Docker

Hélène Coullon

IMT Atlantique

Table of contents

Introduction

Monoliths vs micro-services

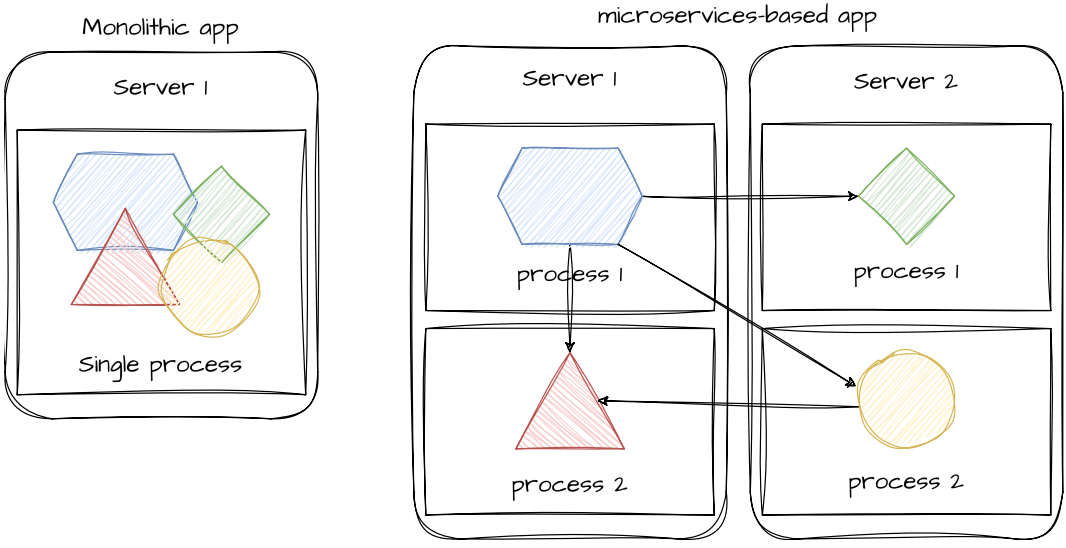
Applications designed as big monoliths

- 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



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

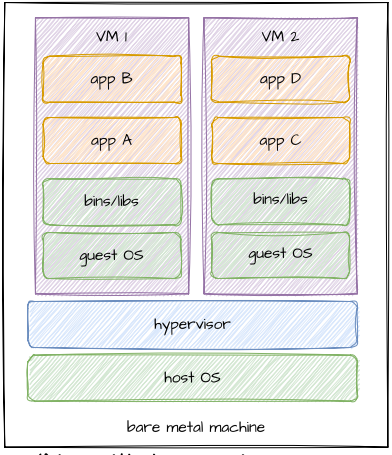
A bit about Linux kernel

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

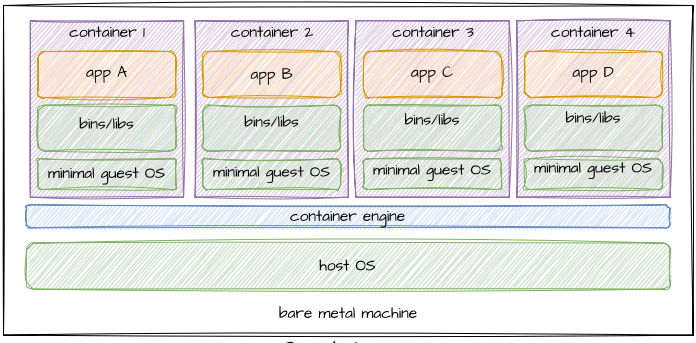
- it is the portion of the OS that is always loaded in memory
- it controls all hardware resources (e.g., I/O, memory, cryptography, CPU) via drivers
- it arbitrates conflicts and concurrency between processes
- it optimizes the utilization of resources (e.g., cache, memory, CPU, file systems, network)

The kernel is one of the first programs loaded on startup

Coarse-grain comparison between VMs and containers



VMs with hypervisor Type1



Containers

Containers

A container is a light virtualization technique

Container technologies

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


Docker

Overview of Docker

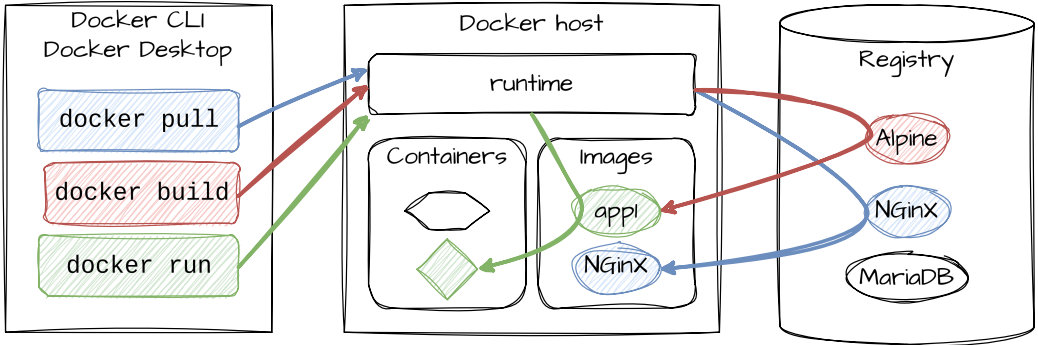
The different pieces involved in the process

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

Docker runtime

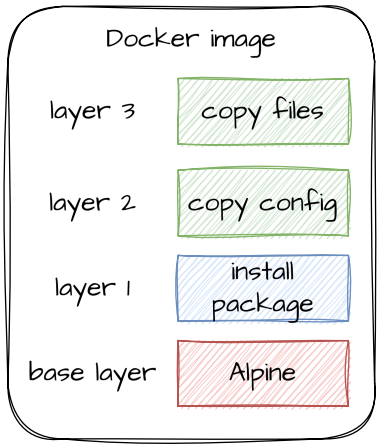
- Start and stop containers
- Manage images
- Manage networks
- Manage volumes
- etc.

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

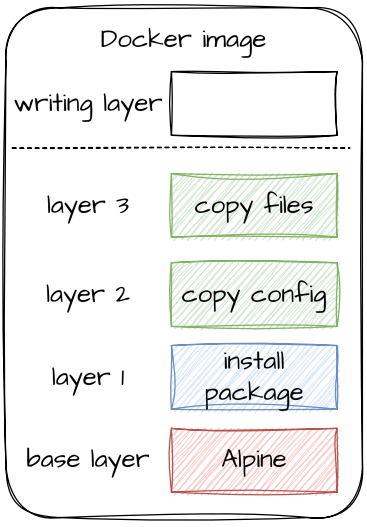
Writing in a container?

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



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.

Ports

Similarly, a port is not statically exposed, port exposition is dynamically created when creating the container

Host volumes

```
> docker run -p src-port:dest-port image_id
```

The *`EXPOSE port`* instruction in a Dockerfile does nothing. But it is a good practice to identify easily the need for a port exposition.

Docker CLI

Nothing better than a tutorial to discover the CLI!

Create an image with a Dockerfile

Dockerfile principles

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/

- *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

To go further

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 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

Reduce image size

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"]
```

Security

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

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

Deploying a software stack with Docker Compose

Automating the deployement 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*

```yaml
version: '3.3'

services:
    lychee:
        image: linuxserver/lychee:4.7.0
        container_name: Gdsn-photos-web
        restart: always
        networks:
            - web
            - default
        volumes:
            - ./conf:/config
            - /srv/gdsn_photos_lychee:/pictures
        labels:
            - "traefik.http.routers.gdsn_photos.rule=Host(`photos.gdsn.fr`)"

    db:
        image: mysql:5.7
        container_name: Gdsn-photos-db
        volumes:
            - /srv/gdsn_photos_db_data:/var/lib/mysql
        restart: always
        networks:
            - default
        environment:
            MYSQL_ROOT_PASSWORD: secret
            MYSQL_DATABASE: lychee
            MYSQL_USER: lychee
            MYSQL_PASSWORD: secret
        labels:
            - "traefik.enable=false"

networks:
    web:
        name: traefik_web
        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.

Try to Compose

It is time to try Docker Compose!

You can also explore samples at this link