↘ Speakers: Davide Zanetti
↘ Stefano Monti
↘ Enrico Grillini

Imola, 18/06/2018

Docker Ecosystem and Tools

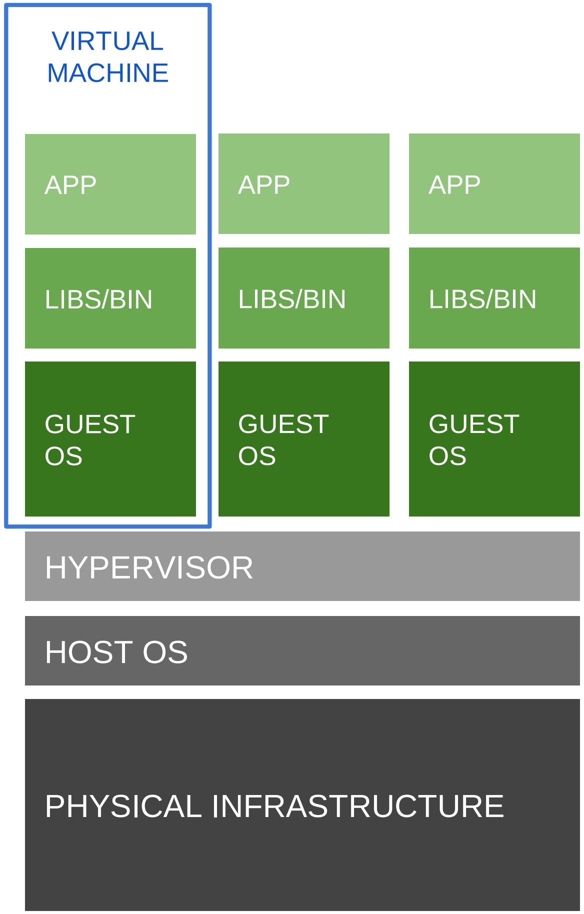
Agenda

1. Containers & Docker ecosystem
    1. Docker basics
    2. Docker basics - hands on
    3. Web GUIs (e.g Portainer) & Debug
    4. Docker-compose
    5. Docker-compose - hands on

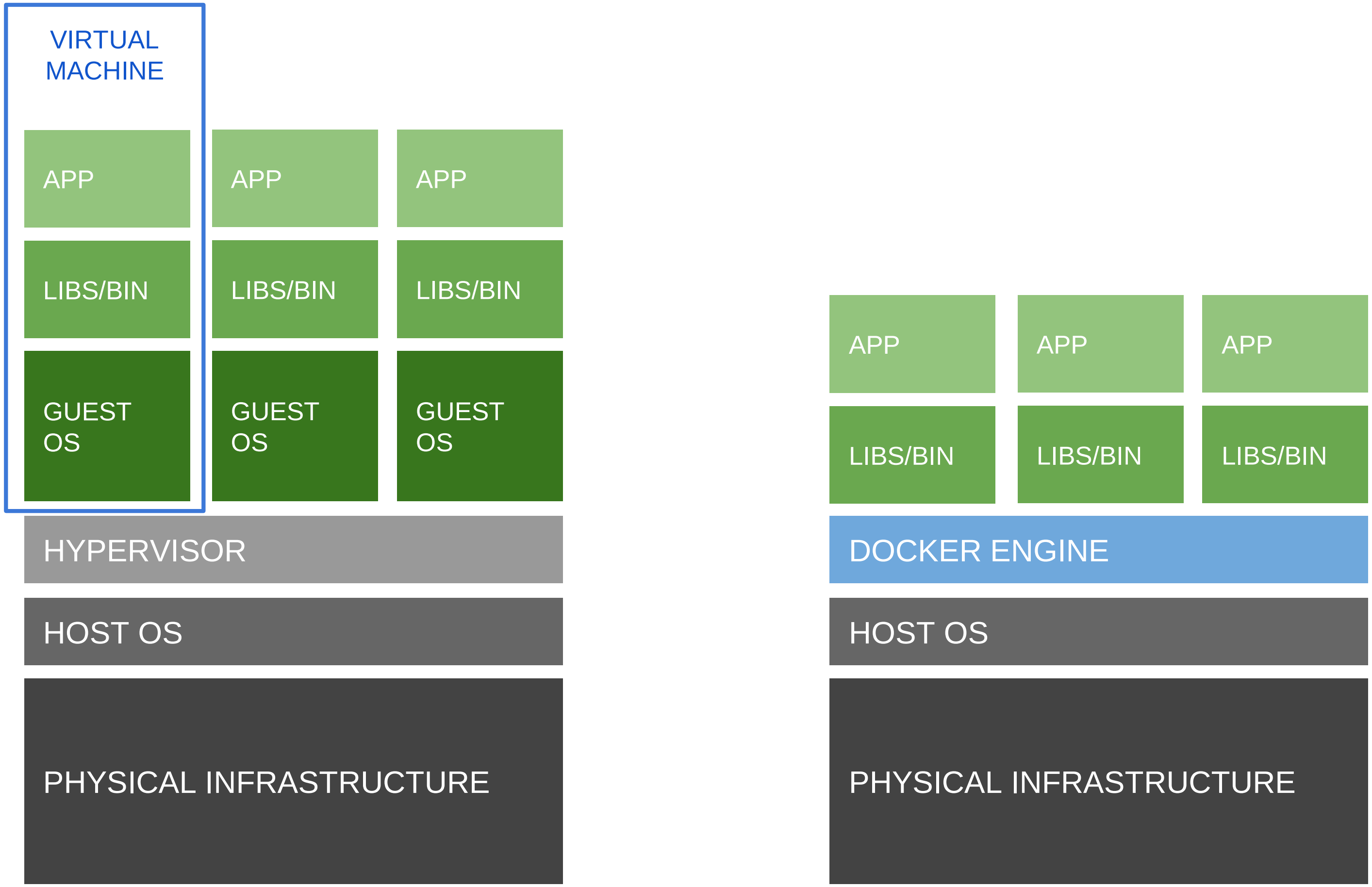
2. Orchestration tools
    1. Overview
    2. Kubernetes

Agenda

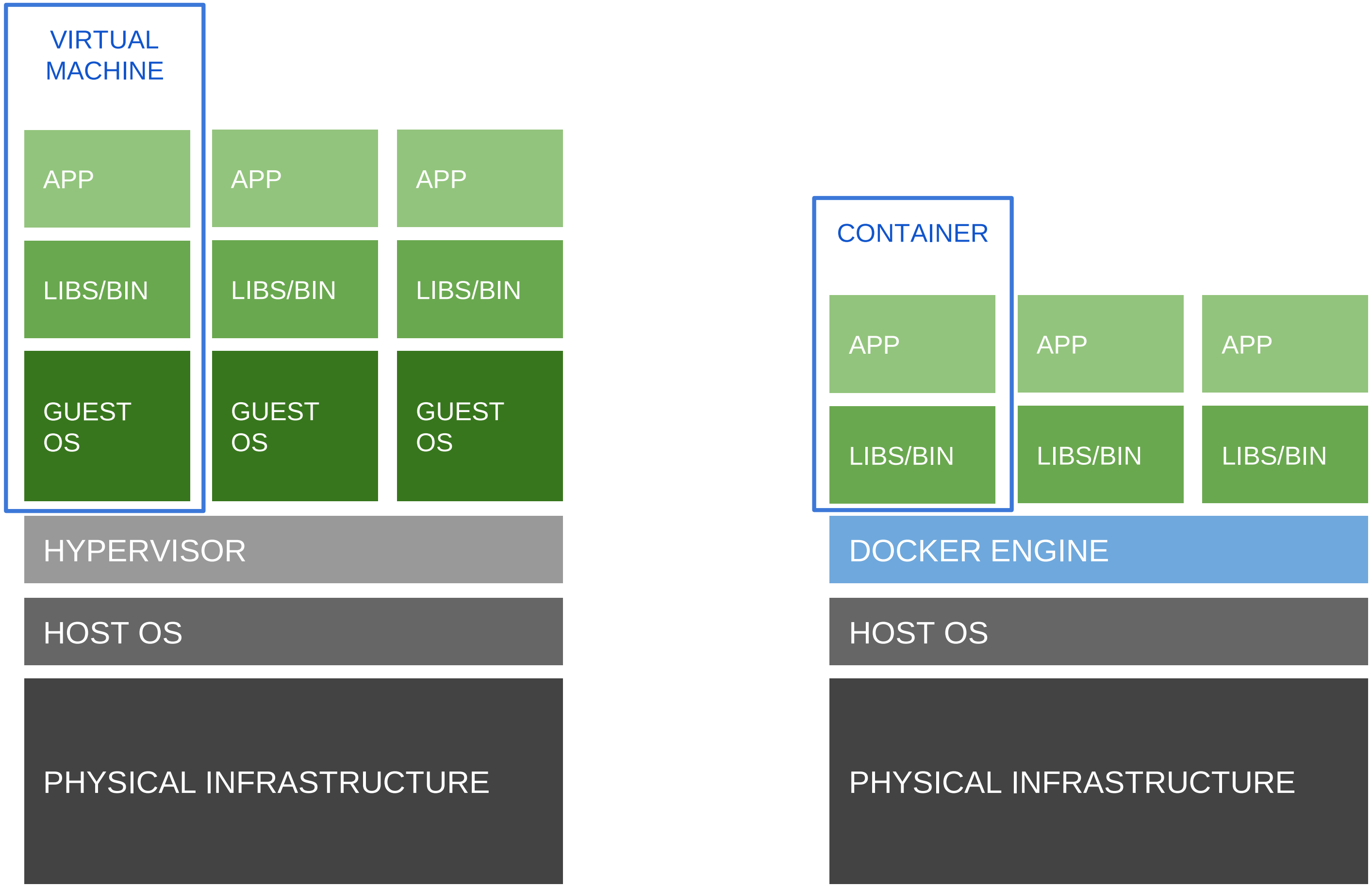
Virtualization vs Containerization



Virtualization vs Containerization



Virtualization vs Containerization



Containerization vs Virtualization

- containers include an **application/service** together with its dependencies
- containers **share kernel** with other containers
- containers run as **isolated processes**

- higher efficiency w/r to virtualization
- **images** are the cornerstone in crafting declarative/automated, easily repeatable, and scalable services and applications

Running Docker on Windows/MacOS (as of 03/2018)

- On **Windows** (Windows 10 Pro 17.09 «Falls Creator Update»):
  - «**Docker for Windows**» official tool
    - Linux containers → run on a Hyper-V Linux VM
    - Windows containers → run on a Hyper-V «Windows server kernel» VM

    Limitation: other hypervisors (eg. VirtualBox) cannot run if Hyper-V if enabled
- On **Windows Server** (Windows Server 2016)
  - **native** Windows Server Containers (no need for VM)

- On **MacOS**: «Docker for Mac» official tool
  - run Linux containers on a HyperKit VM

- ... or you can always manually create a Linux or Windows Server VM with VirtualBox/VMWare with shared folders and install Docker on it

What is Docker?

*An **open** platform for **distributed applications** for **developers and sysadmins***

*Docker allows you to **package an application** with all of its dependencies into a **standardized unit** for software **development**.*

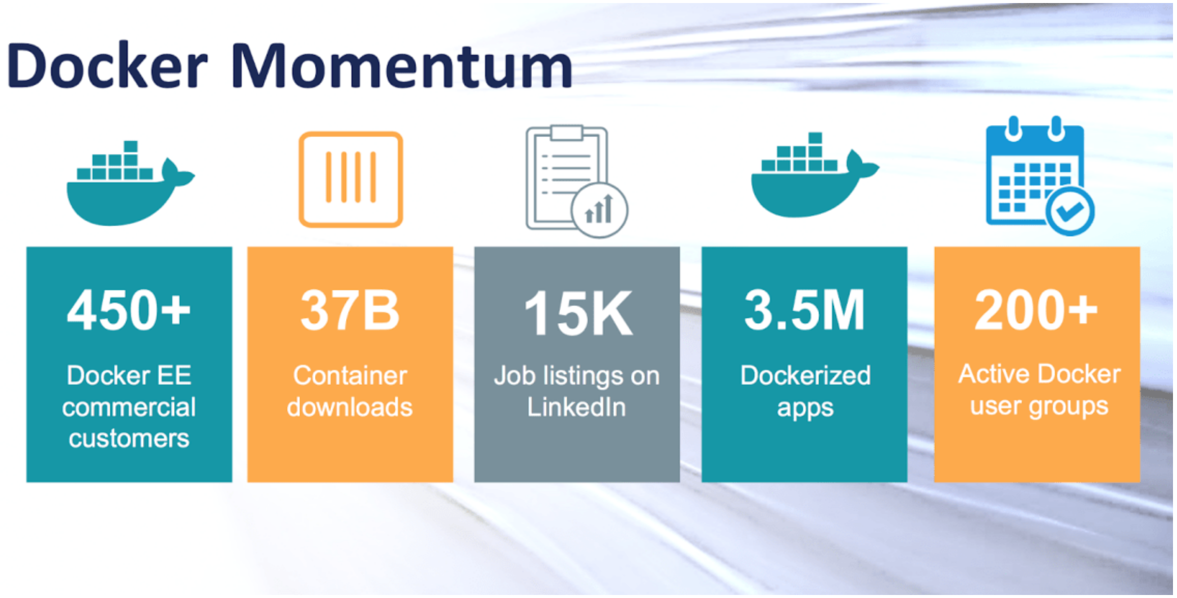
https://docs.docker.com/engine/

What is Docker?

- Docker consists of:
  - The Docker Engine - our lightweight and powerful open source containerization technology combined with a work flow for building and containerizing your applications.
  - Docker Hub - our SaaS service for sharing and managing your application stacks.

Docker inception

- **2013**: Docker comes to life as an open-source project at *dotCloud Inc.*
- **2014:** company changed name to "Docker Inc." and joined the Linux Foundation
- **2015**: tremendous increase in popularity
- **Today:**



https://blog.docker.com/2018/03/5-years-later-docker-journey/

Docker - Under the hood

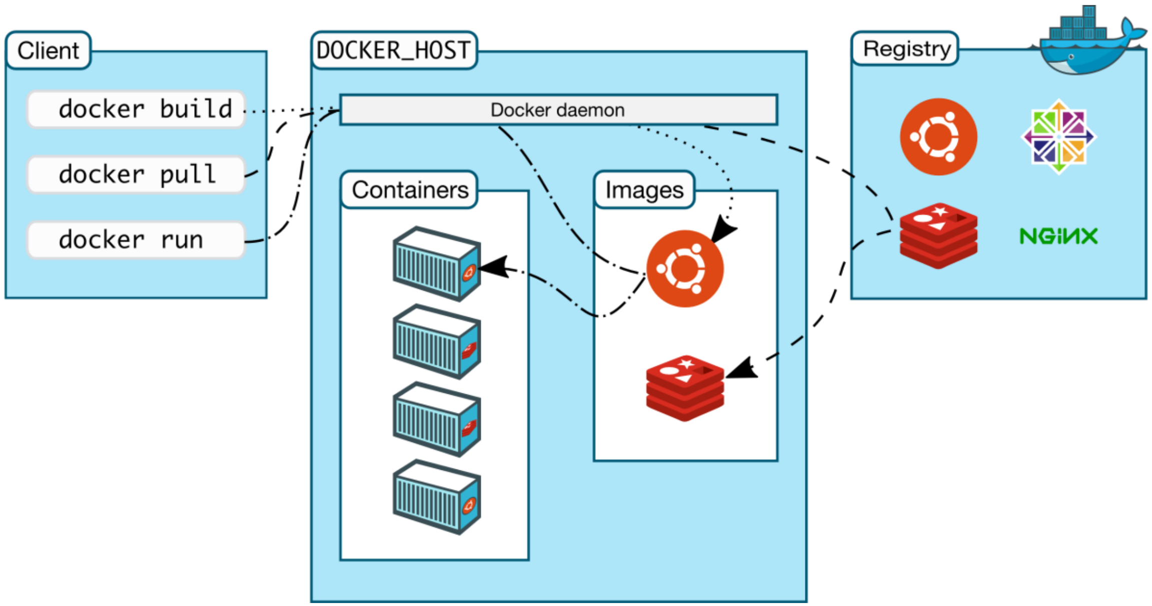
- Standard Bodies: **Open Container Initiative** (OCI), **Cloud Native Computing Foundation** (CNCF)
  - OCI Image specification
  - OCI Runtime Specification

- **runc** runtime (formerly libcontainer)
  - an abstraction/unification layer to decouple Docker from kernel-specific container features (e.g. LXC, libvirt, ...)

- The Docker **Images:**
  - **copy-on-write** filesystems (e.g. AUFS)

- The **Go programming language**
  - a statically typed programming language developed by Google with syntax loosely based on C

Docker Architecture



- **Docker daemon** – The Docker daemon listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes.
- **Docker client** – The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to the docker daemon, which carries them out.
- **Docker registries** – A Docker registry stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry. Docker registries are the distribution component of Docker.

Docker objects

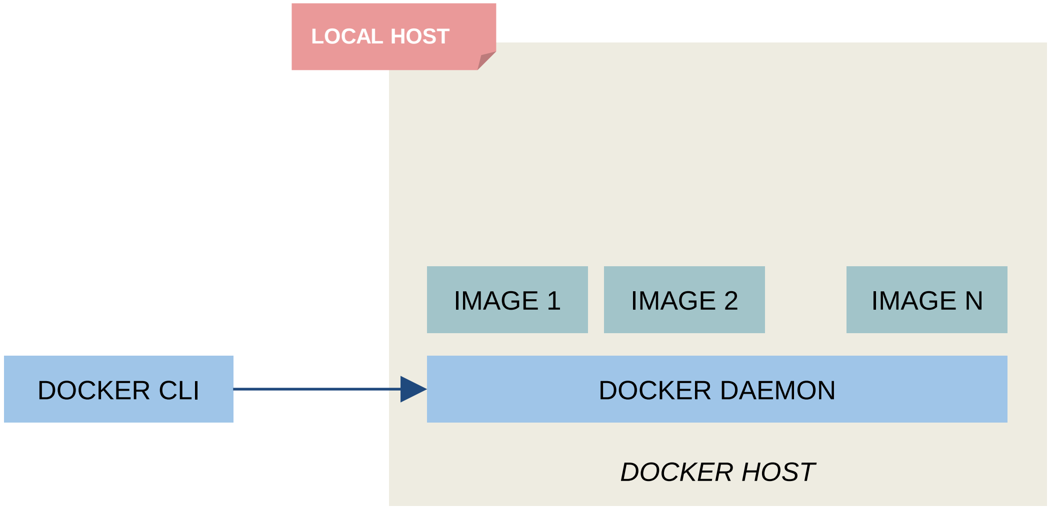
Docker images

*A Docker image is a **read-only template**. For example, an image could contain an Ubuntu operating system with Apache and your web application installed. Images are **used to create Docker containers**. Docker provides a simple way to **build new images** or **update existing images**, or you can **download** Docker images that other people have already created. Docker images are **the build component of Docker**.*

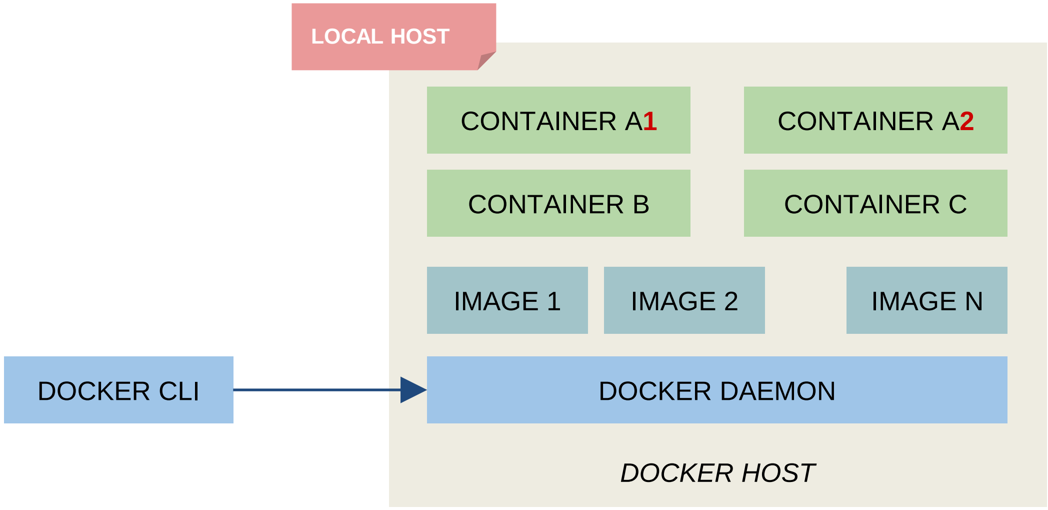
Docker containers

*Docker containers are similar to a directory. A Docker container holds **everything that is needed for an application to run**. Each container is created from a Docker image. Docker **containers can be run, started, stopped, moved, and deleted**. Each container is **an isolated and secure application platform**. Docker containers are **the run component of Docker**.*

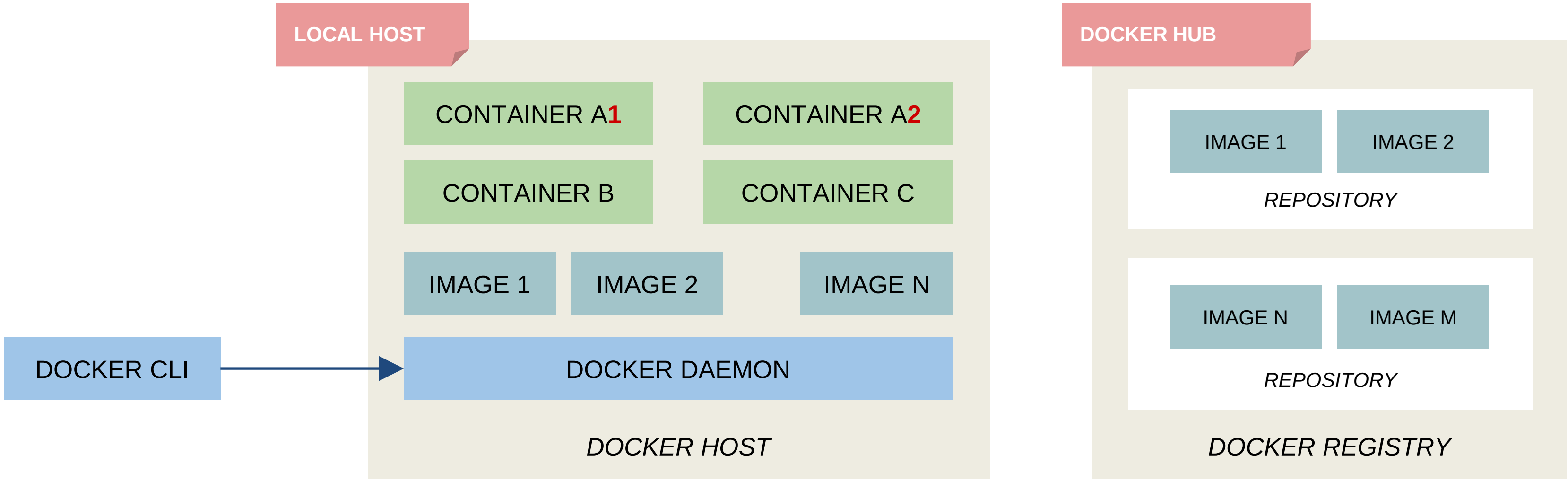
Docker components



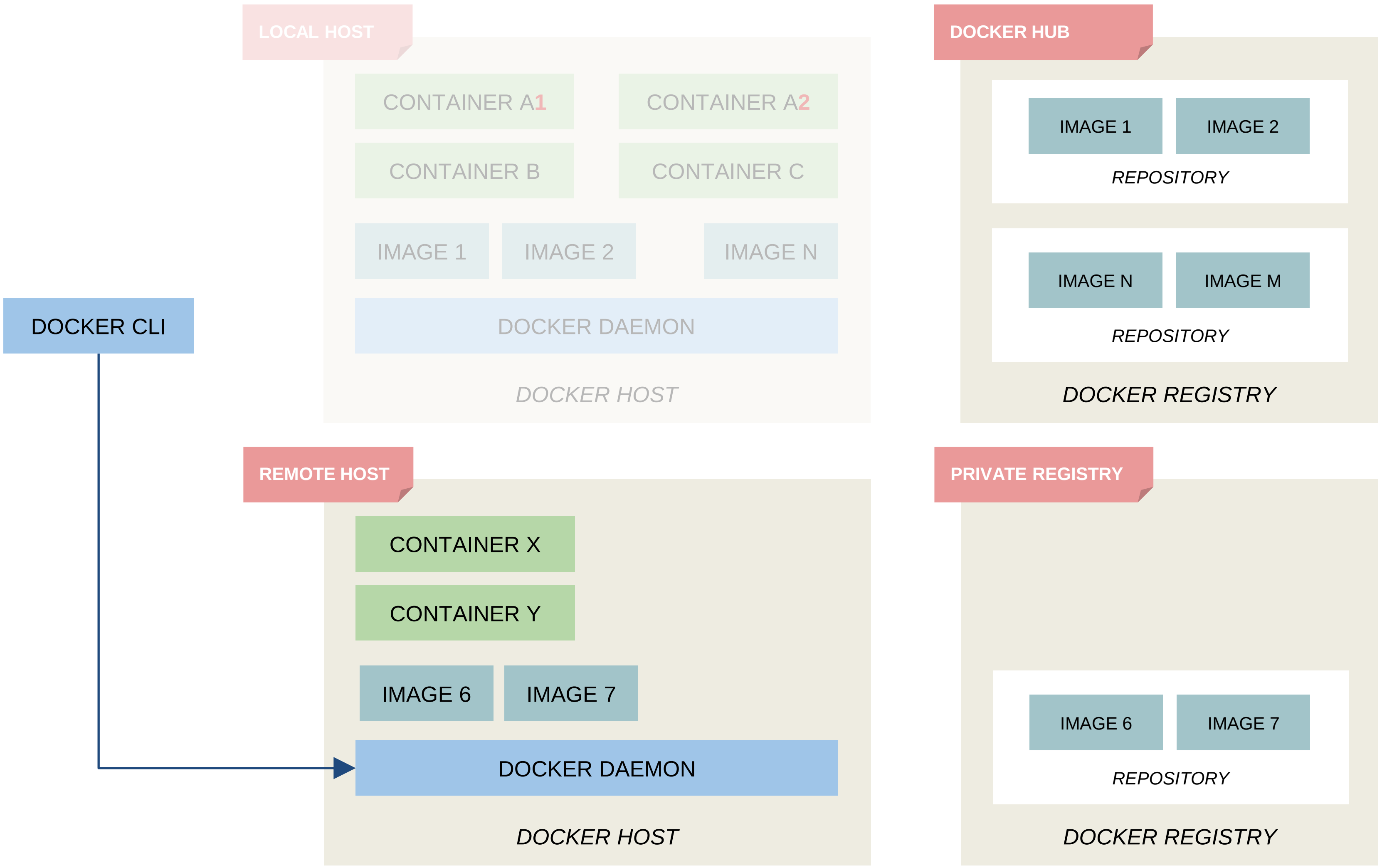
Docker components



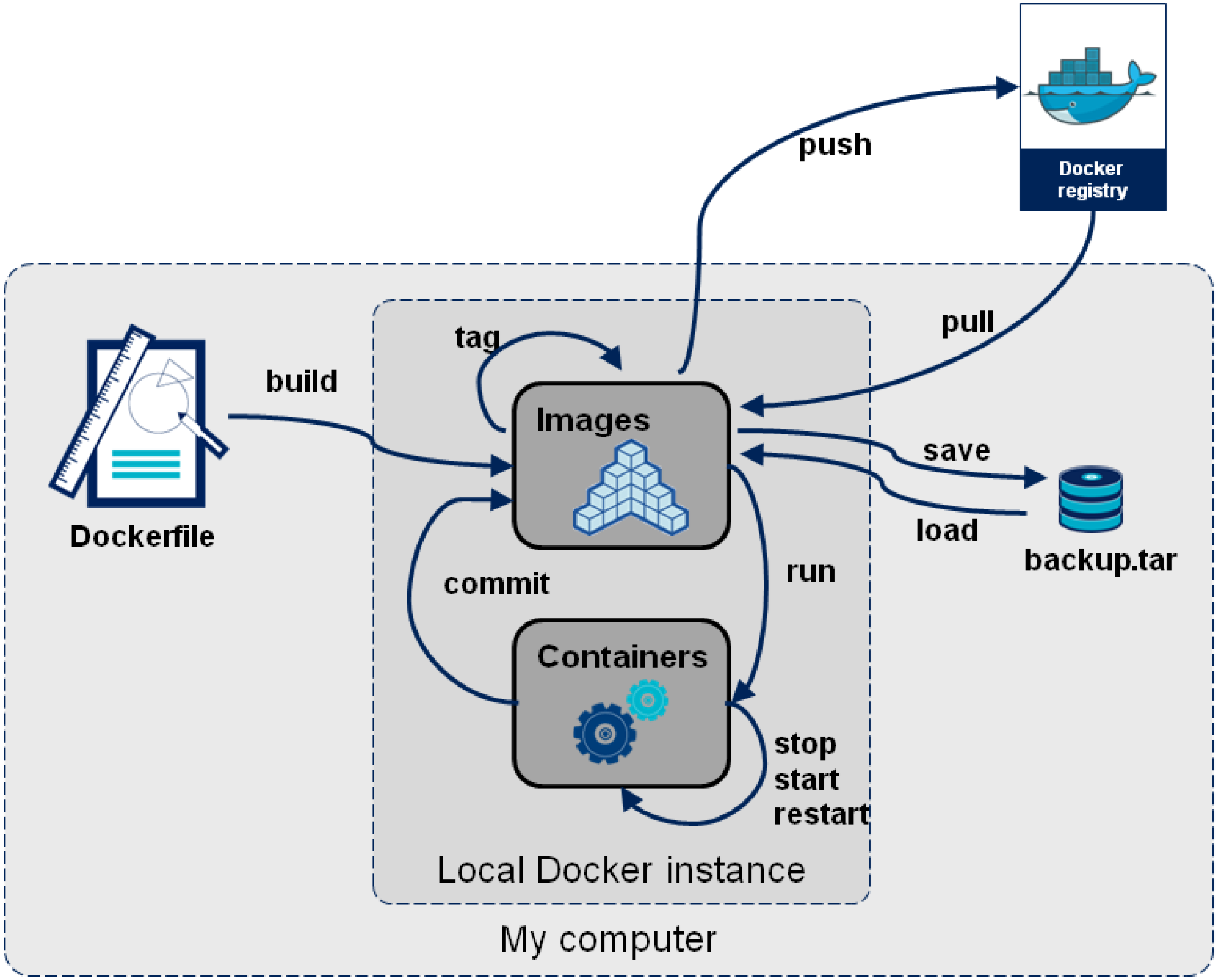
Docker components



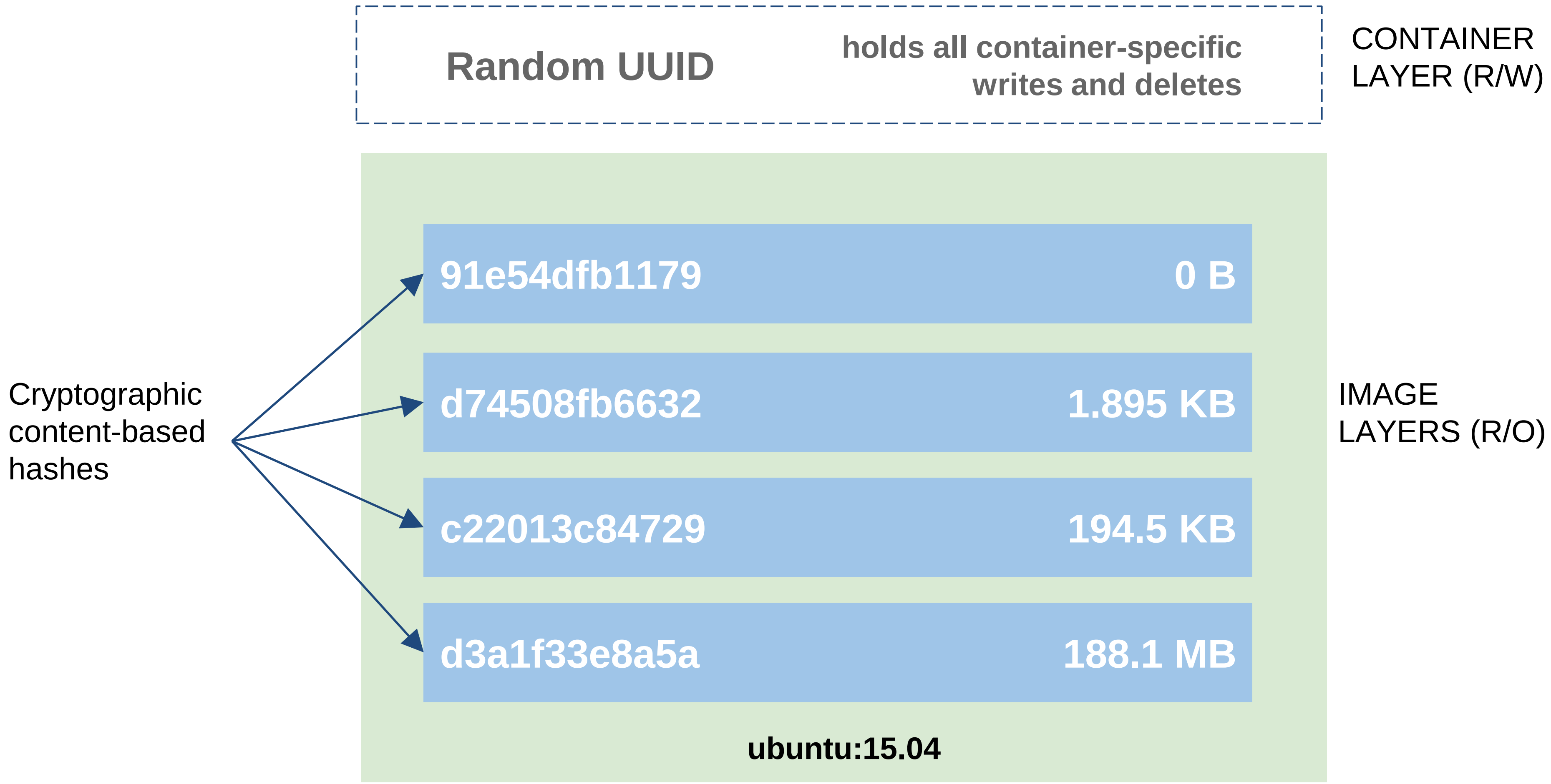
Docker components



Docker Container Lifecycle



Docker images



Docker Images

- Docker images are **read-only** stacks of **layers** → copy-on-write approach
- each layer is uniquely identified by a **cryptographic content-based hash** (>=v.1.10)
  - **collision detection** mitigation
  - strong and efficient **content comparison mechanism**

- This approach is hugely beneficial
  - **efficient disk usage**
    - each new layer keeps only differences from preceding layers
    - layers can be shared among images, e.g. "base" layers such as OS layers (fedora:latest, ubuntu:latest)
  - **ease of modification**
    - new images may be built by simply stacking new layers on top of preceding ones, leaving the below layers unmodified

Docker Images - Naming convention

[hostname[:port]]/[username]/reponame[:tag]

Hostname/port of **registry** holding the image. If missing, defaults to Docker Hub public registry.

Username. If missing, defaults to **library** username on Docker Hub, which hosts official, curated images.

Reponame. Actual image repository.

Tag. Optional image specification (e.g., version number). If missing, defaults to **latest.**

Docker CLI

- **docker pull** – get image from registry

  `$ docker pull` ***imagename***

  copy image from **registry** to **localrepo**

- **docker build** - builds an image from a Dockerfile

  `$ docker build .`

  builds a **new image** based on a **Dockerfile located** on the current directory (**.**)

  `$ docker build -t` ***imagename*** `.`

  builds a **new image** based on a **Dockerfile located** on the current directory (**.**) and **names that image as *imagename***

Docker CLI

- **docker run** - runs a command in a new container, based on a specific image

  `$ docker run hello-world`

  runs the **default command** on a newly created container, based on the **public hello-world image**

  `$ docker run -it ubuntu /bin/bash`

  runs the **bash command interactively** on a newly created container, based on the **public ubuntu image**

  `$ docker run -d tomcat:8.0`

  runs the **default command (*catalina.sh*)** on a newly created container, based on the **public tomcat V.8.0 image**, and **detaches (-d) it to background**

- **docker stop** - stops a running container

  `$ docker stop containerId`

  stops container identified by *containerId*

- **docker start** – starts a stopped container

  `$ docker start containerId`

  start container identified by *containerId*

Docker CLI

- **docker exec** – runs a command in an already **running container**

  `$ docker exec -it containerId /bin/bash`

  runs the **bash command interactively** on container ***containerId***

- **docker container** – manage container

  `$ docker container comando`

  - List containers

    `$ docker container ls` — `docker ps`

    lists **running** containers

    `$ docker container ls -a` — `docker ps -a`

    lists all containers (including stopped ones)

  - Remove container

    `$ docker container rm containerName`

    remove container **containerName** — `docker rm containerName`

Docker CLI

- **docker image** – manage image

  $ docker image comando

  - List containers

    $ docker image ls **docker images**

    lists **images**

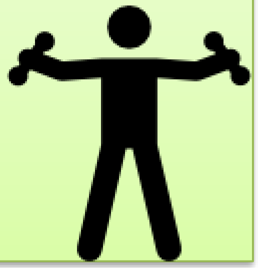
  - Remove image

    $ docker image rm imageName

    remove image **imageName** **docker rmi imageName**

Docker images

```
# Browse https://hub.docker.com/_/httpd/

docker pull httpd
docker history httpd
docker run httpd
docker ps
docker stop ???
docker rm ???

# Browse https://hub.docker.com/explore
```



Dockerfile example - PostgreSQL

```
FROM ubuntu
MAINTAINER SvenDowideit@docker.com

RUN apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys B97B0AFCAA1A47F044F244A07FCC7D46ACCC4CF8

RUN echo "deb http://apt.postgresql.org/pub/repos/apt/ precise-pgdg main" > /etc/apt/sources.list.d/pgdg.list
RUN apt-get update && apt-get install -y python-software-properties software-properties-common postgresql-9.3 postgresql-client-9.3
postgresql-contrib-9.3

USER postgres

RUN    /etc/init.d/postgresql start &&\
    psql --command "CREATE USER docker WITH SUPERUSER PASSWORD 'docker';" &&\
    createdb -O docker docker

RUN echo "host all  all    0.0.0.0/0  md5" >> /etc/postgresql/9.3/main/pg_hba.conf
RUN echo "listen_addresses='*'" >> /etc/postgresql/9.3/main/postgresql.conf

EXPOSE 5432

VOLUME  ["/etc/postgresql", "/var/log/postgresql", "/var/lib/postgresql"]

CMD ["/usr/lib/postgresql/9.3/bin/postgres", "-D", "/var/lib/postgresql/9.3/main", "-c",
"config_file=/etc/postgresql/9.3/main/postgresql.conf"]
```

Dockerfile Reference

- **FROM**: sets the **base image** for subsequent instructions
- **MAINTAINER**: reference and credit to image author
- **RUN**: runs a command and commits changes to a layer on top of previous image layers; the committed image will be visible to the next steps in the Dockerfile
- **ADD**: copies files from the source on the host (or remote URL) into the container's filesystem destination
- **COPY**: copies files from the source on the host into the container's filesystem destination (no URL, no automatic archive expansion support)
- **CMD**: sets the default command for an executing container
- **ENTRYPOINT:** sets/overrides the default entrypoint that will (optionally) execute the provided CMD
- **ENV**: sets environment variables
- **EXPOSE**: instructs Docker daemon that containers based on the current image will listen on the specified **network port**
- **USER**: sets the user name or UID to use when running the image and for any RUN, CMD and ENTRYPOINT instructions that follow it in the Dockerfile
- **VOLUME**: creates a mount point for external data (from native host or other containers)
- **WORKDIR**: sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile
- **LABEL**: adds metadata to an image

Dockerfile reference - CMD vs ENTRYPOINT

Both CMD and ENTRYPOINT instructions define what command gets executed when running a container. There are few rules that describe their co-operation.

- Dockerfile should specify at least one of CMD or ENTRYPOINT commands.
- ENTRYPOINT should be defined when using the container as an executable.
- CMD should be used as a way of defining default arguments for an ENTRYPOINT command or for executing an ad-hoc command in a container.
- CMD will be overridden when running the container with alternative arguments

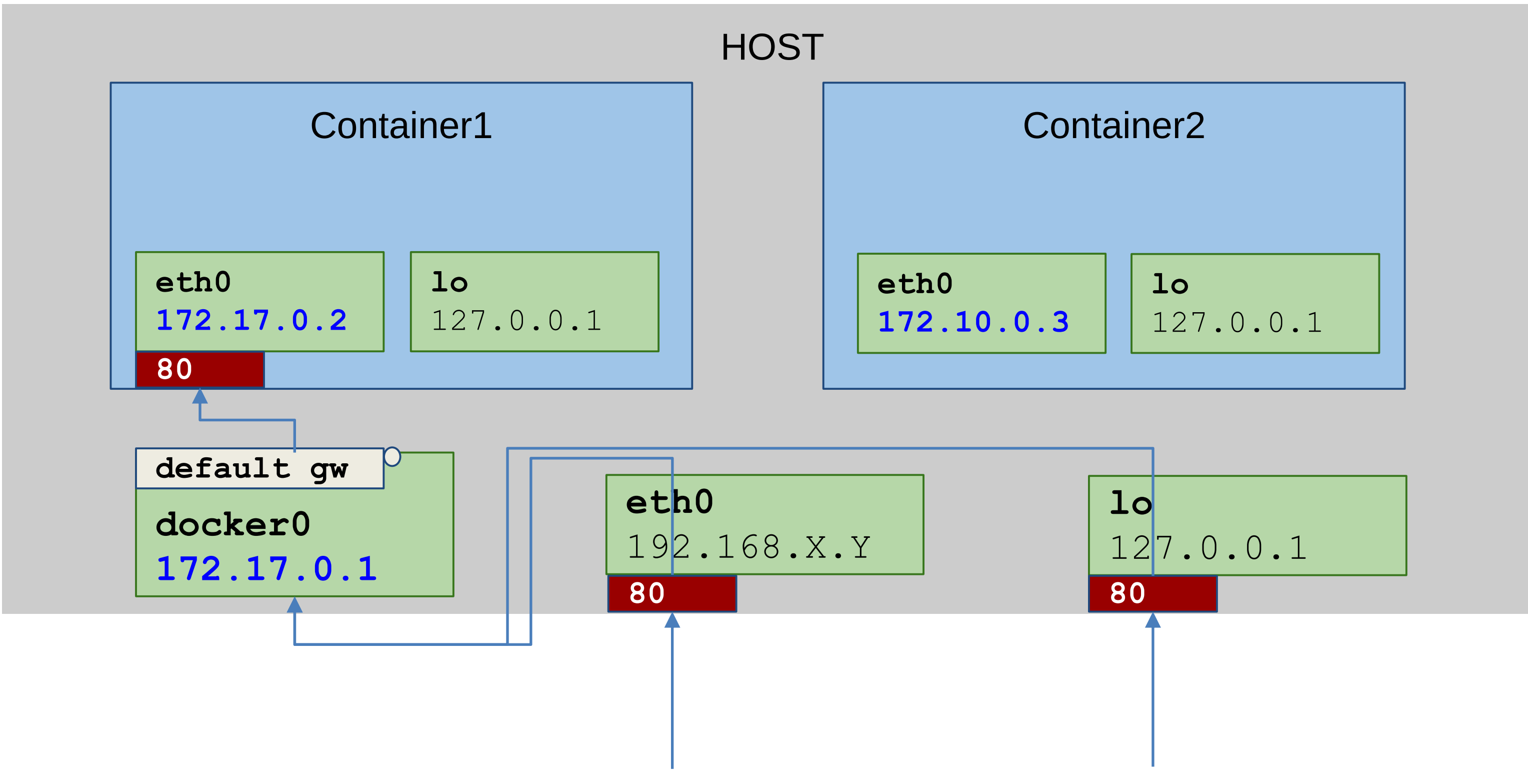
	No ENTRYPOINT	ENTRYPOINT [“entry_s1”, “entry_s2”]	ENTRYPOINT entry_s1 entry_s2
No CMD	error, not allowed	entry_s1 entry_s2	/bin/sh -c entry_s1 entry_s2
CMD [“cmd_s1”, “cmd_s2”]	cmd_s1 cmd_s2	entry_s1 entry_s2 cmd_s1 cmd_s2	/bin/sh -c entry_s1 entry_s2
CMD cmd_s1 cmd_s2	/bin/sh -c exec_cmd p1_cmd	entry_s1 entry_s2 /bin/sh -c exec_cmd p1_cmd	/bin/sh -c entry_s1 entry_s2

Docker networking

- docker networking provides full isolation for containers
- isolation can be overwritten to make containers communicate with each other
- docker engine creates **3 default networks**
  - **bridge** → **default network** for containers; points to **docker0** (virtual) network interface
  - **none** → container lacks network interfaces; only **loopback address** is available
  - **host** → adds container to the host network stack

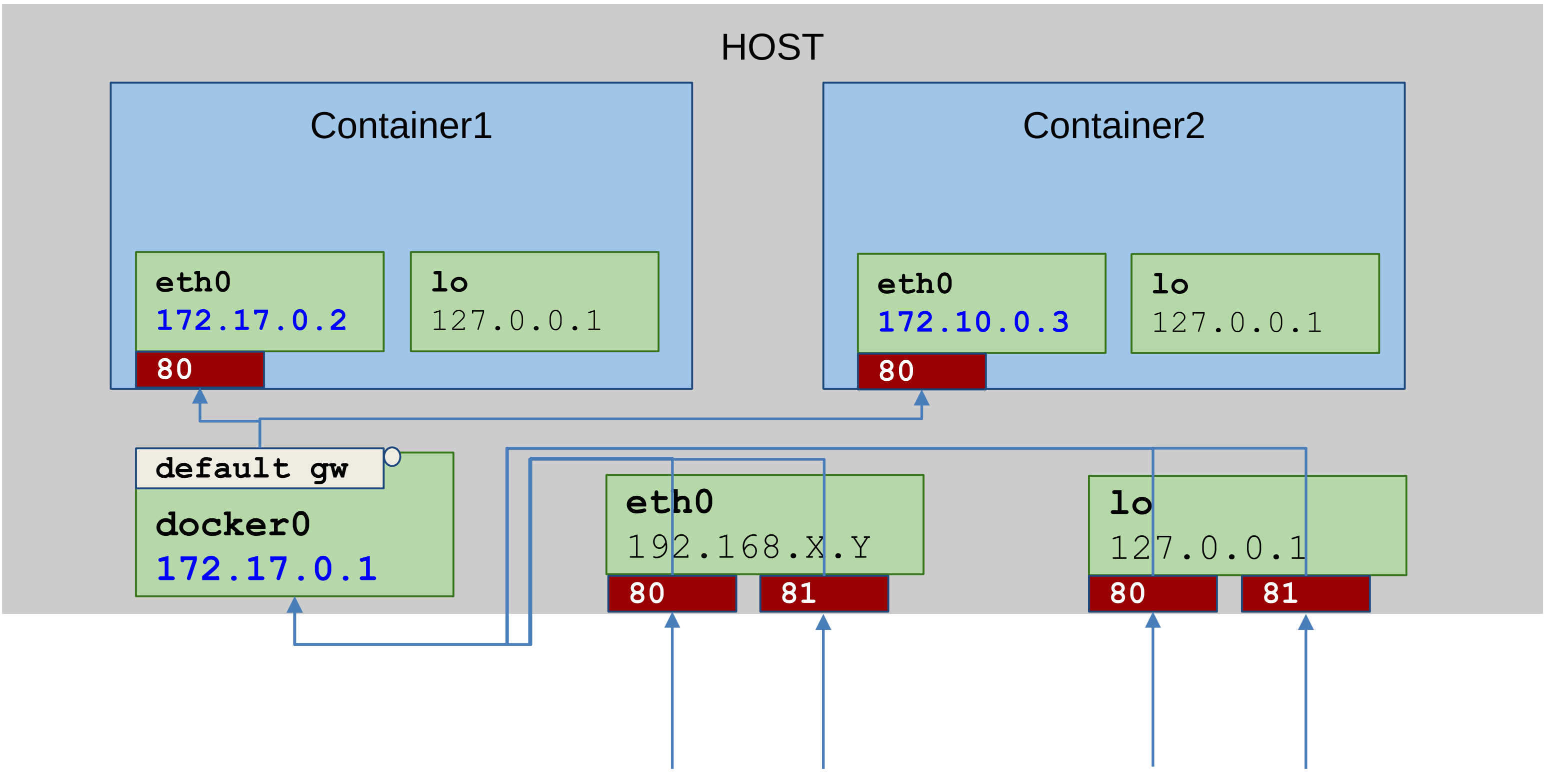
- docker allows users to **create user-defined networks**

Docker networking - port forwarding



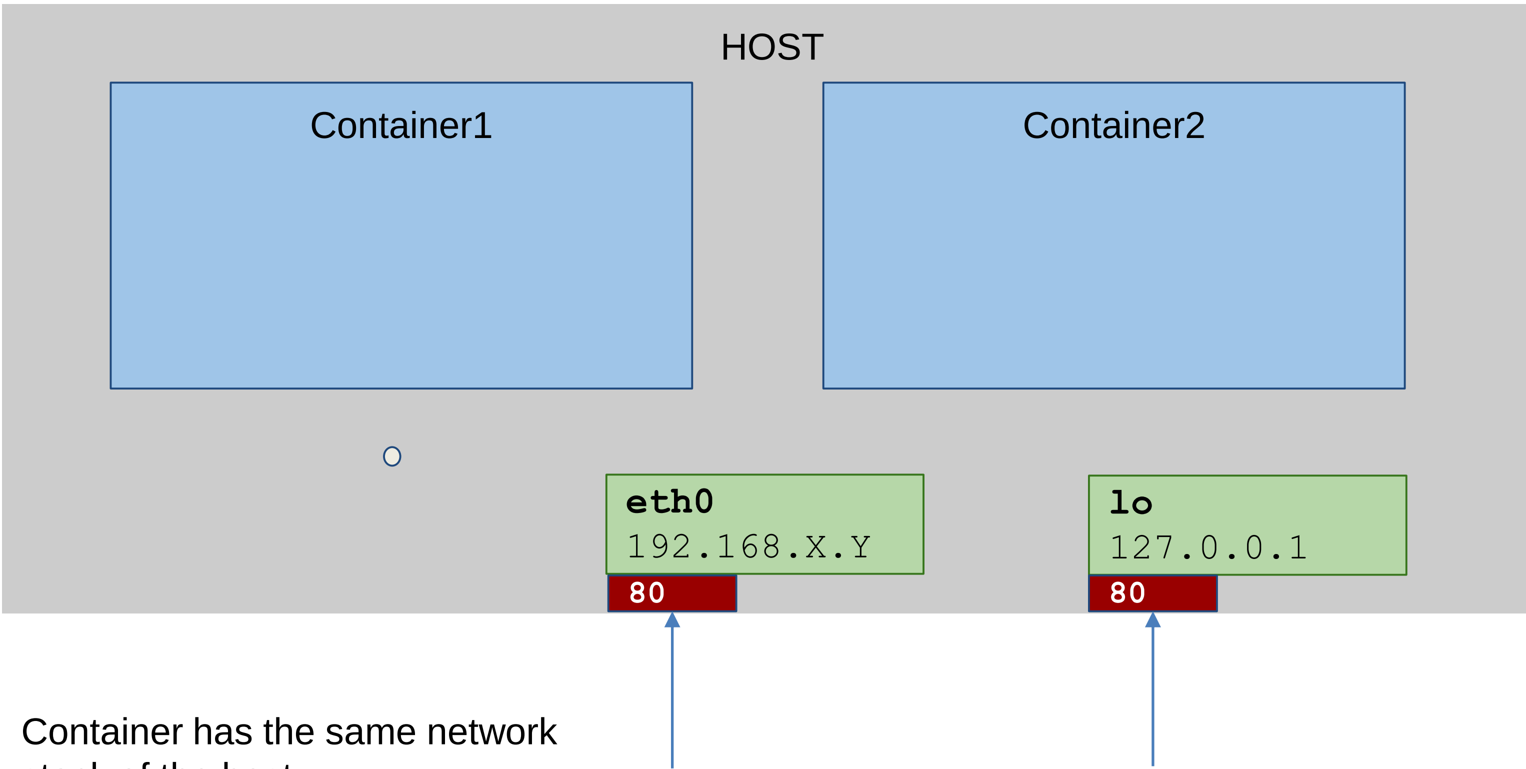

```
docker run -d -p 80:80 httpd:2.4

docker inspect --format '{{ .NetworkSettings.IPAddress }}' $(docker ps -q)
```

Docker networking - port forwarding




```
docker run -d -p 80:80 --name container1 httpd:2.4
docker run -d -p 81:80 --name container2 httpd:2.4
```

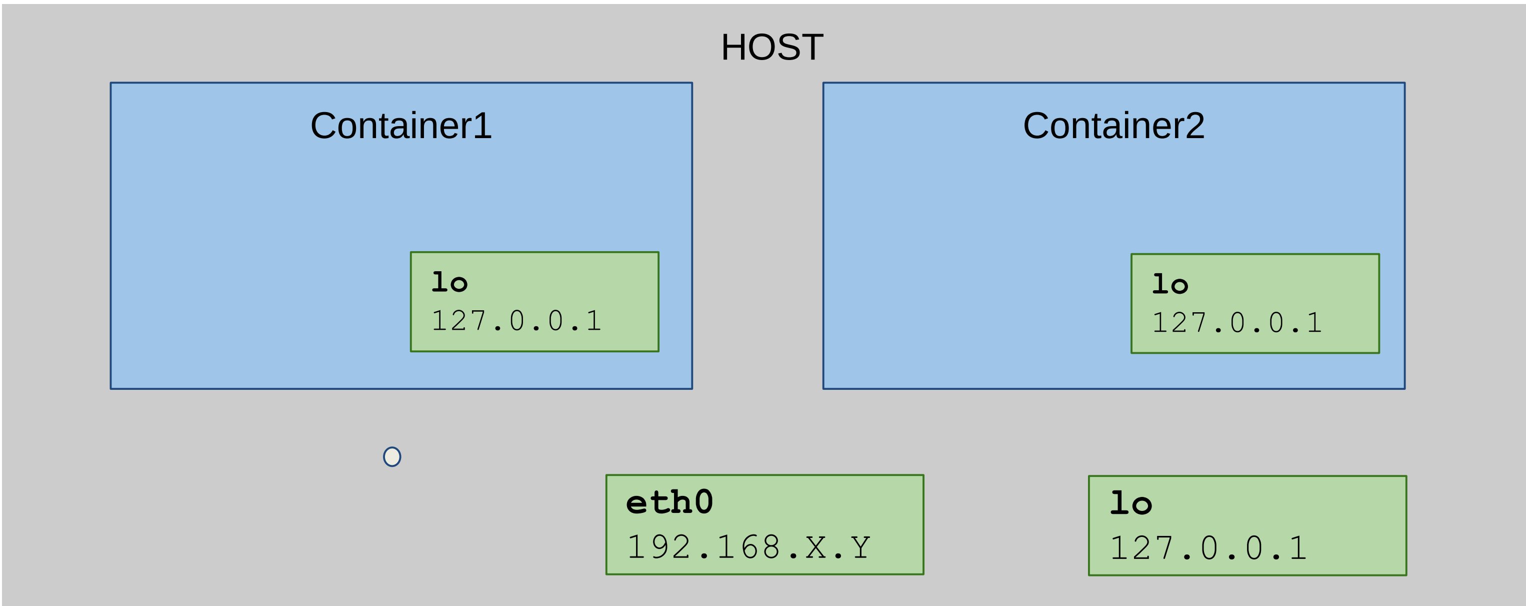
Docker networking - host



Container has the same network stack of the host

```
docker run -d --network host httpd:2.4
```

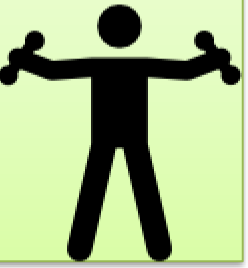
Docker networking - none



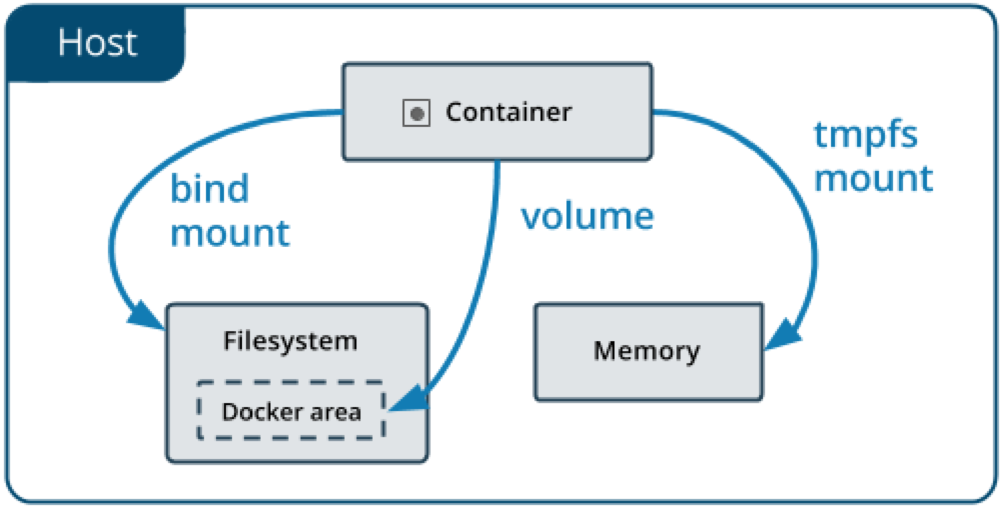

```
docker run -d --network none httpd:2.4
```

Docker Network

```
# From host
docker network ls
docker network inspect networkInterface

# From container
docker exec -it containerId /bin/bash
ip a
```



Container data persistence



- **Volumes** are stored in a part of the host filesystem which is managed by Docker (/var/lib/docker/volumes/ on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker. Volumes may be named or anonymous.
- **Bind mounts** may be stored anywhere on the host system. Non-Docker processes on the Docker host or a Docker container can modify them at any time.
- **tmpfs mounts** are stored in the host system's memory only, and are never written to the host system's filesystem.

Note: You can mount even a single file in container filesystem.

Good use cases for volumes

- **Sharing data among multiple running containers**. If you don’t explicitly create it, a volume is created the first time it is mounted into a container. When that container stops or is removed, the volume still exists. Multiple containers can mount the same volume simultaneously, either read-write or read-only. Volumes are only removed when you explicitly remove them.

- When the Docker host is not guaranteed to have a given directory or file structure. Volumes help you **decouple the configuration of the Docker host from the container runtime**.

- When you want to store your container’s data on a remote host or a cloud provider, rather than locally.

- When you need to back up, restore, or migrate data from one Docker host to another, volumes are a better choice. You can stop containers using the volume, then back up the volume’s directory (such as /var/lib/docker/volumes/<volume-name>).

Good use cases for bind mounts

- **Sharing configuration files from the host machine to containers**. For example Docker by default provides DNS resolution to containers by default, by mounting /etc/resolv.conf from the host machine into each container.

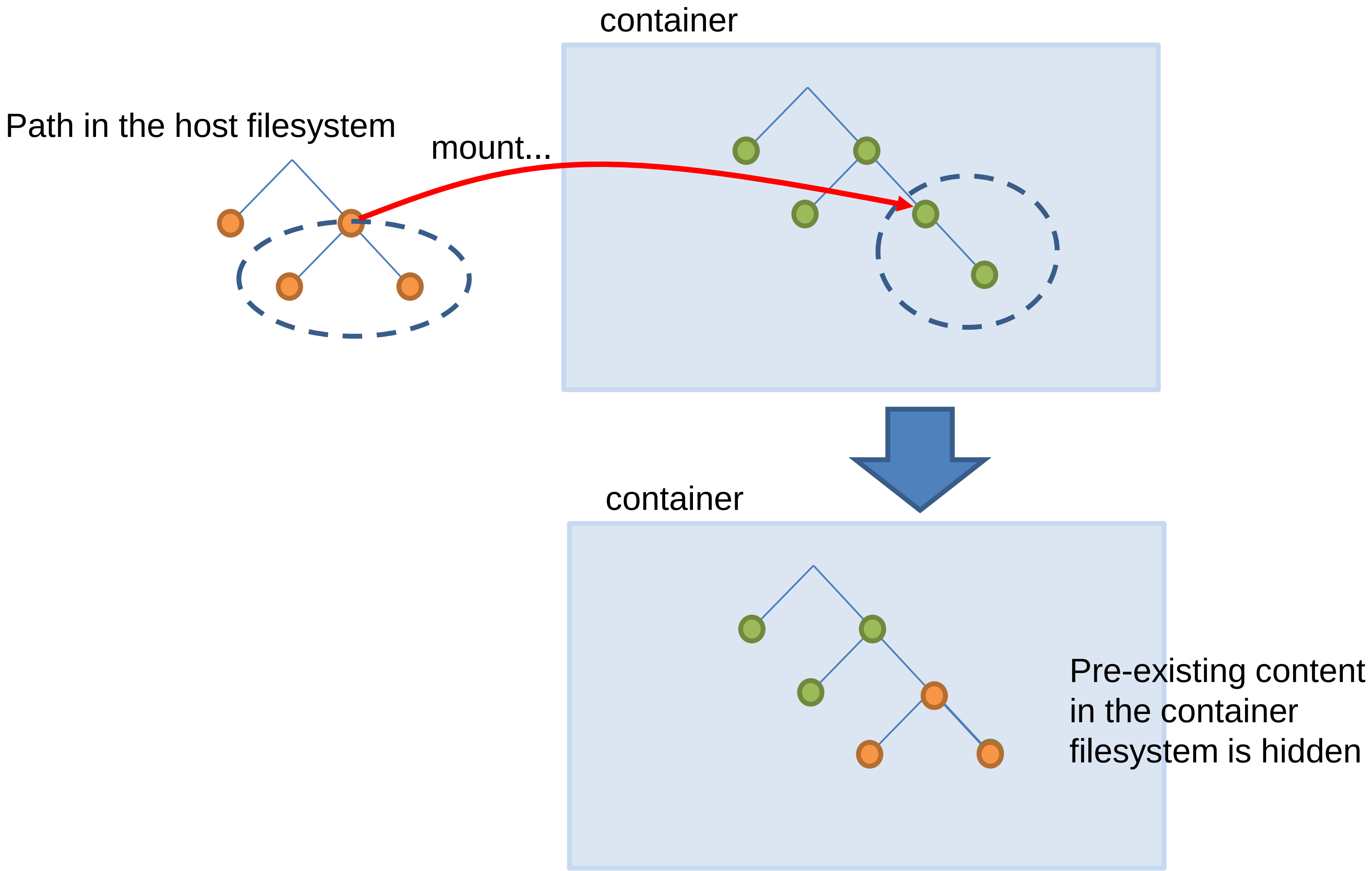
- Sharing source code or build artifacts between a development environment on the Docker host and a container. For instance, you may mount a Maven target/ directory into a container, and each time you build the Maven project on the Docker host, the container gets access to the rebuilt artifacts. **Don't use this modality in production environment (embed the artifact in the image).**

- When the file or directory structure of the Docker host is guaranteed to be consistent with the bind mounts the containers require.

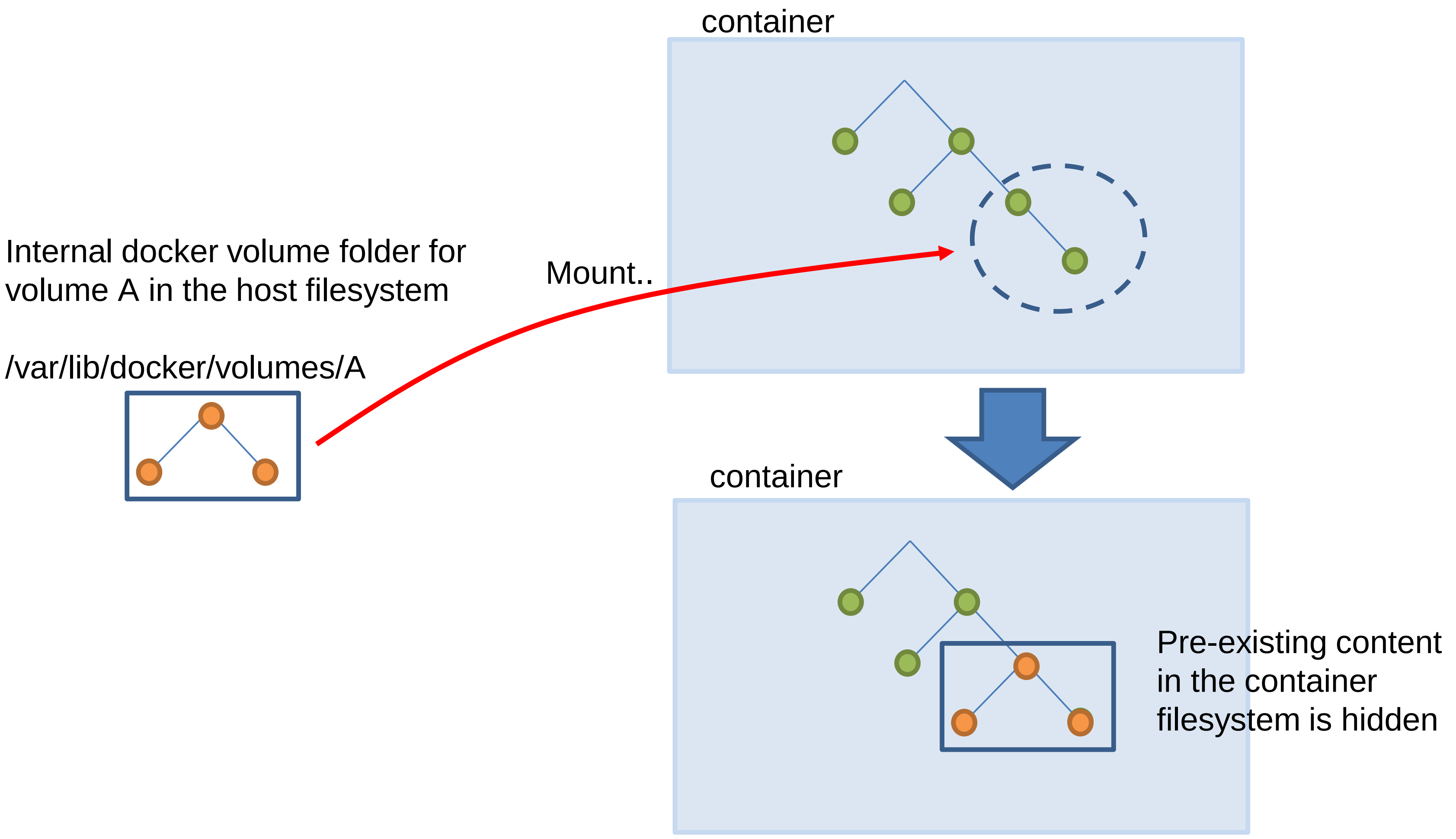
Good use cases for tmpfs mounts

- tmpfs mounts are best used for cases when you do not want the data to persist either on the host machine or within the container. This may be for security reasons or to protect the performance of the container when your application needs to write a large volume of non-persistent state data.

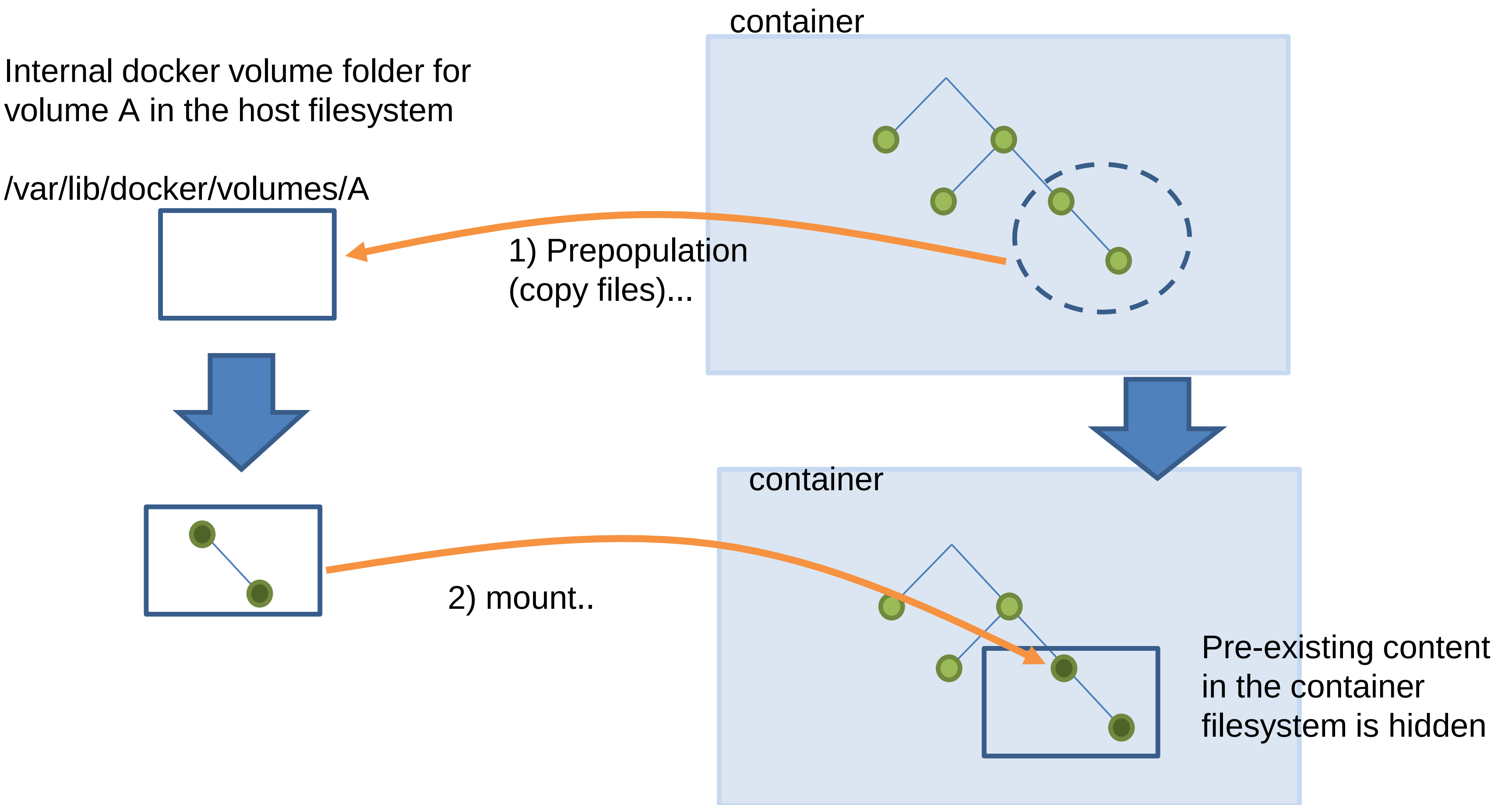
Bind mount


container
Path in the host filesystem
mount...
container
Pre-existing content
in the container
filesystem is hidden

Not-empty named/anonymous volume



Mounting empty named/anonymous volumes



NOTE: prepopulation does not happen for bind mounts!

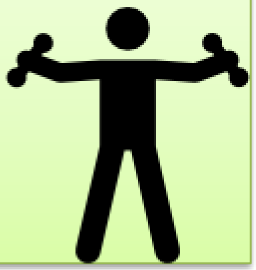
Docker volumes - container data persistence

- **Container filesystem** is visible and persistent as long as the container is available (running/stopped/restarted).
- **Docker volumes**
  - can be shared/reused among different containers
  - persist even after container deletion

```
# mounts a specific host directory (usually, in the /var/lib/docker/… FS tree)
# to /webapp mountpoint within the container
docker run -d -v /webapp tomcat:8.0

# mounts /host_fs_folder host directory to /webapp mountpoint within the container
docker run -d -v /host_fs_folder:/webapp tomcat:8.0

# create and mount a named volume
docker volume create tomcat-webapps
docker run -d -v tomcat-webapps:/webapp tomcat:8.0
```

Docker Volume

```
docker volume ls
docker volume inspect volumeName


docker volume prune
```



Agenda

1. Containers & Docker ecosystem
    1. Docker basics
    2. Docker basics - hands on
    3. Web GUIs (e.g Portainer) & Debug
    4. Docker-compose
    5. Docker-compose - hands on

1. Orchestration tools
    1. Overview
    2. Kubernetes

Docker - Hands-on

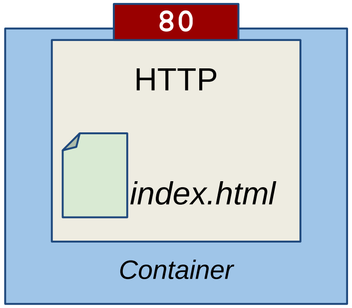
1 - Web Hello World

Goals

- HTTPD (a.k.a. APACHE) Web Server up and running on standard HTTP port 80, and host-accessible
- the default HTML page (index.html) greets users with a HELLO WORLD

Hints

- Docker Hub hosts publicly available images
- COPY statement in a Dockerfile allows to copy content from host to container filesystem




```
git clone http://git.imolinfo.it/Unibo/docker-seminar-templates.git
cd docker-seminar-templates/Exercise1-Docker/1.1-HelloWeb/
```

Docker - Hands-on

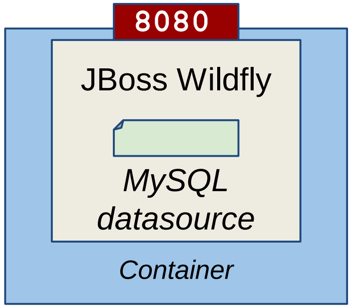
2 - Real-world JEE Application Server

Goals

- JBoss **Wildfly** JEE AS Server up and running on standard HTTP port 8080, and host-accessible
- **MySQL datasource** configured
- check datasource connectivity via CLI

Hints

- Docker Hub hosts publicly available images
- default JBoss Wildfly image comes with a stock configuration file that uses an embedded database
  → **example configuration files** are provided in the exercise template
- COPY statement in a Dockerfile allows to copy content from host to container filesystem



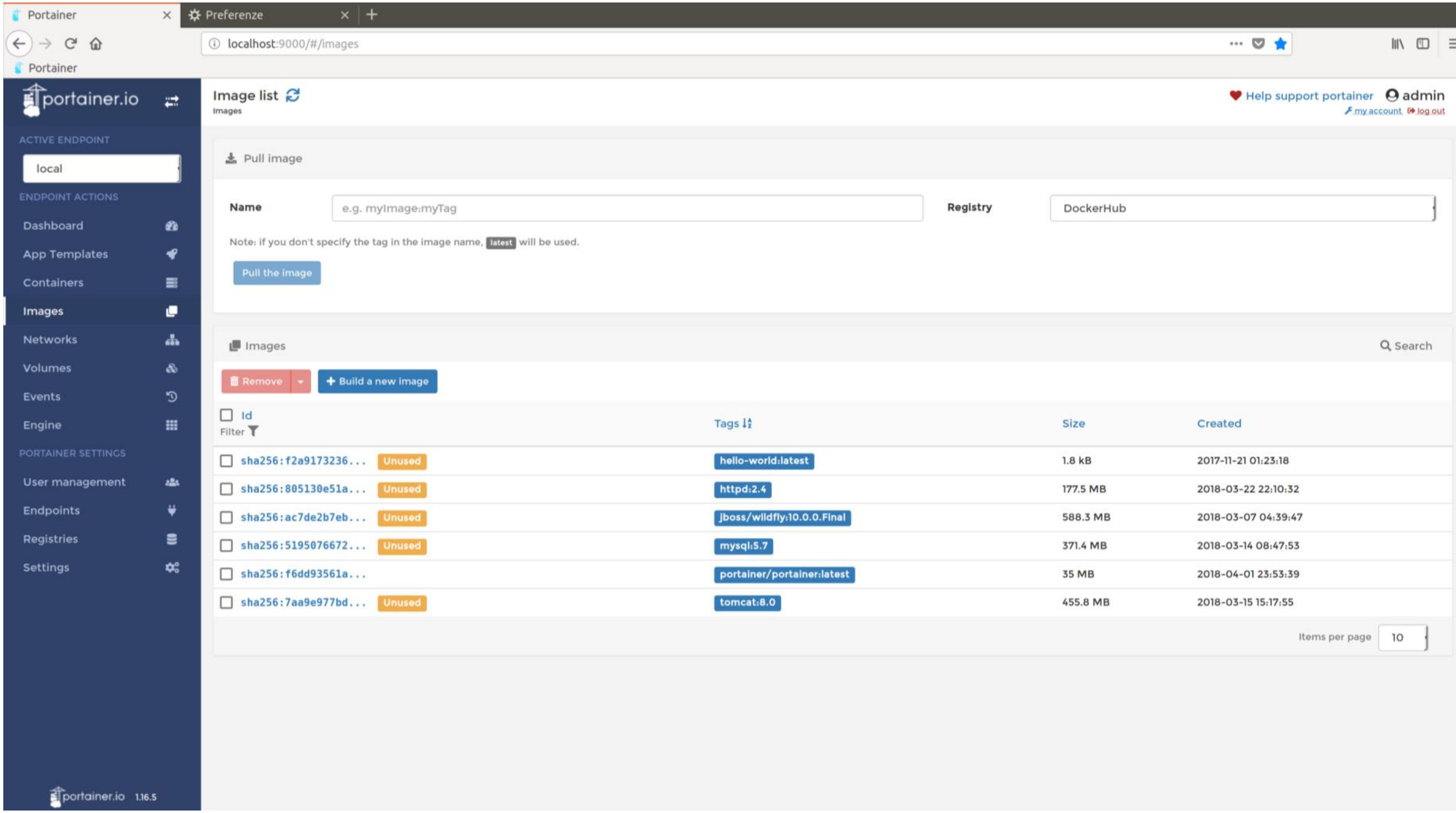

```
git clone http://git.imolinfo.it/Unibo/docker-seminar-templates.git
cd Exercise1-Docker/1.2-WildflyMysql/
```

Agenda

1. Containers & Docker ecosystem
    1. Docker basics
    2. Docker basics - hands on
    3. Web GUIs (e.g Portainer) & Debug
    4. Docker-compose
    5. Docker-compose - hands on

1. Orchestration tools
    1. Overview
    2. Kubernetes

Portainer




```
docker run -d -p 9000:9000 --restart always -v /var/run/docker.sock:/var/run/docker.sock -v
/opt/portainer:/data portainer/portainer
```

Debugging running container

```
# Copy file from container
docker cp containerId:containerFilePath hostFilePath

# Copy file into container
docker cp hostFilePath containerId:containerFilePath

# Install tool via yum/apt if package manager is installed into container:
apt update && apt install iputils-ping -y                # ping
apt update && apt install iproute2 -y                    # ip
apt update && apt install net-tools -y                   # netstat

# elsewhere copy file:
docker cp /bin/less containerId:/bin/less                # less
docker cp /bin/ping containerId:/bin/ping                # ping
docker cp /bin/ip containerId:/bin/ip                    # ip
```

Other techniques: nsenter – https://stackoverflow.com/a/40352004

Container Restart policy

```
docker update --restart=flag containerId
```

Available flag:

- **no** - Do not automatically restart the container. (the default)
- **on-failure** - Restart the container if it exits due to an error, which manifests as a non-zero exit code.
- **unless-stopped** - Restart the container unless it is explicitly stopped or Docker itself is stopped or restarted.
- **always** - Always restart the container if it stops.

Agenda

1. Containers & Docker ecosystem
   1. Docker basics
   2. Docker basics - hands on
   3. Web GUIs (e.g Portainer) & Debug
   4. Docker-compose
   5. Docker-compose - hands on

2. Orchestration tools

Docker shortcomings

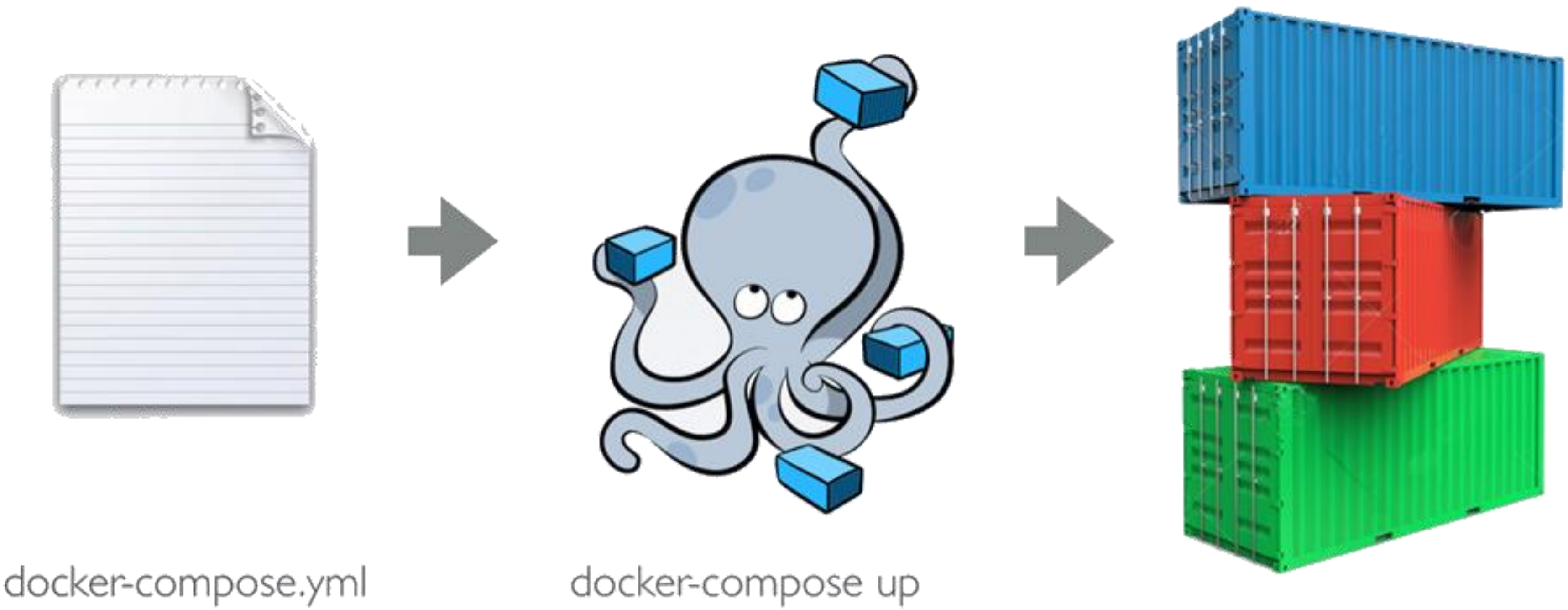
Complex distributed applications are typically composed of a number of interacting services and layers (e.g.: database, cluster of application servers, load balancers, etc…)

Docker promotes encapsulation of reusable pieces of application logic

- **coarse-grained** (e.g., 1 container - N services) containers are easily manageable but fall short on reusability
- **fine-grained** (e.g., 1 container - 1 service) containers are highly reusable (thus generally preferable) but require a higher level of orchestration (e.g., starting up all containers serving an application, in the right order)

Right service granularity requires tradeoff between **modularity and manageability**

Docker-compose



Docker-compose

Compose is a tool for defining and running multi-container Docker applications.
With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Docker-compose allows to orchestrate fine-grained (e.g., single service) containers into a complex application

- **single** container composition **definition file** (docker-compose.yml)
- **single command to build and run** a composition of containers
- containers still available as **single atomic units of deployment**

https://docs.docker.com/compose/

Docker-compose CLI

- **up**
  `$ docker-compose up`
  builds, (re)creates, starts, and attaches to containers for a service; services definition is expected to be on a docker-compose.yml file in the current directory
  `$ docker-compose up -d`
  builds, (re)creates, starts, and attaches to containers for a service; services definition is expected to be on a docker-compose.yml file in the current directory (.); containers run in **background**
- **build**
  `$ docker-compose build`
  builds/rebuilds the services (containers) specified on a docker-compose.yml file in the current directory (.)
- **start**
  `$ docker-compose start`
  starts existing containers for a service composition
- **ps**
  `$ docker-compose ps`
  show running containers

Docker-compose CLI

- **down**

  `$ docker-compose down -v --rmi all`

  removes containers network, volumes (-v) and images (`--rmi all`)
- **create**

  `$ docker-compose create`

  Create containers but do not start them
- **exec**

  `$ docker-compose exec [options] SERVICE COMMAND [ARGS...]`

  Execute a command on an existing service
- **run**

  ```
  $ docker-compose run [options] [-v VOLUME...] [-p PORT...] [-e
       KEY=VAL...] SERVICE [COMMAND] [ARGS...]
  ```

  Create a **new container** for that service and execute a command on it

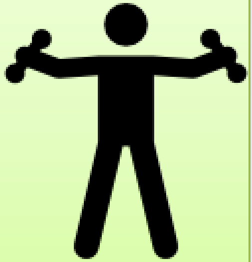
Docker-compose.yml - Wordpress

```yaml
version: '3'

services:
   db:
     image: mysql:5.7
     volumes:
       - dbdata:/var/lib/mysql
     restart: always
     environment:
       MYSQL_ROOT_PASSWORD: somewordpress
       MYSQL_DATABASE: wordpress
       MYSQL_USER: wordpress
       MYSQL_PASSWORD: wordpress

   wordpress:
     depends_on:
       - db
     image: wordpress:latest
     ports:
       - "8000:80"
     restart: always
     environment:
       WORDPRESS_DB_HOST: db:3306
       WORDPRESS_DB_USER: wordpress
       WORDPRESS_DB_PASSWORD: wordpress
volumes:
    dbdata:
```

Docker-compose.yml - Wordpress

```
docker system events
```

```
cd /home/manager/Docker/work2

docker-compose up -d

docker container ls
docker network ls
```



Docker-compose networking

Docker-compose networking extends docker networking model as follows

- a new, **reserved virtual network** is created to host all containers (services) declared in the composition
- containers within the new virtual network can **reach** each other via their **logical service names**

Suppose we are building the previous docker-compose.yml file from /home/user/**wordpressmysql/docker-compose.yml**

- A network called **wordpressmysql_default** is created
- A container is created using **db** configuration. It joins the network **wordpressmysql_default** under the name **db**.
- A container is created using **wordpress** configuration. It joins the network **wordpressmysql_default** under the name **wordpress**.
- Both containers can reach each other via **db**, **wordpress** names

Agenda

1. Containers & Docker ecosystem
    1. Docker basics
    2. Docker basics - hands on
    3. Web GUIs (e.g Portainer) & Debug
    4. Docker-compose
    5. Docker-compose - hands on

2. Orchestration tools

Docker-compose: Hands-on

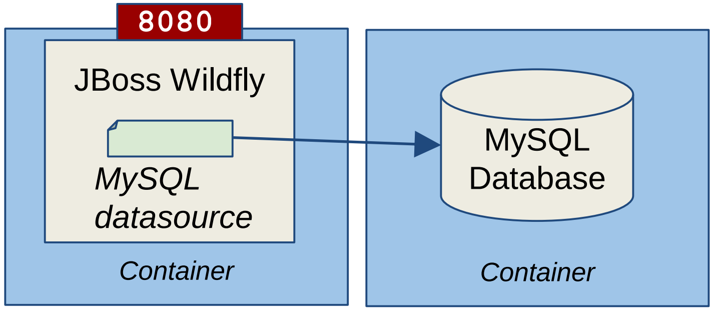
2.1 - Real-world JEE Application Server (cont'd...)

Goals

- JBoss **Wildfly** JEE AS Server up and running on standard HTTP port 8080, and host-accessible
- **MySQL datasource** configured
- **MySQL server** up and running on standard MySQL port

Hints

- [Docker Hub]
- **docker-compose** to ease service composition/orchestration



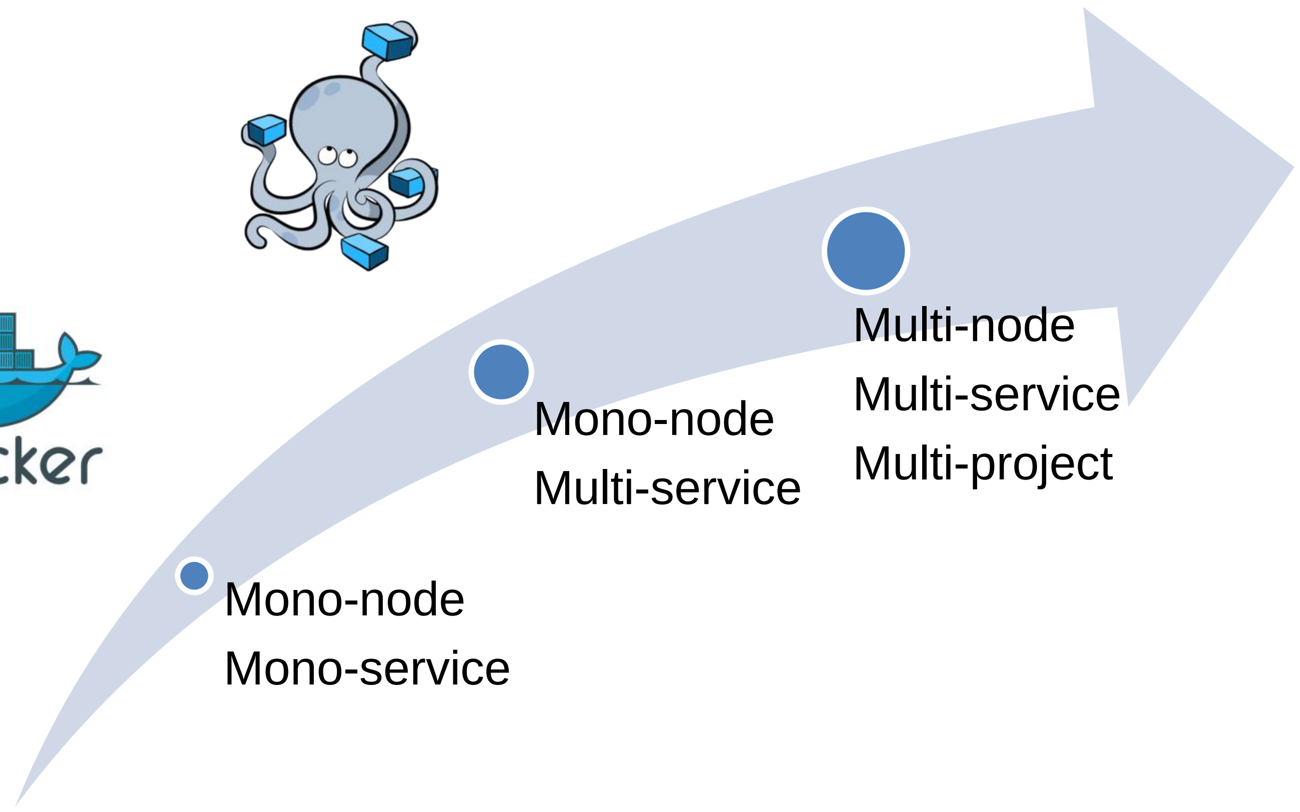

```
git clone http://git.imolinfo.it/Unibo/docker-seminar-templates.git

cd Exercise2-DockerCompose/
```

Agenda

1. Containers & Docker ecosystem
    1. Docker basics
    2. Docker basics - hands on
    3. Web GUIs (e.g Portainer) & Debug
    4. Docker-compose
    5. Docker-compose - hands on

2. Orchestration tools

Orchestrator - Kubernetes



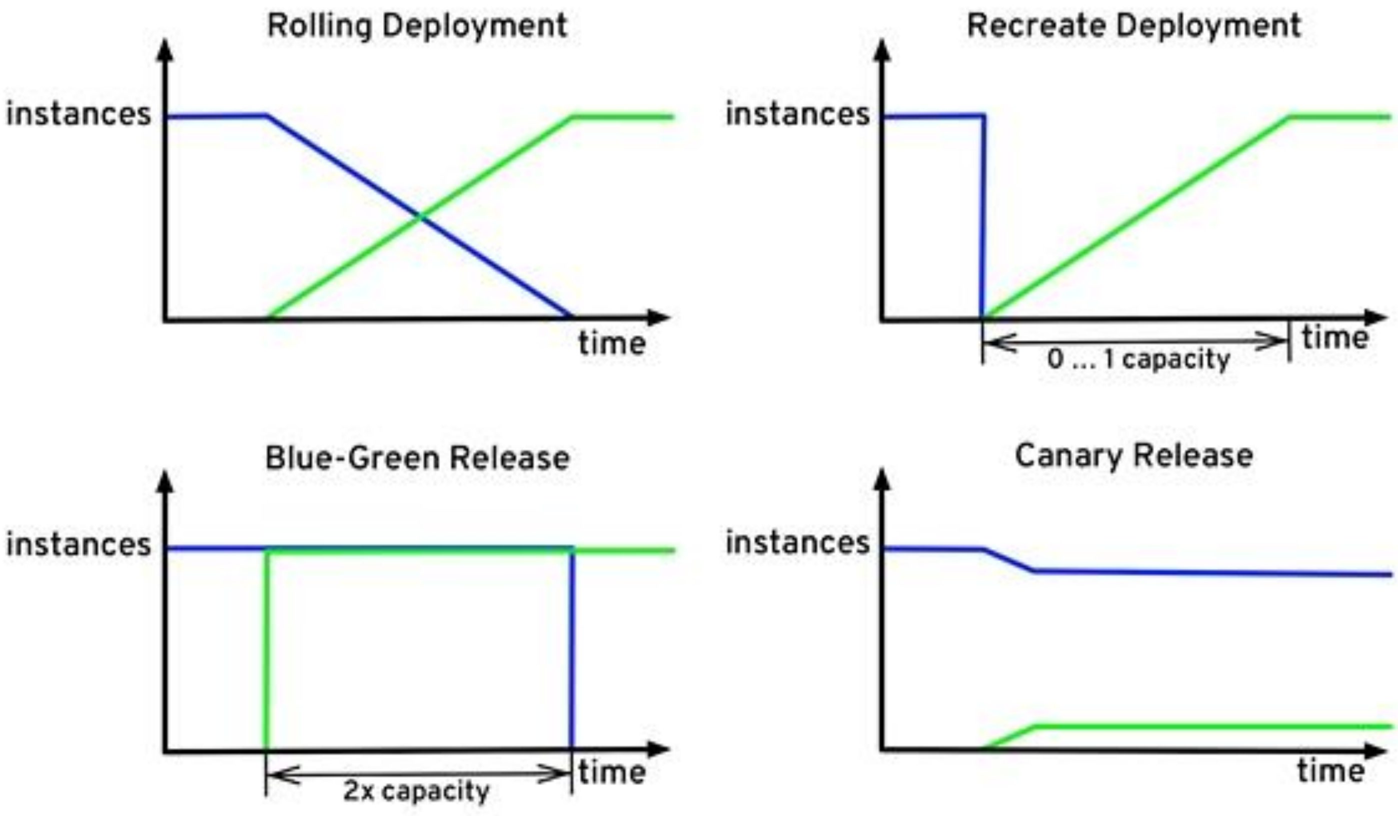
Orchestration

Application orchestration → integrating two or more applications and/or services together to automate a process, or synchronize data in real-time (ESB)

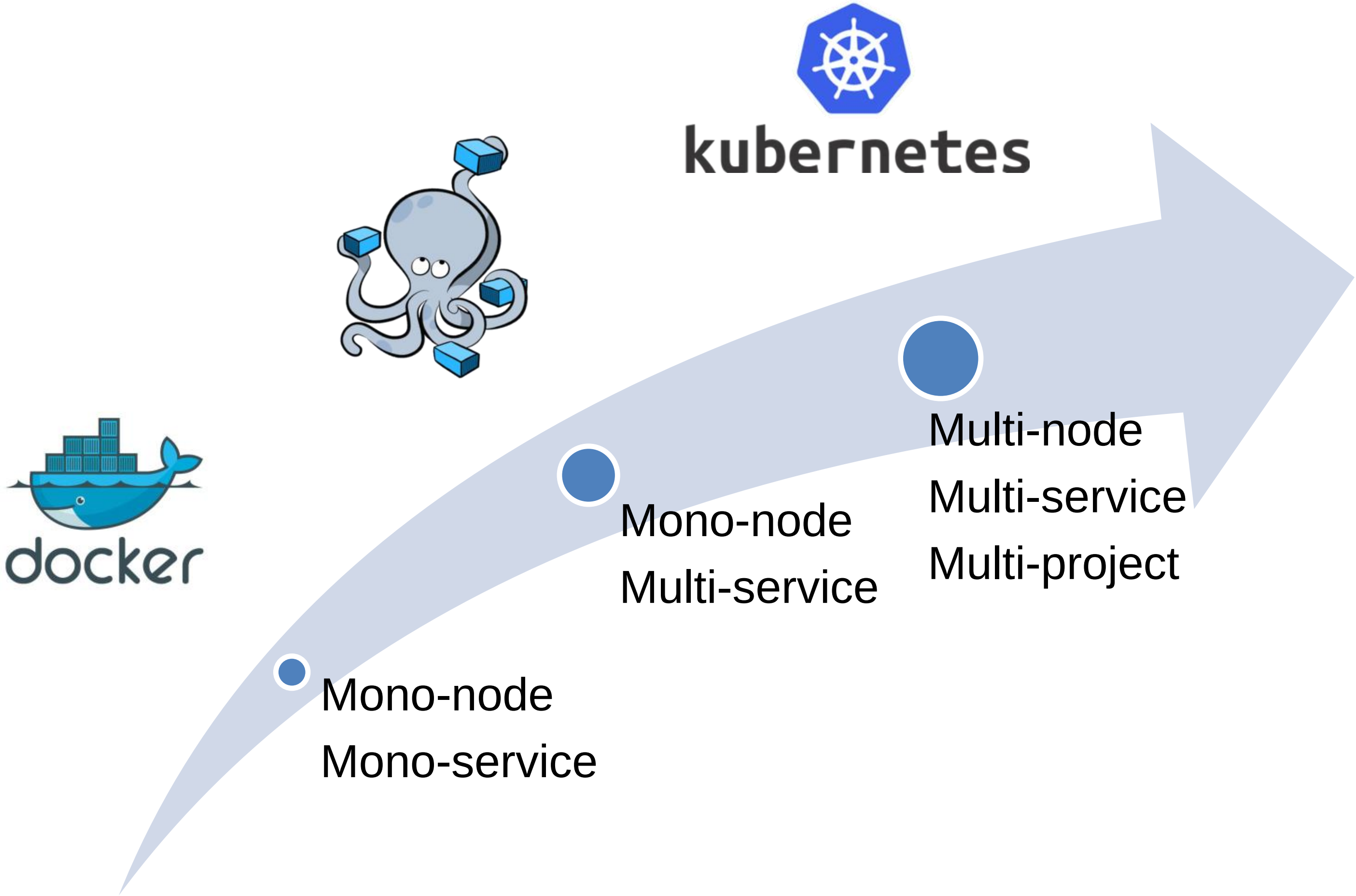
Infrastructure orchestration (Kubernetes, Swarm, Mesos)

- Manage the lifecycle of execution environments (containers) in a cluster
- check the state of the containers in the nodes
- simplify the implementation of :
  - High availability (HA) with load balancers
  - advanced deployment strategies:
    - blue/green deployment, canary release, ...
    - rollback in case of problems
  - health checks

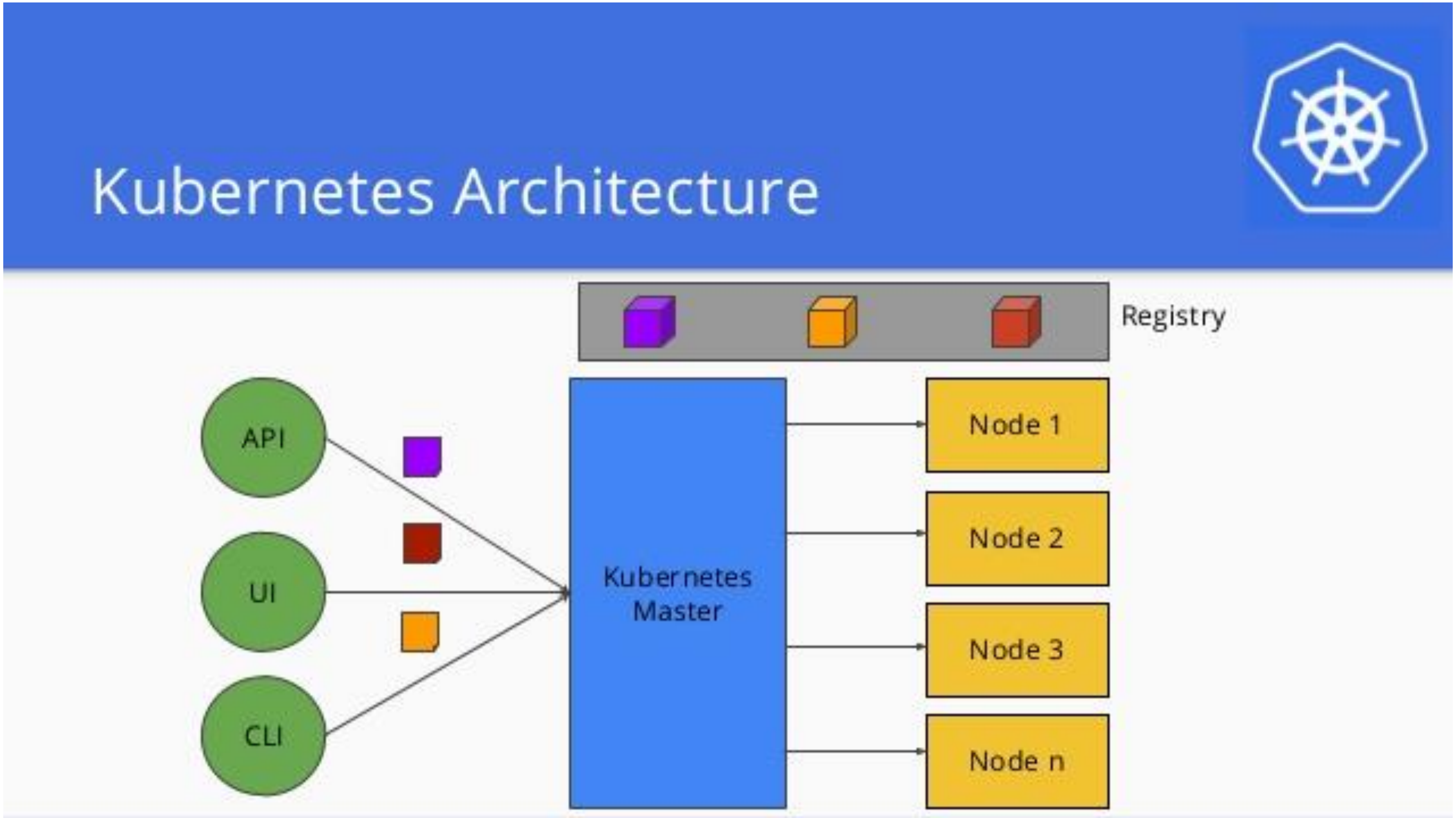
Deployment strategies



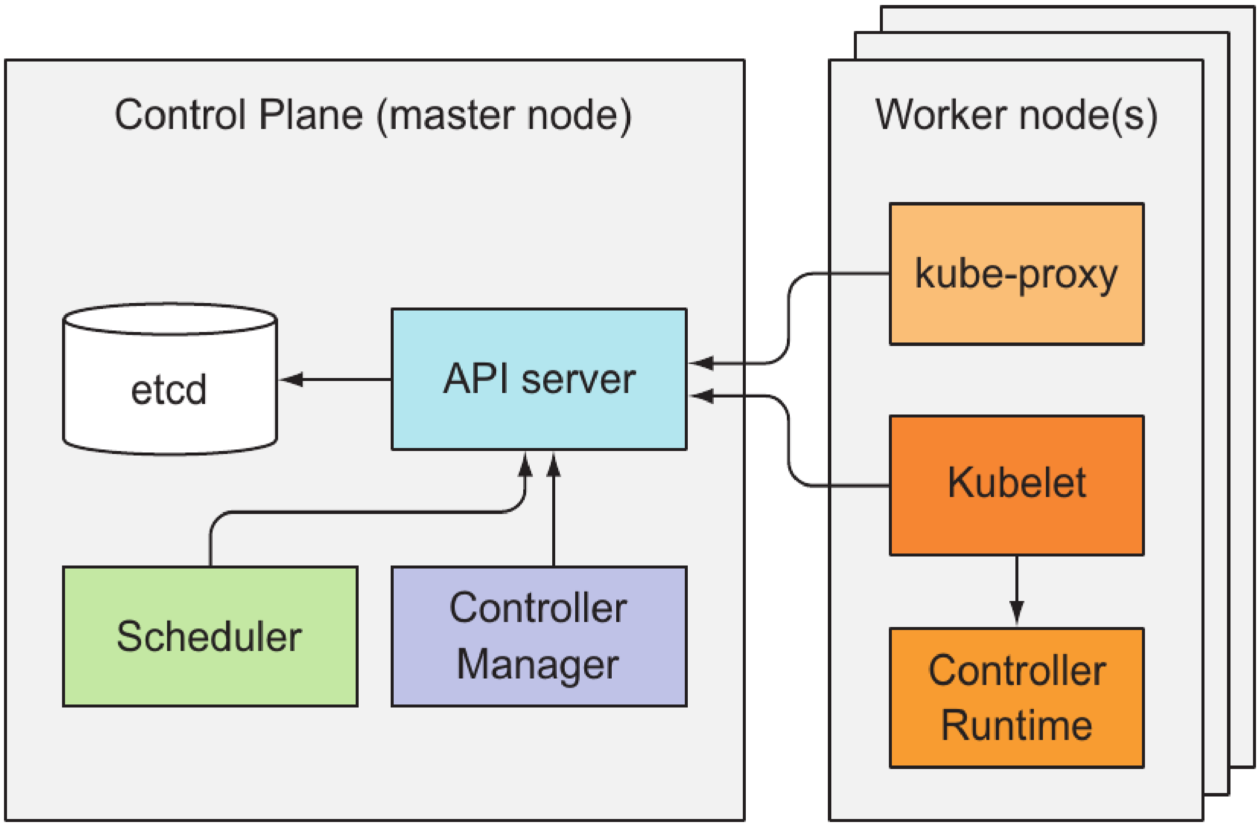
Orchestrator - Kubernetes



Kubernetes – Architecture



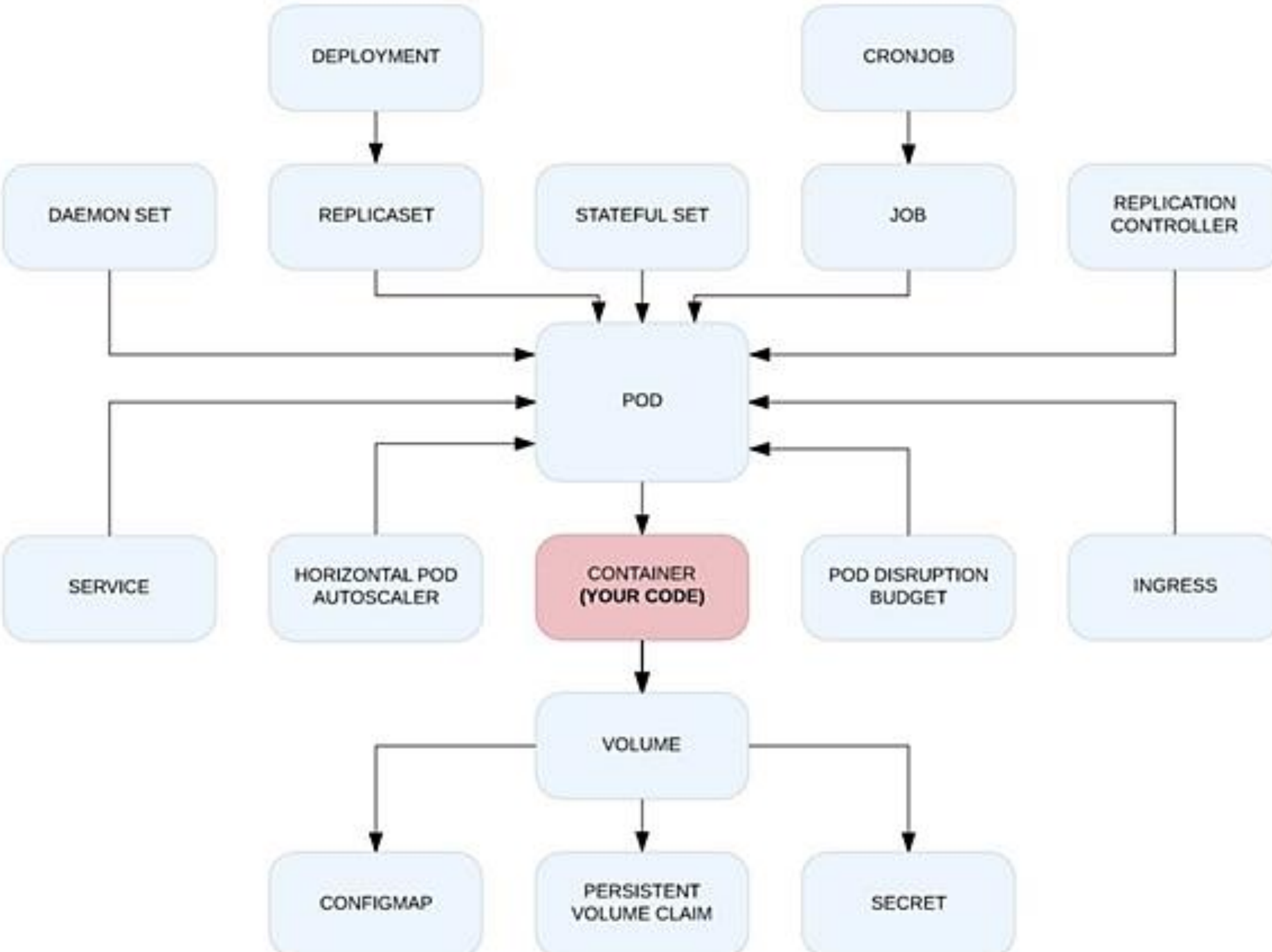
Components of a Kubernetes cluster



Components of a Kubernetes cluster

- In the «Kubernetes master» cluster node:
  - kube-apiserver
  - kube-controller-manager
  - kube-scheduler
- In each non-master cluster node:
  - **kubelet**, which communicates with the Kubernetes Master.
  - **kube-proxy**, a network proxy which reflects Kubernetes networking services on each nod
- Kubernetes Control Plane:
  - record of all of the Kubernetes Objects in the system (etcd)
  - runs continuous control loops to manage those objects' state

Kubernetes resources



Kubernetes resources

- Namespace - for isolating resource pools
- Pod - the deployment unit for a related collection of containers
- Service - service discovery and load balancing primitive
- Volume - for persistent storage
- Controllers (higher-level abstractions):
  - ReplicationController/ReplicaSet - maintain N pod instances
  - DaemonSet - maintainer 1 pod instance in each node
  - Job – an atomic unit of work scheduled asynchronously
  - CronJob - an atomic unit of work scheduled at a specific time in the future or periodically
  - Deployment - manage rollout/rollback of deployments
  - StatefulSet - manage “pets” (pods with identity)

- ConfigMap - distributing configuration data across service instances

- Secret - management of sensitive configuration data

Pod

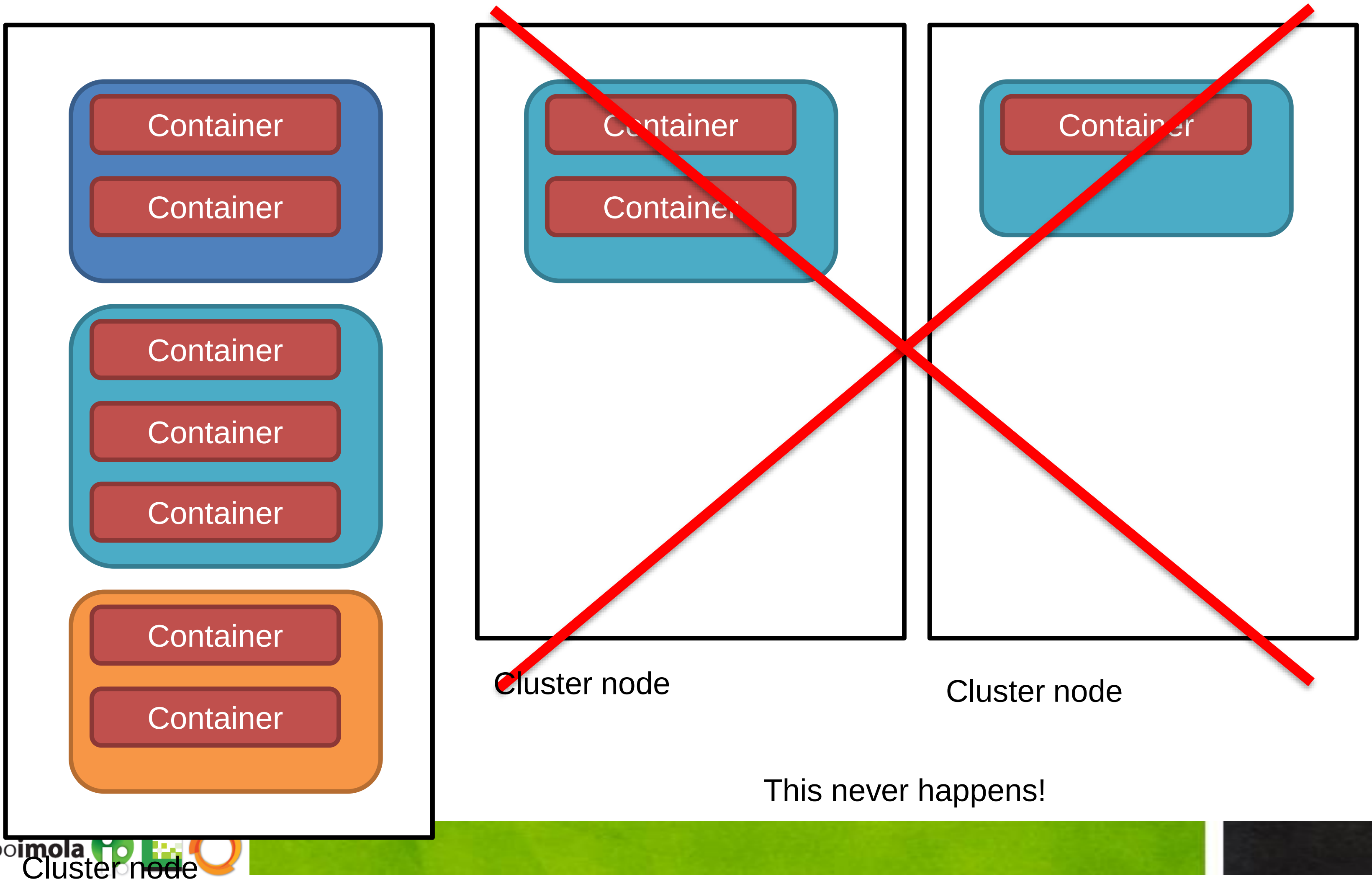
- A Pod is the **basic building block** of Kubernetes

- A Pod encapsulates an application container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run.
- A Pod represents a unit of deployment: a single instance of an application in Kubernetes, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources.

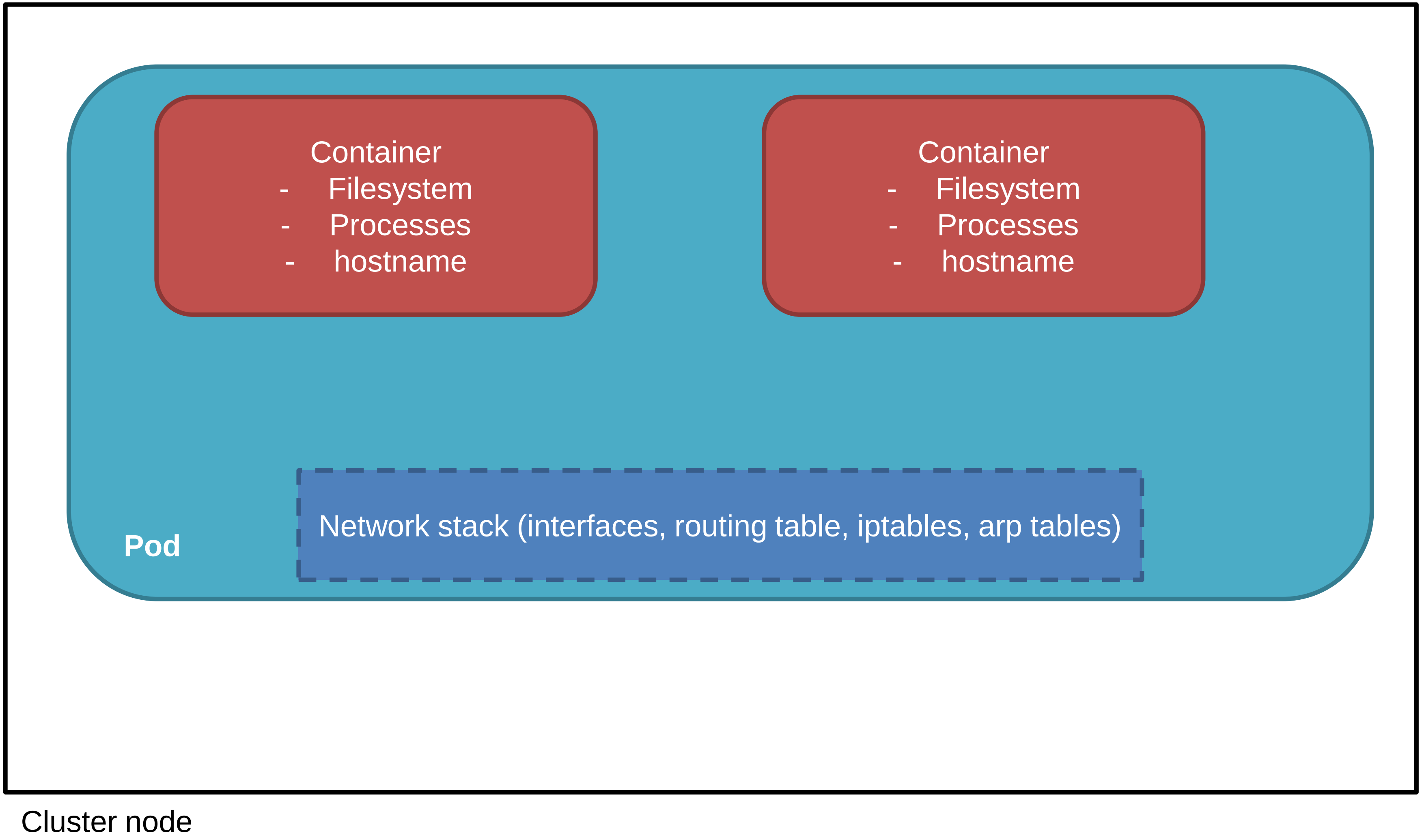
- Docker is the most common container runtime used in a Kubernetes Pod, but Pods support other container runtimes as well.

Containers in a Pod

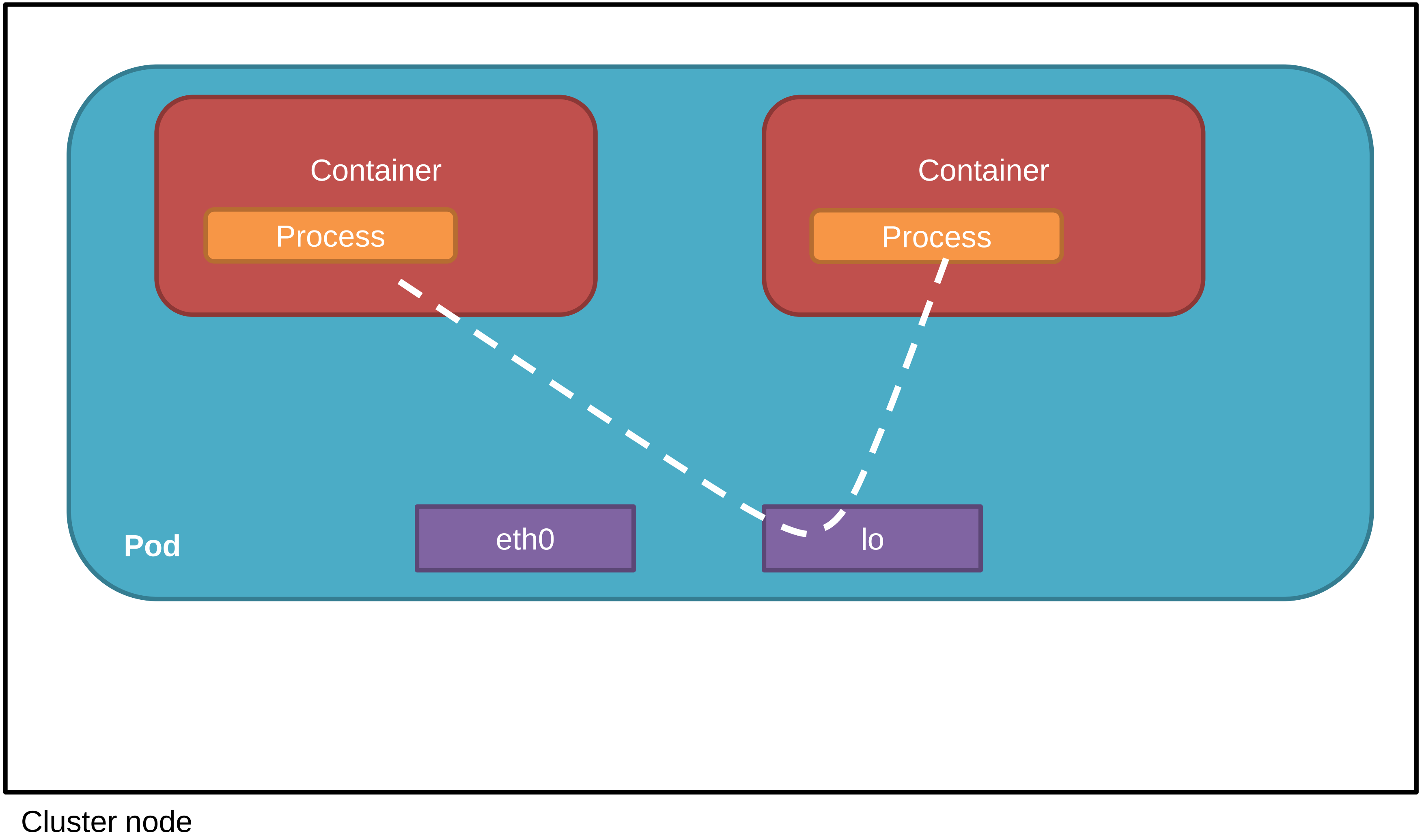
- Containers of the same pod are scheduled in the same node



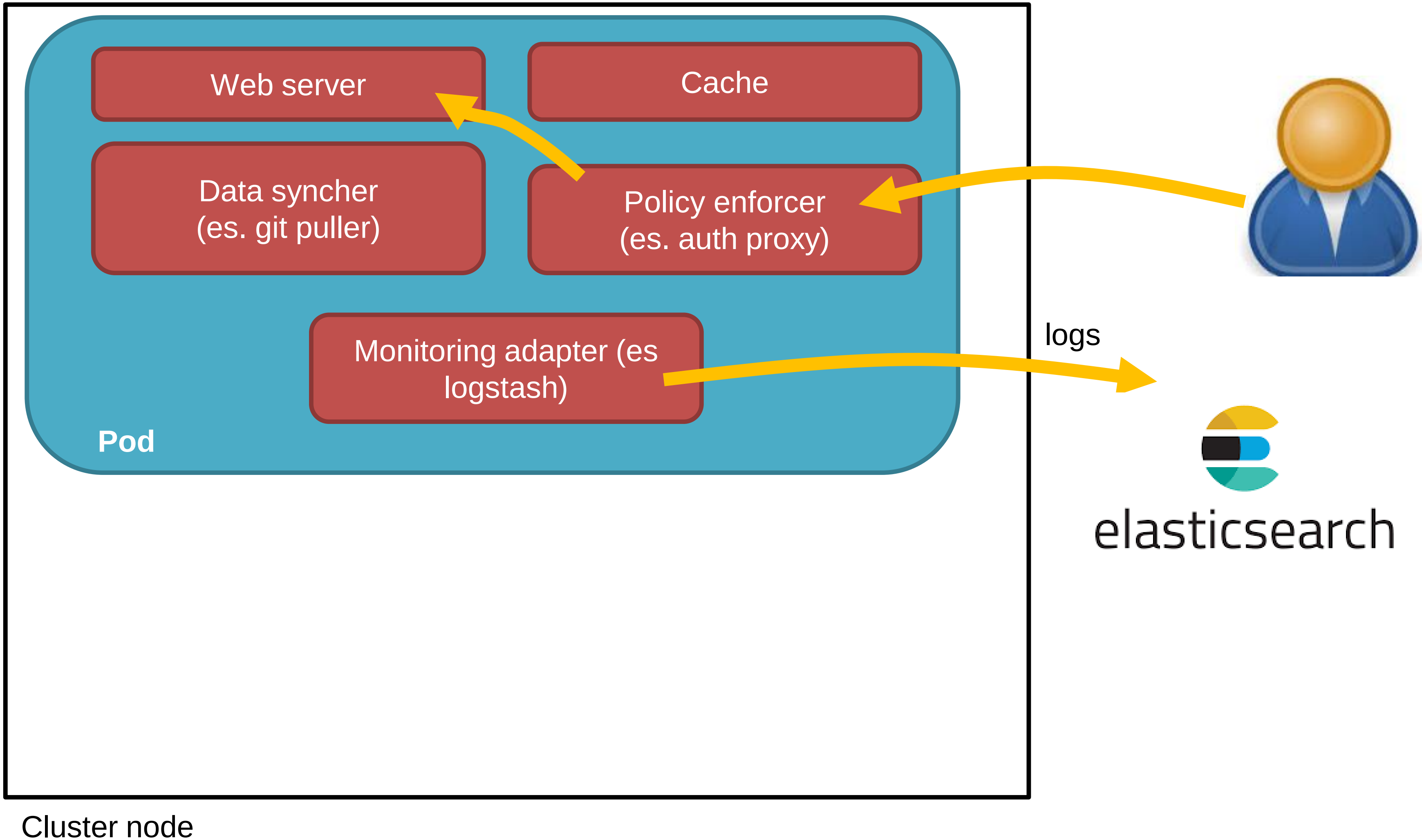
Resource sharing in pod containers



Communication between containers of the same pod

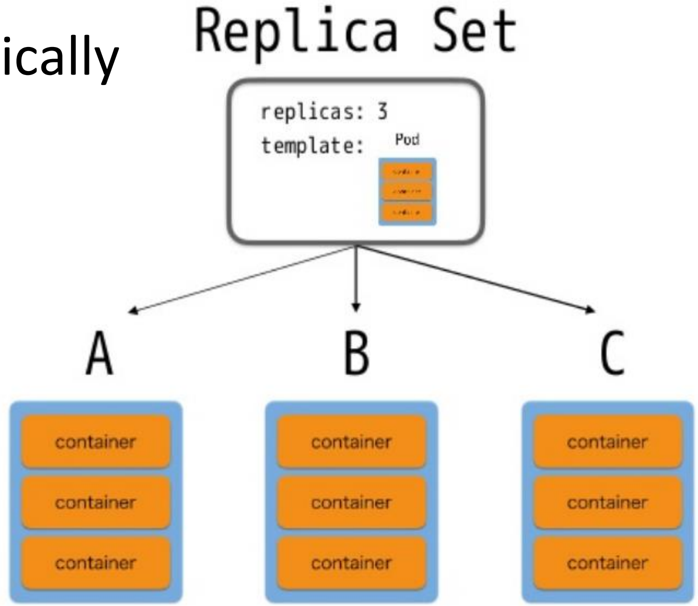


Pod example


Web server
Cache
Data syncher
(es. git puller)
Policy enforcer
(es. auth proxy)
Monitoring adapter (es
logstash)
Pod
logs
elasticsearch
Cluster node

Kubernetes – Replica Set

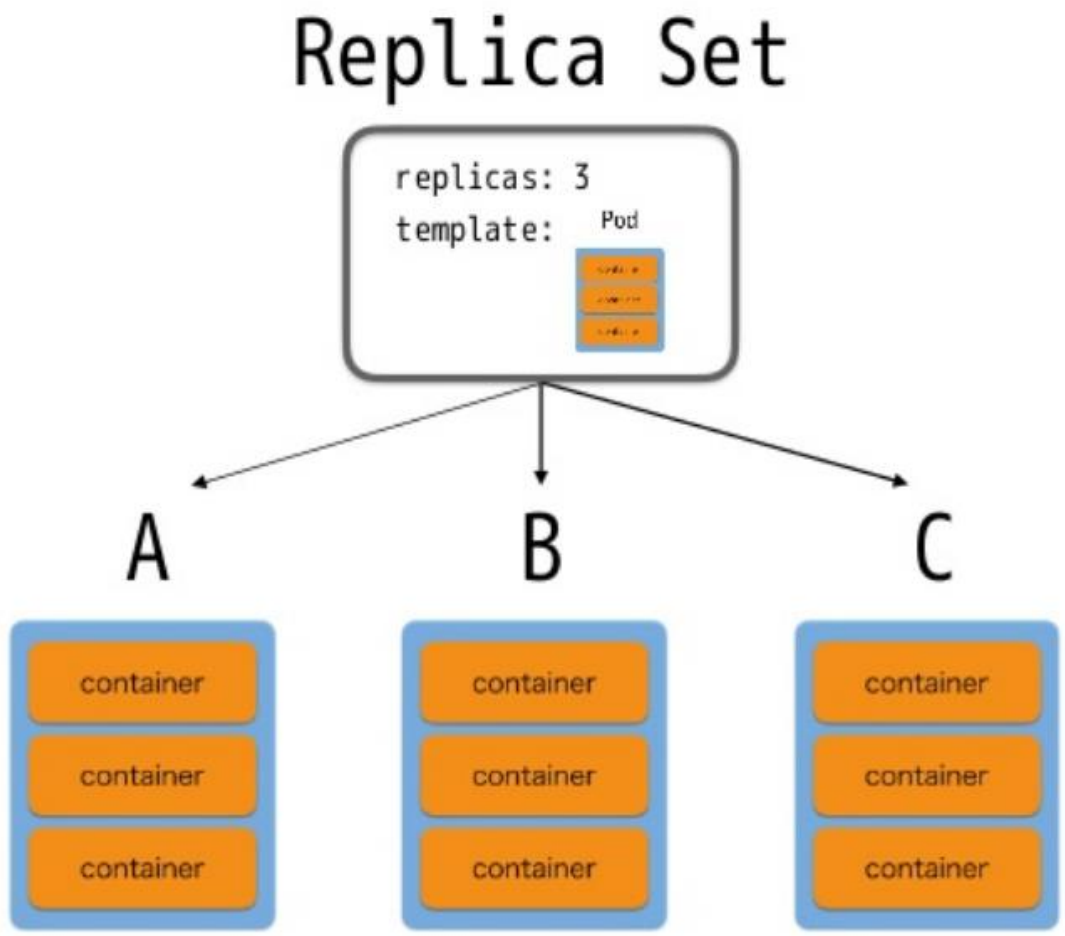
- ReplicaSet is the next-generation Replication Controller.
- Replica Set ensures that a specified number of pod replicas are running at any one time. In other words, Replica Set makes sure that a pod or a homogeneous set of pods is always up and available.
- Replica Set create and destroy Pods dynamically



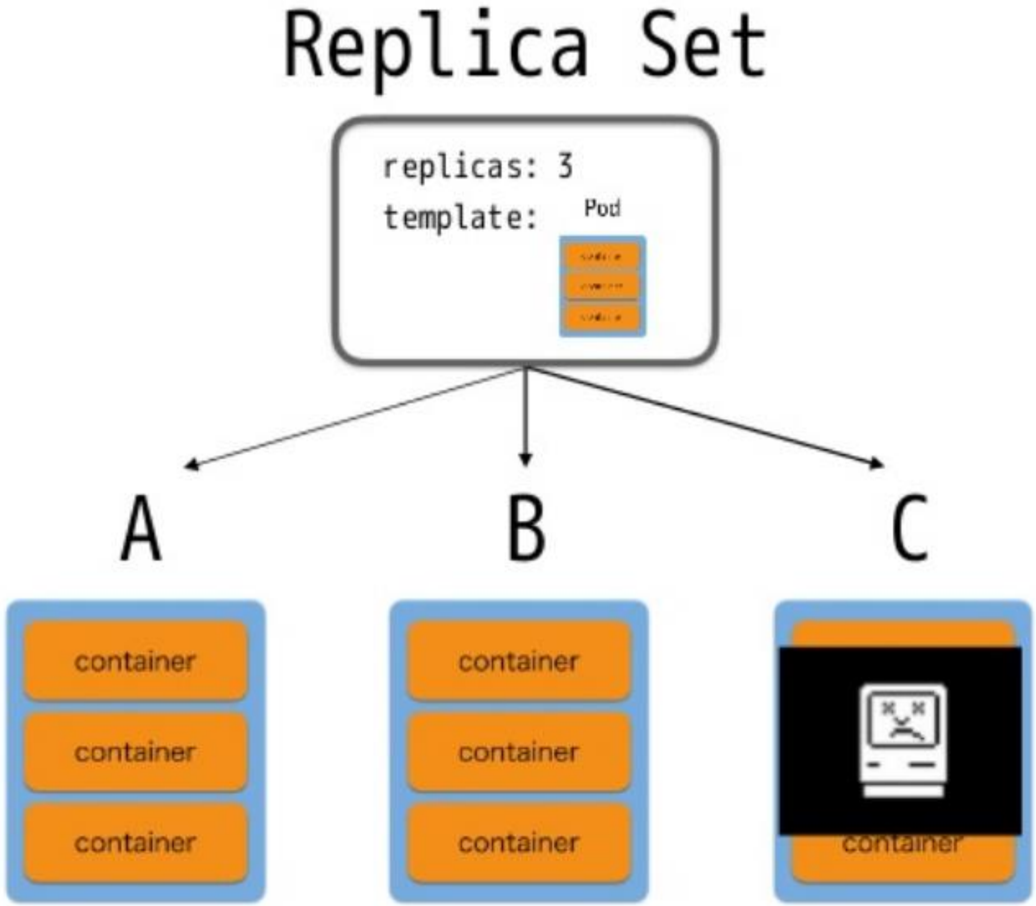
Monitoring - Probe

- Probe:
  - **livenessProbe**: Indicates whether the Container is running. If the liveness probe fails, the kubelet kills the Container, and the Container is subjected to its restart policy. If a Container does not provide a liveness probe, the default state is Success.
  - **readinessProbe**: Indicates whether the Container is ready to service requests. If the readiness probe fails, the endpoints controller removes the Pod's IP address from the endpoints of all Services that match the Pod. The default state of readiness before the initial delay is Failure. If a Container does not provide a readiness probe, the default state is Success
- Some examples of liveness/readiness probes
  - HTTP connection → success= HTTP result success
  - TCP connection → success= connection open
  - Run a command inside the container → success= exit code=0

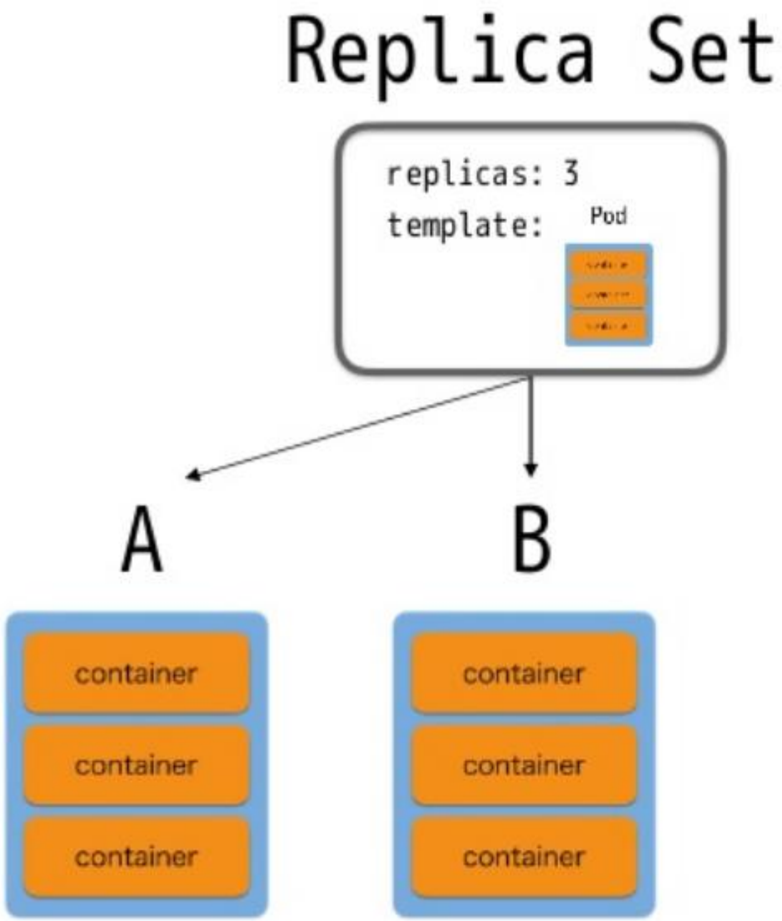
Failing of a readiness probe



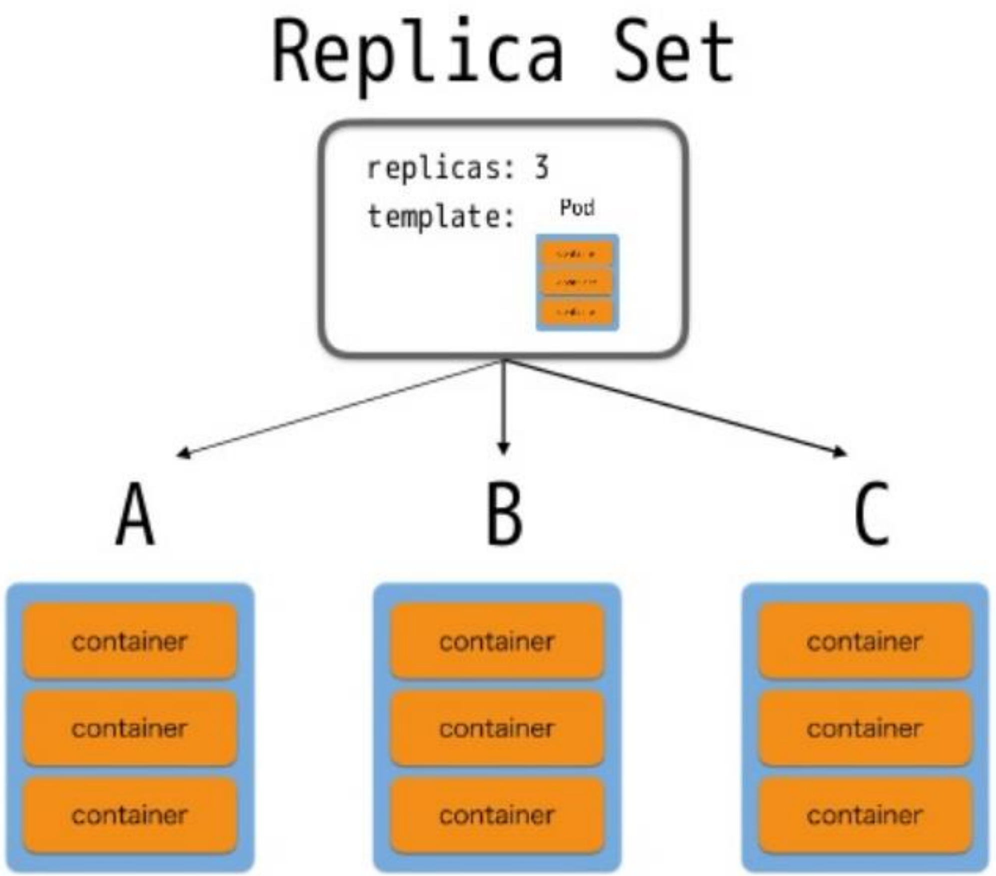
Failing of a readiness probe



Failing of a readiness probe



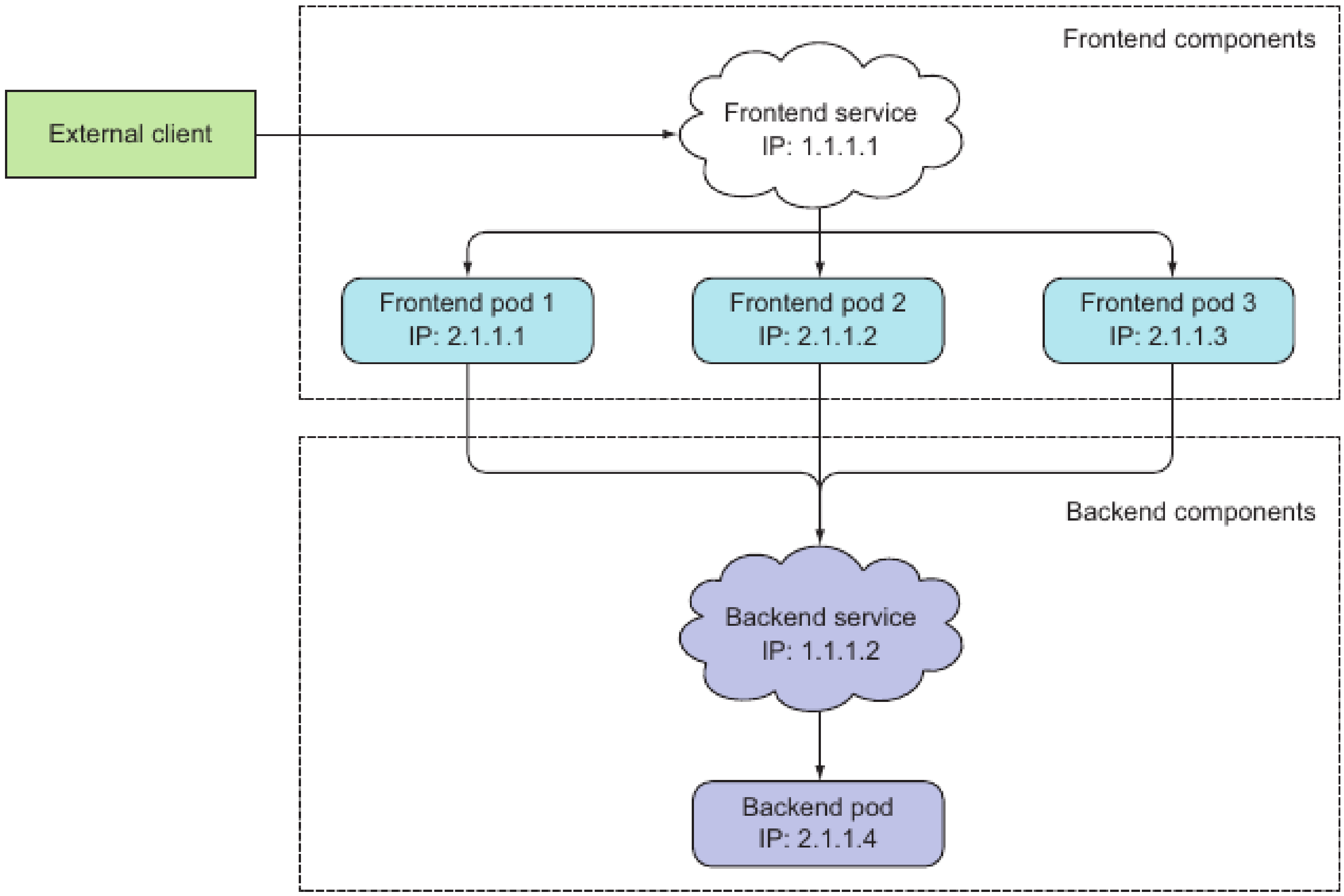
Failing of a readiness probe



Kubernetes – Service

- Service fix an IP/DNS for the pods
- Service is used to expose application IP both outside and inside of kubernetes
- Service handles the load balancing between the pods instances
- Service types:
  - ClusterIP (default) - Exposes the Service on an internal IP in the cluster. This type makes the Service only reachable from within the cluster.
  - NodePort - Exposes the Service on the same port of each selected Node in the cluster using NAT. Makes a Service accessible from outside the cluster using <NodeIP>:<NodePort>. Superset of ClusterIP.
  - LoadBalancer - fixed, external IP
  - ExternalName - CNAME record

Service


Frontend components
External client
Frontend service
IP: 1.1.1.1
Frontend pod 1
IP: 2.1.1.1
Frontend pod 2
IP: 2.1.1.2
Frontend pod 3
IP: 2.1.1.3
Backend components
Backend service
IP: 1.1.1.2
Backend pod
IP: 2.1.1.4

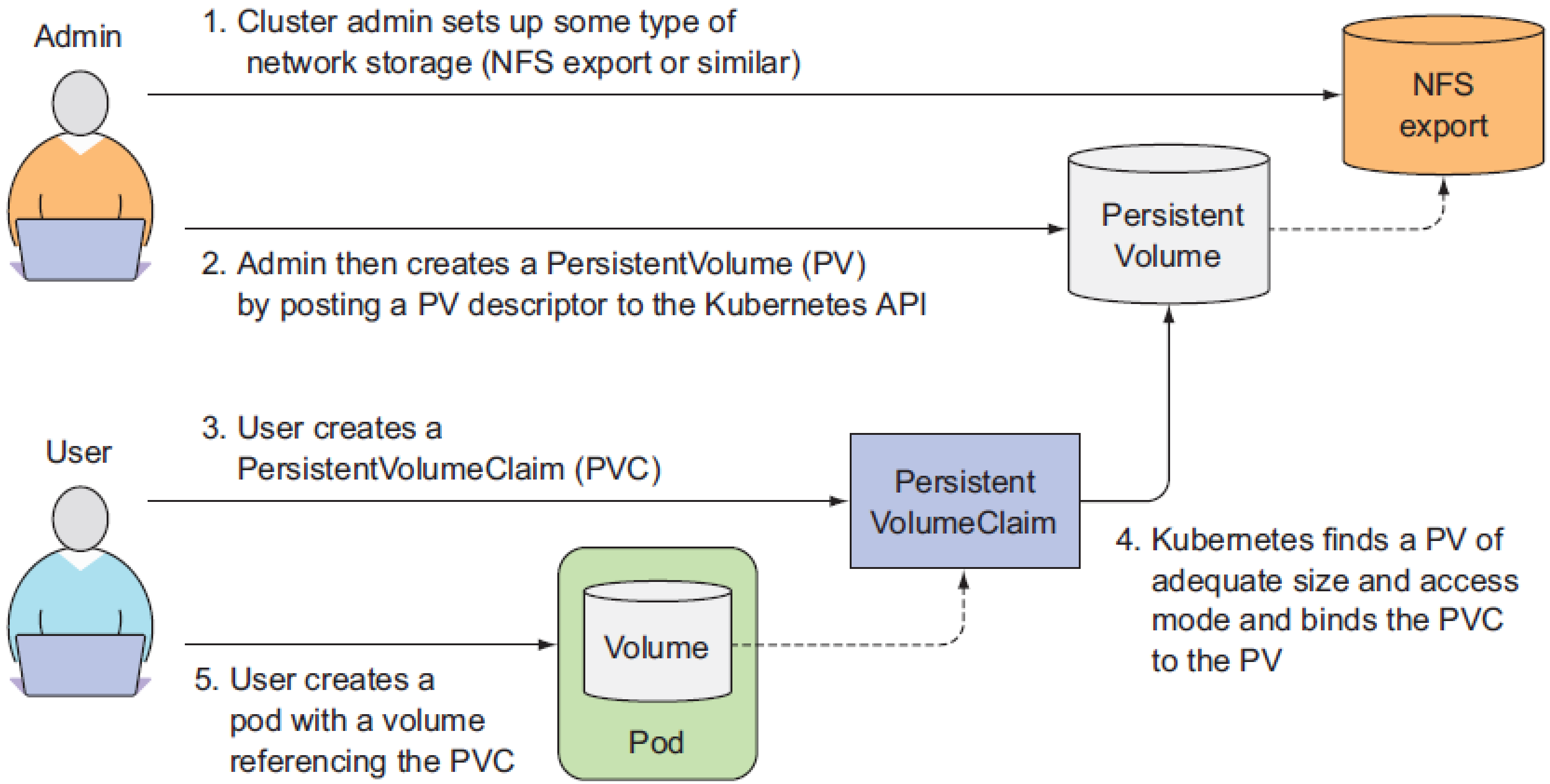
Kubernetes – Volume

- A Kubernetes volume has an explicit lifetime - the same as the Pod that encloses it. Consequently, a volume outlives any Containers that run within the Pod, and data is preserved across Container restarts.
- When a Pod ceases to exist, the volume will cease to exist, too.
- Kubernetes supports many types of volumes, and a Pod can use any number of them simultaneously.
- At its core, a volume is just a directory, possibly with some data in it, which is accessible to the Containers in a Pod. How that directory comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used.

Some kinds of volumes

- **emptyDir**—A simple empty directory used for storing transient data.
- **hostPath**—Used for mounting directories from the worker node's filesystem into the pod.
- **gitRepo**—A volume initialized by checking out the contents of a Git repository.
- **nfs**—An NFS share mounted into the pod.
- **gcePersistentDisk** (Google Compute Engine Persistent Disk), awsElastic-BlockStore (Amazon Web Services Elastic Block Store Volume), azureDisk (Microsoft Azure Disk Volume)—Used for mounting cloud provider-specific storage.
- cinder, cephfs, iscsi, flocker, glusterfs, quobyte, rbd, flexVolume , vsphere-Volume, photonPersistentDisk, scaleIO—Used for mounting other types of **network storage**.
- configMap, secret, downwardAPI— **Special types of volumes** used to expose certain Kubernetes resources and cluster information to the pod.
- **persistentVolumeClaim** —A way to use a pre- or dynamically provisioned persistent storage.

Kubernetes – Persistent volume



Question Time

DOMANDE, DUBBI, CURIOSITÀ?

imola project partners

- Più di **20 anni di esperienza** nell'Enterprise IT
- Consulenza e Skill Transfer su **Architetture**, **Integrazione** e **Processo**
- *OMG* Influence Member, *JSR 312* Expert Group, *CSI*, *WWISA*, *OpenESB* Key Partner, *NetBeans* Strategic Partner

MokaByte
La prima rivista web italiana dedicata a Java®

- La comunita' italiana dedicata a **Java**
- **10 anni di articoli**, pubblicazioni, libri, eventi, training
- Dai programmatori agli architetti
- Piu' di **1.000.000 pagine** lette al mese

ANTREEM INNOVATION PARTNER

- Business partner in progetti con alto grado di **innovazione**
- Padroni in **tecnologie** e **architetture mobile**
- Competenti in **architetture dell'informazione**, **UX** e **Design**