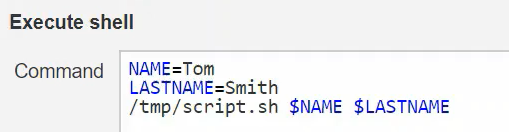
Copy file vào folder docker đang chạy

docker cp script.sh jenkins:/tmp/jenkins.sh

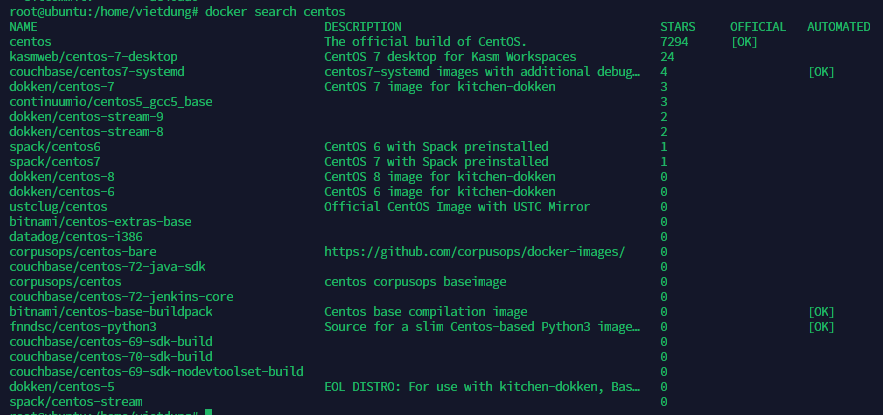
copy file **sript.sh** ở local vào container **jenkins** đường dẫn **/tmp/jenkins.sh**

****

## Docker Image

Iamge official là iamge chính thức của docker hub, ko thay đổi

Còn lại image khác là image public của ng dung đẩy lên, thường hay bị thay đổi, ko nên dùng

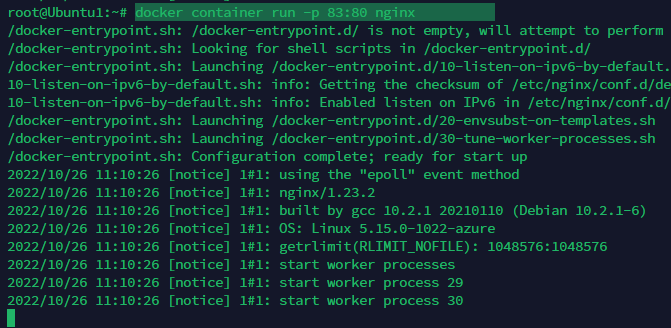
****

# In the background

docker container run -p 83:80 nginx

docker container run --publish 83:80 nginx

2 lệnh giống nhau.



Cơ chế này là chạy ở **foreground,** khi thoát khỏi cái này, cấn ctrl + c thì n ko chạy nữa, vì vậy thường chạy cơ chế **background**

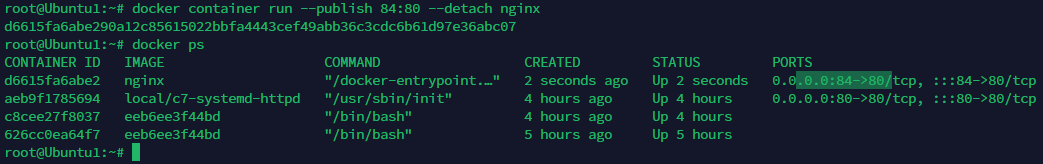
Lệnh này:

* Nó sẽ download image nếu chưa có
* -p là expose port

Cơ chế chạy **foreground** sẽ chạy trên command line, ta muốn chạy nó ở background

docker container run --publish 84:80 --detach nginx

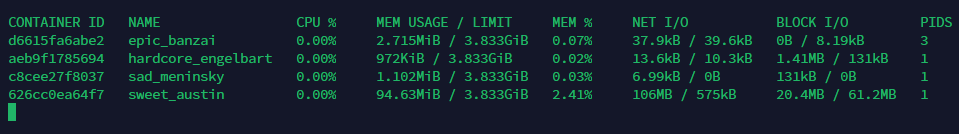
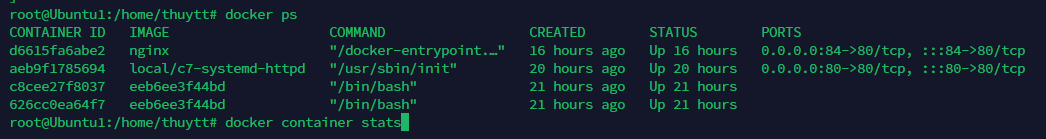
docker container run -p 84:80 -d nginx



2 lệnh là như nhau

docker container inspect : kiểm tra config container

docker container stats : kiểm tra performent container đang chạy



Block I/O là disk performance, nó là cái %CPU

### File .dockerignore

FROM node:14.19.1-alpine AS build

WORKDIR /app

COPY package.json yarn.lock ./

COPY . ./

CMD ["/bin/sh", "-c", "npm install yarn --global"]

Khi build image mà trong thư mục của dockerfile có file .dockerignore liệt kê các package sẽ bị ignore khi build docker

Vì vậy có định nghĩa trong dockerfile thế nào thì cũng ko thể copy file từ thư mục góc vào thư mục root của docker image

WORKDIR /app : tạo ra 1 thư mục root khi exec container

COPY . ./

Hoặc COPY . . : copy tất cả thư mục từ thư mục gốc vào thư mục root của docker container

Thư mục root ở đây là /app

Để xóa file bị ignore khi copy, chỉ có thể update trong thư mục .dockerfile

* Command để dọn dẹp

sudo docker system prune -a –f

## Run && CMD

RUN chỉ chạy 1 lần khi build image, chạy khi build

CMD sẽ luôn được chạy khi 1 container được khởi tạo từ Image, CMD chạy khi run container khởi tạo từ image

COPY và ADD trong Docker phục vụ chung một mục đích đó là copy file từ một nơi nào đó vào trong Image.

COPY nhận vào đối tượng cần copy và đích cần copy tới trong image. Và COPY **chỉ cho phép** ta copy file từ local, từ máy gốc của chúng ta vào trong Image

ADD cũng làm được điều tương tự nhưng nó có thêm 2 chức năng đó là:

* ta có thể copy từ một địa chỉ URL vào trong Image
* ta cũng có thể giải nén một file và copy vào trong Image

Hầu hếu các trường hợp khi ta dùng đến URL là ta muốn download file thì ta sẽ dùng câu lệnh **RUN curl/wget ....** để download file hoặc nếu ta muốn giải nén file thì ta cũng sẽ dùng **RUN tar -xvzf ...** để giải nén

Do đó lời khuyên của mình là luôn dùng **COPY** để làm rõ bạn đang muốn thực hiện hành động nào nhé



Tôi có thể FROM nhiều lần được không ?

Câu trả lời là có nhé, bạn có thể FROM nhiều lần từ nhiều môi trường. Thường dùng trong trường hợp ta chia quá trình build Image thành các giai đoạn (stage), mỗi giai đoạn ta cần 1 môi trường khác nhau. Phần này khi demo với VueJS các bạn sẽ thấy nhé

# Image alpine

## Alpine for nodejs

Thay vì dùng base image nodejs, t dùng base image alpine custom nhẹ hơn nhiều

FROM alpine:latest

WORKDIR /app

COPY . .

RUN apk update && \

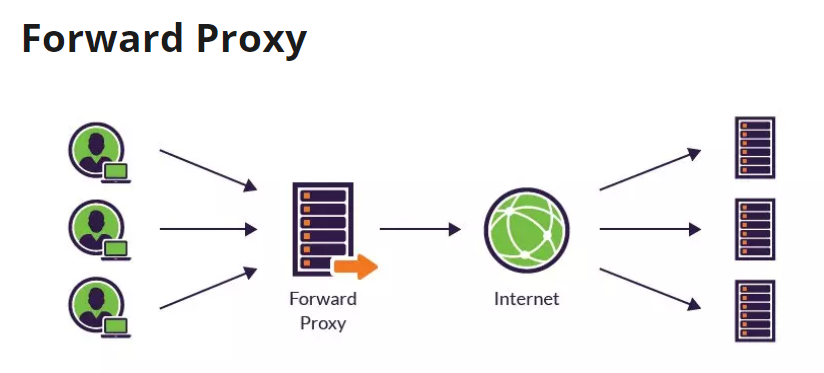
    apk upgrade && \

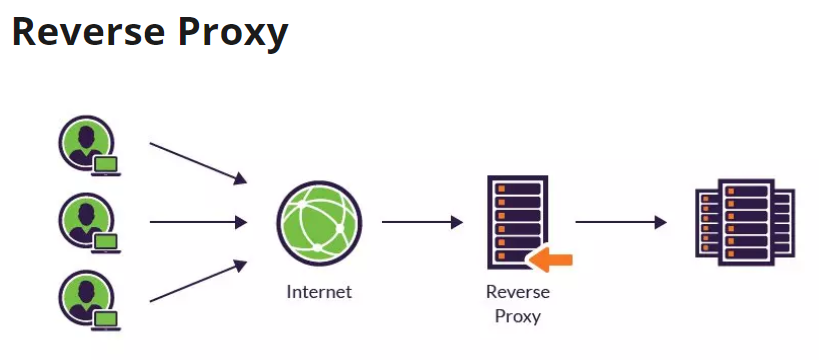
    apk add --no-cache --update nodejs npm

RUN npm install

CMD ["npm", "start"]

Proxy nginx



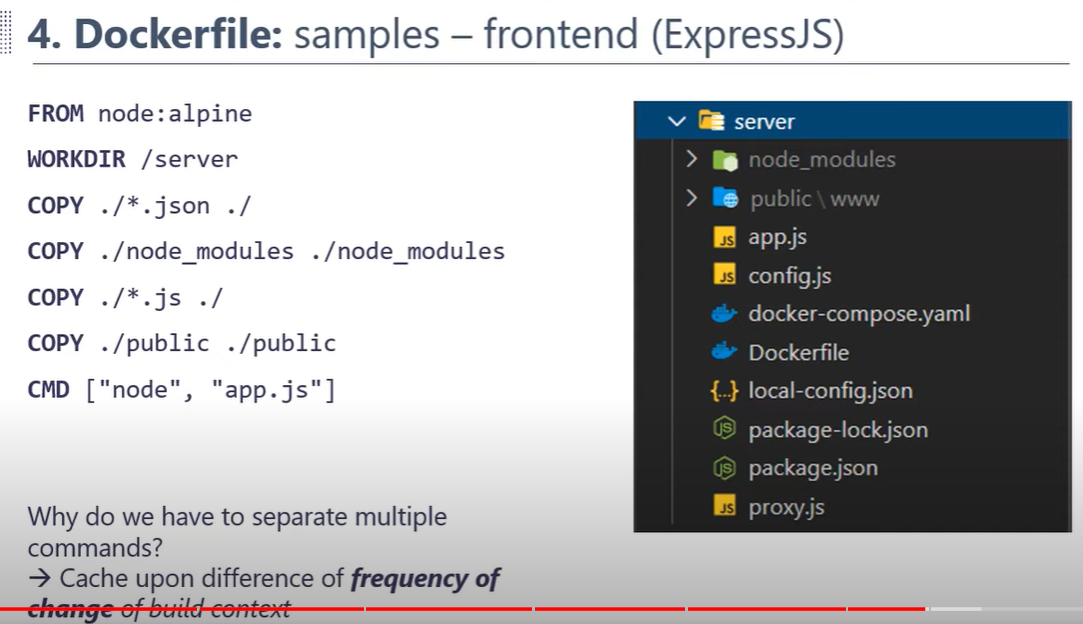


Forward proxy: đứng trước internet – phía ng dùng vào internet, giúp ng dùng tránh web đen

* filter user ng dùng mạng nội bộ bên trong ra ngoài internet
* chặn user ko cho user vào web đen

Reserse Proxy: đứng sau internet

* filter request đến backend – server để server sử lý và trả lại request
* Reserse Proxy: là cân bằng tải LB



Tại sao ko gộp chung trong 1 lệnh COPY

Chúng ta đang tận dụng tính năng **cache, lưu cache image dựa trên tần suất thay đổi của view context**

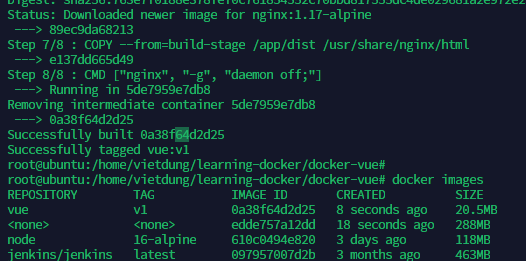
**Quy trình build images**

Image là 1 tập hợp nhiều layer, mỗi layer tương ứng vs 1 câu lênh trong dockerfile

Khi build image, docker sẽ duyệt từng câu lệnh trong dockerfile, qua mỗi câu lệnh, **docker sẽ khởi tạo 1 container tạm thời** , từ layer trước đó tạo ra thay đổi bên trong container theo đúng mô tả câu lệnh, và sau đó sẽ sao lưu container này chạy tiếp layer tiếp theo (cache), và n sẽ hủy container này và tạo tiếp container mới

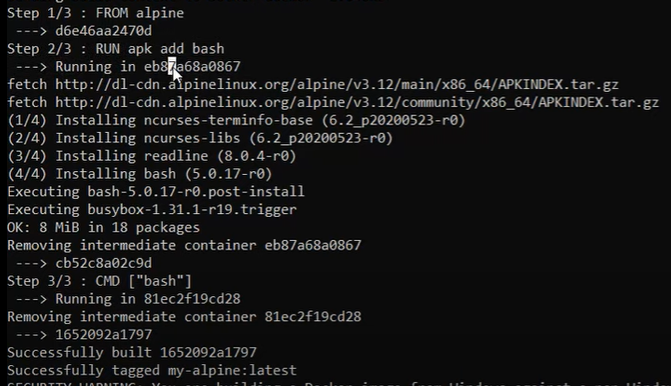
**Sao mỗi layer, docker sẽ tạo ra 1 container tạm thời mới**

Đó là lý do 1 số lần build dockerfile bị lỗi n sẽ trả cho ta 1 image số … ko phải image ta định nghĩa, vì image đó là layer n chạy thành công trước khi n bị failed



Như bạn có thể thấy đây,

* ở step 6/8 tạo 1 container có id = 89ec9da68213 và chạy step 6/8
* đến step 8/8 n tạo 1 container id khác = 5de7959e7db8 từ container 7/8,
* và cuối cùng move tiếp container image 5de7959e7db8 và cuối cùng tạo ra image có id = **0a38f64d2d25**



Container sau được tạo từ container trước



Sao lưu và tạo container từ container trước

* 1 layer = 1 image