

Linux containers and the Docker environment

Hélène Coullon, Lucas Gazeau

IMT Atlantique, FIL A2 & TAF LOGIN FISE A3

Table of contents

1. Introduction
2. What is a container?
3. What is under the hood?
4. Docker
5. Create an image with a Dockerfile
6. Volumes
7. A few good practices
8. Deploying a software stack with Docker Compose

Introduction

Monoliths vs micro-services

Applications designed as big monoliths

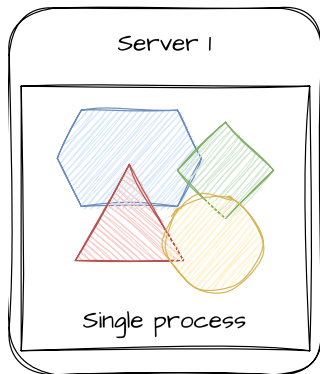
- slow release cycles
- updated infrequently
- lack of flexibility

Micro-services architectures

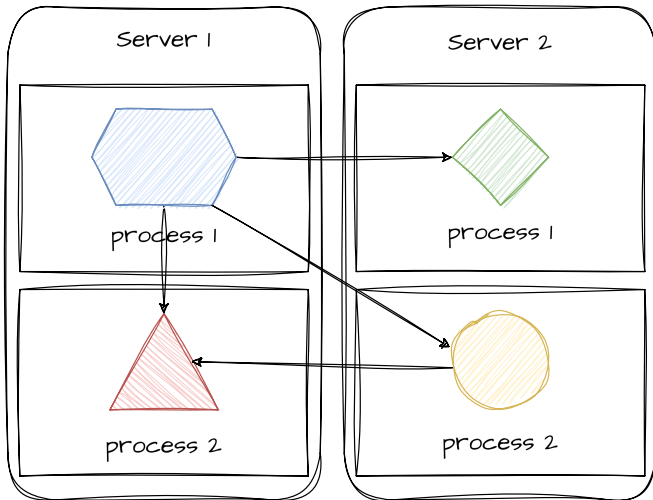
- smaller, independently running components
- decoupled from each other
- short and independent release cycles
 - development
 - deployment
 - update
 - scale

Monoliths vs micro-services

Monolithic app



microservices-based app



Problems of micro-services architectures

With bigger numbers of micro-services and increasingly complex data centers to deploy them

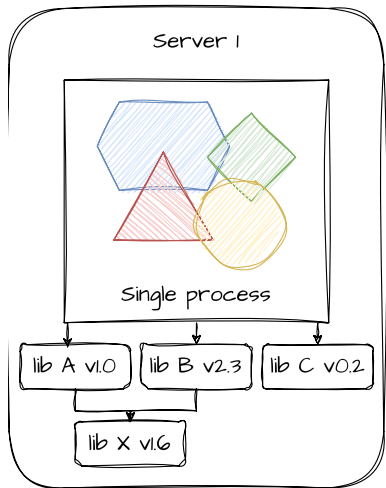
- difficult to correctly configure and deploy the overall system
- difficult to manage the lifecycle of microservices
- difficult to keep the overall system running

Need for automation and orchestration

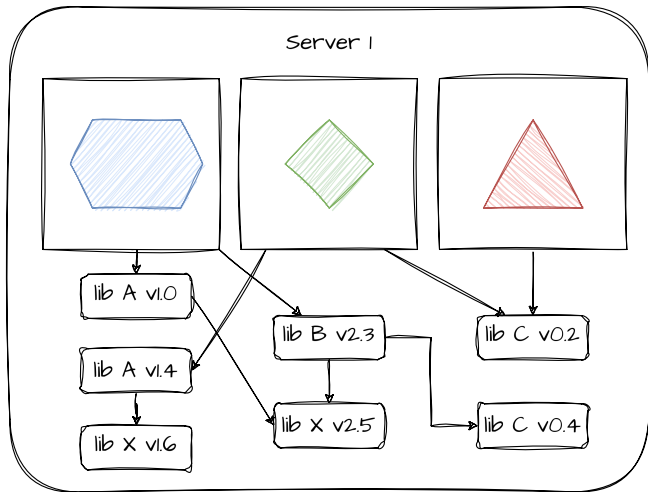
- automatic configuration and deployment (solved with containers and Docker)
- automatic scheduling of micro-services on servers
- automatic supervision and fault-tolerance

Configuration issue example

Monolithic app



Microservices-based app



Why not using VMs?

To solve this issue we could

- start as many VM as the number of services
- automate their configuration and the service deployment with Ansible or Bash

Advantages

- strong isolation
- portability
- better usage of the resources of a machine with co-hosted VMs

Disadvantages

- provision and configure each VM
- data duplication (libraries, kernel)
- performance cost

Containers

A container is a **light virtualization technique**

Container technologies

- Application containers: **Docker**, **podman**, **rkt**, **containerd**
- OS container: **LXC Linux**
- and others like **Singularity** for safe HPC containers



What is a container?

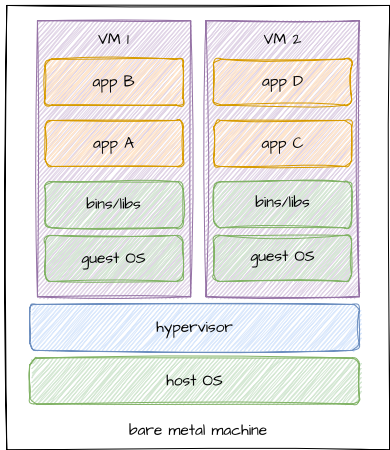
A bit about Linux kernel

The kernel is the core of the operating system ([DEVOS course](#))

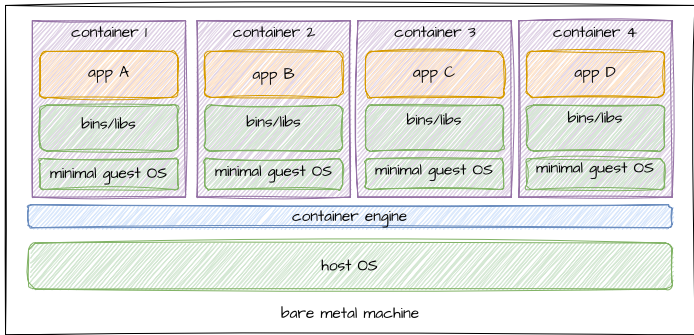
- it is the portion of the OS that is always loaded in memory
- it controls all hardware resources (e.g., I/O, memory, cryptography, CPU) via drivers
- it arbitrates conflicts and concurrency between processes
- it optimizes the utilization of resources (e.g., cache, memory, CPU, file systems, network)

The kernel is one of the first programs loaded on startup

Coarse-grain comparison between VMs and containers



VMs with hypervisor Type1



Containers

Advantages of containers

- isolation (*)
- portability (*)
- limitations of duplicated resources
- limited impact on performances
- fast startup

The model is different and the way applications are deployed is different!

What is under the hood?

A bit more about Linux

In Linux/Unix everything is a file: a file, a directory, a device etc.

The file system

- hierarchical organization of files
- `/` is **root of the file system**
- `/sys` contains system files
- `/etc` contains config files and scripts
- `/media` contains hard drives partitions, devices etc.
- etc.

An image and a container are also a set of files!

Images

A container image is an archive of **files** containing

- a root file system
- libraries, packages etc. (i.e., dependencies)
- the application or service to run

The image contains the required environment to run the application or the service on top of the host kernel.

This environment is **portable** from one host to another if a compatible kernel is present (**WSL on Windows!**)

Registry of images

- DockerHub *<https://hub.docker.com/>*
- your own registry can be deployed

Alpine example

The docker image of Linux Alpine is often used by containers

- it is a very light Linux distribution
- https://hub.docker.com/_/alpine

pull an alpine Docker image

```
> docker pull alpine
```

run a container by using the alpine image and start an interactive sh prompt in it

```
> docker run -it alpine /bin/sh
```

print the file system of the container

```
in alpine> ls -al
```

check there is not any kernel

```
in alpine> ls /boot
```

From an image to a container

As seen before an image is an archive of a file system, creating a container consists in

- giving a **limited amount of resources** to the container
- creating an isolated environment to the container process
- assigning the root file system of the image to the **root file system "/" of the container**

cgroups

Linux Control Groups (cgroups) limit the amount of resources a process can consume (CPU, memory, network bandwidth, and so on)

namespaces

Linux Namespaces make sure each process sees its own personal view of the system (files, processes, network interfaces, hostname, and so on)

chroot, pivot_root

Change the root filesystem of a process

Containers From Scratch by Liz Rice

She shows how to manually create containers using the three concepts presented above

Docker

Overview of Docker

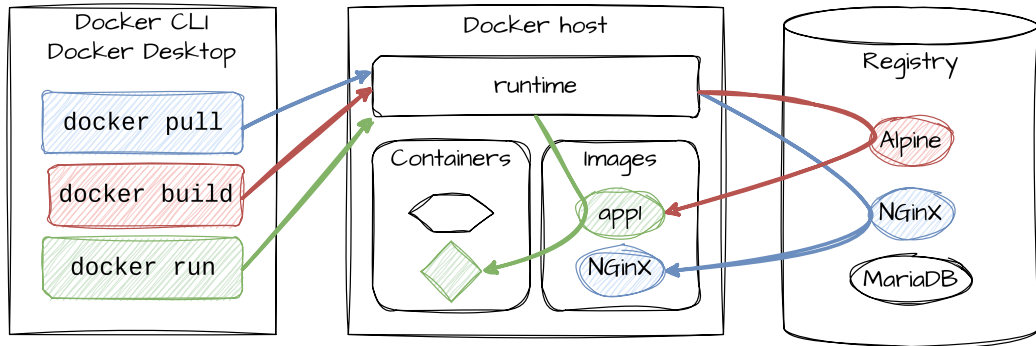
The different pieces involved in the process

- CLI (command line interface)
- Docker runtime
- images and registry
- containers

Docker runtime

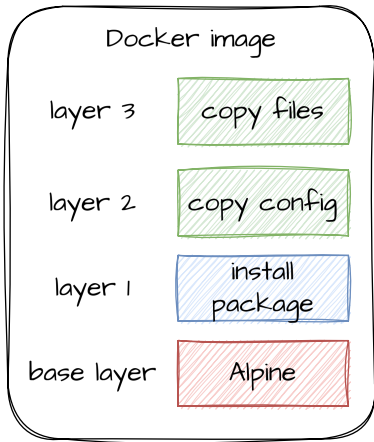
- Start and stop containers
- Manage images
- Manage networks
- Manage volumes
- etc.

Overview of Docker



Structure of Docker images

A Docker image is built by assembling different **layers**



Storage optimization

- layers are shared by different images to optimize storage
- to do that each layer is identified by a hash function according to its content

Writing in a container?

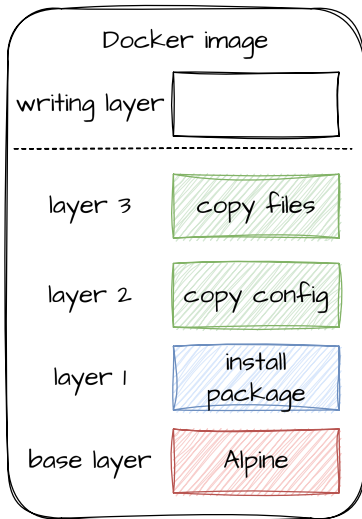
A Docker image is **immutable**!

At runtime, a virtual layer is created on top of the image

- it is possible to write in this layer
- this layer is not shared with other containers
- the layer is destroyed with the container

Volumes

If data has to be **persistently stored** and **shared** between containers, a **volume** has to be used



Nothing better than a tutorial to discover the CLI!

Create an image with a Dockerfile

A Dockerfile contains a set of commands to build a Docker image

- avoid building images manually
- offers a way for Docker to build layers and avoid useless commands
- a Dockerfile is close to a bash (or a set of Ansible tasks) with instructions to apply

Dockerfile through an example

The full documentation is at

[*https://docs.docker.com/engine/reference/builder/*](https://docs.docker.com/engine/reference/builder/)

- *FROM* to indicate the base image used to build our image
- *RUN* to execute a command on top of the base image
- *ENV* to declare some environment variables
- *ENTRYPOINT* the command to execute when starting the container

```
1 FROM alpine
2 RUN apt update
3 RUN apt install -y htop
4 ENV TERM=xterm
5 ENTRYPOINT /bin/htop
```

Another example

- *FROM* with an image version
- *WORKDIR* to indicate the working directory when starting the container
- *ADD* to add some files from the local machine to the container image
- *CMD* the command to execute when starting the container

```
1 FROM python:3.8-alpine
2 WORKDIR /app
3 ADD . /app/
4 RUN pip install -r requirements.txt
5 CMD ["python", "movie.py"]
```

Reference documentation

What are the differences between ADD and COPY? What are the differences between ENTRYPOINT and CMD?

Build an image with a Dockerfile

docker build [OPTIONS] PATH

> docker build . -t "monapp:latest"

- *docker build* is the command to build a docker image
- *.* is the path to find the Dockerfile
- *-t* is an option to give a name to the image
- by default the Dockerfile is *PATH/Dockerfile*, you can give another name and use the *-f* option

It is time write a Dockerfile

Volumes

Volumes

Two types of volumes

Host volumes

```
> docker run -v src-dir:dest-dir image_id  
> docker volume ls
```

Named volumes

```
> docker volume create nom_volume  
> docker run -v nom_volume:dest-dir image_id  
> docker volume ls
```

The **VOLUME** `[/app/logs]` instruction in a Dockerfile only creates the mounting point in the container. It works without it, but it is a good practice to identify easily the need for a volume.

It is time write a Dockerfile with a volume

A few good practices

Think about the layers

In the oldest versions of Docker, any line in the Dockerfile created a layer

- too many intermediate layers can be costly
- not enough layers can increase the building time
- not enough layers can make impossible storage optimizations
- nowadays only *RUN*, *COPY* and *ADD* create new layers

Good practice 1

Think about your layers when you use *RUN*, *COPY* and *ADD* instructions in your Dockerfile

Good practice 2

Only install the required dependencies in your Dockerfile

- if using *apt* to install packages use *--no-install-recommends*
- if possible delete intermediate files not required when applying *RUN*

Multi-stage build

Good practice 3 - do multi stage build

- reduces the size of images by removing compilation dependencies in the final image
- the final image contains only the dependencies required to run the service
- a base image well adapted for executable files only is *scratch* or *alpine*

```
1  FROM golang as builder
2  RUN apt update && apt install -y git protobuf-compiler golang-goprotobuf-dev && \
3      git clone https://gitlab.imt-atlantique.fr/url && \
4      cd productcatalogservice && \
5      go mod download && \
6      mkdir genproto && \
7      protoc --go_out=plugins=grpc:genproto -I . productcatalogservice.proto && \
8      CGO_ENABLED=0 go build
9
10 FROM scratch
11 COPY --from=builder /go/productcatalogservice/productcatalogservice /
12 COPY --from=builder /go/productcatalogservice/products.json /
13 ENTRYPOINT ["/productcatalogservice"]
```

Anyone can push a Docker image on Docker Hub!

Good practice 4 - security

- always prefer official Docker images
- verify that the Docker image is regularly updated
- be sure that the image contains what you think (what are the different layers?)
 - > *docker history image_name*
 - tools like *dive*
- make sure to update the images you are using!

Additional good practices

- Exposing ports in Dockerfiles
 - `EXPOSE 80`
 - `EXPOSE 53/udp`
- Adding information with labels
 - `LABEL maintainer="helene.coullon@imt-atlantique.fr"`
- Add environment variables
 - `ENV ADMIN_USER="mark"`
 - `docker run -e ADMIN_USER="john"`
- Add volumes
 - `VOLUME /myapp/data`

It is time write a multi-stage Dockerfile

Deploying a software stack with Docker Compose

Automating the deployment of containerized applications



- easily deploys a containerized software stack
- define your deployment with a single YAML file (containers, volumes, networks, etc.)
- deployment files easy to share, version control, etc.

Structure of `compose.yaml`

Full specification

- *services*

- name of the service
 - Docker *image* or *build* path to the Dockerfile
 - *ports* exposed by the service
 - *networks* used by the service
 - *volumes* used by the service
 - *environment* variables used by the service with a value
 - *depends_on* another service

- *volumes*

- *networks*

```
1 version: '3.3'
2
3 services:
4   lychee:
5     image: linuxserver/lychee:4.7.0
6     container_name: Gdsn-photos-web
7     restart: always
8     networks:
9       - web
10      - default
11     volumes:
12       - ./conf:/config
13       - /srv/gdsn_photos_lychee:/pictures
14     labels:
15       - "traefik.http.routers.gdsn_photos.rule=Host('photos.gdsn.fr')"
16
17   db:
18     image: mysql:5.7
19     container_name: Gdsn-photos-db
20     volumes:
21       - /srv/gdsn_photos_db_data:/var/lib/mysql
22     restart: always
23     networks:
24       - default
25     environment:
26       MYSQL_ROOT_PASSWORD: secret
27       MYSQL_DATABASE: lychee
28       MYSQL_USER: lychee
29       MYSQL_PASSWORD: secret
30     labels:
31       - "traefik.enable=false"
32
33 networks:
34   web:
35     name: traefik_web
36     external: true
```

It is very important to understand that *Docker compose* creates a DNS so that containers can call each other without knowing their IP addresses!

Full CLI documentation

A few important commands

- *build* to build and rebuild services
- *up* to create and start services, networks, etc.
- *stop* to stop containers, networks, etc.
- *down* to stop and remove containers, networks, etc.

It is time to try Docker Compose!

You can also explore samples at [this link](#)