

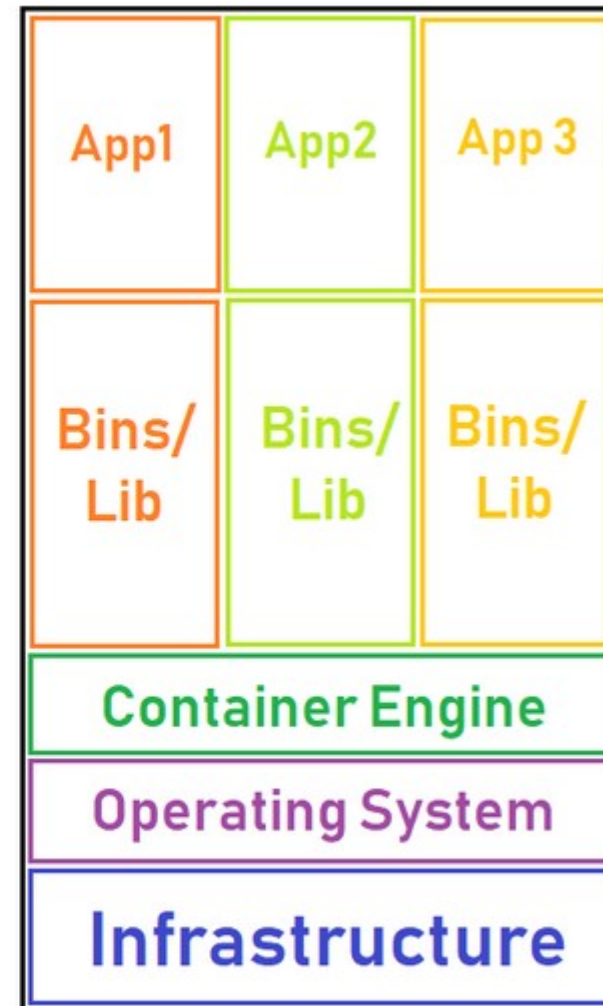


Images and Containers

Virtualization vs Containerization



Virtualization



Containerization

Virtualization vs Containerization

| Virtualization | Containerization |
|--------------------------------------------------------|--------------------------------------------------------|
| More secure and fully isolated. | Less secure and isolated at the process level. |
| Heavyweight, high resource usage. | Lightweight, less resource usage. |
| Hardware-level virtualization. | Operating system virtualization. |
| Each virtual machine runs in its own operating system. | All containers share the host operating system. |
| Startup time in minutes and slow provisioning. | Startup time in milliseconds and quicker provisioning. |



Containerization runtimes

LXC containers

LXC containers are part of the Linux open-source program. They allow you to run multiple isolated Linux systems on one host for application environments that resemble a VM. LXC containers operate independently instead of being managed by a central-access program.

Docker

Docker is a collection of platforms that can be used to create, manage, and deploy Linux application containers at the OS-level. Docker containers are hosted on a Docker Engine that is the client-server application host.

CRI-O

CRI-O is the implementation of Kubernetes Container Runtime Interface (CRI) for Open Container Initiative (OCI) runtime. It's an open-source tool that's a lightweight container engine replacement for Docker in Kubernetes. Using CRI-O enables Kubernetes to use OCI-compliant runtime for running pods.

Podman

Podman is an open-source containerization engine that, unlike Docker, doesn't use a central daemon, enabling the creation and deployment of self-sufficient and isolated containers. The design of Podman containers is security-focused through isolation and user privileges with standard non-root access.

Containerd

Containerd is a daemon that's compatible with both Windows and Linux environments. It's an abstracted interface layer between the container engines and runtime, allowing for easier management of containers. It's an open-source project that originated as one of the primary building blocks of Docker before separating.

runC

runC is a lightweight container runtime that's OS-universal. It started out as a low-level Docker component that helped with the security and architecture of the platform. Using the stand-alone version of the tool, runC is a container runtime that isn't tied to a specific container type, cloud provider, or hardware.

Docker Engine

- The Docker engine is the core software that runs and manages containers

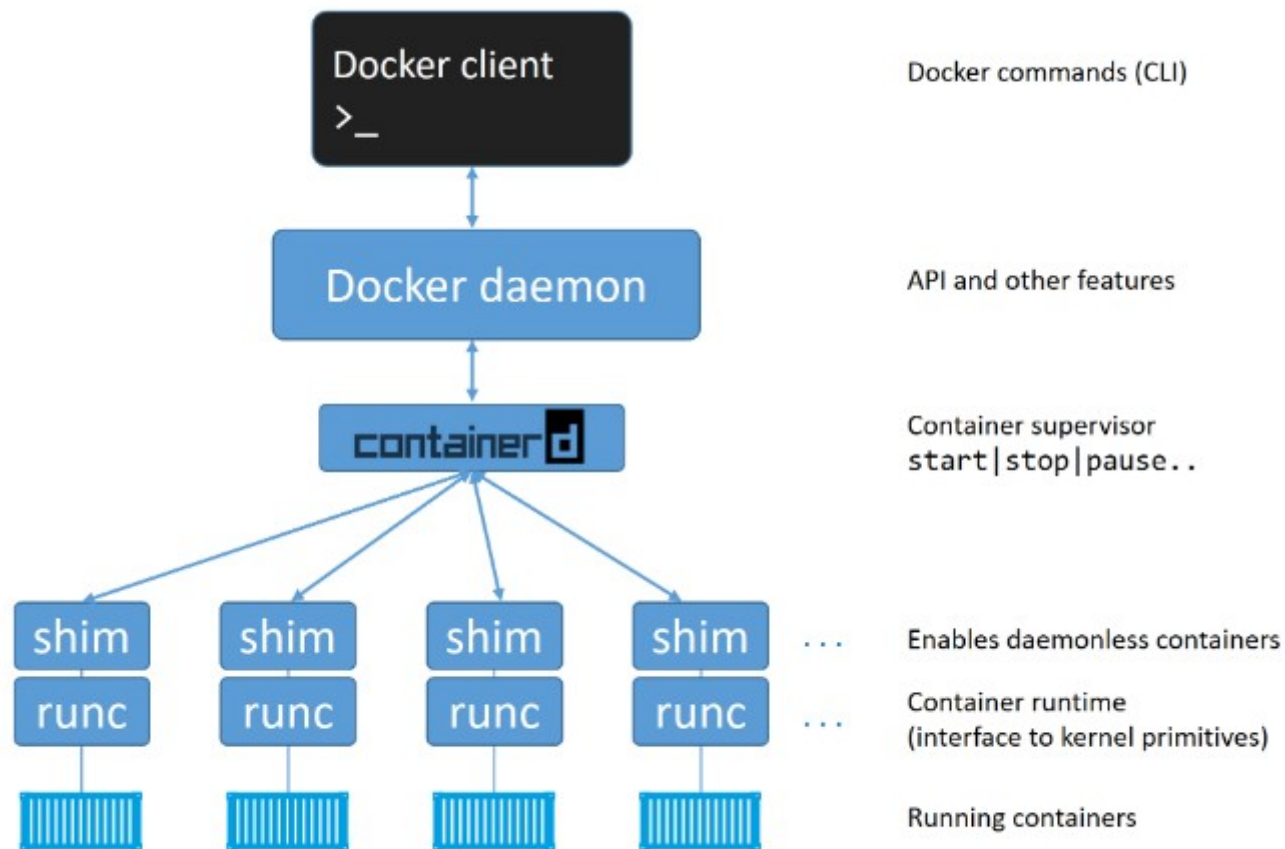
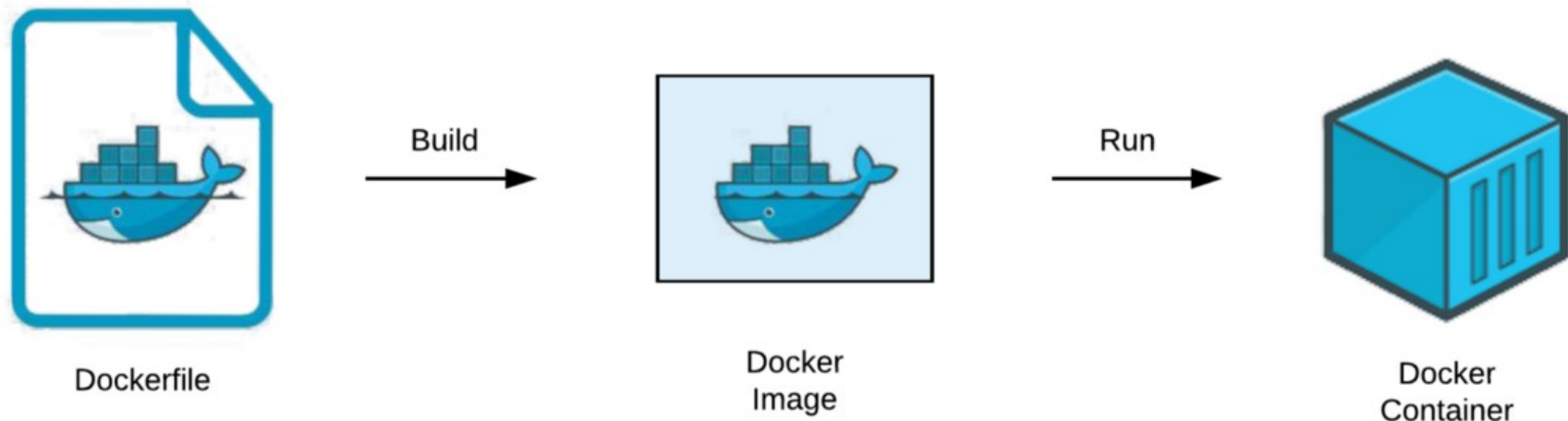


Figure 5.3

From Dockerfile to container

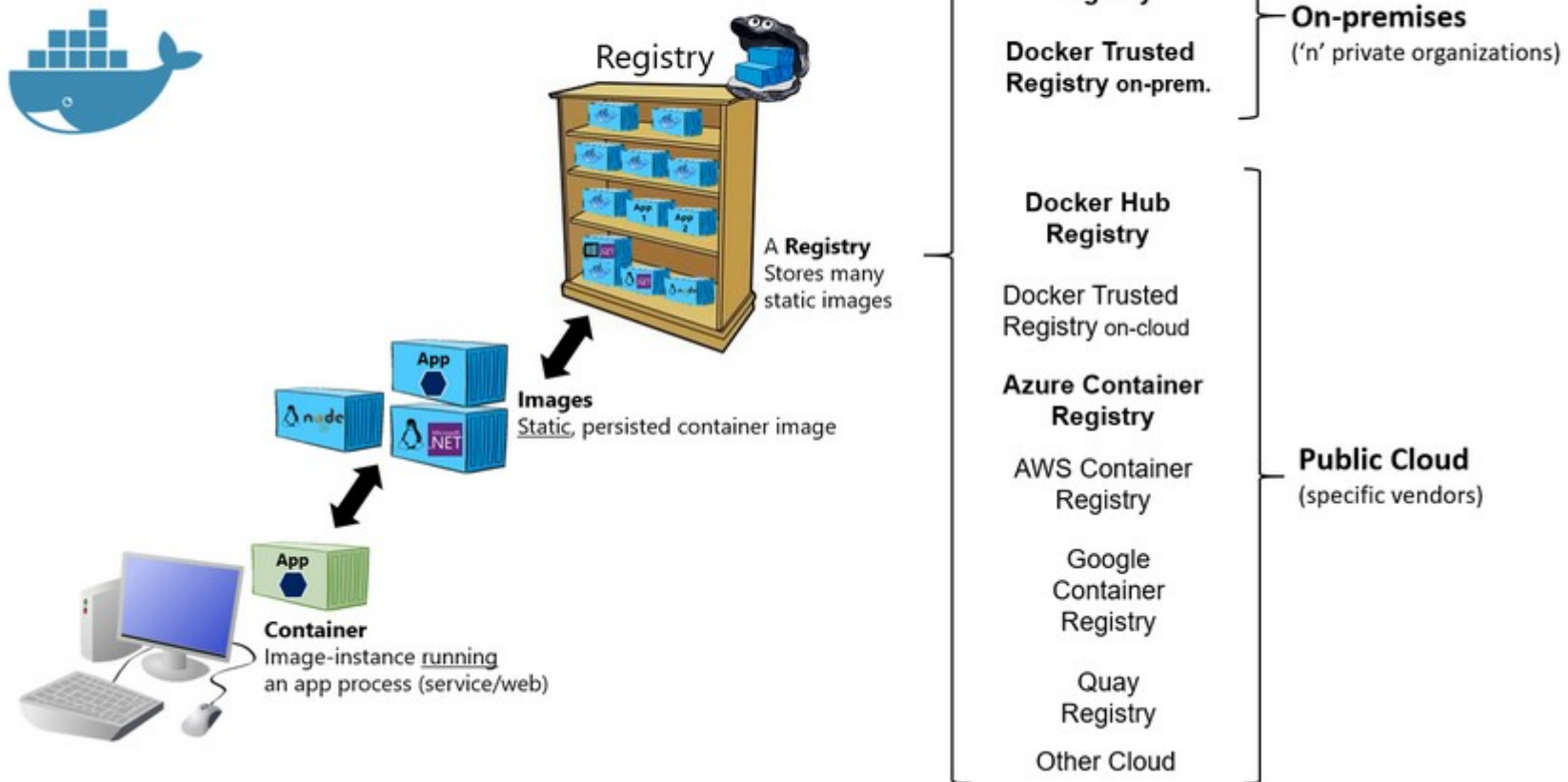
- The Dockerfile takes the place of the provisioning script. A Docker image can be run as an application. This running application sourced from a Docker image is called a Docker container.



Docker registry

The images can be stored in a registry private or public

Basic taxonomy in Docker





Quickstart Docker

- Sign in <https://hub.docker.com/>
- Install docker <https://docs.docker.com/engine/install/>
- <https://docs.docker.com/engine/install/linux-postinstall/>

Basic docker commands

Create a file run.py with the following content:

```
print("Hello Docker - PY")
```

| | |
|-------------------|-----------------------------------|
| FROM python | ← reuse an already existing image |
| ADD . . | ← add src code |
| CMD python run.py | ← execute runtime |

```
docker build . -t test-python
docker run test-python
docker inspect test-python
```

Basic docker commands (IMAGES)

docker images -> list existing images

docker search ubuntu -> find images in public registry

docker image pull ubuntu:latest -> download image from registry to local

docker info -> get info on the engine

sudo ls /var/lib/docker -> browse docker engine files

docker ps -> list instances

Basic docker commands (BUILD)

ENV: available a runtime

ARG: available only during build

Assignement:

Edit the file run.py adding an env variable MY_VAR

```
import os
my_var = os.environ['MY_VAR']
print("Hello Docker – {my_var}")
```

provide the env var to the dockerfile, build and run the image

```
docker build . testynov2
```

```
docker run -e MY_VAR="ynov" testynov2
```

Basic docker commands (RUN)

docker run -it ubuntu:latest -> run an image interactive

docker run -it ubuntu:latest ls -> override default command (bash) with ls

docker run -it --entrypoint ls ubuntu:latest /bin -> override the entrypoint and add CMD at the end

ENTRYPOINT: is used when we want the user to avoid overriding the default executable

CMD: is the default command it can be overridden by user

execution order: ENTRYPOINT + CMD

Assignment:

Modify the Dockerfile to provide arguments at runtime

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('-H', '--host', default='localhost', dest='host', help='Provide destination host. Defaults to localhost', type=str)
args = parser.parse_args()
print('Quiet mode is %r.' % args.quiet)
```

Basic docker commands (others)

docker run -it -v \$(pwd)/datadir:/usr/local/datadir ubuntu:latest -> run a container with datadir available

docker run -it -u 1000 ubuntu:latest -> run a container as user 1000

docker run -n myubuntu -d ubuntu:latest sleep 5000 -> run a container demonized

docker exec -it myubuntu -> exec in a container

docker stop myubuntu -> stop a container

docker rm myubuntu -> rm a container

Dockerfile best practices

- Separation of concern: Ensure each Dockerfile is, as much as possible, focused on one goal. This will make it so much easier to reuse in multiple applications.
- Avoid unnecessary installations: This will reduce complexity and make the image and container compact enough.
- Reuse already built images: There are several built and versioned images on Docker Hub; thus, instead of implementing an already existing image, it's highly advisable to reuse by importing.
- Have a limited number of layers: A minimal number of layers will allow one to have a compact or smaller build. Memory is a key factor to consider when building images and containers, because this also affects the consumers of the image, or the clients.

Dockerfile tuning

1. always use versioning on images:

es. use FROM python:slim-bullseye instead of python

2. reduce size of image

docker image ls python:latest → 921MB

docker image ls python:slim-bullseye → 126MB

docker image ls python:3.10-alpine → 48.7MB



Dockerfile tuning

3. run executables as unprivileged users:

define a USER in the Dockerfile with limited privileges

4. use 2 stages build to separate between build stage and execution stage: this results in lightweight containers

Requirements

- Create SSH key:
<https://linuxhint.com/generate-ssh-key-ubuntu/>
- Add public key to GITHUB
- Git clone ssh repo git@github.com
:sinaure/ynov-docker.git
- ADD docker user and password as secrets in the workflow tab

Assignment 1

Divide in groups for language preference

- Nodejs
- Java
- C++

adapt the secure Dockerfile to build and execute a helloworld with the new language

- Optionel: adapter la pipeline github action pour automatiser le build de l'image



Assignment 2

adapt the docker-compose project to let
interact the application with the database



Basic orchestration with docker compose

Docker compose has limited orchestration capabilities:

use `depends_on` to define start priorities