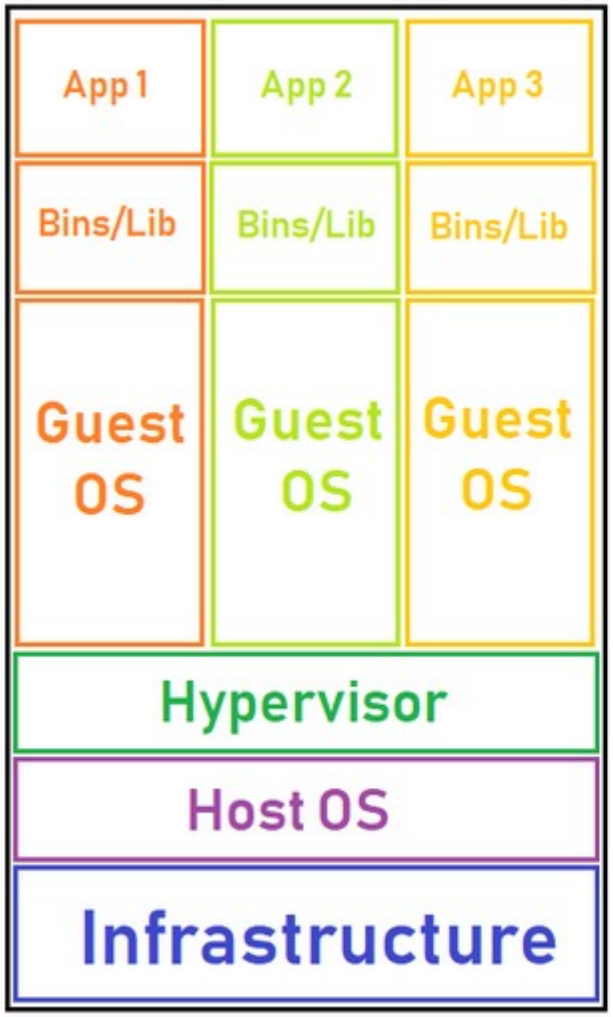
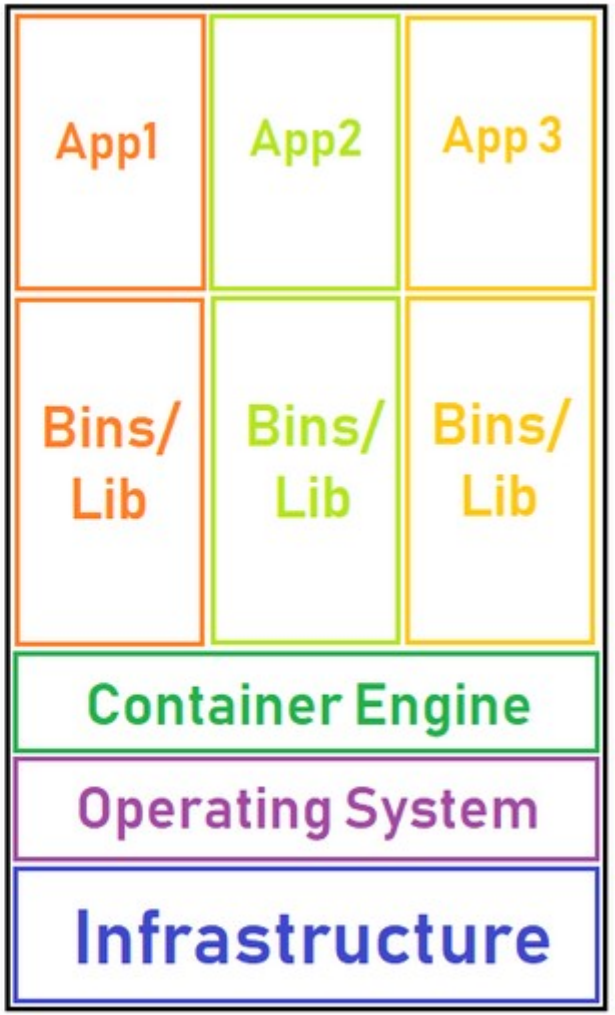
Images and Containers

Virtualization vs Containerization



Virtualization



Containerization

OBJECTIVES DES CONTENEURS

Portabilité : Les applications autonomes livrées avec leurs dépendances permettent aux conteneurs d'être prévisibles. Le code s'exécute de la même manière sur n'importe quelle machine, car les conteneurs apportent leur environnement reproductible.

Efficacité : Les conteneurs exploitent non seulement le noyau existant de leur machine hôte, mais ils existent souvent seulement le temps nécessaire à l'exécution du code qui leur incombe. De plus, les conteneurs sont souvent conçus pour n'exister que tant que le processus ou l'application qu'ils exécutent est nécessaire, et s'arrêtent une fois la tâche accomplie.

Non-conservation d'état : La non-conservation d'état est un idéal dans la conception des conteneurs qui vise à ce que le code s'exécute toujours de la même manière, sans nécessiter la connaissance des exécutions de code passées ou futures. Cependant, cette approche isolante a été confrontée aux besoins du monde réel et s'est concrétisée avec la plupart des moteurs de conteneurs proposant une solution pour les données persistantes, souvent sous forme de stockage de volume pouvant être partagé entre les conteneurs.

Réseau : Bien que les conteneurs soient souvent complètement isolés les uns des autres, il existe de nombreux cas où les conteneurs doivent communiquer entre eux, avec la machine hôte ou avec un autre groupe de conteneurs. Docker propose à la fois une mise en réseau pont pour les conteneurs qui existent sous la même instance de Docker afin de communiquer, et une mise en réseau superposée pour les conteneurs entre différentes instances de Docker.

Désolidarisation de l'application et de l'infrastructure : La logique de l'application doit démarrer et s'arrêter à l'intérieur du conteneur qui en est responsable. La gestion et le déploiement des conteneurs eux-mêmes dans différents environnements peuvent être pris en charge par les équipes d'infrastructure.

Spécialisation et microservices : La répartition des responsabilités entre plusieurs conteneurs permet l'utilisation de microservices plutôt que d'applications monolithiques.

TERMINOLOGIE

Système d'exploitation : Il s'agit du logiciel qui gère tous les autres logiciels sur votre ordinateur, ainsi que le matériel. Souvent abrégé en "SE", un exemple en est Linux, qui lui-même possède de nombreuses distributions telles qu'Ubuntu.

Noyau : Il s'agit de la composante du SE qui se spécialise dans l'interfaçage le plus basique et de bas niveau avec le matériel d'une machine. Il traduit les demandes de ressources physiques entre les processus logiciels et le matériel. Les ressources comprennent la puissance de calcul du CPU, l'allocation de mémoire de la RAM et les E/S du disque dur.

Conteneurs : Il s'agit d'une unité logicielle qui regroupe le code avec ses dépendances requises afin de s'exécuter dans un environnement isolé et contrôlé. Ils virtualisent un SE, mais pas un noyau.

Image de conteneur : Il s'agit de packages de logiciels requis par les conteneurs qui contiennent le code, le runtime, les bibliothèques système et les dépendances. En général, ils partent d'une image d'un SE comme Ubuntu. Ces images peuvent être construites manuellement ou être récupérées depuis un registre d'images.

Registre d'images : Il s'agit d'une solution de stockage et de partage d'images de conteneurs. L'exemple le plus remarquable est Docker Hub, mais il existe d'autres registres publics d'images et il est possible de mettre en place des registres d'images privés.

Répertoire d'images : Il s'agit d'un emplacement spécifique pour une image de conteneur au sein d'un registre d'images. Il contient l'image la plus récente ainsi que son historique au sein du registre.

TERMINOLOGIE

Runtimes de conteneurs : Ils gèrent le démarrage et l'existence d'un conteneur, ainsi que la manière dont un conteneur exécute réellement du code. Quelle que soit la solution de conteneur que vous choisissez, vous optez essentiellement pour différents types de runtimes de conteneurs en tant que base. De plus, les solutions de conteneurs complètes utilisent souvent plusieurs runtimes en conjonction. Les runtimes de conteneurs sont eux-mêmes divisés en deux groupes :

Open Container Initiative (OCI) : Il s'agit du runtime de base qui crée et exécute un conteneur, avec peu d'autres fonctionnalités. Un exemple en est runc, qui est le plus courant, utilisé par Docker et écrit en Golang. Une alternative est crun de Red Hat, écrit en C, et censé être plus rapide et léger.

Interface de runtime de conteneur : Ce runtime est plus axé sur l'orchestration de conteneurs. Les exemples incluent le dockershim initial, qui a progressivement été abandonné en faveur de containerd plus complet de Docker, ou l'alternative CRI-O de Red Hat.

Moteurs de conteneurs : Solutions complètes pour les technologies de conteneurs, un exemple étant Docker. Lorsque les gens parlent de la technologie des conteneurs, ils font souvent référence aux moteurs de conteneurs. Cela inclut le conteneur, le runtime de conteneur qui le sous-tend, l'image de conteneur et les outils pour les construire, et peut éventuellement inclure les registres d'images de conteneurs et l'orchestration de conteneurs. Une alternative à Docker serait une pile de Podman, Buildah et Skopeo de Red Hat.

Orchestration de conteneurs : Automatisation du déploiement de conteneurs. L'orchestration comprend la fourniture, la configuration, la planification, la mise à l'échelle, la surveillance, le déploiement, et bien plus encore. Kubernetes est un exemple d'une solution d'orchestration de conteneurs populaire."

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.

Container runtime

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. Example: PROXMOX

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: 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: 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.

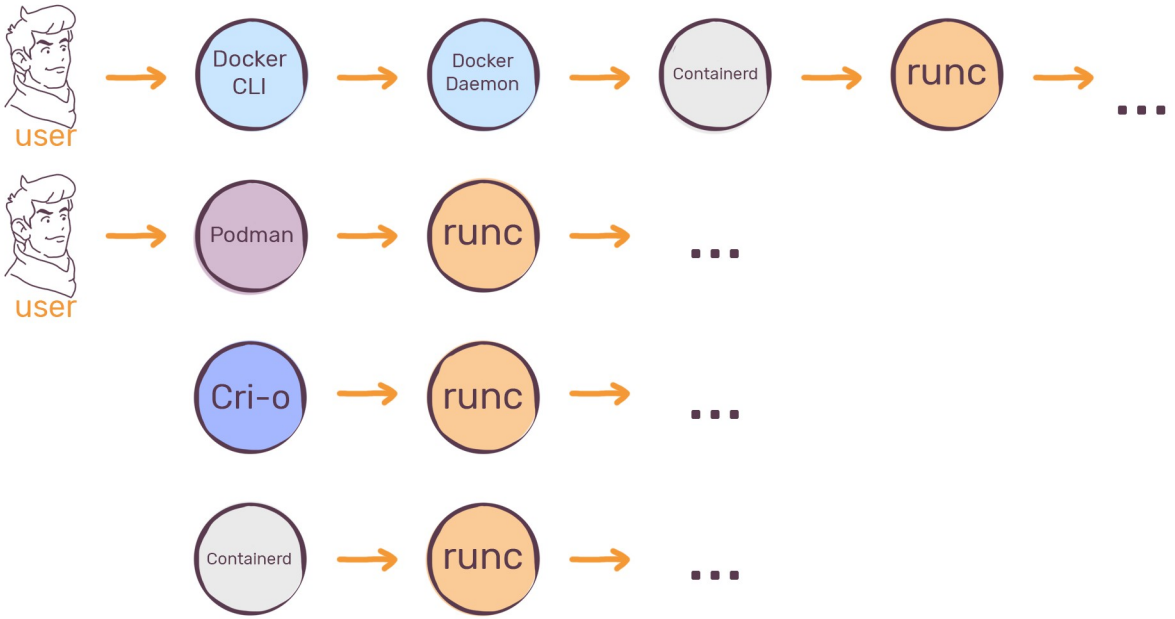
Containerd: is a daemon that's compatible with both Windows and Linux environments. It's an abstracted interface layer between the container engines and runtime, allowing for easier management of containers. It's an open-source project that originated as one of the primary building blocks of Docker before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

RunC: 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.

Container runtime - RUNC

https://selectel.ru/blog/en/2017/06/06/managing-containers-runc/

https://mkdev.me/posts/the-tool-that-really-runs-your-containers-deep-dive-into-runc-and-oci-specifications



OCI vs CRI

OCI define specs for images and runtimes

https://github.com/opencontainers

While OCI specs defines a single container, CRI (container runtime interface) describes containers as workload(s) in a shared sandbox environment called a pod. Pods can contain one or more container workloads. Pods let container orchestrators like Kubernetes handle grouped workloads that should be on the same host with some shared resources such as memory and vNETs.

Docker Engine

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

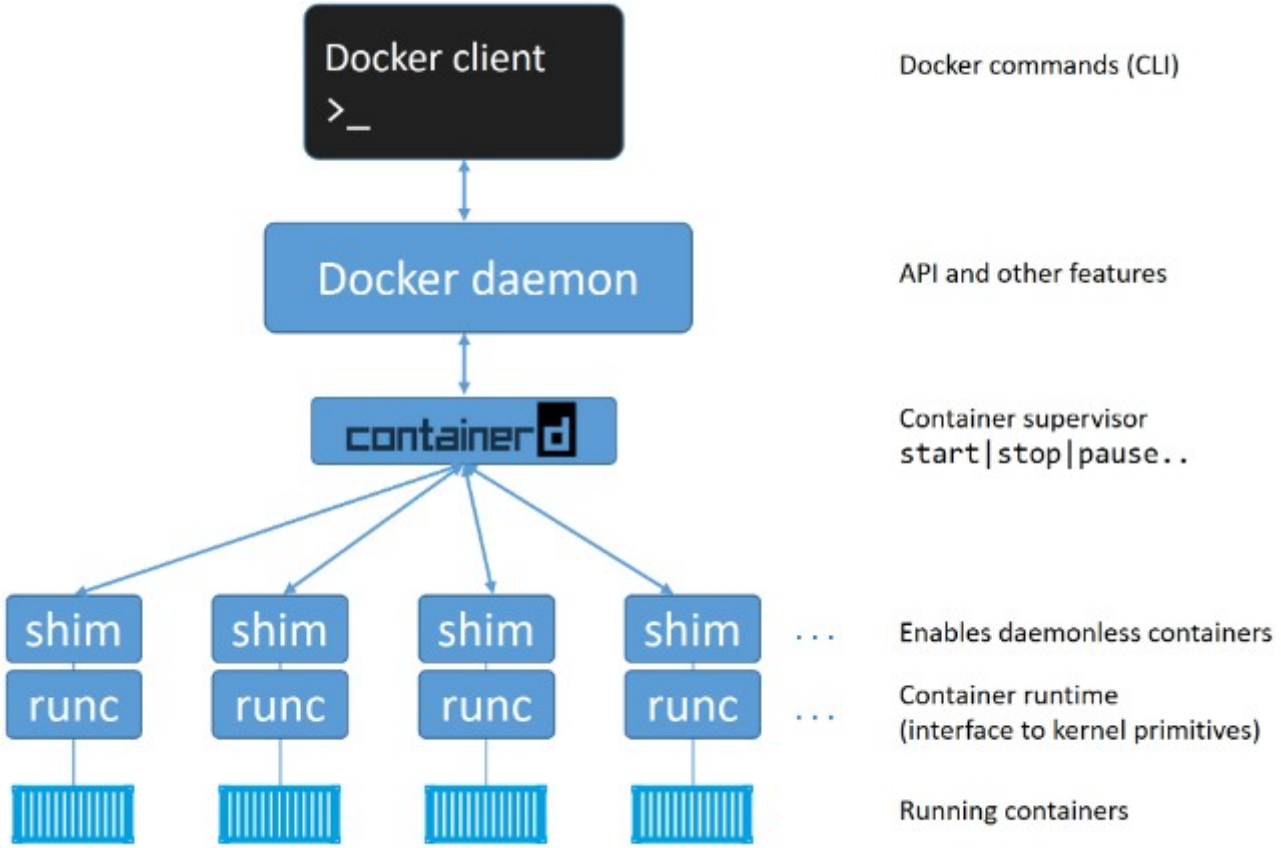
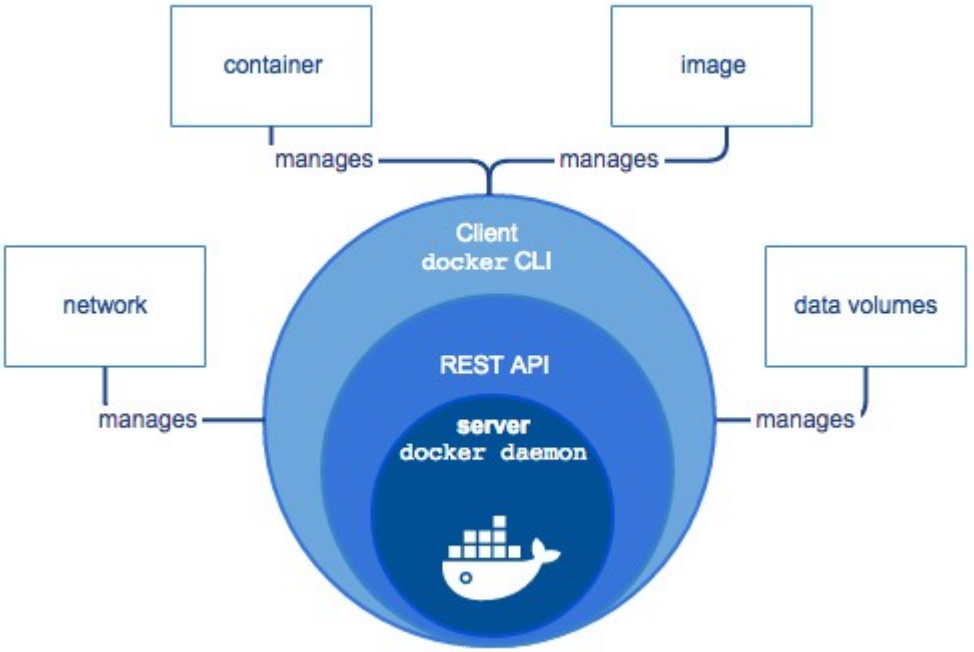


Figure 5.3

Docker daemon

- dockerd is the persistent process that manages containers. Docker uses different binaries for the daemon and client. To run the daemon you type dockerd.
- Several configurations can be customized: Daemon socket option and proxy for instance
- Create a file */etc/*docker/daemon.json with following content

{"hosts": ["tcp://0.0.0.0:2375", "unix:///var/run/docker.sock"]}



Docker daemon - socket

- A socket is a pseudo-file that represents a network connection. Once a socket has been created (identifying the other host and port), writes to that socket are turned into network packets that get sent out, and data received from the network can be read from the socket.
- Sockets are similar to pipes. Both look like files to the programs using them. Both facilitate interprocess communication. Pipes communicate with a local program; sockets communicate with a remote program. Sockets also offer, as you mention, bidirectional communication (much like a pair of properly connected pipes could).

Docker CLI – Rest API

```
curl \
    -s \
    "http://localhost:2375/containers/create?name=test" \
    -X POST \
    -H "Content-Type: application/json" \
    -d '{ "Image": "alpine:latest", "Cmd": [ "sh" ], "Tty":true }'
```

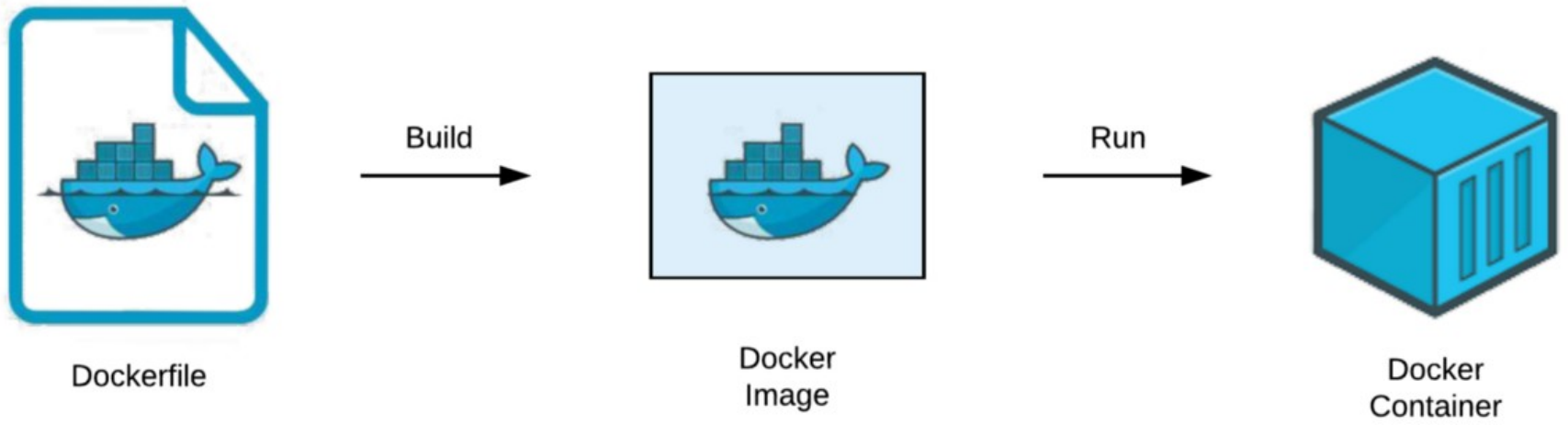
```
curl \
    -s \
    "http://localhost:2375/containers/test/start" \
    -X POST \
    -H "Content-Type: application/json"
```

Docker daemon - systemd

- A socket is a pseudo-file that represents a network connection. Once a socket has been created (identifying the other host and port), writes to that socket are turned into network packets that get sent out, and data received from the network can be read from the socket.
- Sockets are similar to pipes. Both look like files to the programs using them. Both facilitate interprocess communication. Pipes communicate with a local program; sockets communicate with a remote program. Sockets also offer, as you mention, bidirectional communication (much like a pair of properly connected pipes could).

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