Docker
The Complete Guide with Hands-on Tutorial

Presenter: Bach Minh Nam

Oct 2020

✉ nambach2@gmail.com youtube.com/c/nbstudio2

Agenda

1. Docker
2. Docker compose
3. Docker Swarm

Agenda

1. **Docker**
    1. *What is Docker? Why do we need Docker?*
    2. *Basic concepts: Docker Engine, container, image, registry, basic commands*
    3. *Core concepts of Container*
    4. *How to build my own image? Dockerfile AZ*
2. Docker compose
3. Docker Swarm

Docker

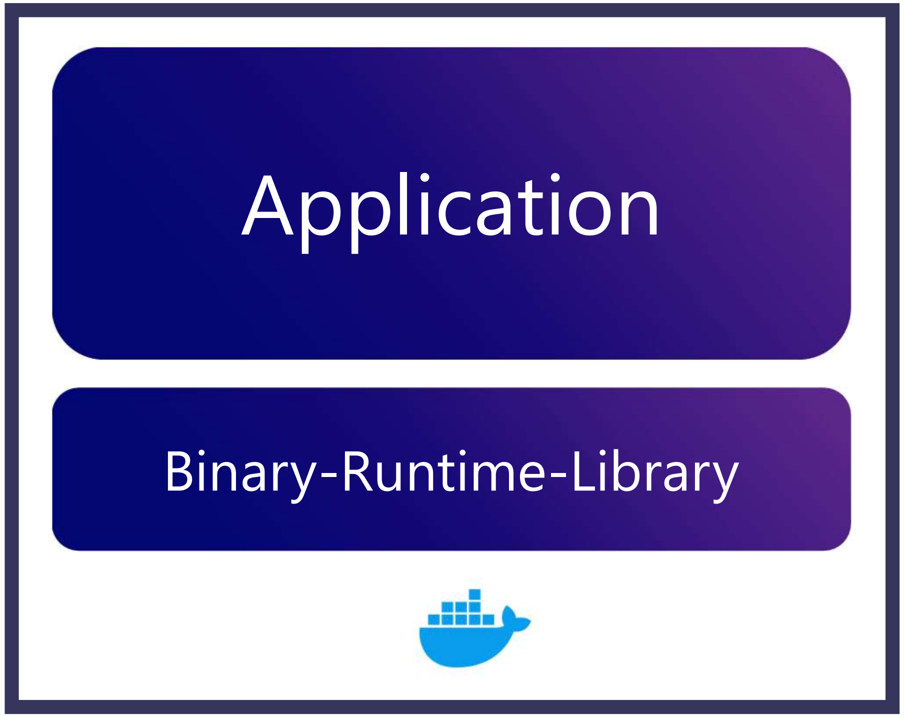
1

1. What is Docker? Why do we need Docker?

1. What is Docker?

When developing application, it usually needs ***belonging dependencies*** such as binary files, libraries, runtime environment…

Docker is a technology allows us to wrap the *application* and its *dependencies* into **one package,** which are **portable** (run anywhere) and **executable** (run anytime).



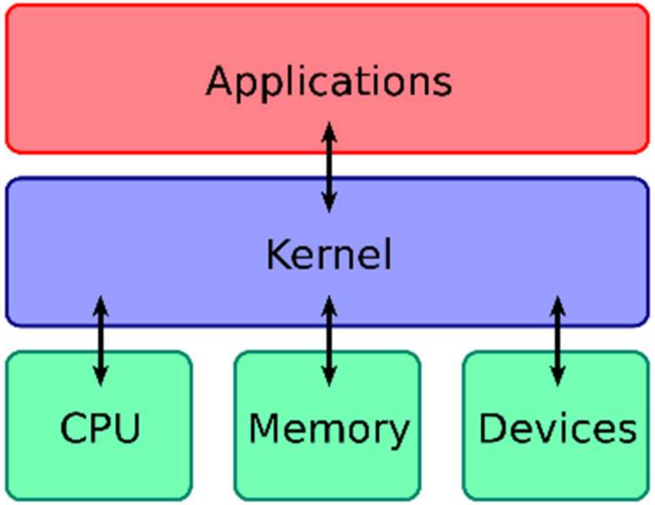
1. What is Docker: *container, image*

“Docker is a technology allows us to wrap the *application* and its *dependencies* into **one package,** which are **portable** (run anywhere) and **executable** (run anytime)”

- Those packages are called **images**.
- When executing an image, we get a **container**.
- Containers functions like a virtual machine with fully provided features such as file system, network interface, process tree…
- However, containers are not virtual machines at all. They don’t have their own OS kernel, but **share the same kernel** with the physical machine.
- In the end, they are just ***processes*** running on the OS and managed by Docker.

1. **What is Docker:** OS kernel

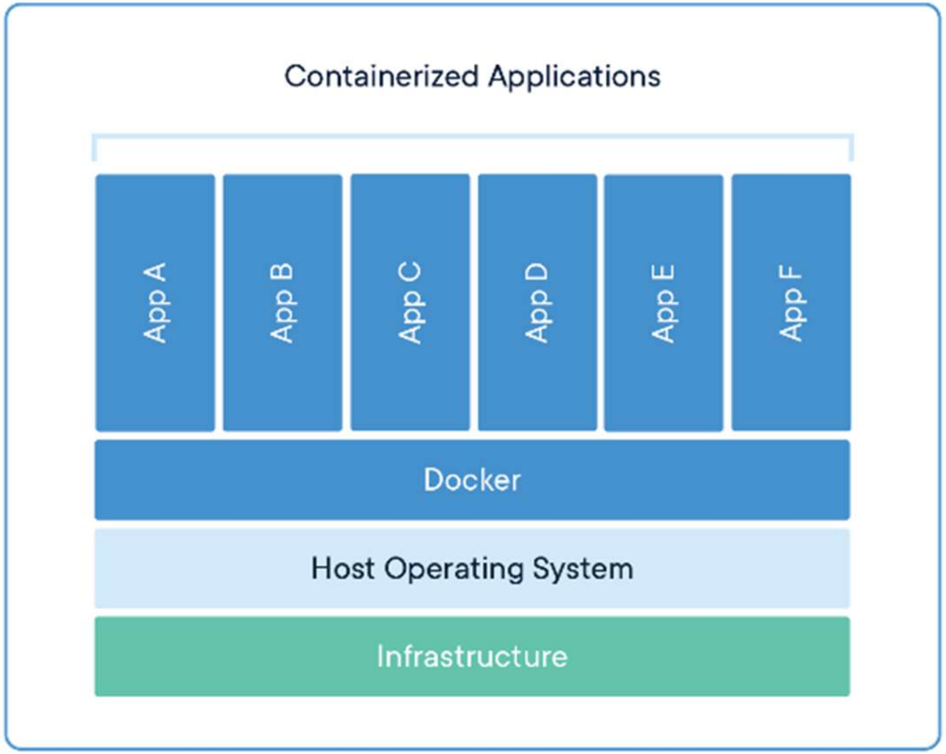
- Kernel is the core program, the heart of the operating system.
- It controls over everything in the system.
- It facilitates interactions between hardware and software components



1. What is Docker: *container vs VM*

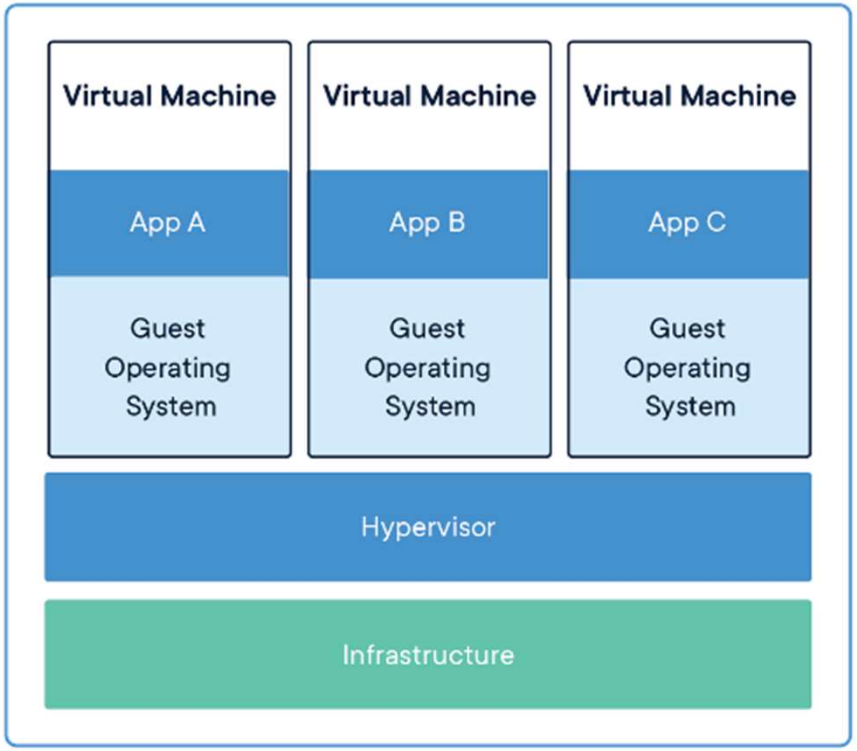
Container

Abstraction at *software* layer - share same OS kernel
Lightweight (megabytes)



Virtual Machine

Abstraction at *hardware* layer – run its own OS
Heavy (gigabytes)



1. **What is Docker:** underlying technology

Docker is written in the **Go** programming language and takes advantage of several features of the Linux kernel to deliver its functionality:

- Namespaces: pid, net, ipc, mnt, uts
- Control groups: hardware resources
- Union file systems: layers creation
- Container format: format of Docker image

1. Why do we need Docker?

Docker helps to package (containerize) and ship and run application more easily.

Manually

git
Java
Maven™
JS
node
PM2

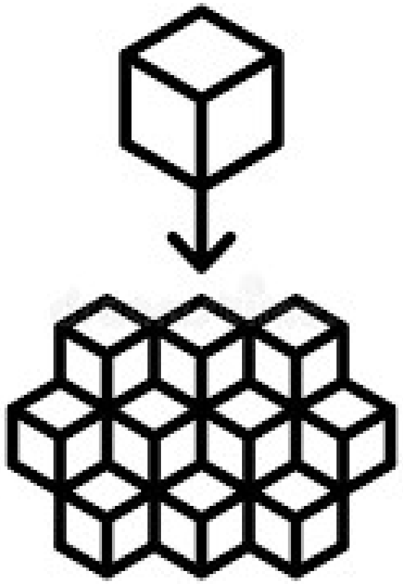
With Docker

1. Why do we need Docker?

Docker helps to package (containerize) and ship and run application more easily.

Microservices

- *Immutable*: same behavior
- *Lightweight*: fast creation
- *Stateless*: disposable and ephemeral



Docker

1

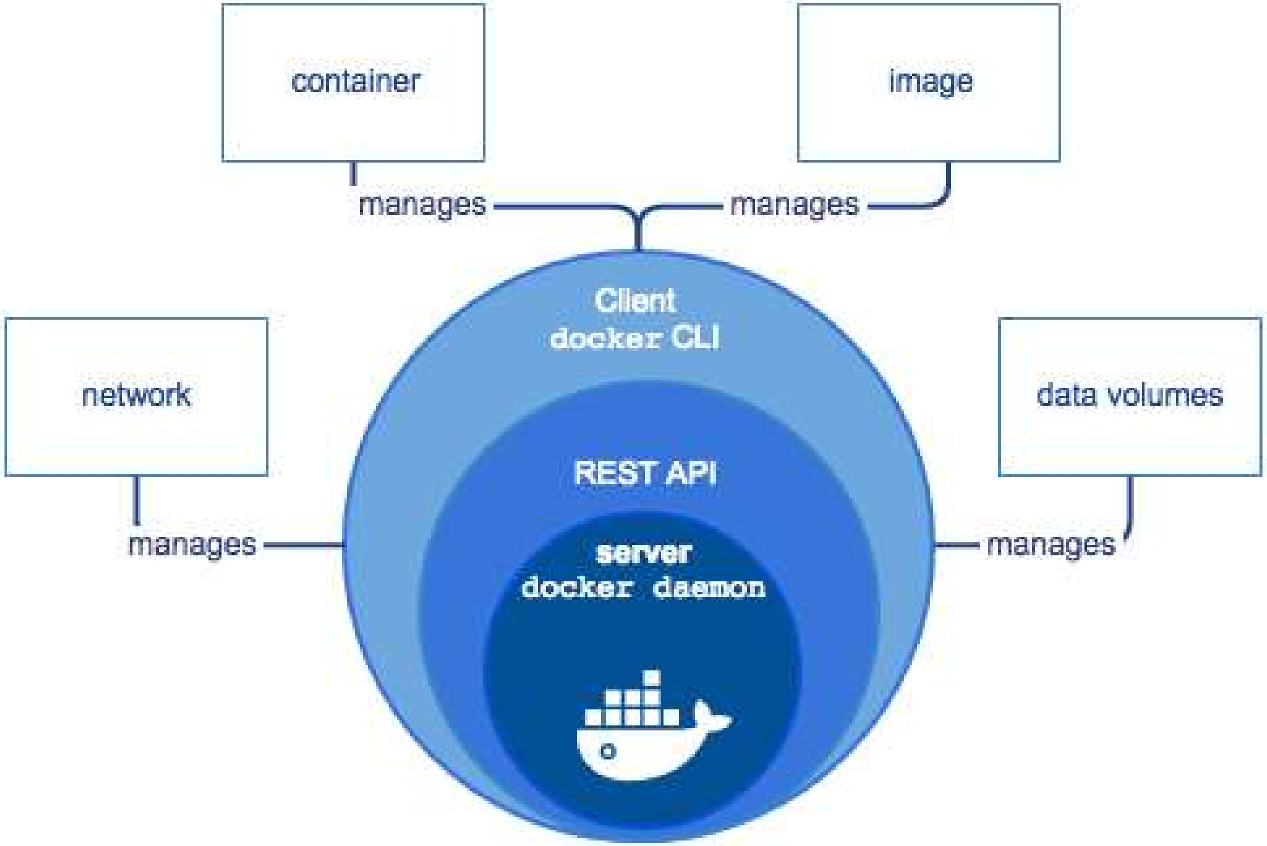
2. Basic concepts: Docker Engine, container, image, registry

2. Basic concepts: *Docker Engine*

Docker Engine

- Docker Daemon & exposed APIs
- Docker client (cli)

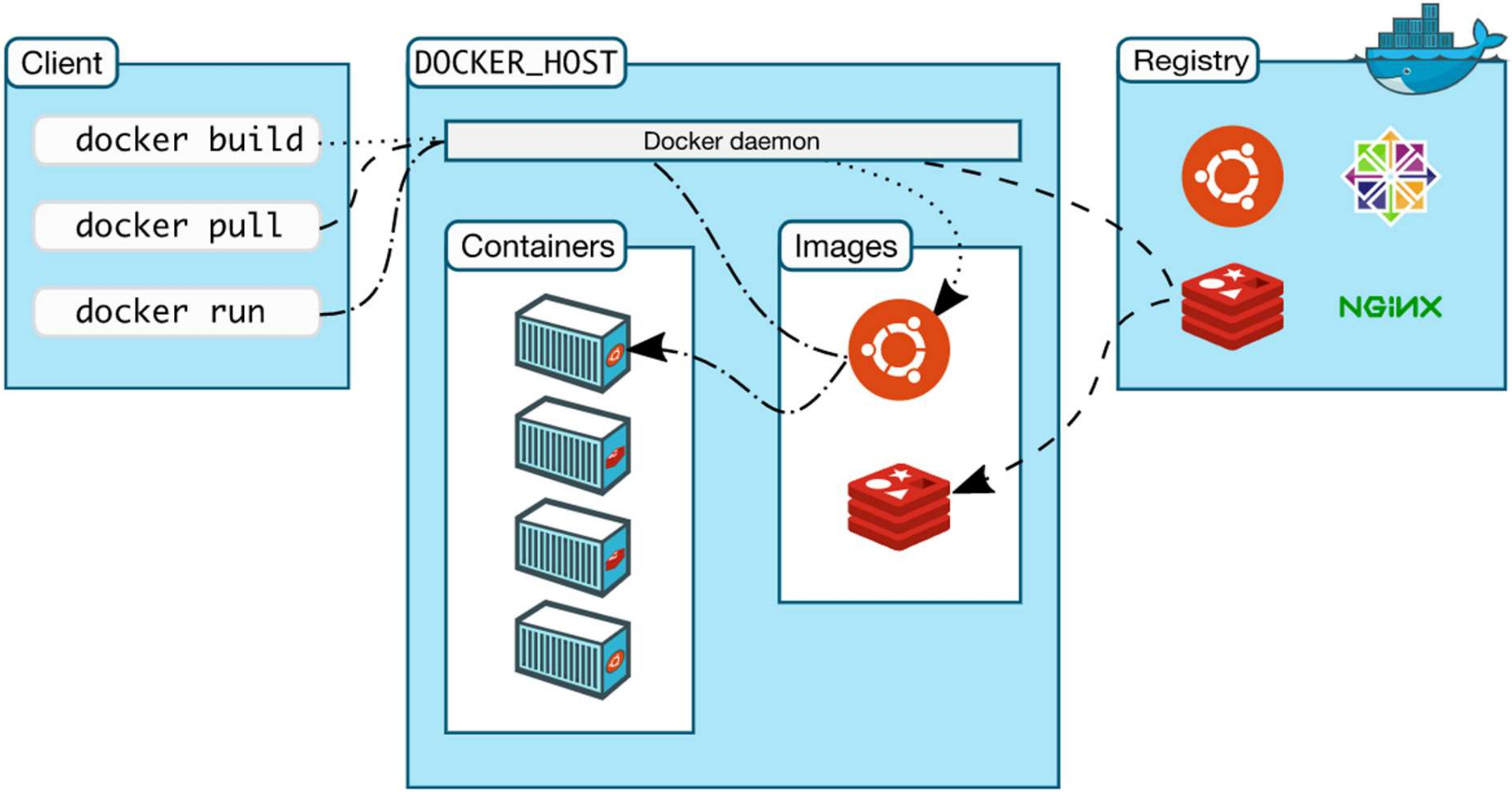
Share OS kernel
⇒ OS dependent
⇒ On Windows must install a Linux virtual machine to run Linux containers



2. Basic concepts : *registry*

- Docker packages (containerizes) application and its dependencies into **image**.
- Images are stored in Docker repository call **container registry**.
- Some Docker registry providers:
    - Docker Hub
    - Amazon Elastic Container Registry (ECR)
    - Google Container Registry (GCR)
    - Azure Container Registry (ACR)

2. Basic concepts: *overall*



Docker

1

2. Basic commands: *demo*

2. Basic commands

Syntax

```
docker <component> <command>
```

Components:

- image
- container
- network
- volume
- …

Commands:

- ls: list
- run
- exec
- stop
- pull
- prune
- …

2. Basic commands: *image*

```
docker image pull <image>
docker image pull <image>:<tag>
docker image push <image>:<tag>
docker image ls | docker images
docker image prune
```

Short-hand:

```
  docker pull
  docker push
```

2. Basic commands: *container*

```
docker container run <image>
docker container ls | docker container ls –a
docker ps | docker ps -a
docker container stop <container_id>
docker container prune
docker container exec <container_id> <command>
```

Short-hand:

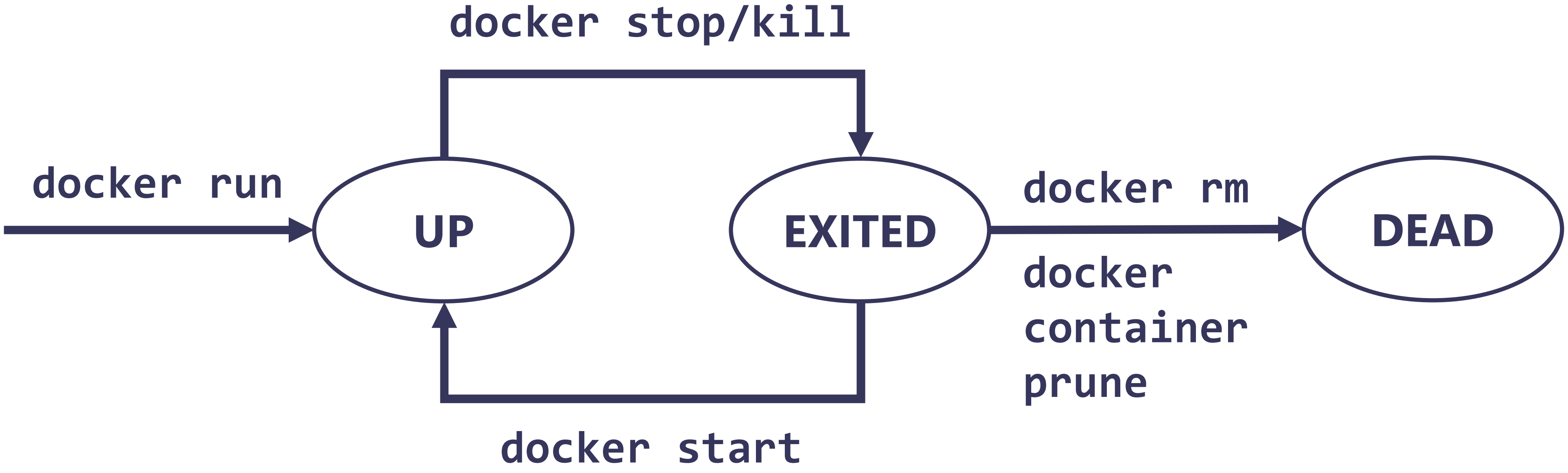
```
  docker run
  docker stop
  docker exec
```

Docker

1

3. Core concepts of Container

3. Container Life Cycle



3. Container Life Cycle in Depth: PID 1

- Life cycle of a container depends on life cycle of the process running insides that has **PID = 1** (main process). The container will remain alive *as long as* the PID 1 process is alive.
- When using `docker stop`, Docker will send **SIGTERM** (terminating signal) to stop process PID 1 insides, then the container will stop with **EXITED (0)**.
- Within *10 seconds*, if container does not exit, Docker will send **SIGKILL** (kill signal) and the container will stop immediately with **EXITED (137)**.

3. Container Life Cycle in Depth: PID 1

`docker run alpine`

- Why does the container exit immediately?

`docker run -it alpine`

- `-i or --interactive`: keep STDIN open even if not attached
- `-t or --tty`: allocate a pseudo-TTY (TeleTYpewriter)
- To exit the shell within a container, press **Ctrl + D** because Ctrl + C does not work.
- Exit the shell also makes the container exit. How to keep the container alive?
- Attach – detach mode: press **Ctrl + P + Q** to detach container from local terminal; **`docker attach <container_id>`** to re-attach container stdin and stdout into local terminal.

`docker run -d <image>`: to run a container in detach mode

3. Execute commands insides container

Syntax

- `docker exec <container_id> <command>`

Example

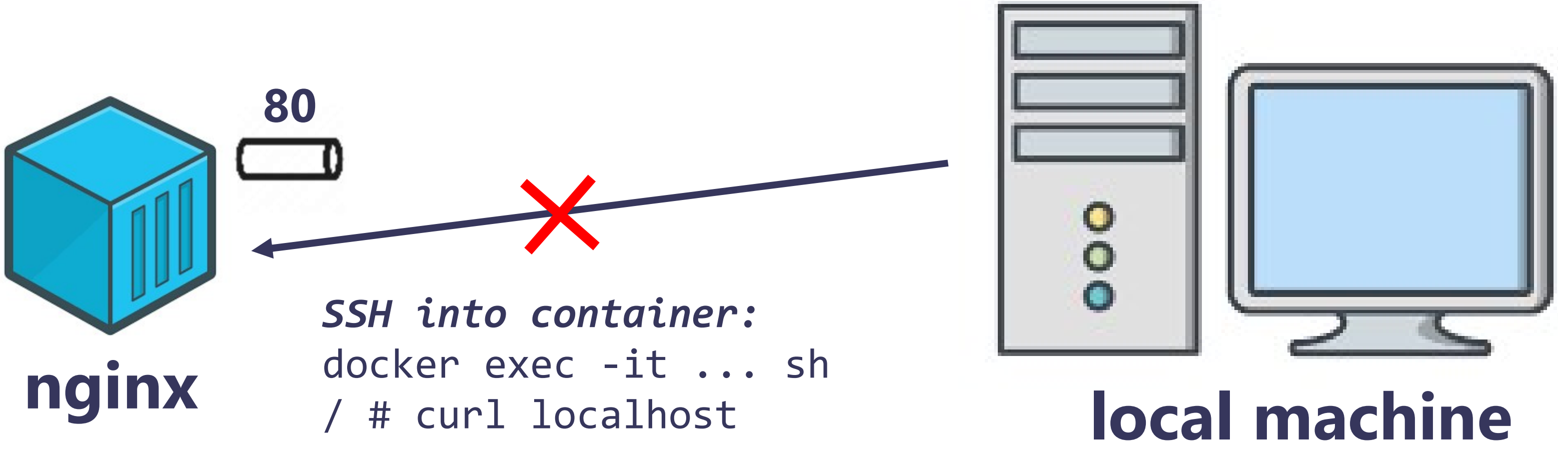
- `docker exec <container_id> echo Hello World`
- `docker exec <container_id> echo $PATH`
- `docker exec <container_id> sh -c “echo $PATH”`
- `docker exec <container_id> cat /etc/os-release`

Good To Remember

To SSH into any container, use exec shell or bash with -it

- `docker exec -it <container_id> sh`
- `docker exec -it <container_id> bash`

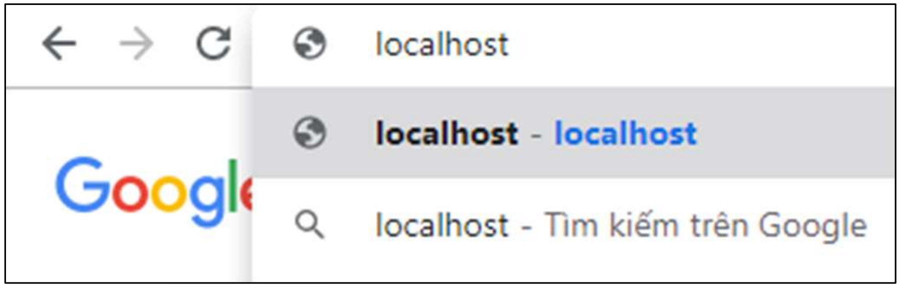
3. Port Mapping



SSH into container:

```
docker exec -it ... sh
/ # curl localhost
```

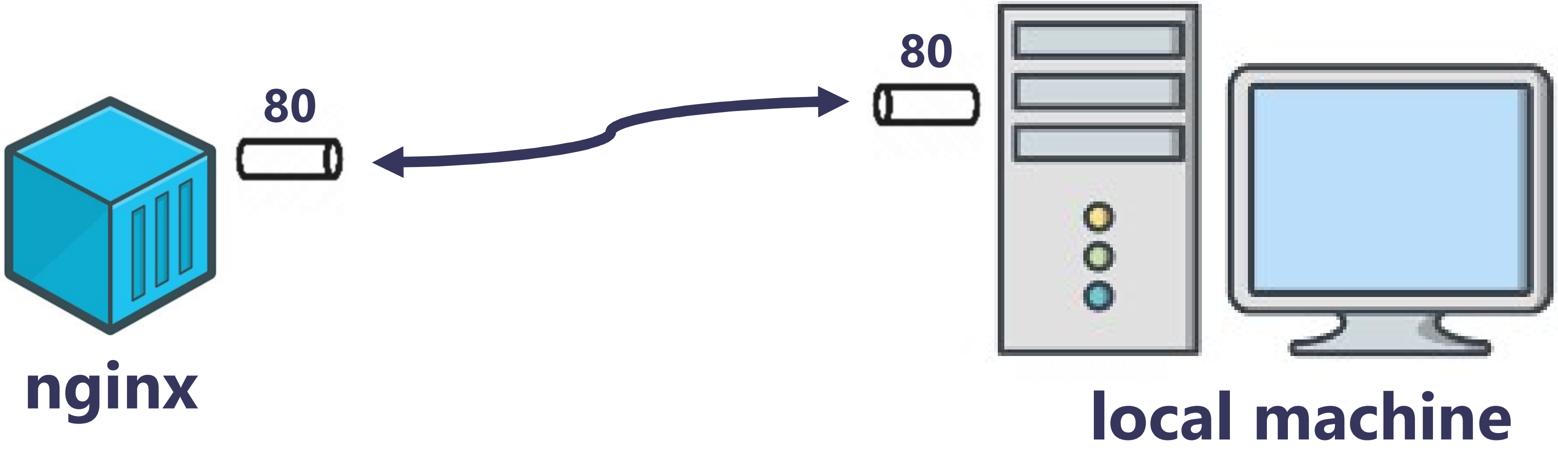
3. Port Mapping



localhost

localhost - localhost

localhost - Tìm kiếm trên Google



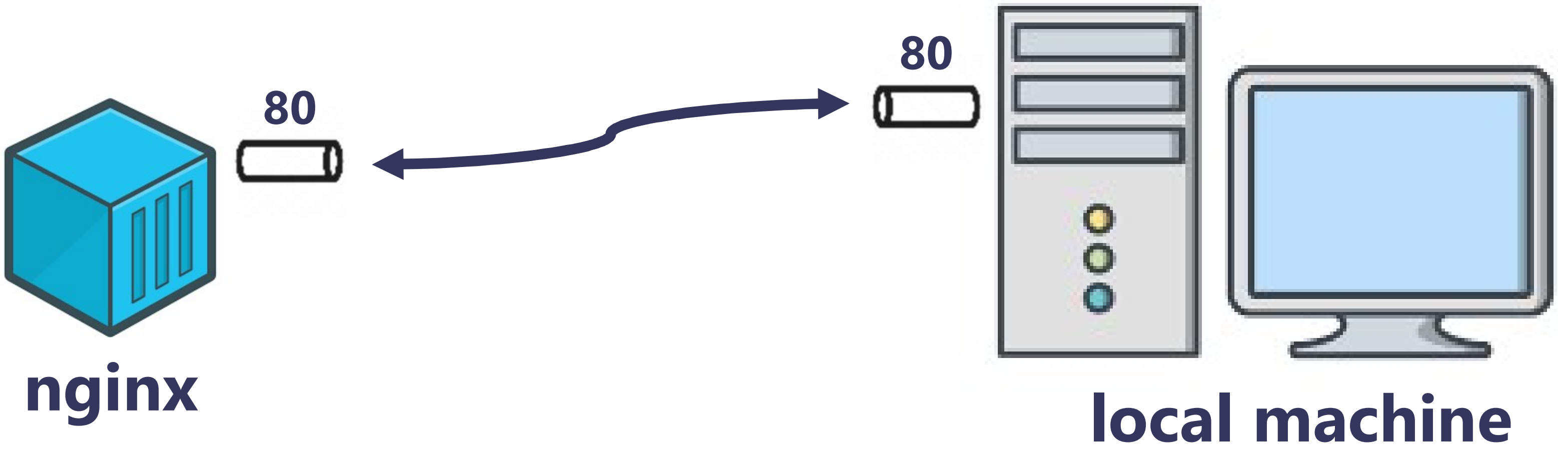
3. Port Mapping

Syntax

- `docker run -p <target_port>:<container_port> ...`

Example

- `docker run -p 80:80 nginx`



3. Logs trace

Syntax

```
docker logs -f <container_id>
```

- `-f` : keep following the log output

3. Volume – bind mount

Image is immutable. Container is stateless.

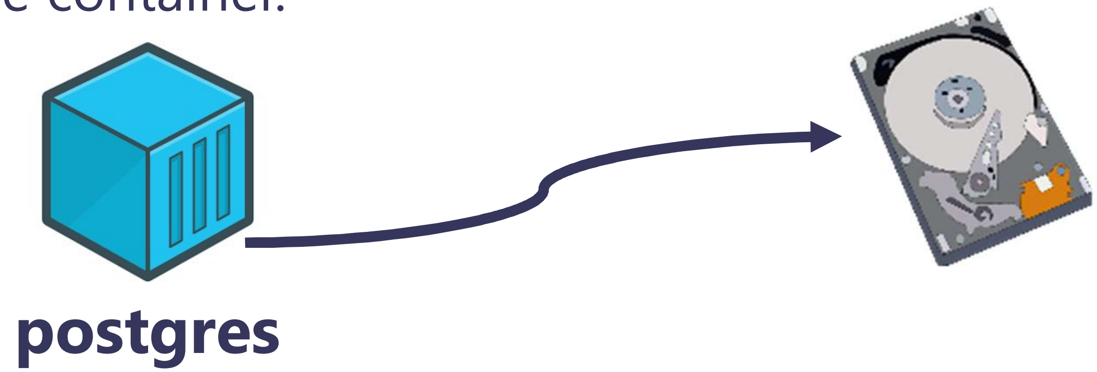
```
docker run -p 5432:5432 postgres
```

- How to persist data in postgres? How to make container stateful?

  → Use volume

<u>Volume</u>

- Volume indicates the partition of memory that Docker uses to persist data inside container.



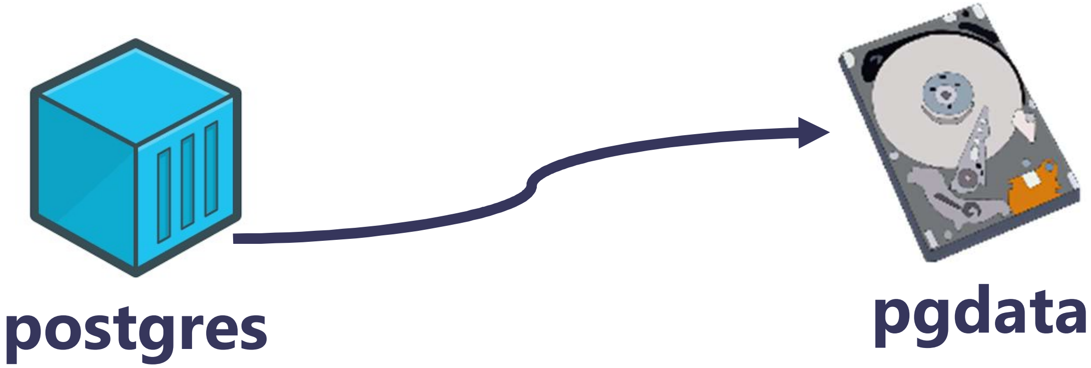
3. Volume – bind mount

Syntax

```
  docker volume create [volume_name]
  docker run -v [local_dir/volume]:[container_dir]    ...
```

Example

```
docker volume create pgdata
docker run -v pgdata:/var/lib/postgresql/data -p 5432:5432 postgres
docker run -v /usr/data:/var/lib/postgresql/data -p 5432:5432 postgres
```



3. Volume – bind mount

Syntax

```
  docker volume create [volume_name]
  docker run -v [local_dir/volume]:[container_dir]    ...
```

Example

```
docker volume create pgdata
docker run -v pgdata:/var/lib/postgresql/data -p 5432:5432 postgres
docker run -v /usr/data:/var/lib/postgresql/data -p 5432:5432 postgres

docker run -v “C:\users\html”:/usr/share/nginx/html –p 80:80 nginx
```

3. **Container** immutability & statelessness **- image**

```
docker run -it alpine
#sh: apk add bash
docker run -it alpine bash
```

- Error! Modifying container does not affect the original image.
- How to save the desired state of the container?

  → Build our own image

Docker

1

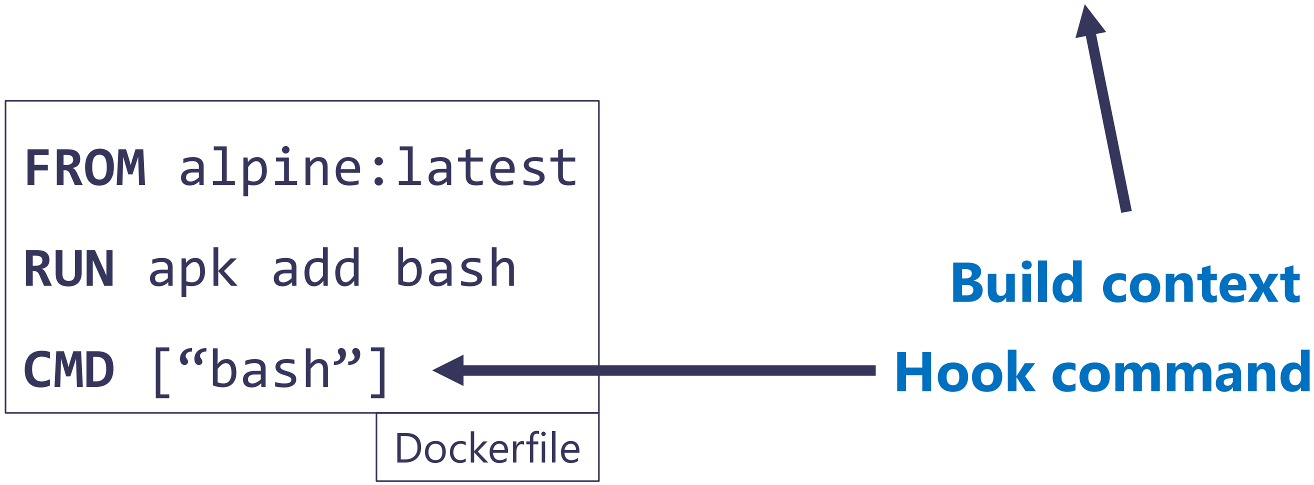
4. How to build my own image? Dockerfile AZ

4. Dockerfile

- Dockerfile is a template that allows us to instruct Docker to build our own image step by step.
- Back to our example with alpine having bash

Syntax

- `docker build -t <image_name>:<tag> .`



```
FROM alpine:latest
RUN apk add bash
CMD [“bash”]
```

Dockerfile

Build context

Hook command

4. Build context - .dockerignore

- “Build context” refers to the folder that contains Dockerfile. All contents insides that folder is call build context.
- Docker client will send build context to Docker Daemon insides Linux machine to build the image.
- Be careful: remember put all necessary resources (files, images…) into the build context, as well as remove all unnecessary things, in order to utilize the size of request payload.

.dockerignore

- When sending build context to Docker Daemon, it will ignore files and folders listed in the .dockerignore – just the same as .gitignore

4. Demo

```
docker build -t my-alpine .
docker tag <image_name> <new_name>
```

```
FROM alpine:latest
RUN apk add bash
CMD [“bash”]
```

Dockerfile

4. Dockerfile keywords

```
FROM <image>
RUN <command>
WORKDIR <directory>
COPY <src> <dest>
ADD <src/URL> <dest>
EXPOSE <port>
```

```
CMD command argument1 argument2...
```

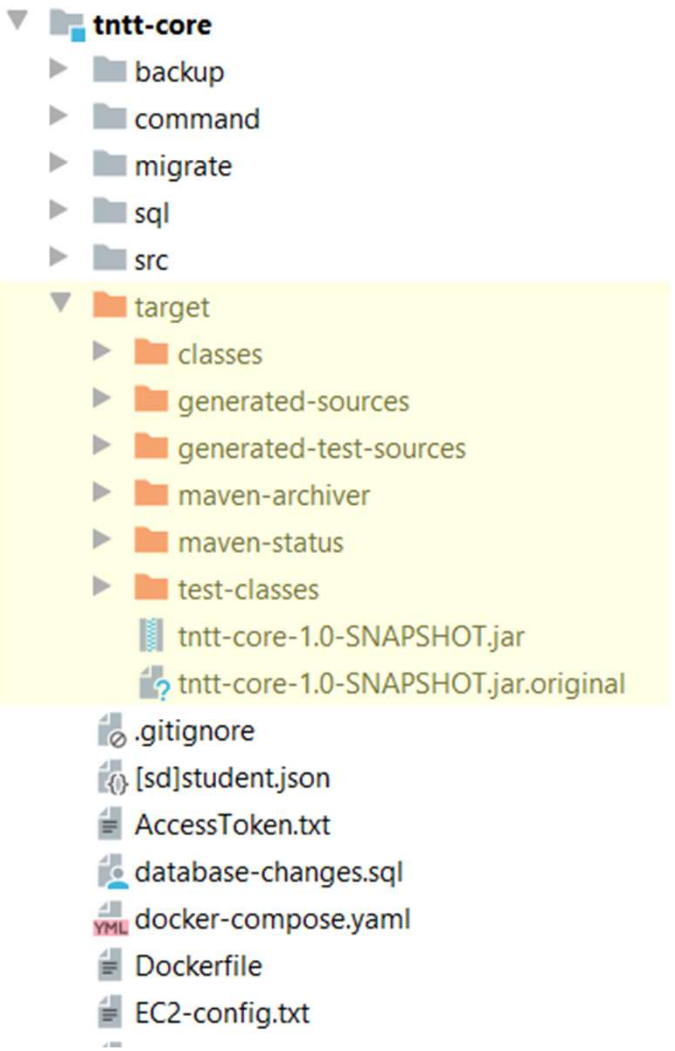
Shell form

```
CMD [“command”, “argument1”, “argument2”, ...]
```

Exec form
(prevent shell injection – recommend)

4. **Dockerfile:** samples – backend (Java)

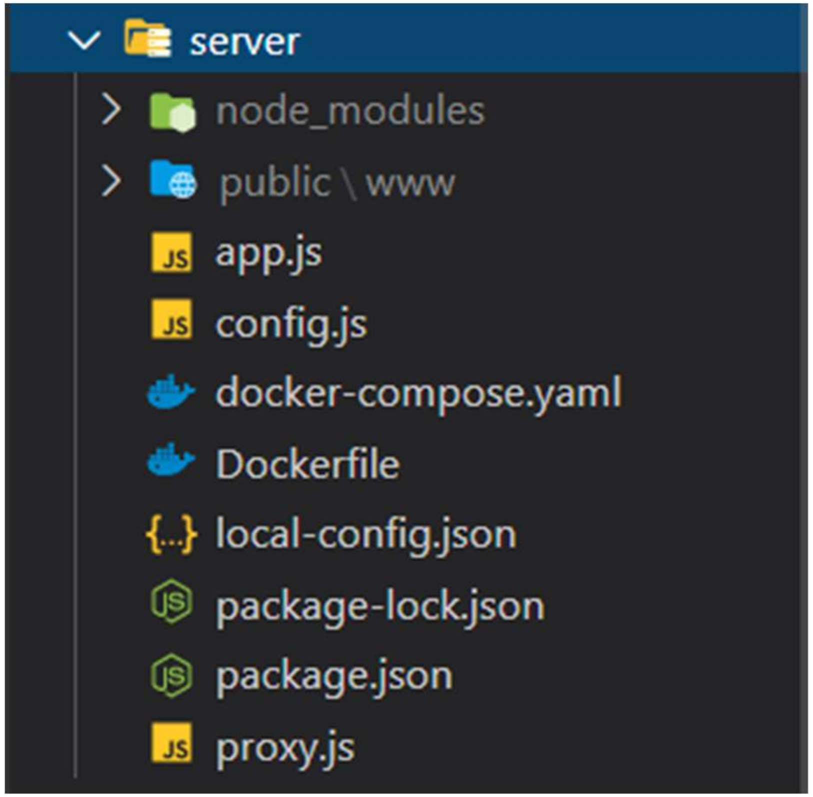
```
FROM openjdk:8-jre

EXPOSE 8080

WORKDIR /app

COPY ./target/app-1.0-SNAPSHOT.jar .

CMD ["java", "-jar", "app-1.0-SNAPSHOT.jar"]
```



```
▼ tntt-core
  ▶ backup
  ▶ command
  ▶ migrate
  ▶ sql
  ▶ src
  ▼ target
    ▶ classes
    ▶ generated-sources
    ▶ generated-test-sources
    ▶ maven-archiver
    ▶ maven-status
    ▶ test-classes
      tntt-core-1.0-SNAPSHOT.jar
      tntt-core-1.0-SNAPSHOT.jar.original
    .gitignore
    [sd]student.json
    AccessToken.txt
    database-changes.sql
    docker-compose.yaml
    Dockerfile
    EC2-config.txt
```

4. **Dockerfile:** samples – frontend (ExpressJS)

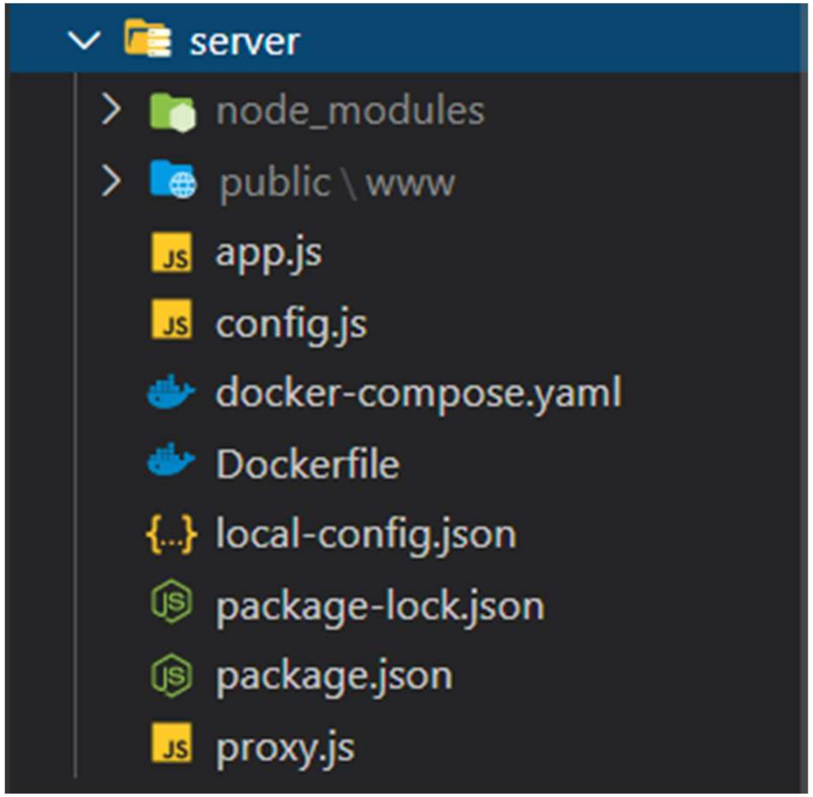
```
FROM node:alpine
WORKDIR /server
COPY ./*.json ./
COPY ./node_modules ./node_modules
COPY ./*.js ./
COPY ./public ./public
CMD ["node", "app.js"]
```



```
server
  node_modules
  public \ www
  app.js
  config.js
  docker-compose.yaml
  Dockerfile
  local-config.json
  package-lock.json
  package.json
  proxy.js
```

4. **Dockerfile:** samples – frontend (ExpressJS)

```
FROM node:alpine
WORKDIR /server
COPY ./*.json ./
COPY ./node_modules ./node_modules
COPY ./*.js ./
COPY ./public ./public
CMD ["node", "app.js"]
```



```
server
  node_modules
  public \ www
  app.js
  config.js
  docker-compose.yaml
  Dockerfile
  local-config.json
  package-lock.json
  package.json
  proxy.js
```

Why do we have to separate multiple commands?
→ Cache upon difference of ***frequency of change*** *of build context*

4. Dockerfile: Utilizing size by layers

Build process

- Image is an ordered set of layers. Each layer is according to a command in Dockerfile.
- With each command, Docker will create a temporary container from previous layer, then execute command inside that container, take a snapshot of it (commit) into a new layer, and finally remove the temporary container which is no longer needed.
- In order to minimize the size of final image, we can combine multiple command using && and clear cached data after those commands.

```
RUN apt-get update \
    && apt-get install <app> \
    && rm -rf /var/lib/apt/lists/*
```

4. Dockerfile: Utilizing size by layers

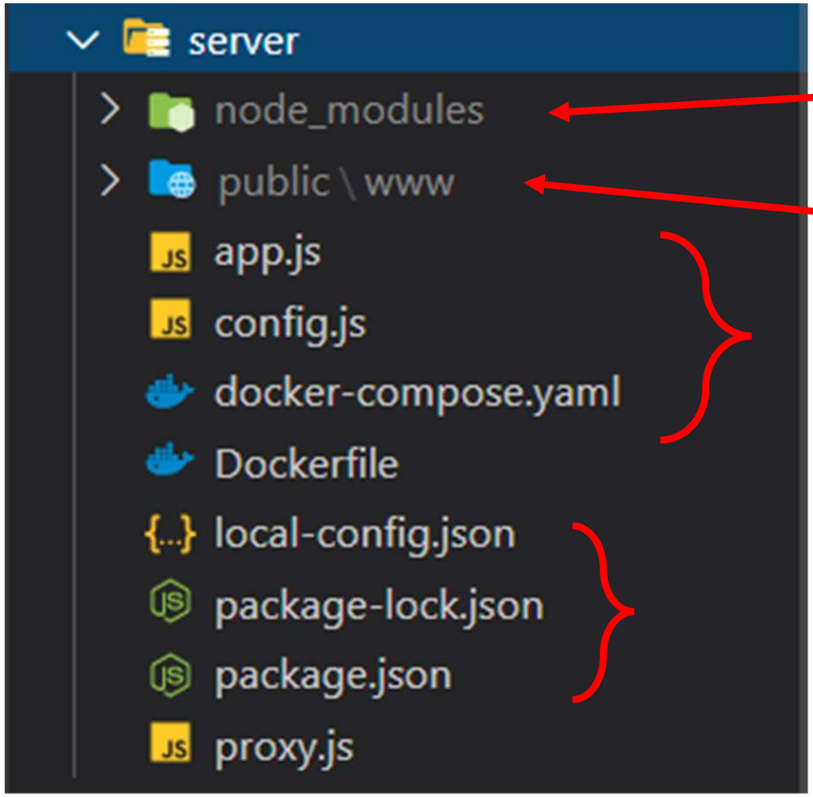
Pushing image

- Login into the registry with **`docker login`**
- Push image: **`docker push <image>:<tag>`**

Image cache

- Docker caches layers that remain unchanged (by computing hash value of output of the command). Only changed layers will be pushed.
- To clear cache, use argument **`--no-cache`** when building image
- Good to remember:
  - It is nice if we could arrange the commands reasonably to minimize the changes made to the image, so that minimize the request payload sent to Docker registry when pushing image.

4. **Dockerfile:** Utilizing size by layers



server
- node_modules
- public \ www
- app.js
- config.js
- docker-compose.yaml
- Dockerfile
- local-config.json
- package-lock.json
- package.json
- proxy.js

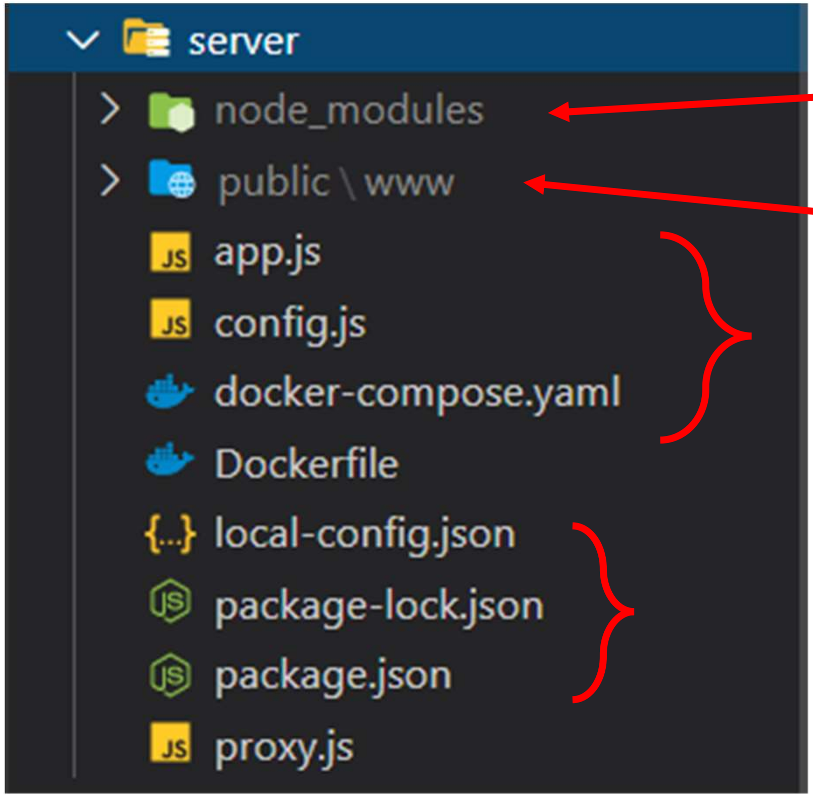
Libraries of ExpressJS

Static HTML files

ExpressJS source code

Manifest files

4. Dockerfile: Utilizing size by layers



server
- node_modules
- public \ www
- app.js
- config.js
- docker-compose.yaml
- Dockerfile
- local-config.json
- package-lock.json
- package.json
- proxy.js

1. Libraries of ExpressJS

3. Static HTML files

2. ExpressJS source code

1. Manifest files

4. **Dockerfile:** Utilizing size by layers

```
FROM node:alpine
WORKDIR /server
COPY ./*.json ./
COPY ./node_modules ./node_modules
COPY ./*.js ./
COPY ./public ./public
CMD ["node", "app.js"]
```

```
FROM node:alpine
WORKDIR /server
COPY ./public ./public
COPY ./*.json ./
COPY ./node_modules ./node_modules
COPY ./*.js ./
CMD ["node", "app.js"]
```

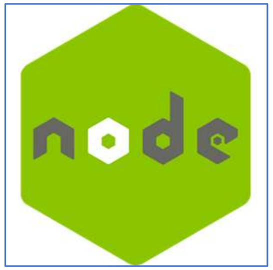
Which one is better?

2 Docker compose

Agenda

1. Docker
2. **Docker compose**
    1. *Multiple containers: configurations and communication*
    2. *docker-compose.yaml*
    3. *Build image easily with compose*
    4. *Run container easily with compose*
3. Docker Swarm

1. Multiple containers



Frontend
port: 80
envs: [...]

Backend
port: 8080
envs: [...]

Database
port: 5432
volume: /usr/data
envs: [...]

1. docker-compose

docker-compose.yaml

Frontend
port: 80
envs: [...]

Backend
port: 8080
envs: [...]

Database
port: 5432
volume: /usr/data
envs: [...]

comment

```
# This is comment
```

object: yaml

```
title: Vừa Nhắm Mắt Vừa Mở Cửa Sổ
author:
  name: Nguyễn Ngọc Thuần
  birthYear: 1972
```

object: json

```
{
  title: “Vừa Nhắm Mắt Vừa Mở Cửa Sổ”,
  author: {
    name: “Nguyễn Ngọc Thuần”
    birthYear: 1972
  }
}
```

array: yaml

```
publishers:
  - name: NXB Trẻ
    year: 2004
  - name: NXB Văn học
    year: 2010
```

array: json

```
{
  publishers: [
    { name: “NXB Trẻ”, year: 2004 },
    { name: “NXB Văn học”, year: 2010 }
  ]
}
```

1. docker-compose.yaml

```
version: ‘3’
services:
  pg:
    image: postgres:9.6-alpine
    ports:
      - 5432:5432
    volumes:
      - pgdata:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: postgres
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres

  frontend:
    image: nambach/frontend:latest
    build:
      context: .

volumes:
  pgdata:
```

2. Build image easily with compose

Place docker-compose.yaml in the same folder of Dockerfile and build.

Syntax

```
docker-compose build <service_name>
```

Example

```
docker-compose build frontend
```

```
version: ‘3’
services:
  frontend:
    image: nambach/frontend:latest
    build:
      context: .
```

3. Run container easily with compose

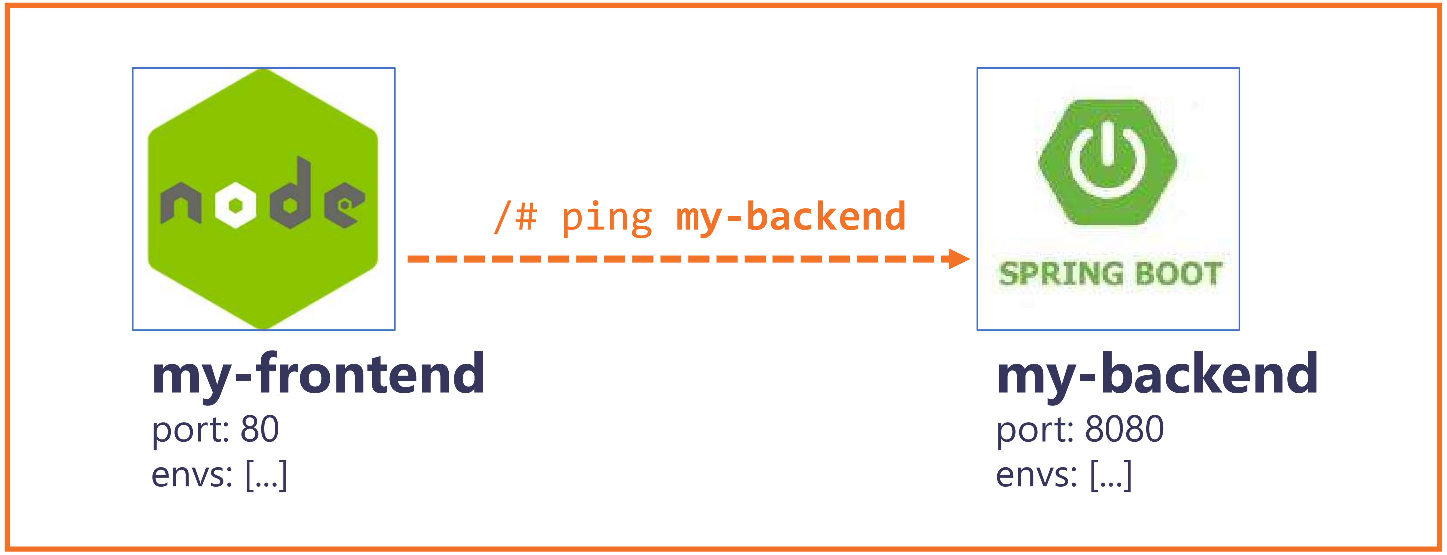
Syntax

```
docker-compose up
docker-compose up <service_name>
docker-compose up -d <service_name>
docker-compose logs -f <service_name>
docker-compose stop <service_name>
docker-compose down
```

4. Network

When attaching containers into same network, thanks to the built-in network resolution, Docker can resolve service’s name into its actual IP. So that containers can now communicate with each other through service names instead of IPs.

Docker network



`/# ping my-backend`

my-frontend
port: 80
envs: [...]

my-backend
port: 8080
envs: [...]

Demo

Syntax

```
docker-compose up
docker-compose up <service_name>
docker-compose up -d <service_name>
docker-compose logs -f <service_name>
docker-compose stop <service_name>
docker-compose down
```

Agenda

1. Docker
2. **Docker compose**
3. Docker Swarm

THANK YOU