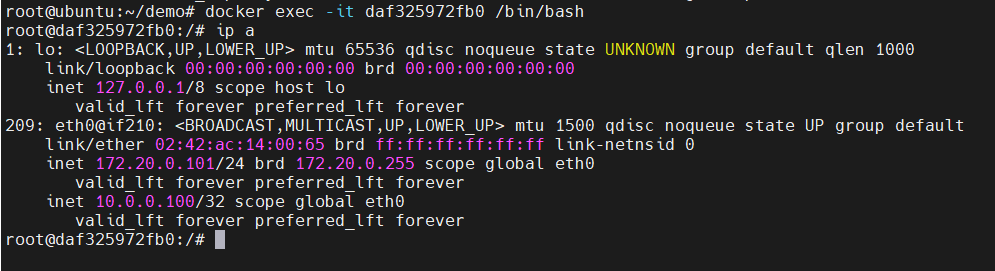
|  |
| --- |
| version: '3.7'  networks:  ha-stack:  ipam:  config:  - subnet: 172.20.0.0/24  services:  web1:  image: nginx:latest  networks:  ha-stack:  ipv4\_address: 172.20.0.11  aliases:  - web1-a.ha.stack  volumes:  - ./web/index1.html:/usr/share/nginx/html/index.html  web2:  image: nginx:latest  networks:  ha-stack:  ipv4\_address: 172.20.0.12  aliases:  - web2-b.ha.stack  volumes:  - ./web/index2.html:/usr/share/nginx/html/index.html  haproxy1:  build: .  networks:  ha-stack:  ipv4\_address: 172.20.0.101  volumes:  - ./keepalived/keepalived-master.conf:/etc/keepalived/keepalived.conf:ro  environment:  - service\_name=haproxy  - service\_2=keepalived  stdin\_open: true  tty: true  cap\_add:  - NET\_ADMIN  haproxy2:  build: .  networks:  ha-stack:  ipv4\_address: 172.20.0.102  volumes:  - ./keepalived/keepalived-backup.conf:/etc/keepalived/keepalived.conf:ro  environment:  - service\_name=haproxy  - service\_2=keepalived  stdin\_open: true  tty: true  cap\_add:  - NET\_ADMIN |



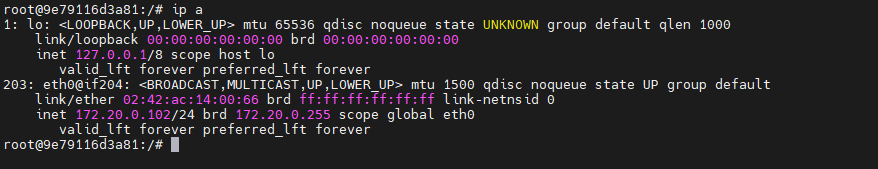
Có 4 container được tạo ra:



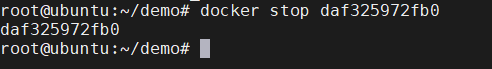
Kiểm tra card mạng trên haproxy, thấy có ip 10.0.0.100



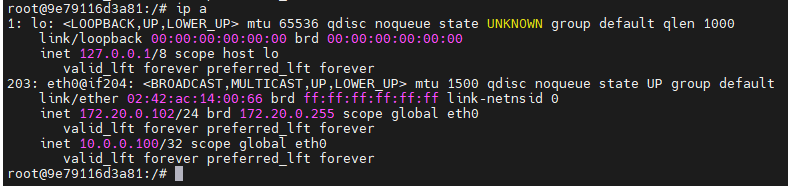
Ip trên container haproxy2:



Thực hiện stop container haproxy1 và kiểm tra ip trên haproxy2:



Trên haproxy2 đã có ip 10.0.0.100



Khi start lại container haproxy1 thì ip 10.0.0.100 trên haproxy2 sẽ mất:

