Docker

Quang Hoang - 2024

Viettel Digital Talent 2024 - Software & Data Engineering

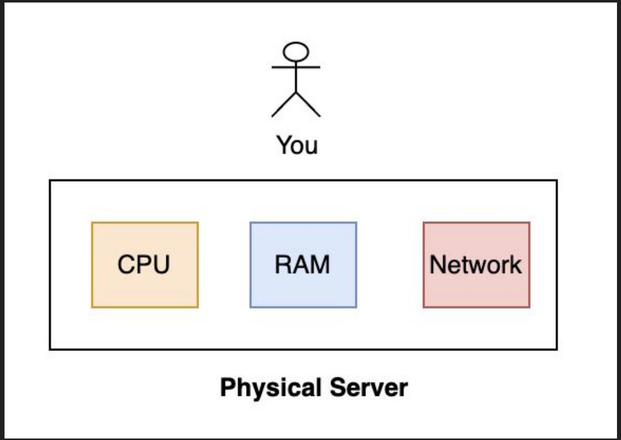
Prerequisite

- Install Docker

Lesson Outline

1. **Docker under the hood**
    - **Virtualization, Hypervisor**
    - **Containerization**
    - **Linux Namespace, Cgroups**
    - **Docker**
    - **Quiz**
2. Practical Docker

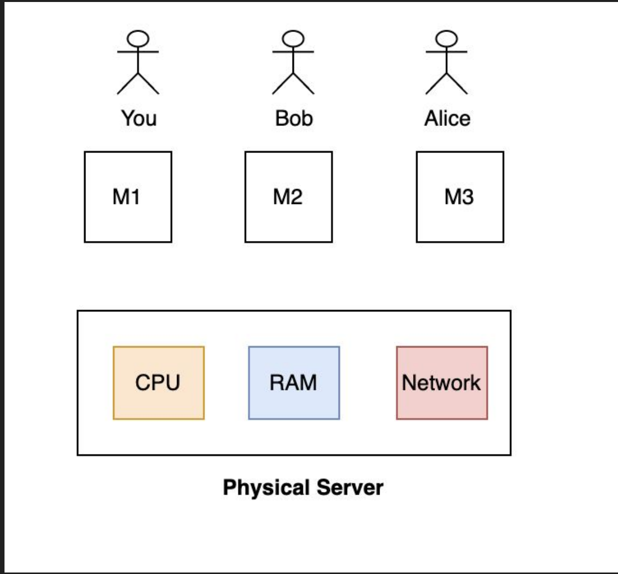
Virtualization



Virtualization

Process of creating a software-based version of your physical server.

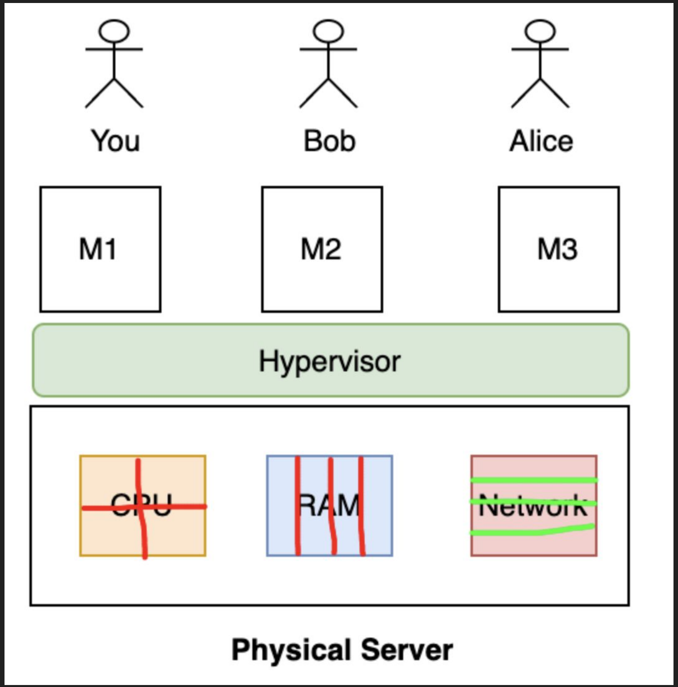
Very old tech (1960s), but still relevant today



Hypervisor

A software that runs on top of your physical server; it divides, isolates the physical resources (CPU, RAM…) and creates multiple virtual servers (VM).

VM = software-based computer

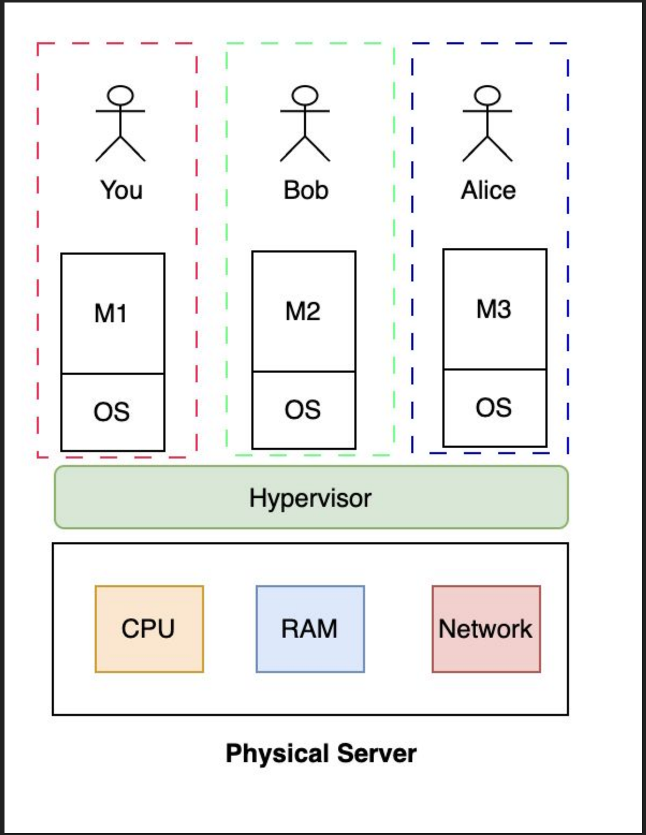


Hypervisor

Isolation:

You, Bob and Alice can’t access each other’s VM.

- Each VM can install it own OS.
- Portable: we can move VM from one hypervisor to another



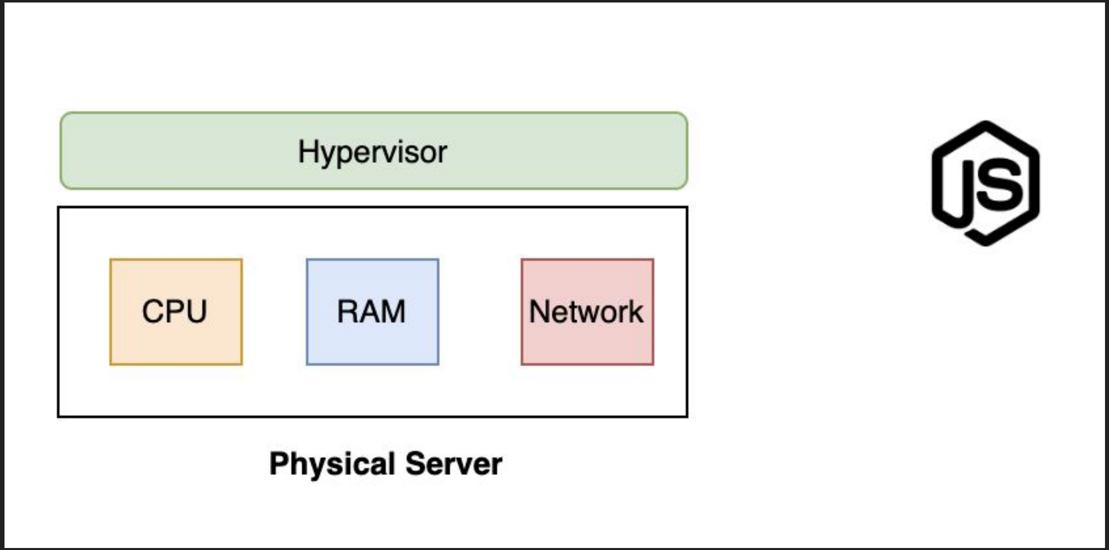
Virtualization

Benefits

- Cost saving
- Speed
- Reduce downtime

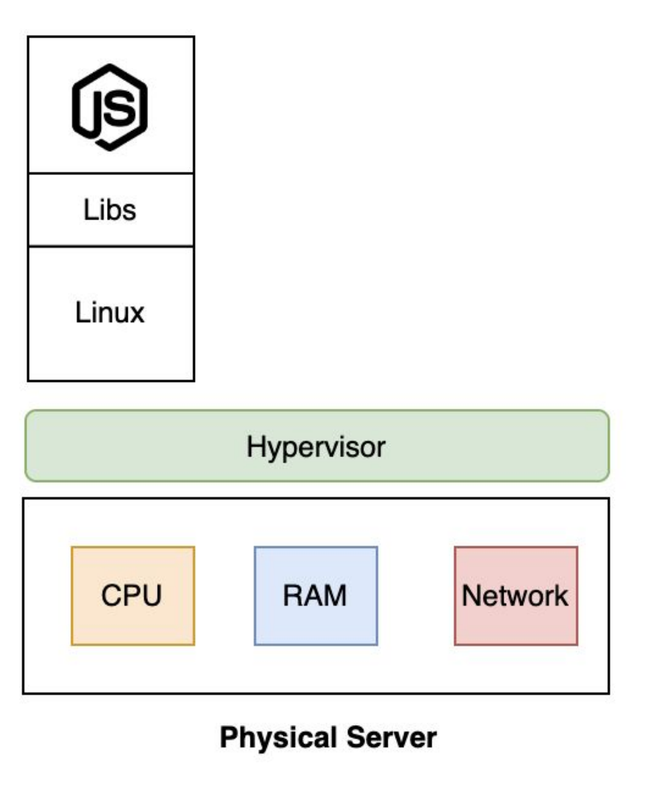
Virtualization

Limitations



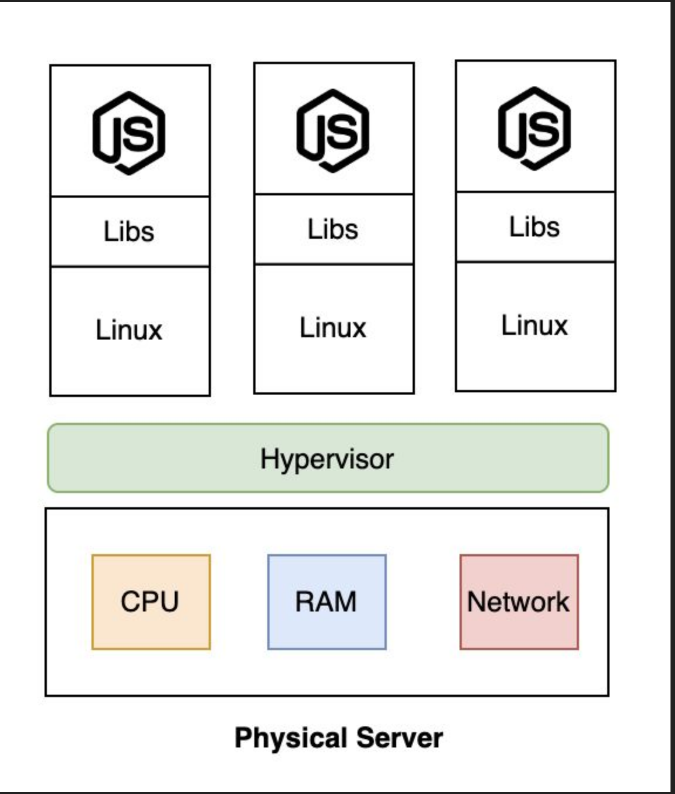
Virtualization

Limitations



Virtualization

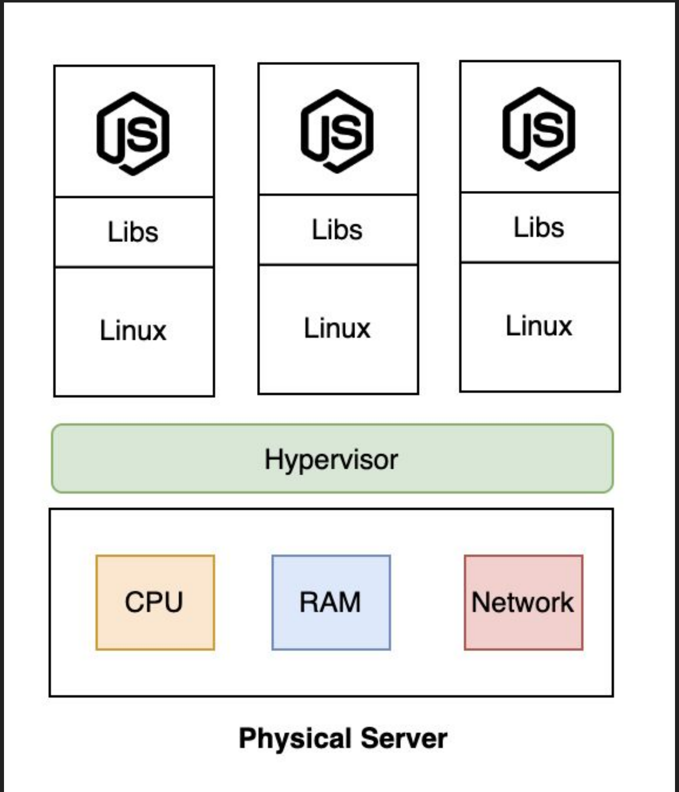
Limitations



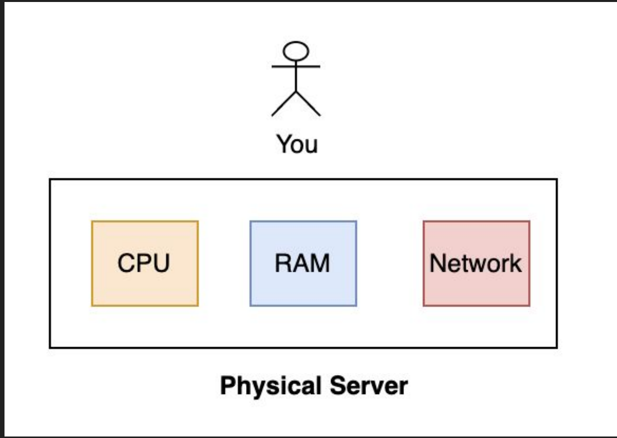
Virtualization

Limitations

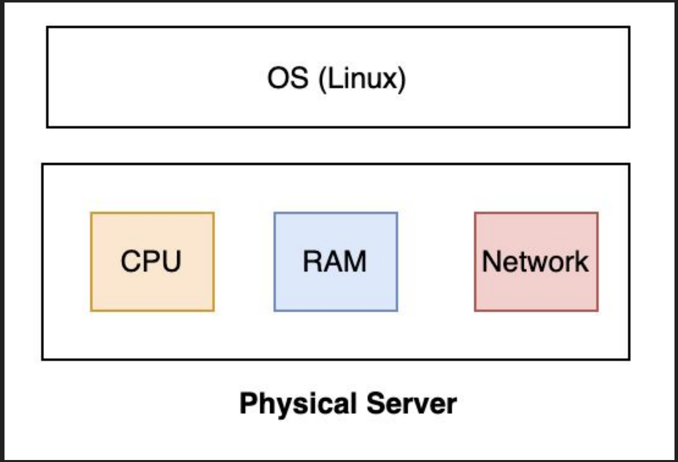
- resource-intensive
- slow start up (faster than set up a physical server but still not fast enough)
- not easy to manage (install related libs)



Containerization



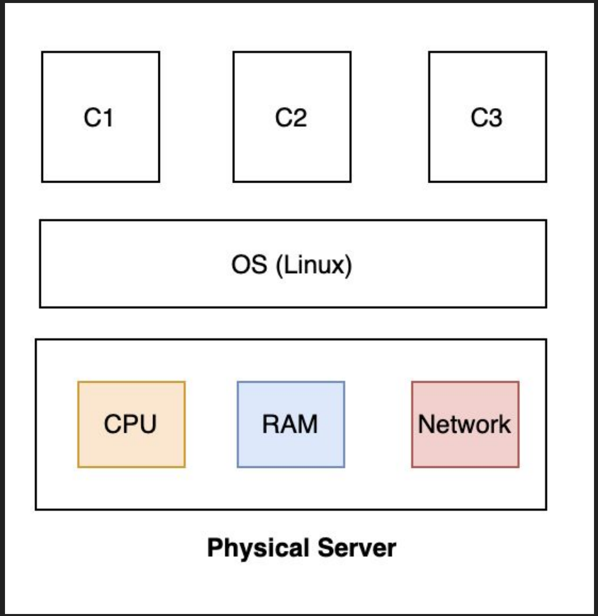
Containerization



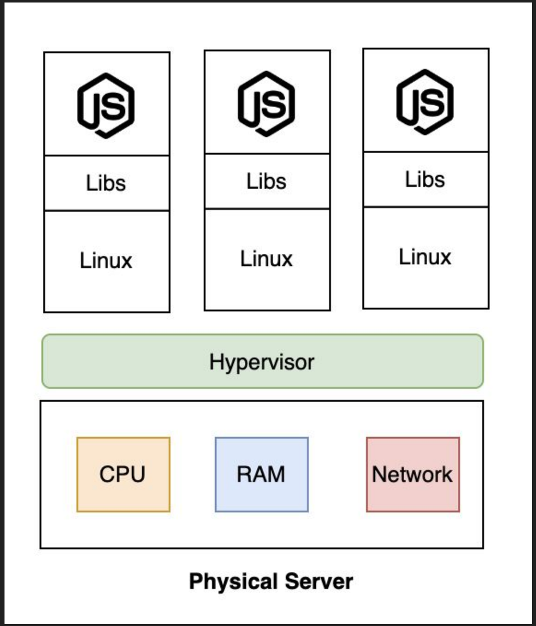
Containerization

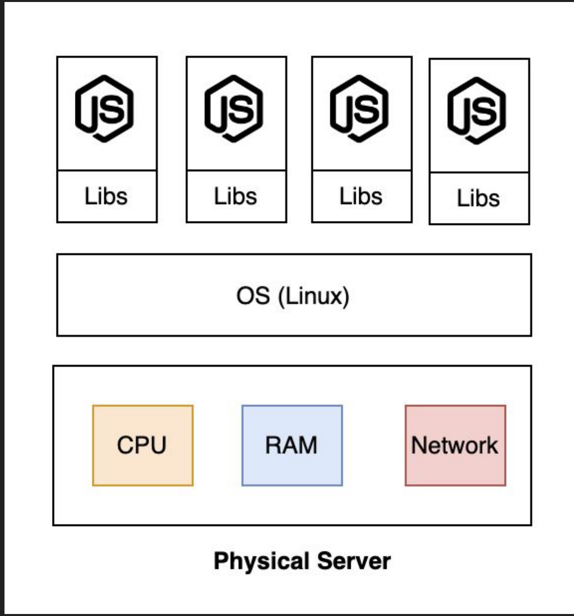
The resource isolation happens at

Operating System level.



Containerization





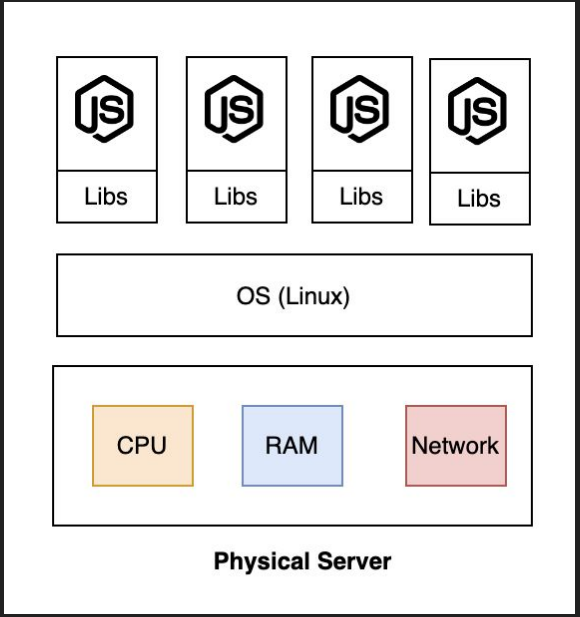
Recap - Q&A

- Virtualization is process of creating a software-based version of your physical server.
    - Hypervisor A software that runs on top of your physical server, device, isolate the physical resources (CPU, RAM…) and create multiple virtual servers (VM)
- Containerization is the packaging of software code with just the operating system libraries and dependencies required to run it.
- Hypervisor virtualize physical hardware, Containerization virtualize OS (typically Linux).

Containerization

Hypervisor allows us to virtualize hardware at machine level.

Which technologies allows us to isolate hardware resource at OS level?

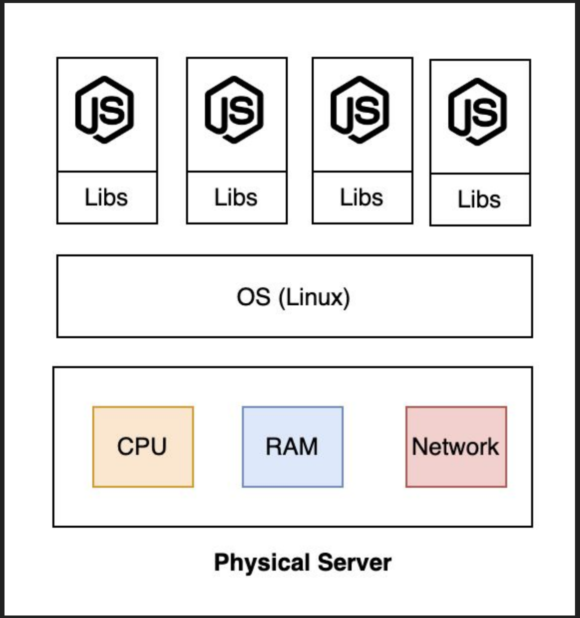


Containerization

Hypervisor allows us to virtualize hardware at machine level.

Which technology allows us to isolate and allocate hardware resource at OS level?

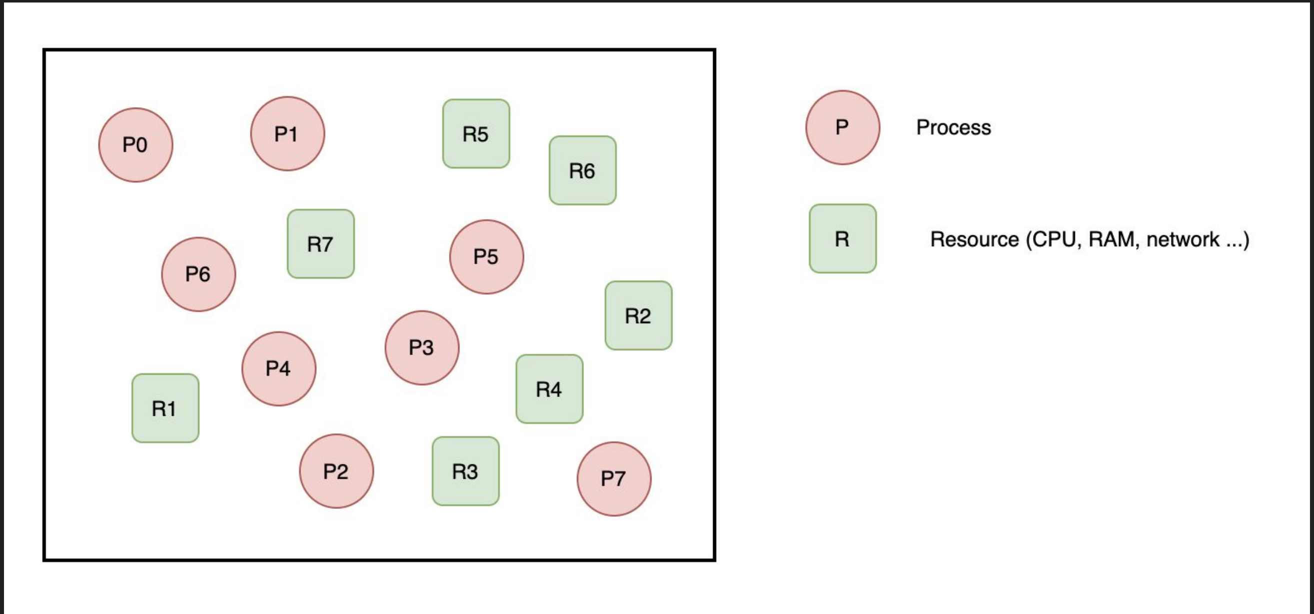
- Linux namespace
- Linux cgroups (2006)



Linux Namespace

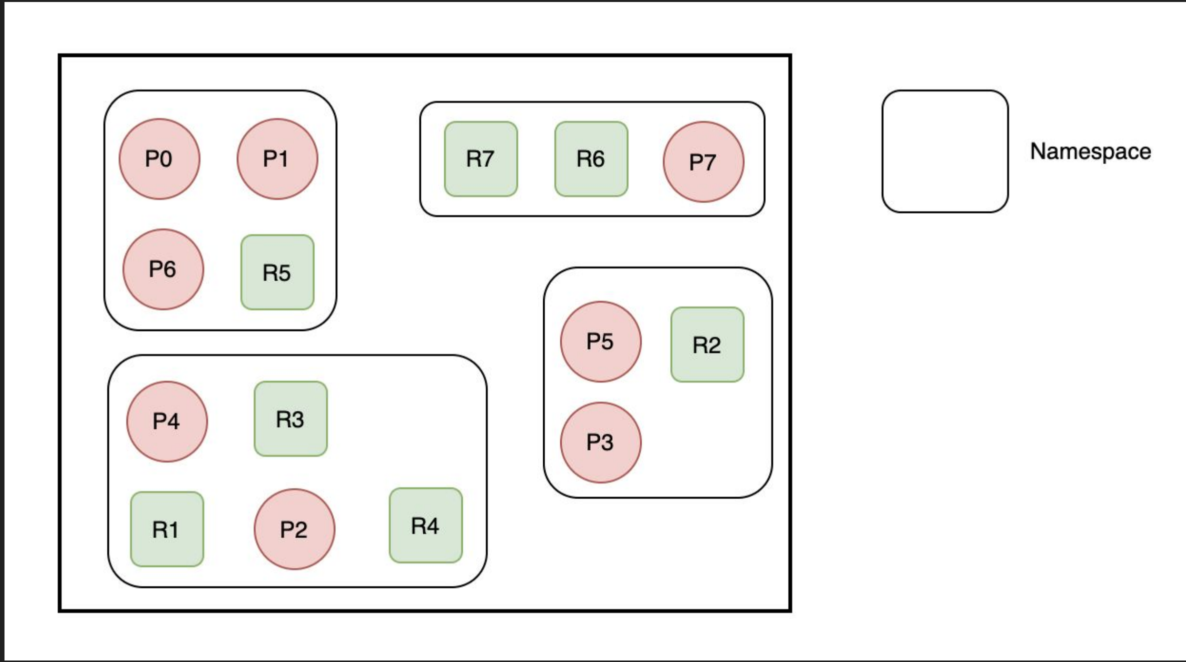
Resources Linux Kernel manages:

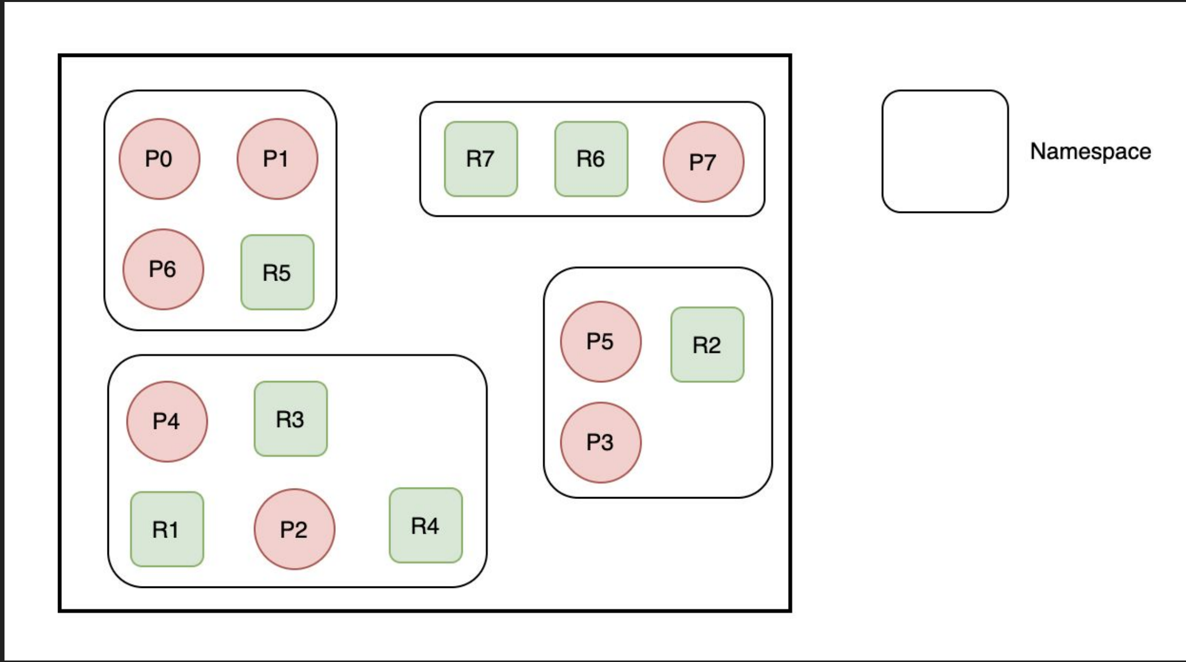
- network
- file system
- process
- time and clock…



Linux Namespace

Namespace (NS) is a feature of Linux Kernel
that wraps around a global system resource
in an abstraction that makes it appear to the
processes within the namespace that they have their
own isolated instance of the global resource.


P0
P1
P6
R5
R7
R6
P7
P5
R2
P3
P4
R3
R1
P2
R4
Namespace


P0
P1
P6
R5
R7
R6
P7
P5
R2
P3
P4
R3
R1
P2
R4
Namespace


Mount
User
PID
Network
Time
IPC

Linux Namespace

Types

1. IPC (Interprocess Communication Namespace)

Isolate IPC so processes cannot accidentally access/ destroy others.

2. Mount

Isolate filesystem hierarchy, make your container look like it has its own entire filesystem.

3. Network

Isolate network device, IP address, port numbers …

Linux Namespace

Types

4. PID

Isolate processes by assigning different process ID numbers.

5. Time

Allow each container to set their own date/time. (run uptime command)

6. User

Isolate user and group ID number from each other. e.g. UID 0 (root) in a user namespace is not the same thing as UID 0 on the host.

Linux Namespace

Types

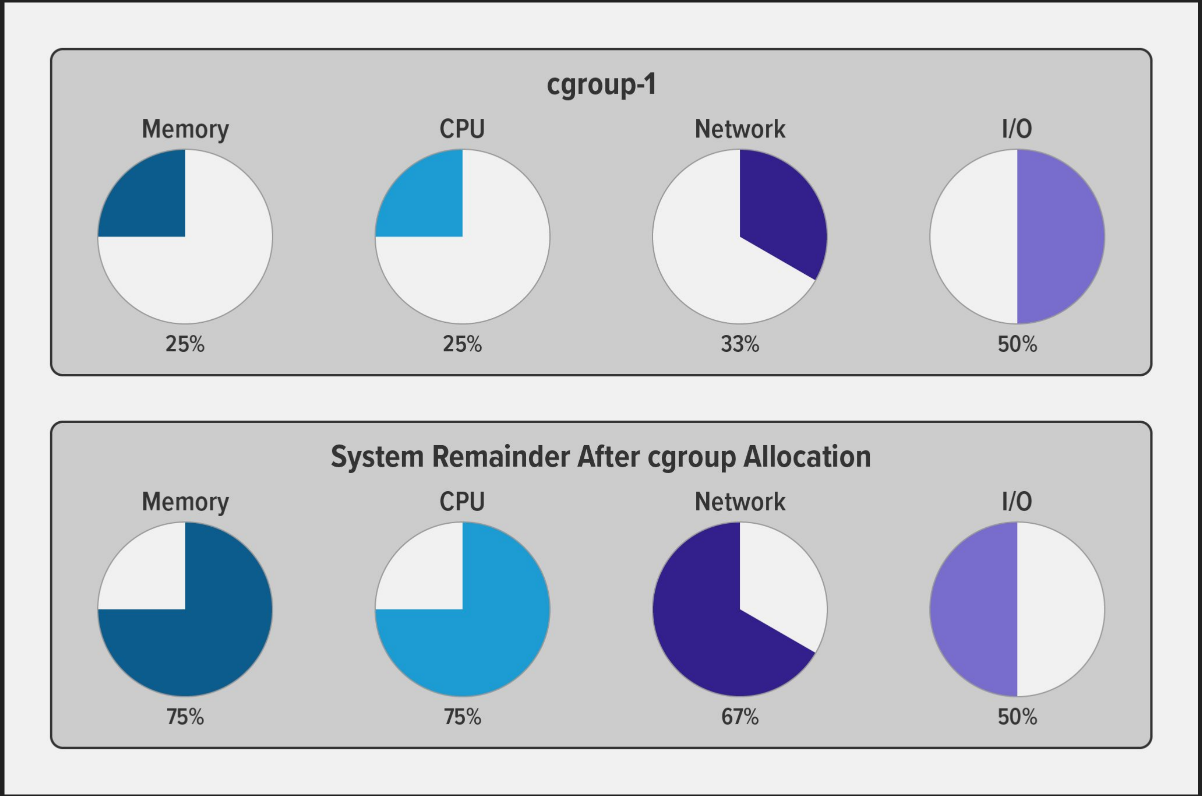
And some other namespaces …

Cgroups

Cgroups (Control groups) is a Linux kernel feature that limits, monitors, and isolates the resource usage (CPU, memory, disk I/O, network, and so on) of a collection of processes. Control groups allow you to set limits on resources for processes and their children.

- resource limit
- prioritization
- monitor
- control

Cgroups



cgroup-1
Memory
CPU
Network
I/O
25%
25%
33%
50%
System Remainder After cgroup Allocation
Memory
CPU
Network
I/O
75%
75%
67%
50%

Linux Container

Fact: with Cgroups and Namespace you can build your own containerization technology. (e.g)

(Try it yourself ?)

Docker

- The first containerization tool.
- First release: 2013

Docker

- The first containerization tool.
- First release: 2013
- Benefits
  - lightweight: no hypervisor, no duplicated OS.
  - isolation from the main system: developer can install multiple version of a software on the same machine.
  - easy packaging software process: application + related dependencies in a single standard **image** format.
  - portable: no need to recompile or repackage. Docker promises an equivalent environment in both development and production

Life before Docker

1. Time-consuming migration: As soon as the software is migrated to the new environment, managers, developers, and the system administrators used to start hunting the bugs produced because of a new environment.

Life before Docker

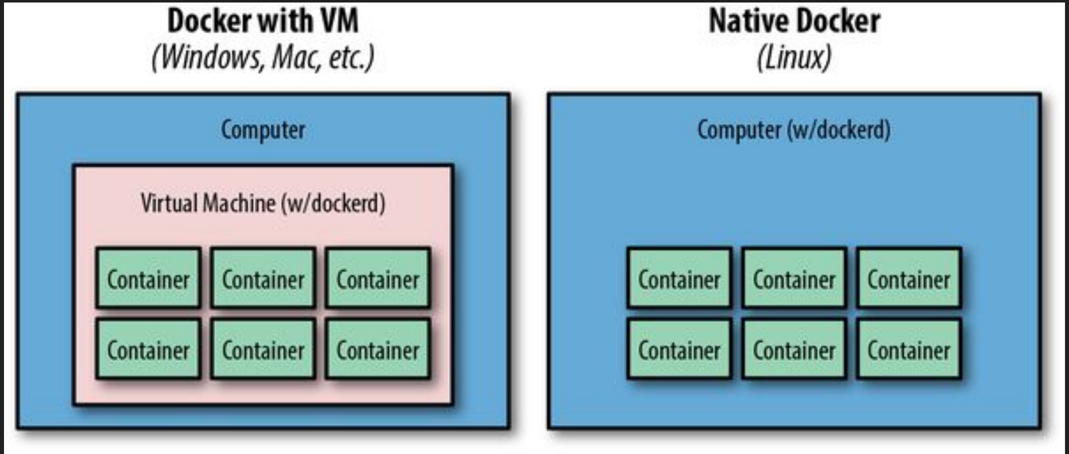
2. "It works on my machine!"

This was the biggest problem whenever a new developer joined the team and needed the project to be set up on his system.

How containers work on non-Linux OS?

How containers work on non-Linux OS?

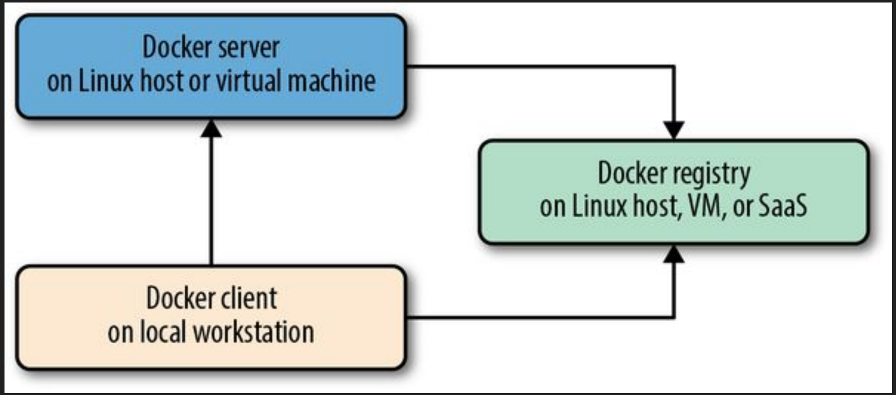
- Docker Desktop for Windows uses the hypervisor technology of Windows (Hyper-V) to create a lightweight Linux VM.
- Docker for MacOS uses Apple Hypervisor Framework to create a lightweight Linux VM.



What Docker is not?

- Not a virtualization platform (e.g. VMware, KVM…)
- Not a cloud platform (e.g. OpenStack, CloudStack…)
  - both allows horizontally scale, but
  - It only handles deploying, running, and managing containers on pre-existing Docker hosts.
  - It doesn't allow you to create new host systems (instances), object stores, block storage.
- Configuration management (e.g. Puppet, Chef…)
- Workload management tool (e.g. K8S, Mesos…)
  - coordinate work across a pool of Linux container hosts

Docker components



1. Docker client
   the primary way that Docker users interact with Docker. (command or API)
2. Docker server (daemon)
   listens for Docker API requests and manages Docker objects.
3. Docker image (OCI image)
   a read-only template with instructions for creating a Docker container
4. Container
   a runnable instance of an image

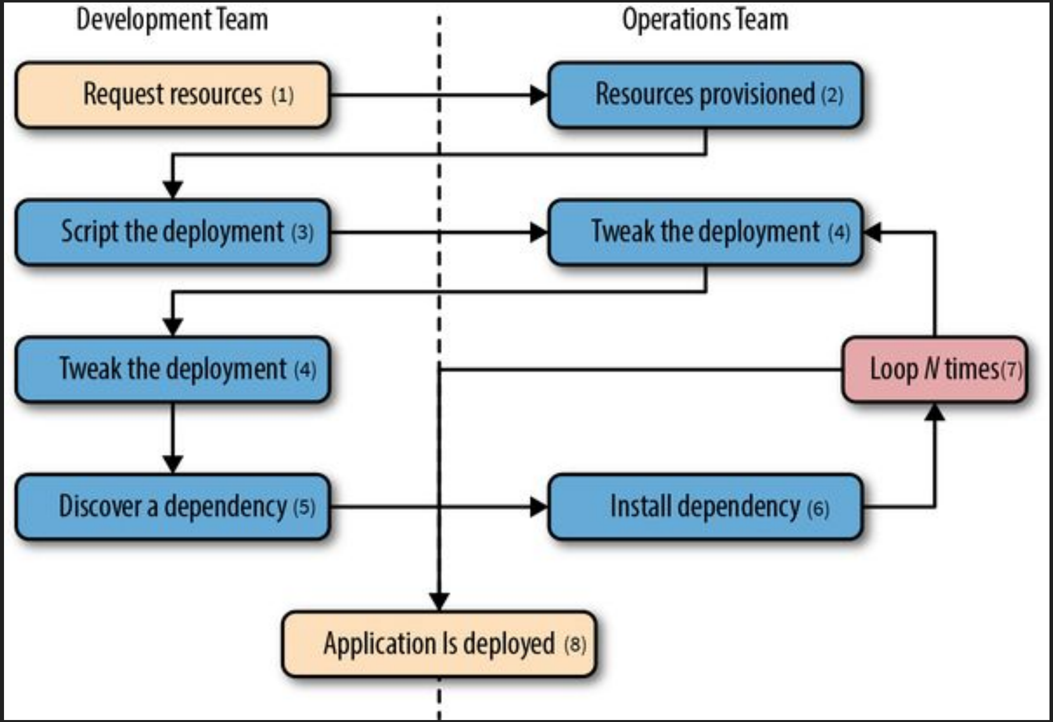
Docker components

5. Docker registries

Docker images storage.

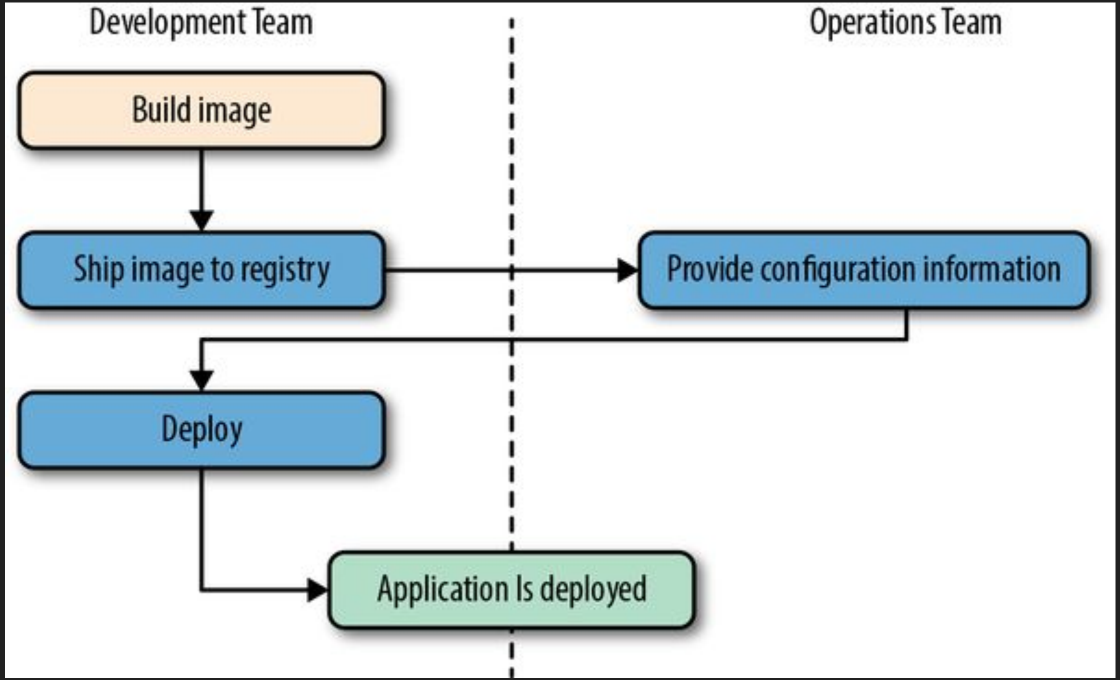
Docker in Software Development Cycle

Before Docker



Docker in Software Development Cycle

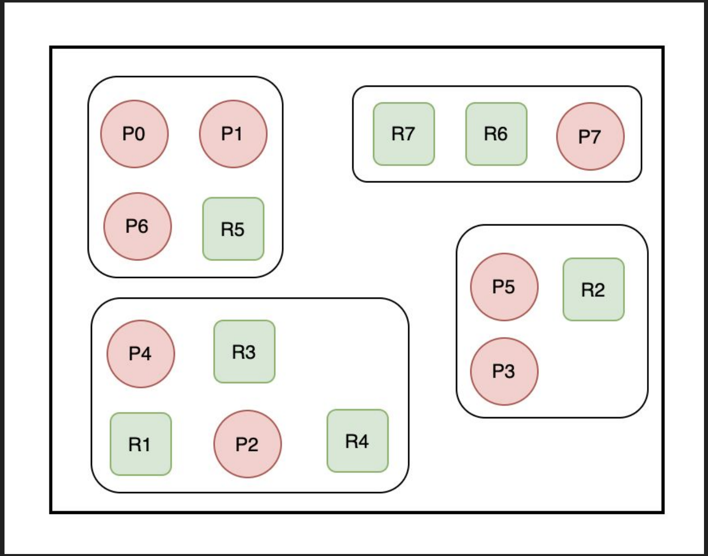
After Docker



Docker Image

A read-only template with instructions for creating a Docker container.

If Docker container is a cell, docker image is the DNA.



Docker Image

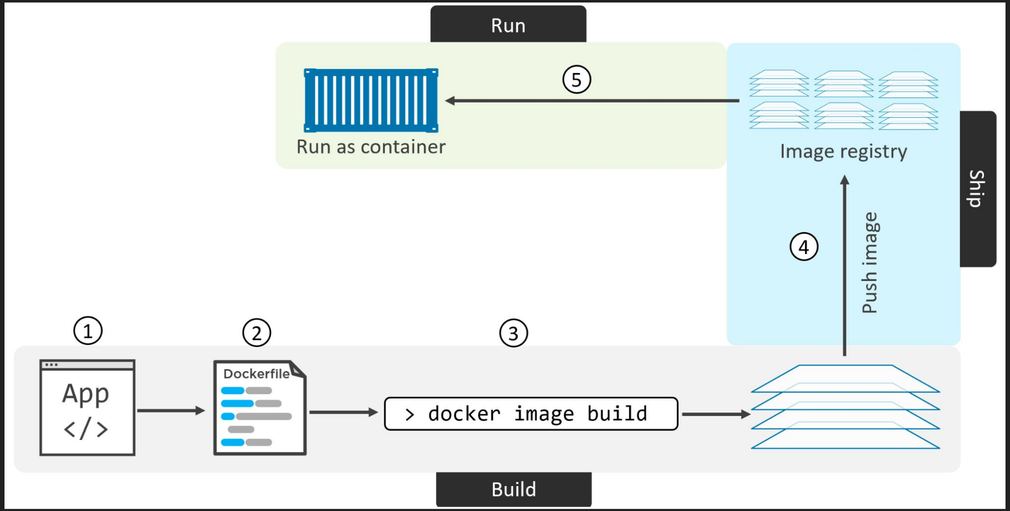
To launch a container, we need to either:

- download a public image from Docker Hub.
- or create our own image.

Docker Image is represented by a Dockerfile

Containerizing an app process

1. Start with your application code and dependencies
2. Create a Dockerfile that describes your app, its dependencies, and how to run it
3. Build the Image based on Dockerfile
4. Push the new image to a registry (optional)
5. Run container from the image



Containerizing an app

Let’s consider a very simple Nodejs application

```javascript
const express = require('express');

// Constants
const PORT = process.env.PORT || 3000;
const HOST = '0.0.0.0';

// App
const app = express();
app.get('/', (req, res) => {
    res.send('Hello World, My name is ' + process.env.NAME);
});

app.listen(PORT, HOST, () => {
    console.log(`Running on http://${HOST}:${PORT}`);
});
```

```json
{
  "name": "docker_web_app",
  "version": "1.0.0",
  "description": "Node.js on Docker",
  "author": "First Last <first.last@example.com>",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.16.1"
  }
}
```

```dockerfile
# instead of building a node instance from scratch
# we often build on top of an existing official image
# on Docker Hub. Here we use node version 16
FROM node:16

# set variables and their default values,
# which are only available during the image build process
ARG email="quanghd.95vn@gmail.com"

# Docker runs all processes as root by default
# your can change it to other user.
RUN useradd -ms /bin/bash quangh
USER quangh

# set shell variables that can be used by your running application
ENV PORT 8080
ENV NAME Quang

# set the working directory of container
WORKDIR /usr/src/app

# copy file package.json from local to
# container
COPY package.json ./

# run a command
RUN echo $email
# install dependencies of nodejs app
RUN npm install

# copy source code
COPY . .

# documentation the port that our webapp will run
EXPOSE 8080

# defines the command that launches the process that
# you want to run within the container. if you define multiple
# CMD, only the last one takes effect.
CMD [ "node", "server.js" ]
```

Dockerfile syntax

`FROM <image_name:version>` build image on top of existing image from [Docker Hub]

`ARG <variable_name>=<value>` define a variable to use during build image process

`ENV <variable_name> <value>` define a variable to use in running process inside container

`RUN <command>` run a Linux command

`USER <username>` define the user will run the process inside container, default is root

`COPY <local path> <container path>` copy files from local to container

`CMD [<command>]` define the command that launch the process within container

Build Image from Dockerfile

```
docker image build -t <image-name> <path>
```

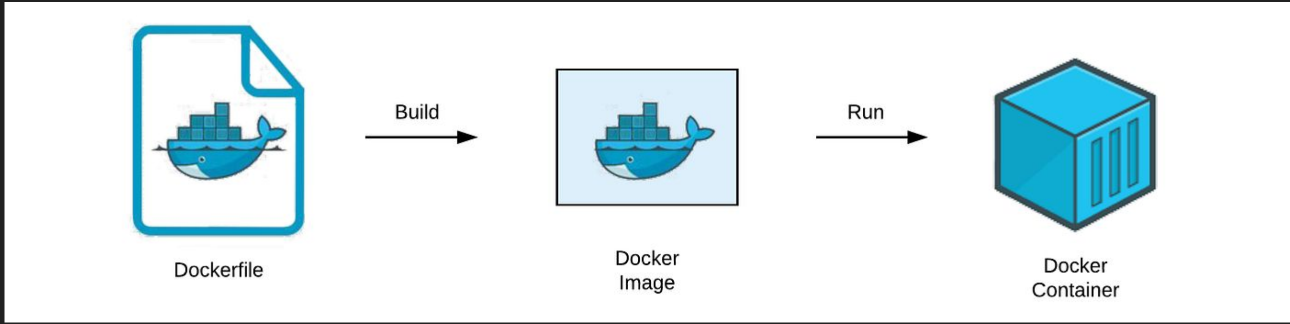
e.g:

```
docker image build -t quang/my-nodejs-app:latest .
```

Docker will look for a Dockerfile in the current folder and build an image.

The build process is cached.

```
docker image ls
```


Build
Run
Dockerfile
Docker
Image
Docker
Container

Run the Image

`docker container run <options> <image-name>`

e.g

`docker container run --rm -d -p 3000:8080 quang/my-nodejs-app:latest`

`-d` run the container in background

`-p` map the port of container to the port of host

`-it` take you inside the container

`-rm` Automatically remove the container when it exits

Demo

Access a running container

```
docker exec -it <container-name> <command>
```

Docker container - Storage

All data in container will be deleted when container exits.

e.g.

```
docker container run --name mysql -d \
    -e MYSQL_ROOT_PASSWORD=change-me \
    -p 3306:3306 \
    mysql:8

docker stop mysql
docker rm mysql
```

Docker container - Storage

All data in container will be deleted when container exits.

In many case, we need data to persist, e.g. database.

Docker run command has a flag for this: `-v target:source`

e.g.

```
docker container run -d \
    -e MYSQL_ROOT_PASSWORD=change-me
    -p 3306:3306 \
    -v mysql:/var/lib/mysql \
    mysql:8
```

`/var/lib/mysql` in container will be mapped to `mysql` folder in host.

Resource quota

we can limit the resource of a container (using what ?)

- CPU
- memory

Resource quota

CPU shares are relative

- If only one container is active it can use all the CPU
- When there is a contention, a container configured for 1024 shares of a cpu will get twice as much cpu time as a container that requested 512 cpu shares.

CPU quota is absolute

- If you set cpu=1, even if there is no contention, the container only utilized 1 core.

Resource quota

1. CPU

100% ~ 1024

50% ~ 512

```
docker container run --cpu-shares 512 --rm -d -p 3000:8080 quang/my-nodejs-app:latest
```

```
docker container run --cpus="0.5" --rm -d -p 3000:8080 quang/my-nodejs-app:latest
```

Resource quota

2. Memory

`--memory`

`b, k, m, g` representing bytes, kilobytes, megabytes, or gigabytes.

```
docker container run --memory 512m --rm -d -p 3000:8080
quang/my-nodejs-app:latest
```

Docker Network

By default, containers run in isolation and don’t know anything about other processor/ containers on the same machine.

To allow a container to talk to another, we need to create a network for them.

2 containers can talk to each other if and only if they are on the same network.

Docker Network

create a network:

```
docker network create my-network
```

attach a container to a network

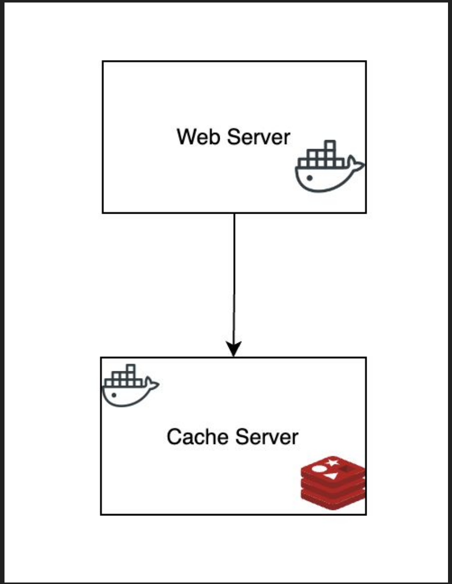
```
docker run --network my-network …
```

Docker Compose

- Distributed Monolith
  - web frontend
  - web backend
  - database
  - cache
- Microservices
  - E-commerce app:
    - Order
    - Product
    - Account
    - Payment
    - Chat
- Docker compose is a tool, built on top of Docker, to manage multi-container Docker apps.

Docker Compose

- use a yaml file to define all the containers.
    - default name: docker-compose.yml

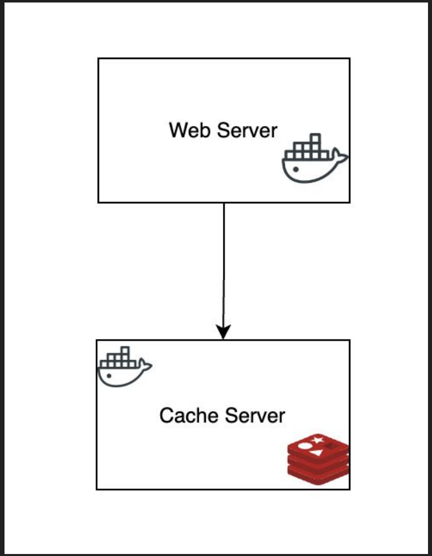


Source

```yaml
version: "3.8"
services:
  web-fe:
    build: .
    command: python app.py
    ports:
      - target: 5000
        published: 5000
    networks:
      - counter-net
    volumes:
      - type: volume
        source: counter-vol
        target: /code
  redis:
    image: "redis:alpine"
    networks:
      counter-net:
```

```yaml
networks:
  counter-net:

volumes:
  counter-vol:
```



Source

`version` mandatory, version of the Compose file.

`services` define the different application, each application is one container.

    `build .` build a new image using the instructions in the Dockerfile in the current directory (.)

    `command: python app.py` run a Python app called app.py as the main app in container (technically, this is optional as we defined CMD in Dockerfile)

    `ports:` map port 5000 inside the container (-target) to port 5000 on the host

    `networks:` attach the service's container to a network

    `volumes:` mount the counter-vol volume (source:) to /code (target:) inside the container

`networks` define a network for containers to communicate with each other

`volumes` define new volumes.

Docker Compose

`cd` to the directory containing the docker-compose.yml file.

```
docker-compose up -d
```

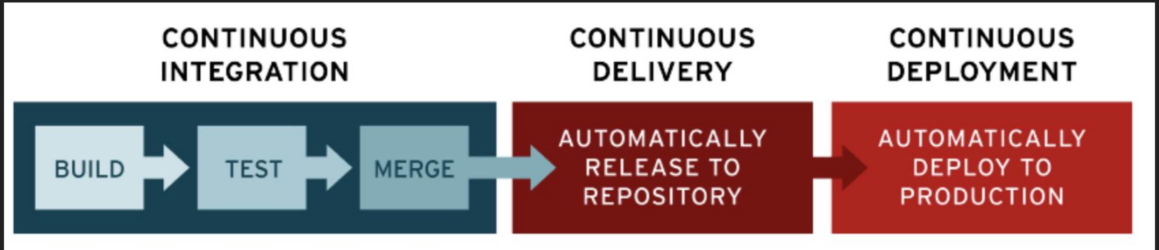
```
docker-compose ps
```

```
docker-compose down
```

Docker and CI/CD

CI/CD is a method to frequently deliver apps to customers by introducing automation into the stages of app development.

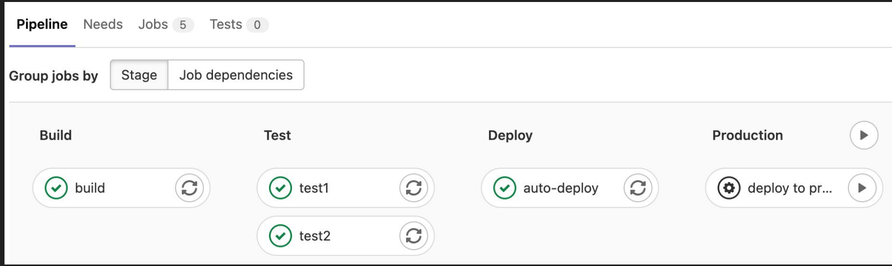
- CI ~ continuous integration: new code changes to an app are regularly built, tested, and merged to a shared repository automatically.
- CD
  - Continuous delivery: applications are automatically packaged and uploaded to a repository (e.g. Docker Hub)
  - Continuous deployment: automatically releasing a developer's changes from the repository to production

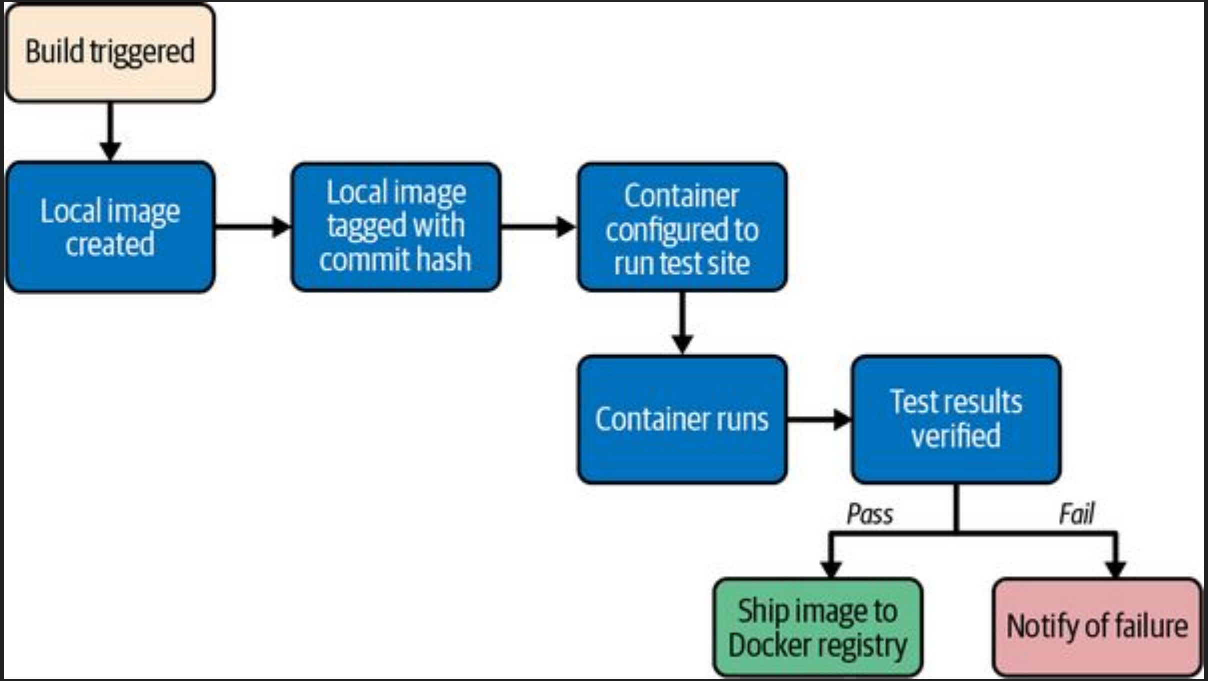


Docker and CI/CD

CI/CD workflow with Docker:

1. A build is triggered by some means (e.g. manual trigger by a developer, a webhook call from a source code repository)
2. Build server kicks off a container image build. the image is created on the build server.
3. A new container, based on the newly built image, is configured to run the test suite.
4. The test suite is run against the container, and the result is captured by the build server.
5. The build is marked as passing or failing.
6. Passed builds are shipped to an image registry or other storage mechanism.




Build triggered
Local image created
Local image tagged with commit hash
Container configured to run test site
Container runs
Test results verified
Pass
Fail
Ship image to Docker registry
Notify of failure

Challenge

- create a simple hello-world webapp (using NodeJS or any web technology you know).
- upload source code to Gitlab.
- configure Gitlab CI such that:
    - commit code, push and merge to master branch => automatically build, run unit test, upload image to Docker hub.
    - optional: Gitlab CI automatically deploy the application to a VM (e.g. Google Cloud, AWS, Digital Ocean).

There are a lot of tutorial on the internet (e.g)

Quiz Time