Containers

- Lightweight virtual environment that groups and isolates a set of processes and resources (memory, CPU, disk, …), from the host and any other containers

Containers

- Containers are an evolution of the chroot tool presented in 1982

- Aplications (group of processes) are encapsulated into containers that:

  - Provide an isolated execution environment

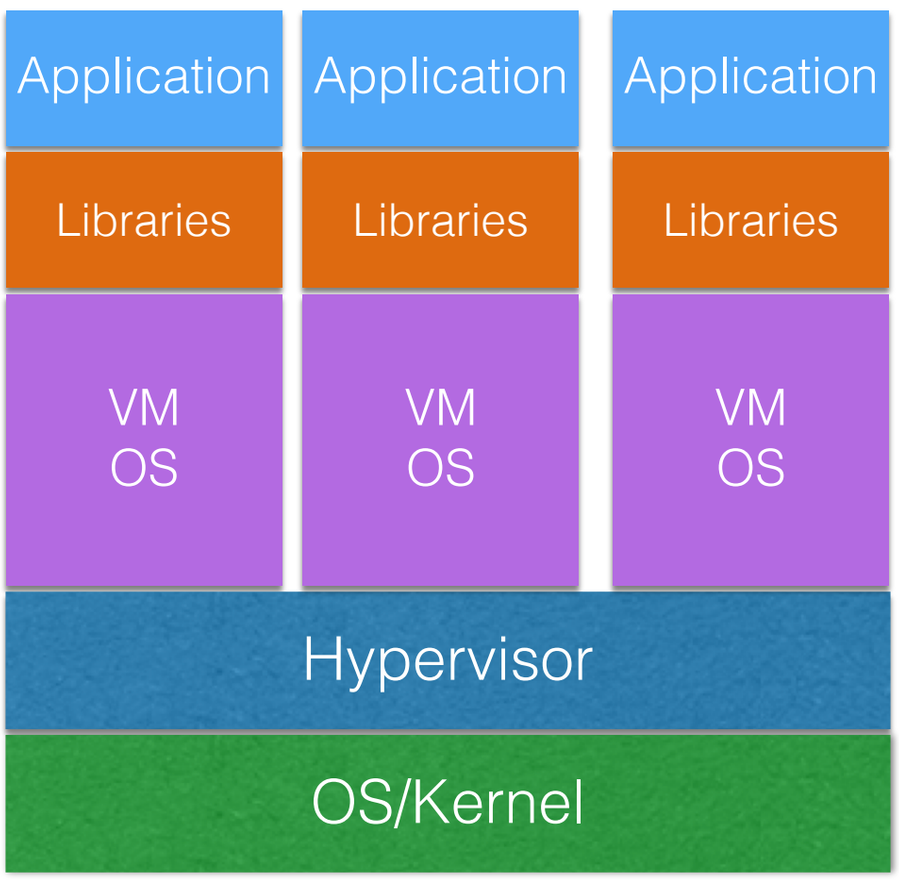
Containers

- Why are containers useful?
  - Running different isolated versions of the same software/application (e.g., database) in a shared OS/Kernel environment
  - Portability/migration across servers
  - Easy packaging of applications and their dependencies

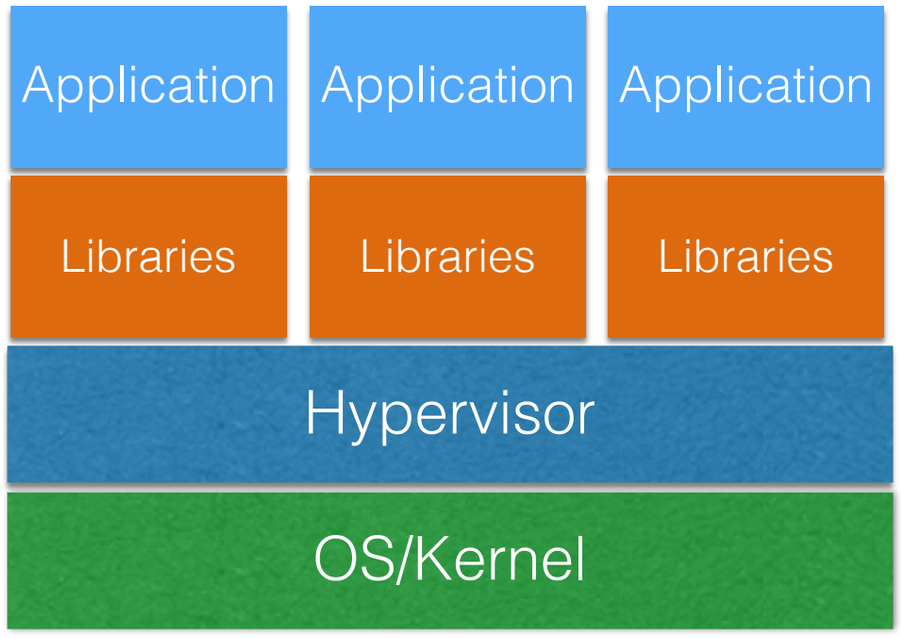
Linux Containers

- Can be seen as a lightweight VM where it is not needed the simulation of all hardware resources of a real computer

VMs



Containers



Linux Containers

- The host system is compartmentalised for each container in terms of CPU and I/O (memory, disk, network)

- Each container shares the hardware and kernel/OS with the host system

- Each container is isolated from others

Linux Containers

Types

- OS Containers
  - Containers are similar to VMs, running multiple processes
  - Kernel is shared, have independent OS
  - Flexibility/isolation vs performance
  - E.g., LXC

Linux Containers

Types

- Application Containers
  - Focused on deploying applications
  - Each application is seen as an independent process
  - Share the Kernel and OS (or some OS functionalities)
  - Flexibility/isolation vs performance
  - E.g., Docker

Linux Containers

Building Blocks

- Namespaces (Isolation)
  - Component of the Linux Kernel that makes a global host resource appear as a dedicate resource for a group of processes running in the same namespace
  - Allows sharing of host resources across different containers without conflicts
    - Network, storage (filesystem), ...

Linux Containers

Building Blocks

- Control Groups (Resource Management)
  - Allows allocating resources (CPU, RAM, Disk I/O, network bandwidth) among groups of processes
  - In other words, this mechanism allows limiting the amount of resources used by each container and prioritising one container over others (e.g., The I/O of a critical database service)

Linux Containers

Building Blocks

- SELinux (Security)
  - Provides additional security over namespaces so that a container is not able to compromise the host system and other containers
  - Enforces access control and security policies

Advantages

Over VMs

- Easier and Faster testing/provisioning/migration
- Better resource utilisation and lower costs
- Can be deployed on both physical and virtualized servers
- Easier management
- Performance

Disadvantages

Over VMs

- Weaker isolation/security (OS/Kernel are shared)
- Flexibility in running different OSs (e.g., Linux, Windows, BSD, ...)

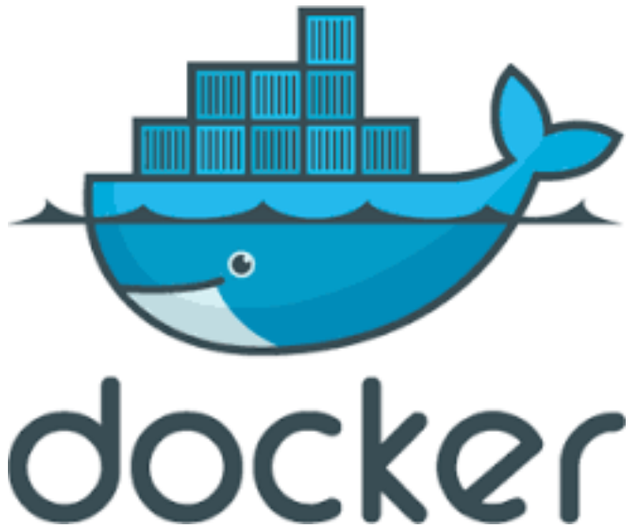
Disadvantages

Over VMs

- Weaker isolation/security (OS/Kernel are shared)
- Flexibility in running different OSs (e.g., Linux, Windows, BSD)

Each virtualization platform presents its own advantages and disadvantages and should be used according to the requirements of application/workloads

Docker



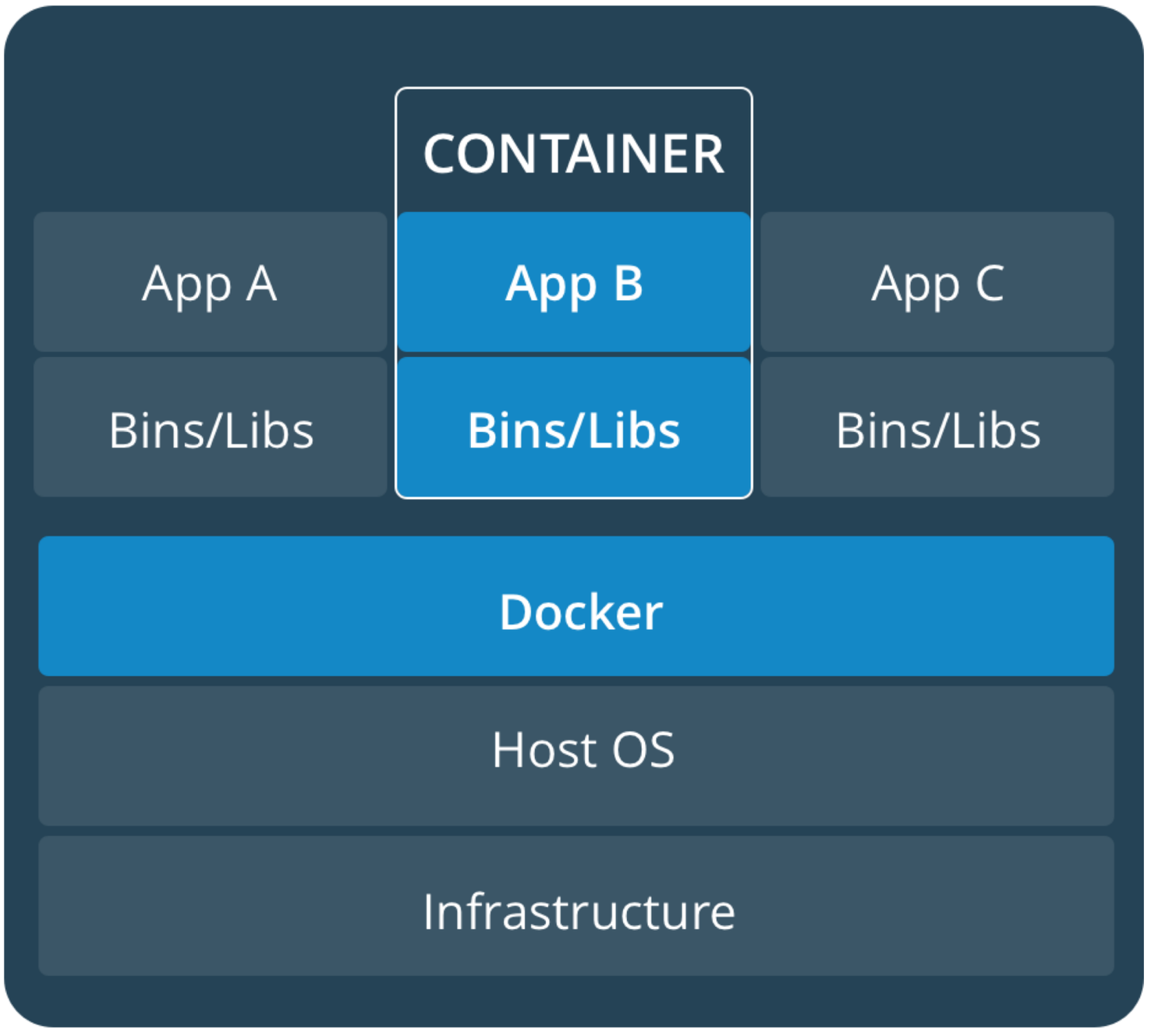
- Most widely-known container platform
- Supports Ubuntu, Fedora, RHEL, CentOS, Windows, etc
- https://www.docker.com

Windows Containers

- Initial version (Docker) relies on Linux Virtual Machines for running docker containers
  - High performance impact

- The newest version (Docker) resorts to a Windows container hypervisor that resorts to similar primitives used in Linux Containers

Docker


CONTAINER
App A
App B
App C
Bins/Libs
Bins/Libs
Bins/Libs
Docker
Host OS
Infrastructure

Docker

Underlying Technology

- Namespaces
- Control groups
- Union file system

Docker

Underlying Technology

- Union file system
    - The container is created by deploying several file system layers on top of each other
    - Each layer can be seen as an intermediate image that supports different services running on the container
    - This approach allows reusing layers across different containers, saving disk space and optimising the time to provision new containers

Docker

Configuration File (Dockerfile)

```
FROM node:argon

# Create app directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Install app dependencies
COPY package.json /usr/src/app/
RUN npm install

# Bundle app source
COPY . /usr/src/app

EXPOSE 8080
CMD [ "npm", "start" ]
```

Docker

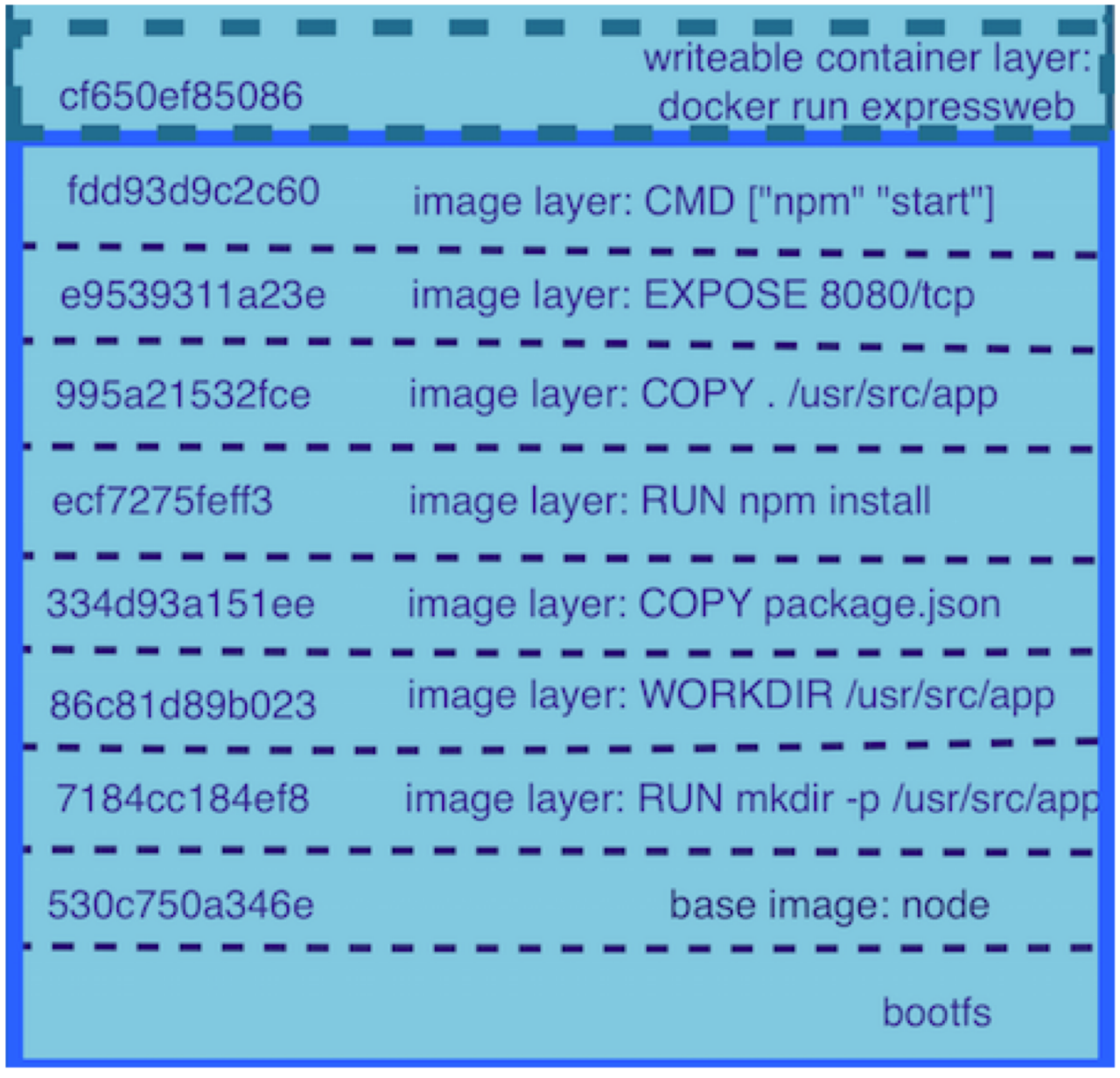
Deployment

```
$ docker build -t expressweb .
Step 1 : FROM node:argon
argon: Pulling from library/node...
...
Status: Downloaded newer image for node:argon
 ---> 530c750a346e
Step 2 : RUN mkdir -p /usr/src/app
 ---> Running in 5090fde23e44
 ---> 7184cc184ef8
Removing intermediate container 5090fde23e44
Step 3 : WORKDIR /usr/src/app
 ---> Running in 2987746b5fba
 ---> 86c81d89b023
Removing intermediate container 2987746b5fba
Step 4 : COPY package.json /usr/src/app/
 ---> 334d93a151ee
Removing intermediate container a678c817e467
Step 5 : RUN npm install
 ---> Running in 31ee9721cccb
 ---> ecf7275feff3
Removing intermediate container 31ee9721cccb
Step 6 : COPY . /usr/src/app
 ---> 995a21532fce
Removing intermediate container a3b7591bf46d
Step 7 : EXPOSE 8080
 ---> Running in fddb8afb98d7
 ---> e9539311a23e
Removing intermediate container fddb8afb98d7
Step 8 : CMD npm start
 ---> Running in a262fd016da6
 ---> fdd93d9c2c60
Removing intermediate container a262fd016da6
Successfully built fdd93d9c2c60
```

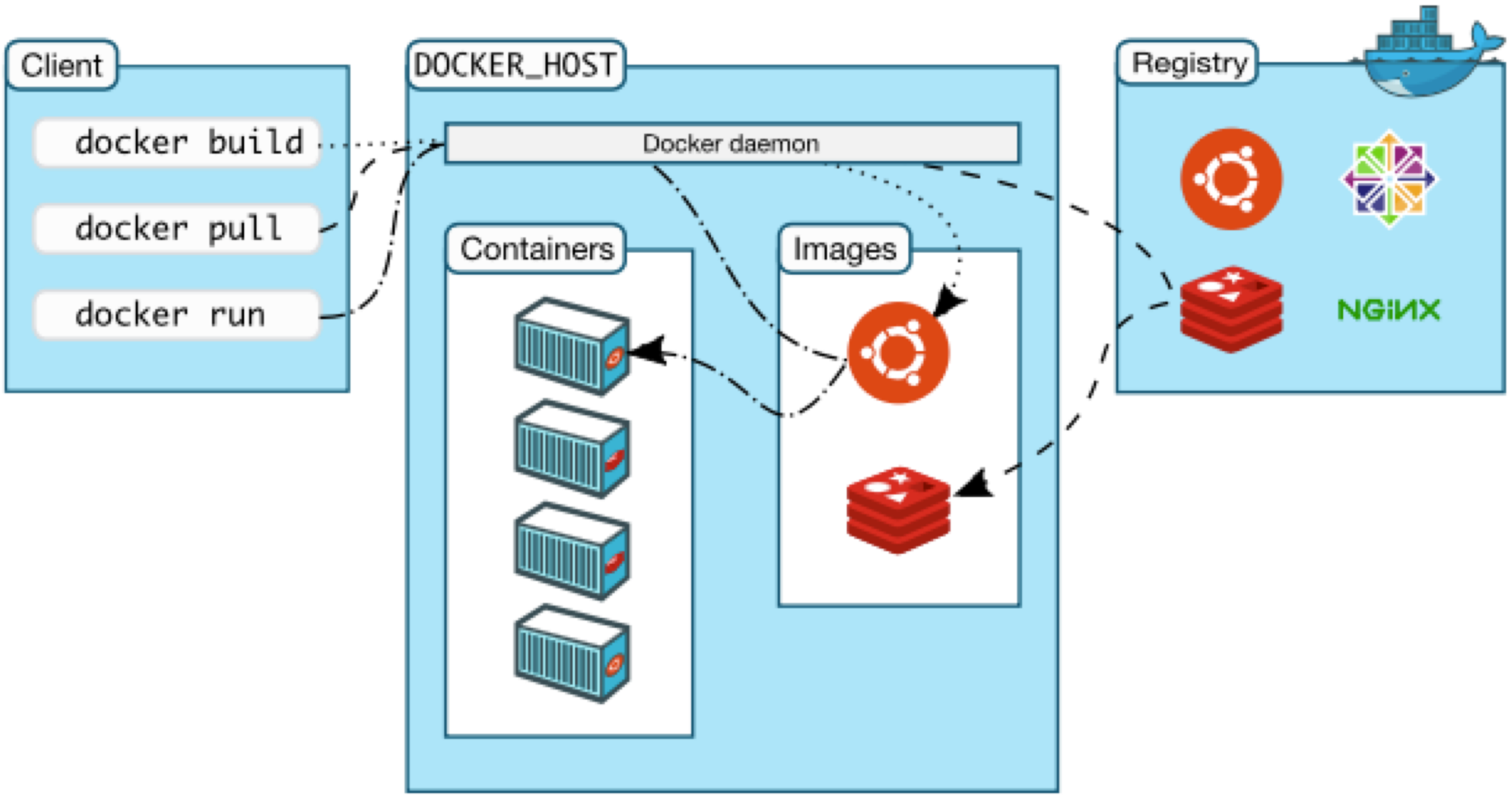
Docker

Deployment

- The first layer is writable while the others are read-only



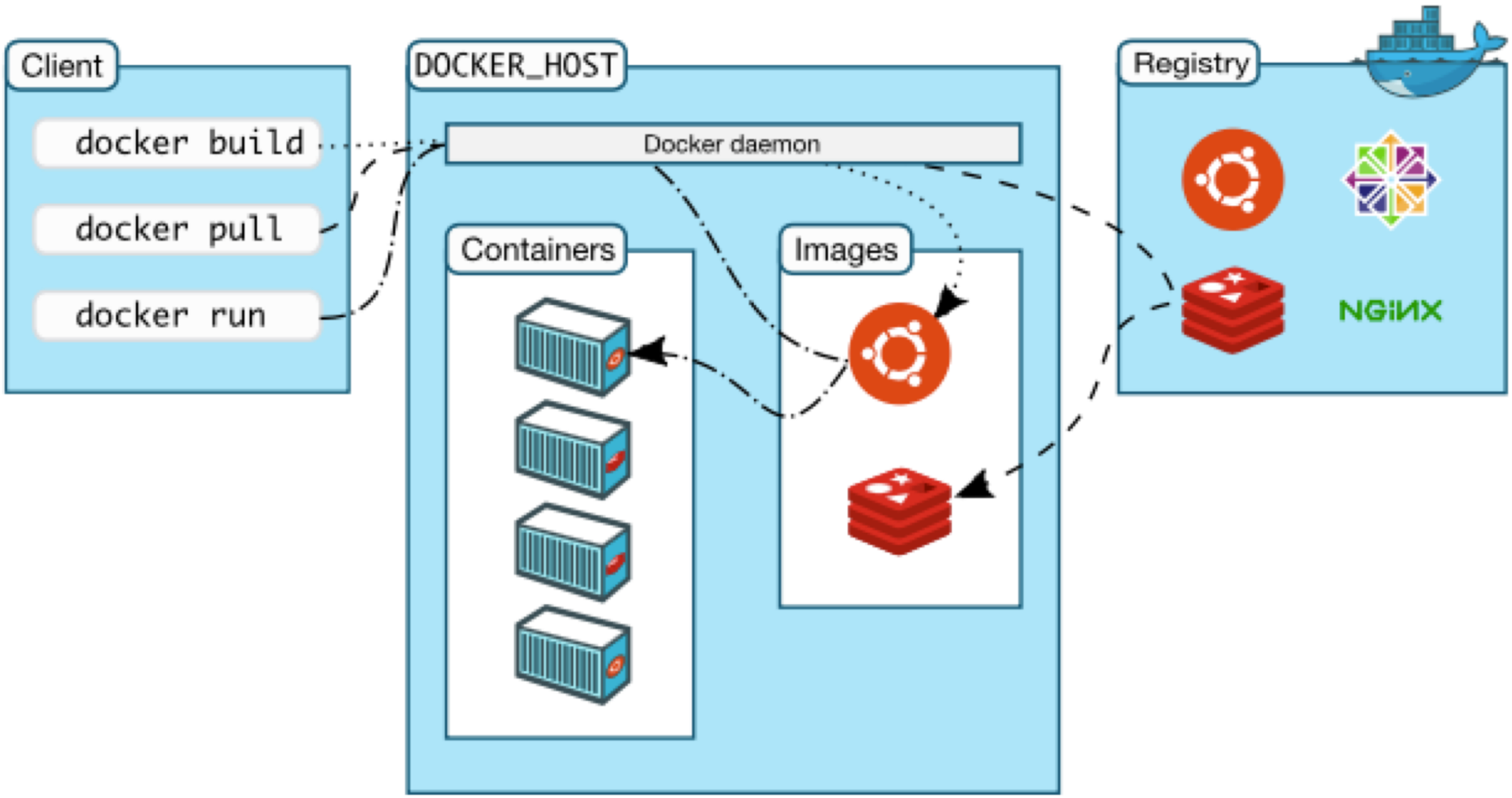
Docker


Client
docker build
docker pull
docker run
DOCKER_HOST
Docker daemon
Containers
Images
Registry
NGINX

Docker

Docker Daemon

- Uses the Docker API for receiving requests from the Docker Client
- Manages Docker Images, Containers, Networks



Docker

Docker Client

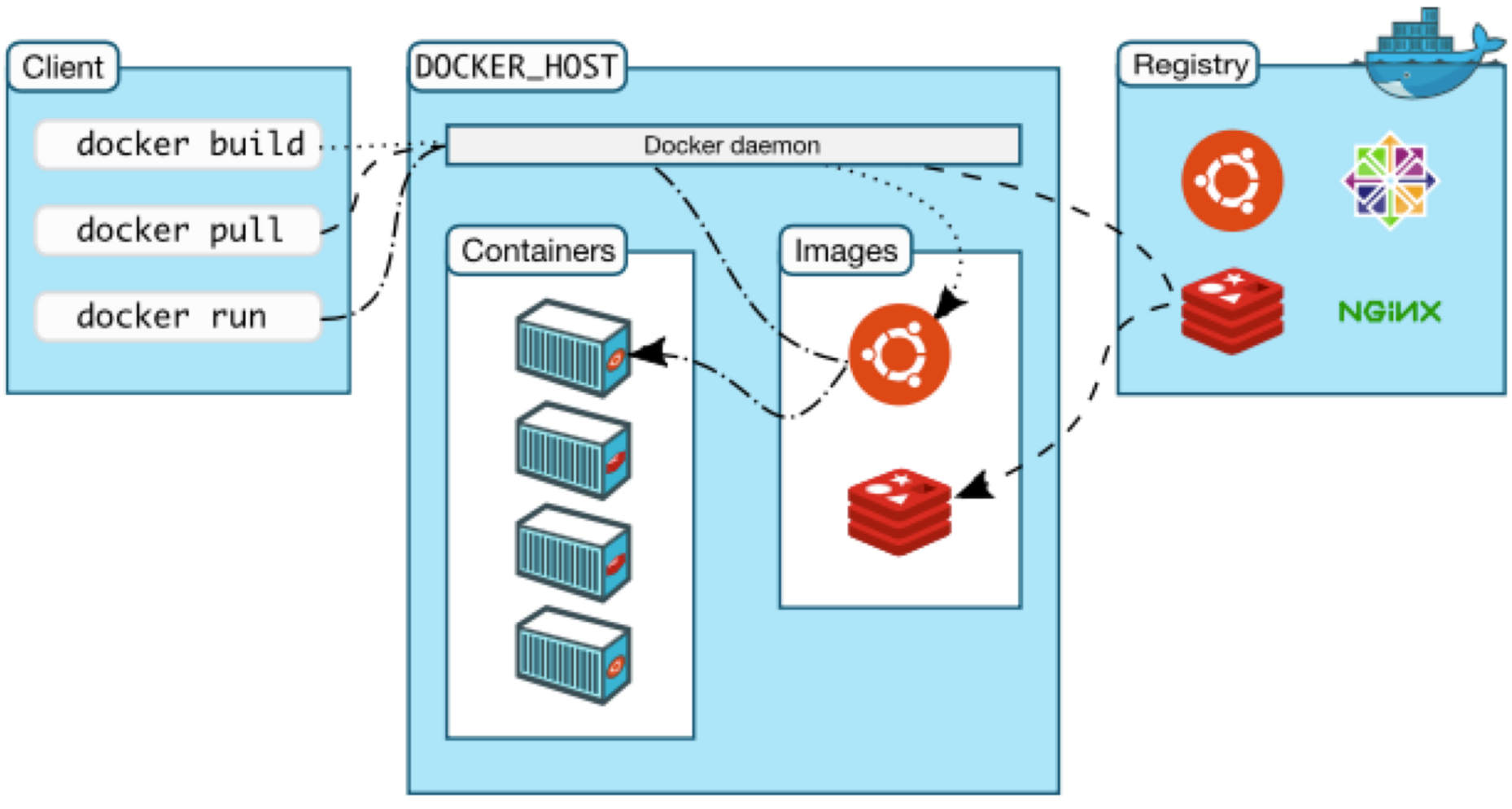
- Component used by users to interact with the Docker platform
- Implements the Docker API for:
  - Running and orchestrating/managing containers, and networks
  - Reading logs and metrics
  - Pulling and managing images

Docker

Docker Objects

- Image - read-only template with instructions for building a Docker container

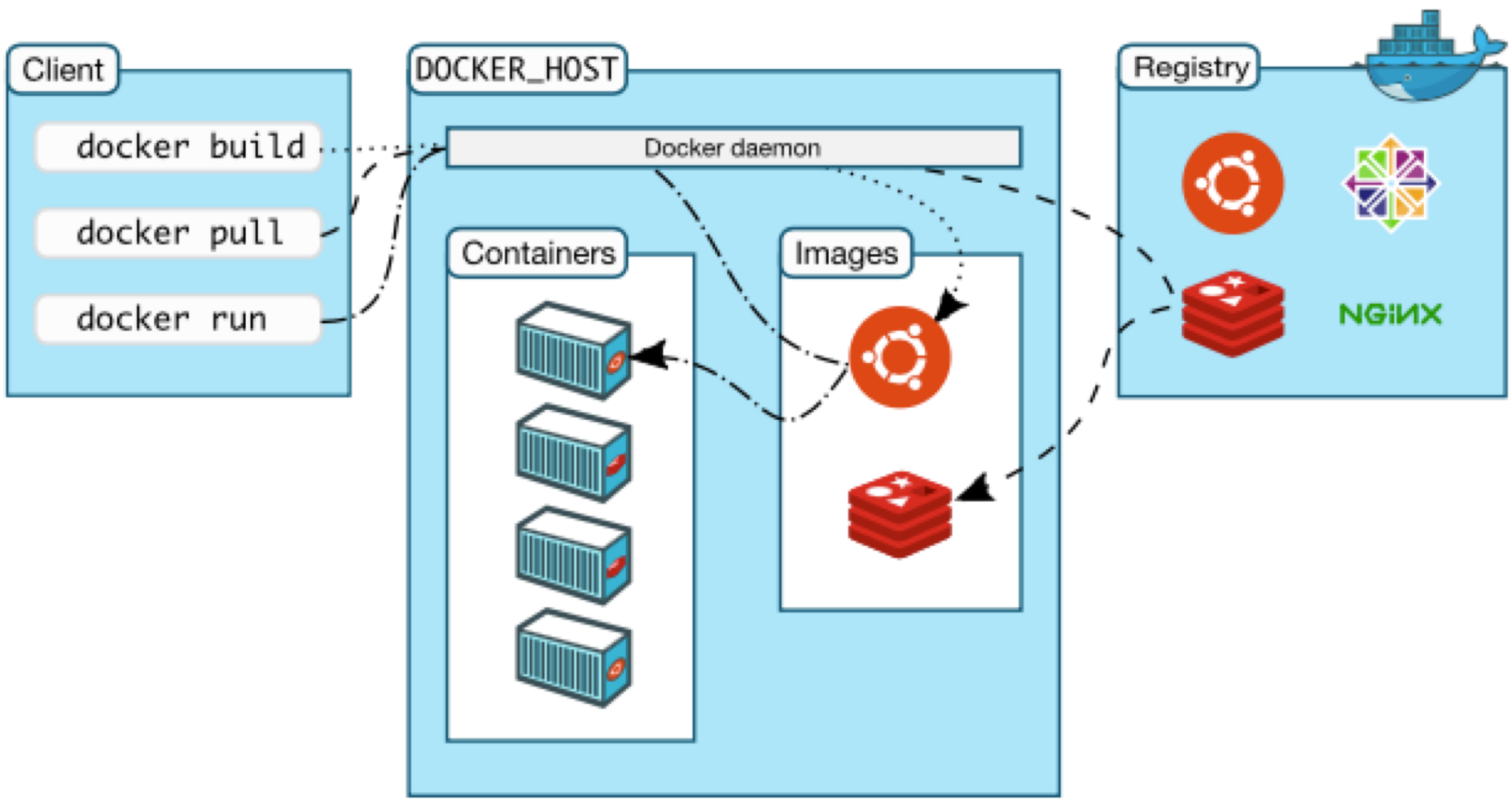
- Container - runnable instance of an Image



Docker

Docker Registry

- Repository of Docker Images
- E.g., https://store.docker.com



Docker

Storage

- Docker volumes can be created by:
  - Attaching a host volume/folder to the container
  - Creating an isolated volume inside the docker container

Docker

Network

- Host
  - Shares the host networking namespace. Container services are presented in the network as if they were run by the host. Ports are shared (e.g., port 80)
- Bridge
  - The container is seen as another node in the physical network

Docker

Services

- In a scalable setup different Docker containers may be running specific components of a more complex application (e.g, database, web server, etc)

- Each container is called a Service (container in production)

- These services must specify the ports that each container exports, how many replicas will be deployed (requires cluster mode), ...

- Such can be done with the Docker-compose mechanism

Docker

Cluster

- For performance and scalability purposes, each service may be deployed on a cluster composed by several nodes (hosts)
- Docker Swarm
  - A swarm is a group of machines that are running Docker engine and are members of a cluster
  - The nodes controlling the cluster are swarm managers while the other nodes are workers
  - Swarm Managers are responsible for managing the workers and the tasks (services) assigned to each (Load Balancing)
  - Other solutions: Kubernetes

Docker

Cluster Network

- Cross-Host communication
- Overlay - create a logical network across Docker hosts
  - Ingress
  - Weave
  - Calico
  - Flannel