↘ Speakers: Filippo Bosi

↘ Fulvio Di Maria

↘ Lorenzo Manzoni

↘ Stefano Monti

↘ Luca Poli

Bologna, 27/05/2016

Docker Ecosystem and Tools

gruppo**imola**

Agenda

1. Containers & Docker ecosystem
   1.1. Docker basics
   1.2. Docker basics - hands on
   1.3. Docker-compose
   1.4. Docker-compose - hands on

2. Docker for developers
   2.1. Integrating Maven and Docker - repeatable and scalable development/testing infrastructure
   2.2. Integrating Maven and Docker - hands on

3. Scaling to a (private, open-source) cloud

Reference templates

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

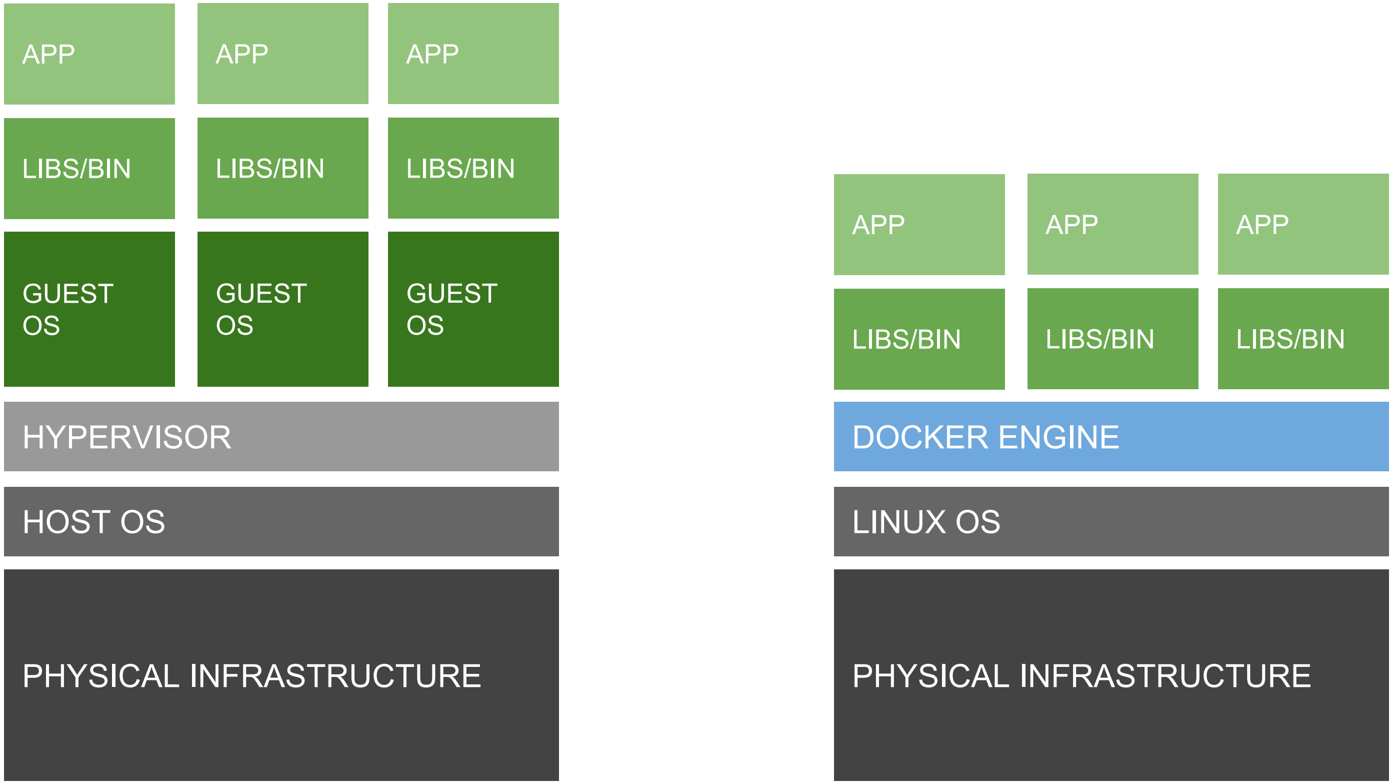
Agenda

1. **Containers & Docker ecosystem**
    1.1. Docker basics
    1.2. Docker basics - hands on
    1.3. Docker-compose
    1.4. Docker-compose - hands on

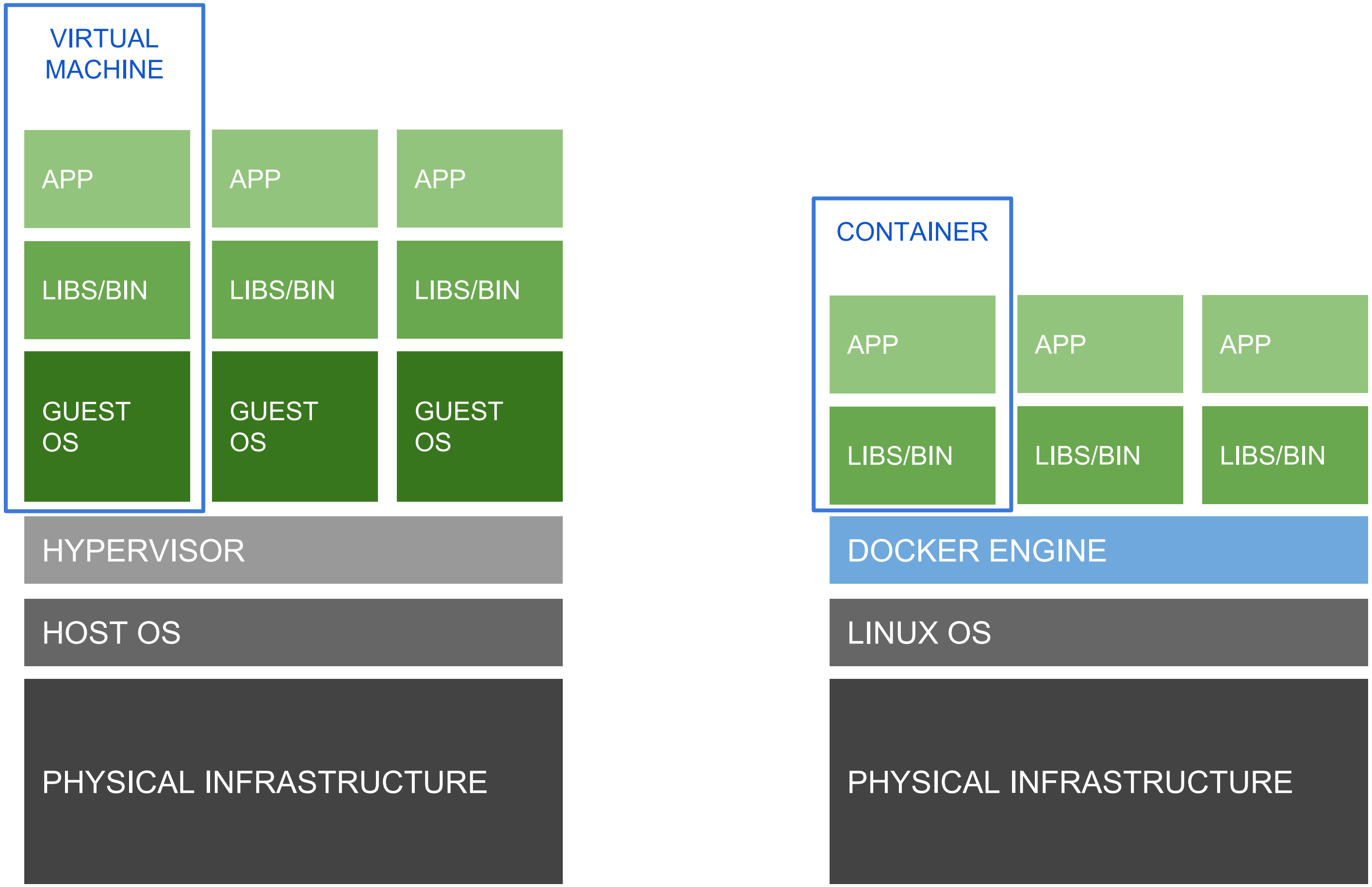
2. Docker for developers
    2.1. Integrating Maven and Docker - repeatable and scalable development/testing infrastructure
    2.2. Integrating Maven and Docker - hands on

3. Scaling to a (private, open-source) cloud

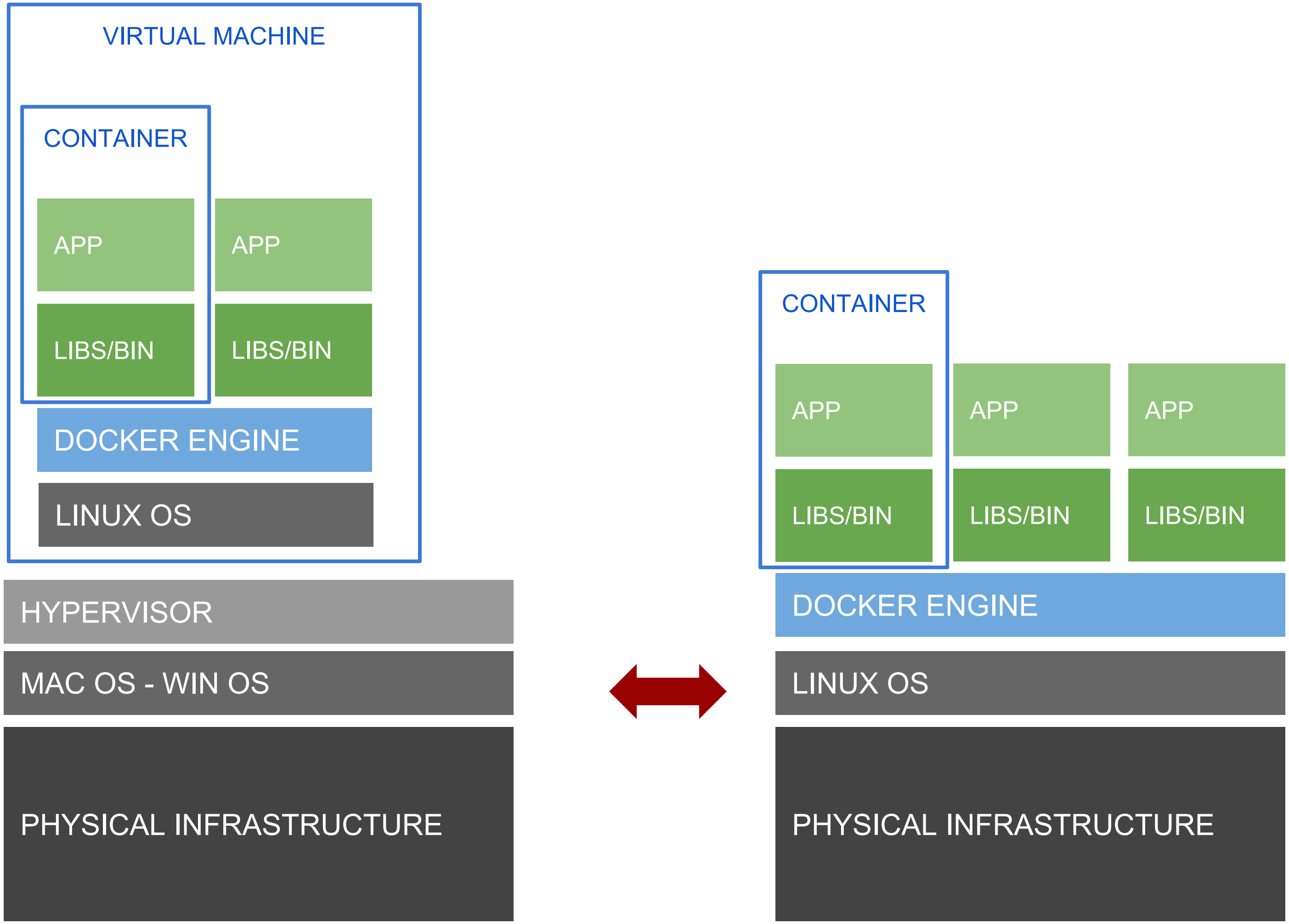
1.1 Virtualization vs Containerization



1.1 Virtualization vs Containerization



1.1 Docker flavors



1.1 Containerization vs Virtualization

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

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

1.1 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/

1.1 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
- **2015-2016**: tremendous increase in popularity
  - Thoughtworks technology radar strongly promotes Docker adoption https://www.thoughtworks.com/radar/platforms
  - 2x Docker image pulls in 3 months, up to 2 billion pulls (as of February 2016)

1.1 Docker - Under the hood

- **Libcontainer** Specification
  - an abstraction/unification layer to decouple Docker from kernel-specific **container features** (e.g. LXC, libvirt, ...)

- The Docker **Image Specification**
  - **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

1.1 Docker key concepts

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**.*

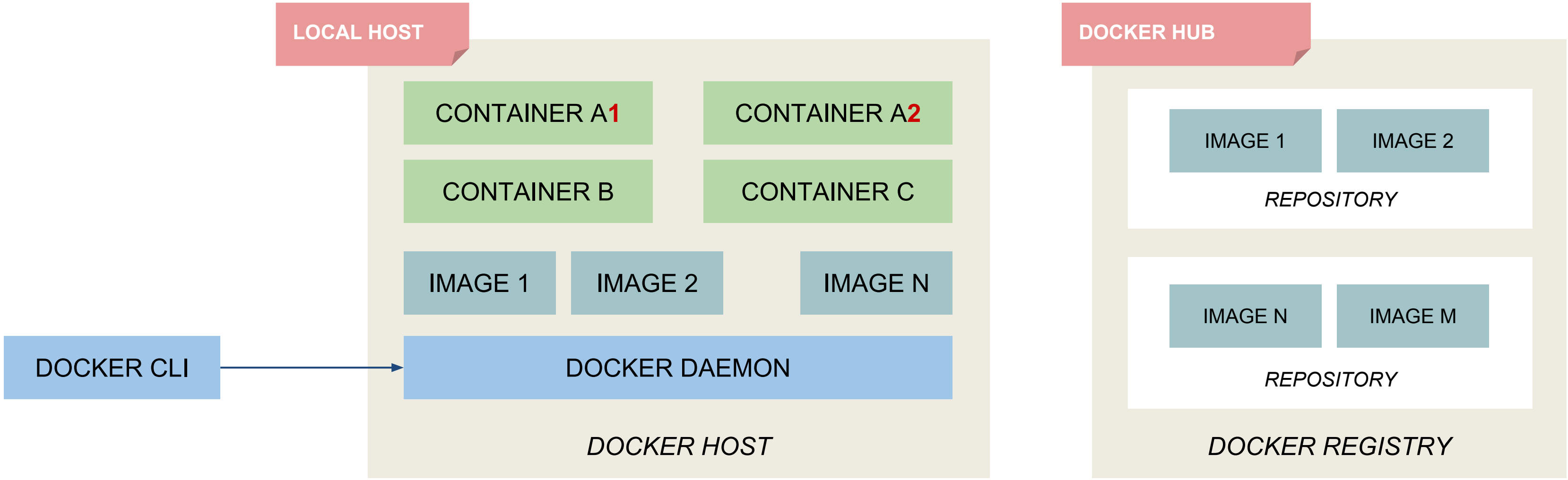
Docker containers

*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**.*

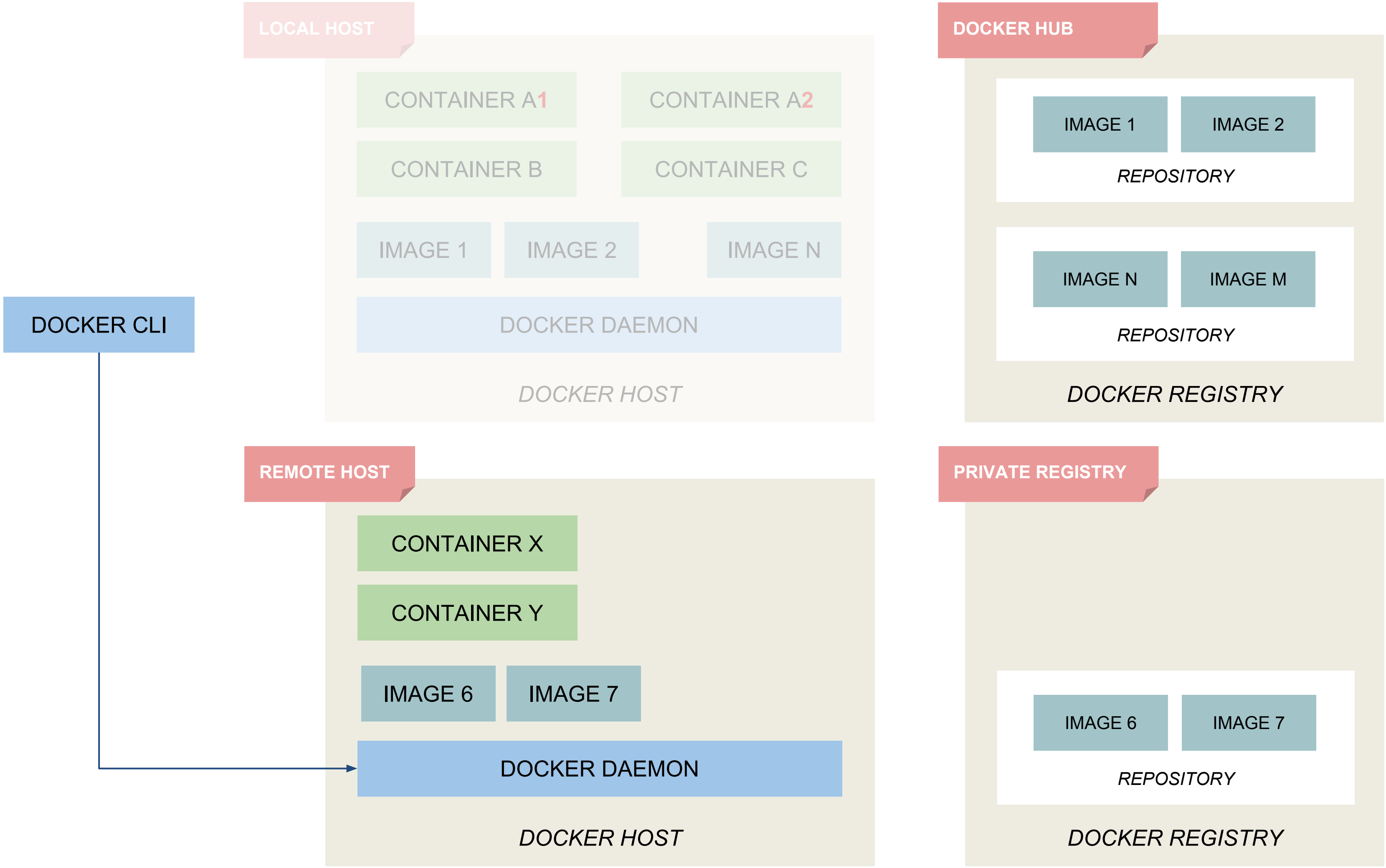
Docker registries

*Docker registries **hold images**. These are **public or private stores** from which you **upload or download** images. The public Docker registry is provided with the Docker Hub. It serves a huge collection of existing images for your use. These can be images you create yourself or you can use images that others have previously created. Docker registries are **the distribution component of Docker**. For more information, go to Docker Registry and Docker Trusted Registry.*

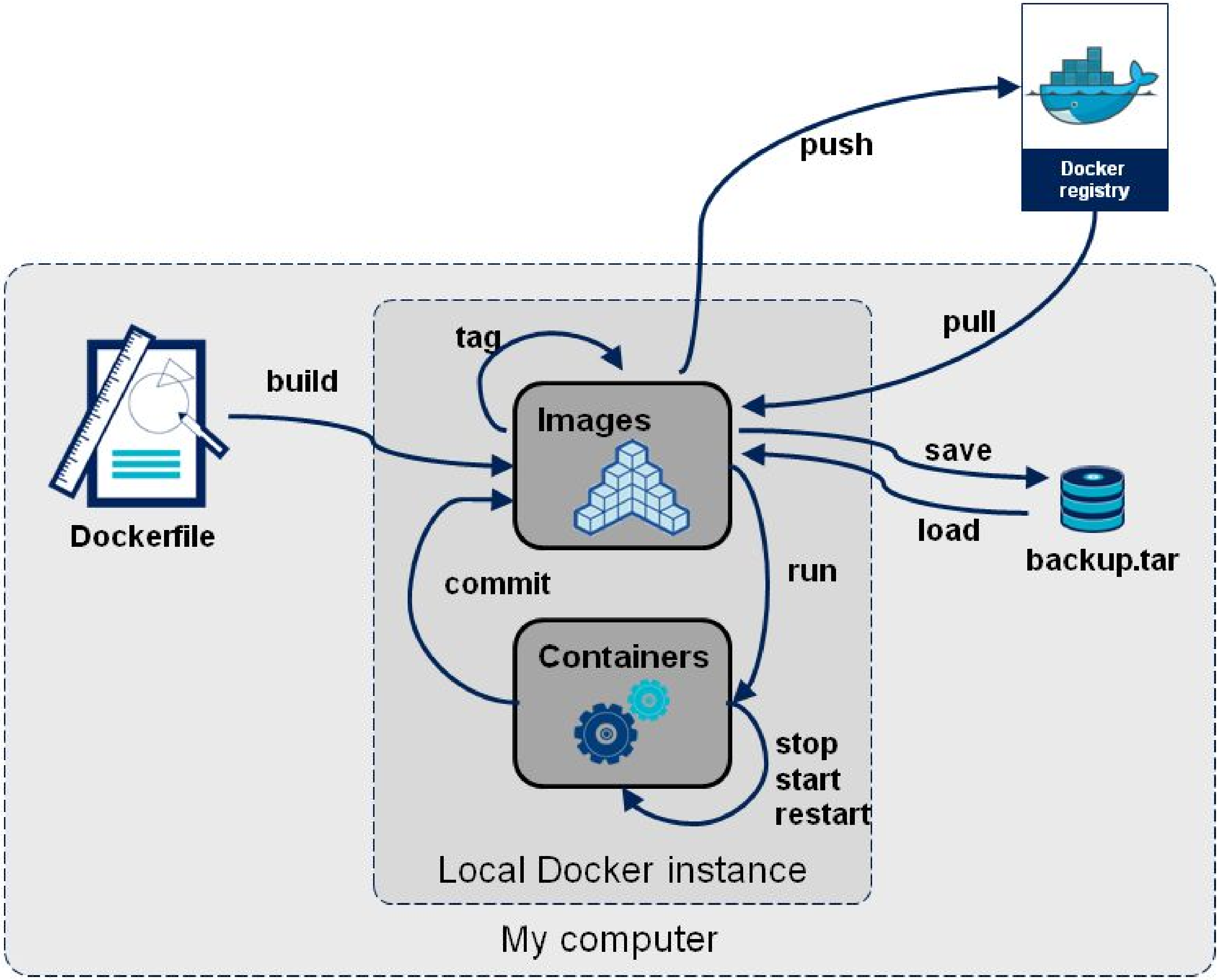
1.1 Docker components



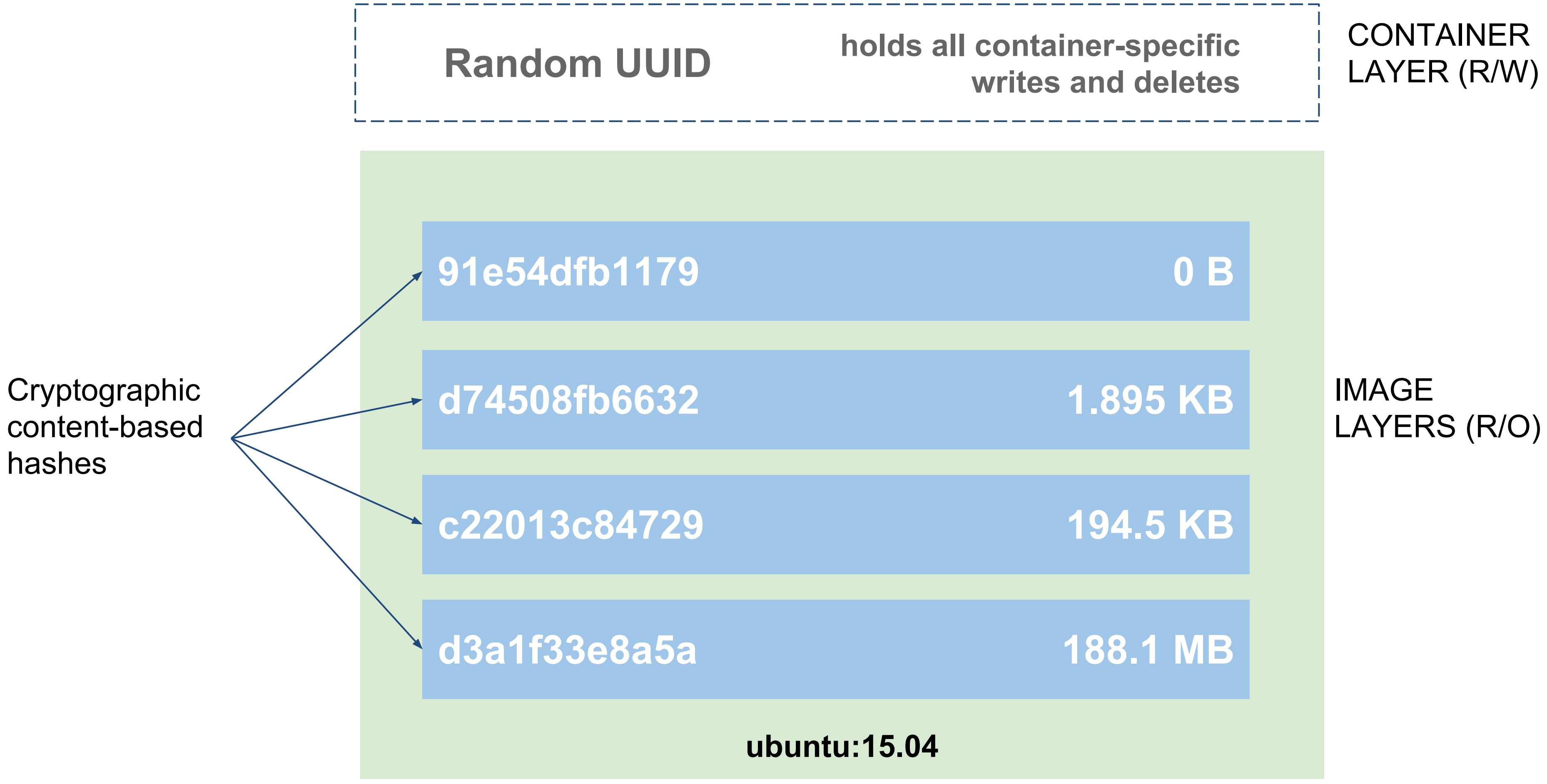
1.1 Docker components



1.1 Docker Container Lifecycle



Docker images



Random UUID

holds all container-specific writes and deletes

CONTAINER LAYER (R/W)

Cryptographic content-based hashes

91e54dfb1179	0 B
d74508fb6632	1.895 KB
c22013c84729	194.5 KB
d3a1f33e8a5a	188.1 MB

IMAGE LAYERS (R/O)

ubuntu:15.04

1.1 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

1.1 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.**

1.1 Docker images

```
docker pull hello-world
docker history hello-world
```

1.1 Docker Images

Browse to: https://hub.docker.com/explore/

1.1 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 restart** - re-runs a previously stopped container, preserving run options such as port forwarding)

```
$ docker restart containerId
```

restarts a container identified by ***containerId***

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

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

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

1.1 Docker CLI

- **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 images** - shows (locally) available images

```
$ docker images
```

1.1 Docker CLI

- **docker ps** - lists running/available containers

```
$ docker ps        lists running containers
$ docker ps -a     lists all containers (including stopped ones)
```

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

```
$ docker stop containerId
```

stops container identified by *containerId*

- **docker rm** - removes containers

```
$ docker rm containerId
```

removes container identified by *containerId*

- **docker rmi** - removes images

```
$ docker rmi imageId
```

removes image identified by *imageId*

1.1 Docker CLI

```
docker run hello-world
docker ps
docker rm
```

1.1 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"]
```

1.1 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:** provides 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 daemot 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

1.1 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:** provides 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 daemot 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

1.1 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

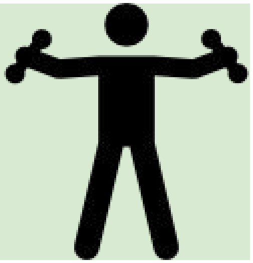
	No ENTRYPOINT	**ENTRYPOINT exec_entry p1_entry**	**ENTRYPOINT [“exec_entry”, “p1_entry”]**
No CMD	*error, not allowed*	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD [“exec_cmd”, “p1_cmd”]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry exec_cmd p1_cmd	exec_entry p1_entry exec_cmd p1_cmd
CMD [“p1_cmd”, “p2_cmd”]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry p1_cmd p2_cmd	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

1.1 Docker networking

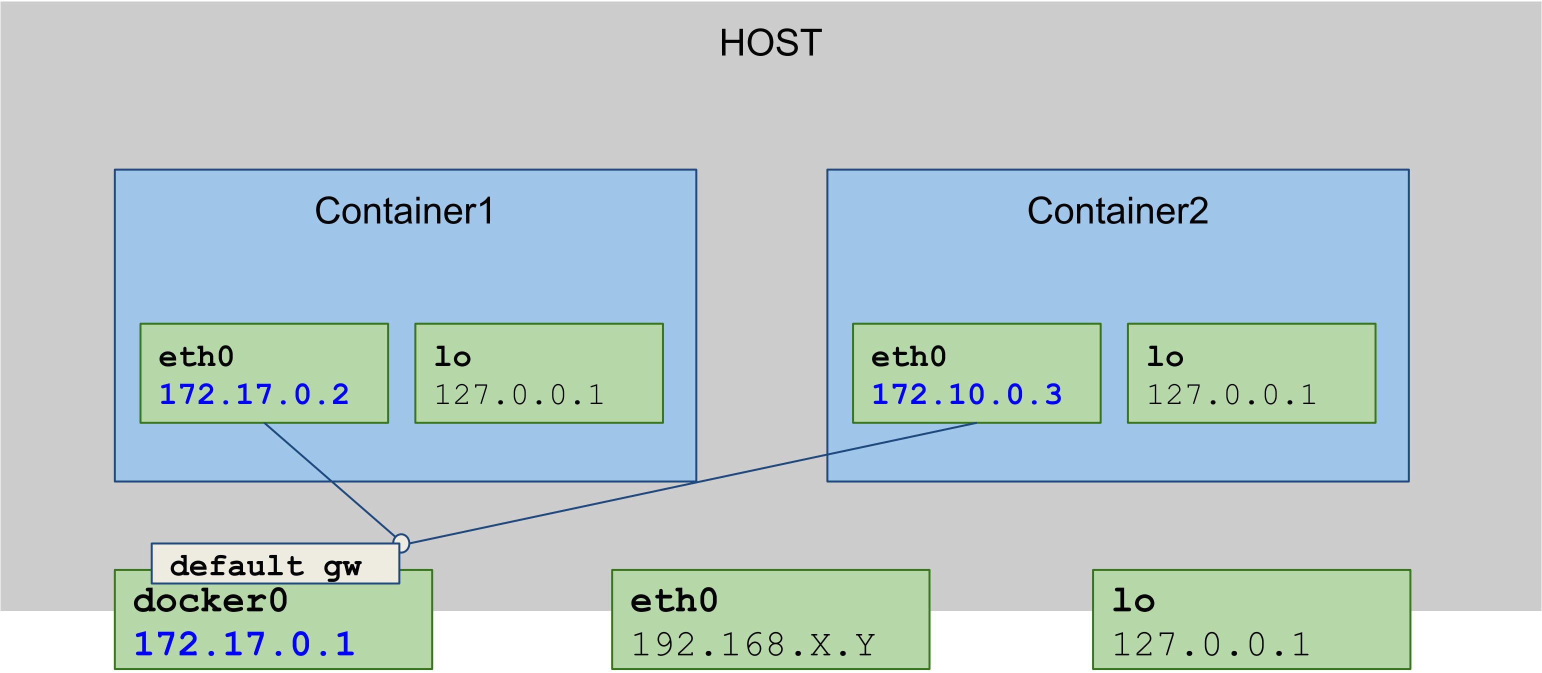
- docker networking provides full isolation for containers
- isolation can beoverwrittento make containers communicate with each other
- docker engine creates **3 default networks**
  - **bridge** → **default network** for containers; points to **docker0** (virtual) network interface
  - **none** → container 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 network ls
docker network inspect bridge
```

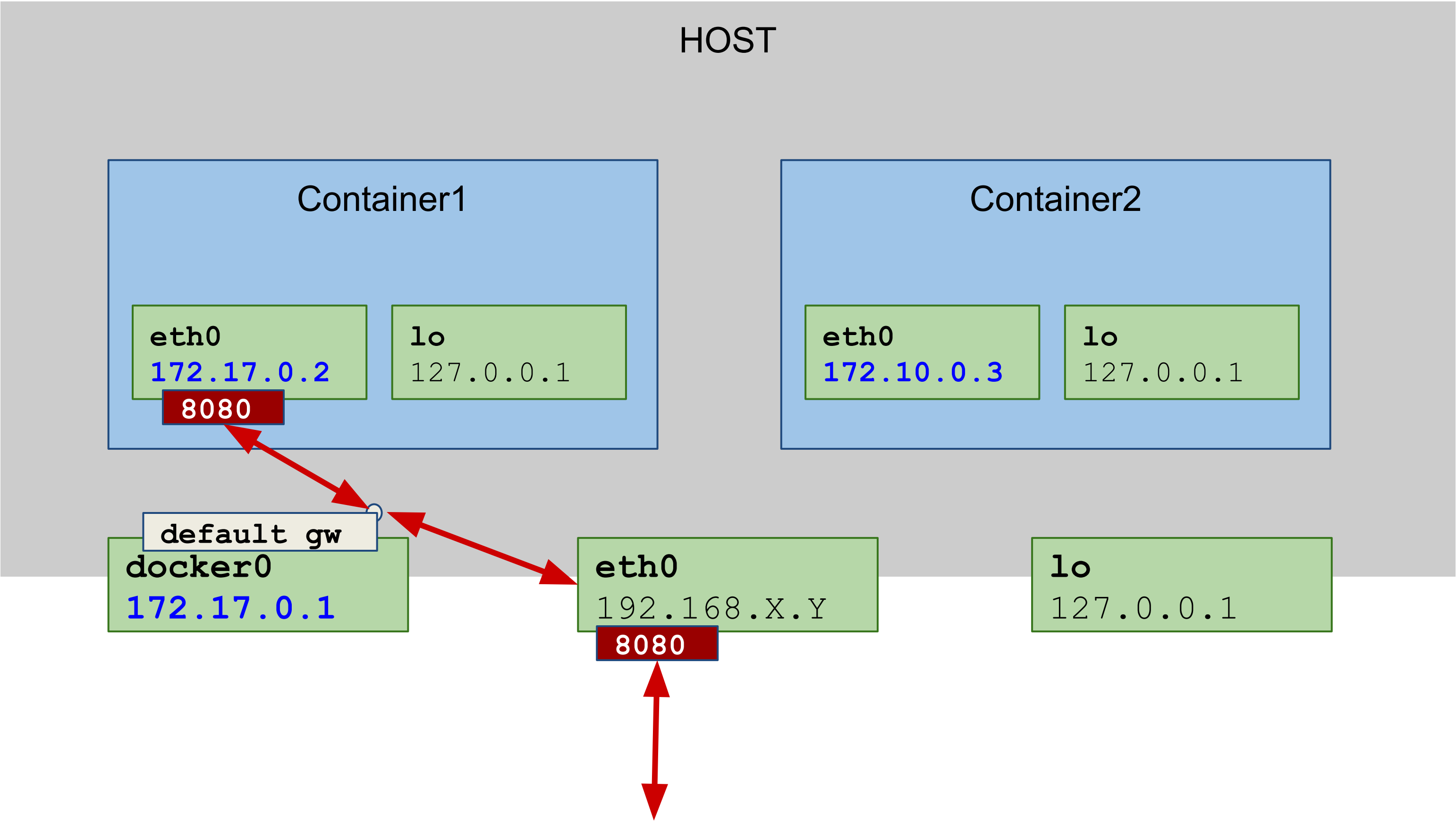


Docker networking - Bridge



```
ifconfig docker0
docker inspect --format '{{ .NetworkSettings.IPAddress }}'
containerId
```

1.1 Docker networking - port forwarding



```
docker run -d -p 8080:8080 tomcat:8.0
```

1.1 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

```
$ docker run -d -v /webapp tomcat:8.0
```

mounts a specific host directory (usually, in the /var/lib/docker/… FS tree) to /webapp mountpoint within the container

```
$ docker run -d -v /host_fs_folder:/webapp tomcat:8.0
```

mounts **`/host_fs_folder`** host directory to /webapp mountpoint within the container

Agenda

1. **Containers & Docker ecosystem**
   - 1.1. **Docker basics**
   - 1.2. **Docker basics - hands on**
   - 1.3. Docker-compose
   - 1.4. Docker-compose - hands on
2. Docker for developers
   - 2.1. Integrating Maven and Docker - repeatable and scalable development/testing infrastructure
   - 2.2. Integrating Maven and Docker - hands on
3. Scaling to a (private, open-source) cloud

1.2 Docker - Hands-on

1.1 - 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 Exercise1-Docker/1.1-HelloWeb/
```

1.2 Docker - Hands-on

1.2 - 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 JBoss 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**
   1.1. **Docker basics**
   1.2. **Docker basics - hands on**
   1.3. **Docker-compose**
   1.4. **Docker-compose - hands on**

2. **Docker for developers**
   2.1. **Integrating Maven and Docker - repeatable and scalable development/testing infrastructure**
   2.2. **Integrating Maven and Docker - hands on**

3. **Scaling to a (private, open-source) cloud**

1.3 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**

1.3 Docker-compose

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/

1.3 Docker-compose example

```yaml
version: '2'
services:
  db:
    image: mysql:5.7
    volumes:
      - "./.data/db:/var/lib/mysql"
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

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

1.3 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** - builds or rebuilds services

```
$ 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

1.3 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**
   - **1.1. Docker basics**
   - **1.2. Docker basics - hands on**
   - **1.3. Docker-compose**
   - **1.4. Docker-compose - hands on**

2. **Docker for developers**
   - **2.1. Integrating Maven and Docker - repeatable and scalable development/testing infrastructure**
   - **2.2. Integrating Maven and Docker - hands on**

3. **Scaling to a (private, open-source) cloud**

1.4 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
   1.1. Docker basics
   1.2. Docker basics - hands on
   1.3. Docker-compose
   1.4. Docker-compose - hands on

2. Docker for developers
   2.1. Integrating Maven and Docker - repeatable and scalable development/testing infrastructure
   2.2. Integrating Maven and Docker - hands on

3. Scaling to a (private, open-source) cloud

2.1 Rationale

Building a complex, real-world application usually requires coordinating a set different moving parts

Typical N-tier applications consist of layers of

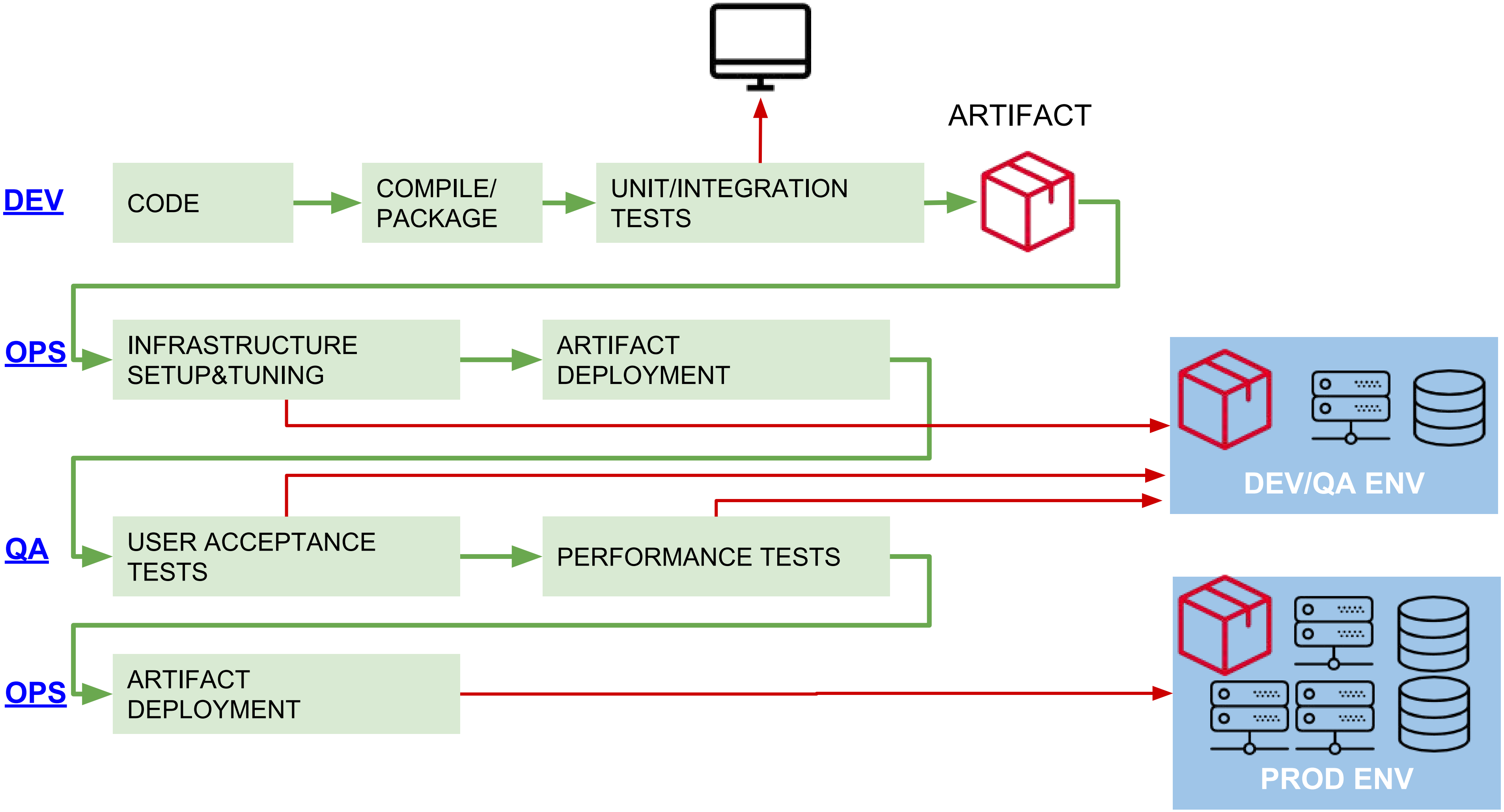
- persistence → relational/NoSQL database
- middle-tier (business logic) → JEE application servers, messaging systems (e.g., JMS-compliant queue managers)
- mediation/integration layers → ESBs
- presentation → APACHE HTTPD front-end, SW/HW Load Balancer/Reverse proxies, etc…

2.1 Rationale

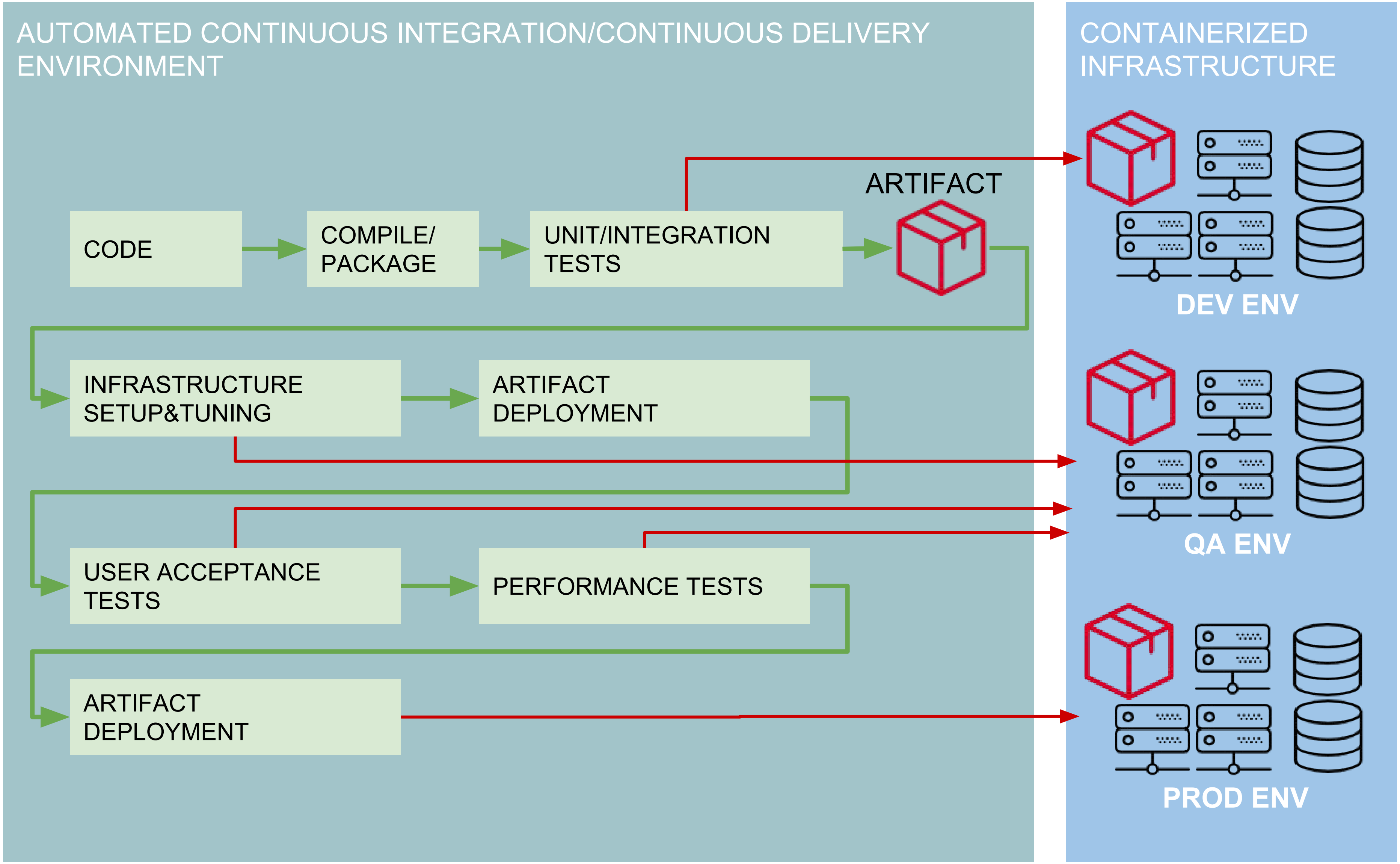
Docker/Docker-compose allow developers to tame architecture/infrastructure complexity

Containers integrate into traditional development/build/test cycles to make build processes easily **scalable and repeatable** → **e.g., no dependency on external server configuration**

2.1 Traditional build environment



2.1 Containerized build environment



Agenda

1. Containers & Docker ecosystem
   - 1.1. Docker basics
   - 1.2. Docker basics - hands on
   - 1.3. Docker-compose
   - 1.4. Docker-compose - hands on

2. Docker for developers
   - 2.1. Integrating Maven and Docker - repeatable and scalable development/testing infrastructure
   - 2.2. Integrating Maven and Docker - hands on

3. Scaling to a (private, open-source) cloud

2.2 Integrating Maven and Docker - hands on

```
cd Exercise3-Maven/exercise3
mvn clean package docker:build
mvn docker:start
mvn docker:stop
```

https://fabric8io.github.io/docker-maven-plugin/

Agenda

1. Containers & Docker ecosystem
    1.1. Docker basics
    1.2. Docker basics - hands on
    1.3. Docker-compose
    1.4. Docker-compose - hands on

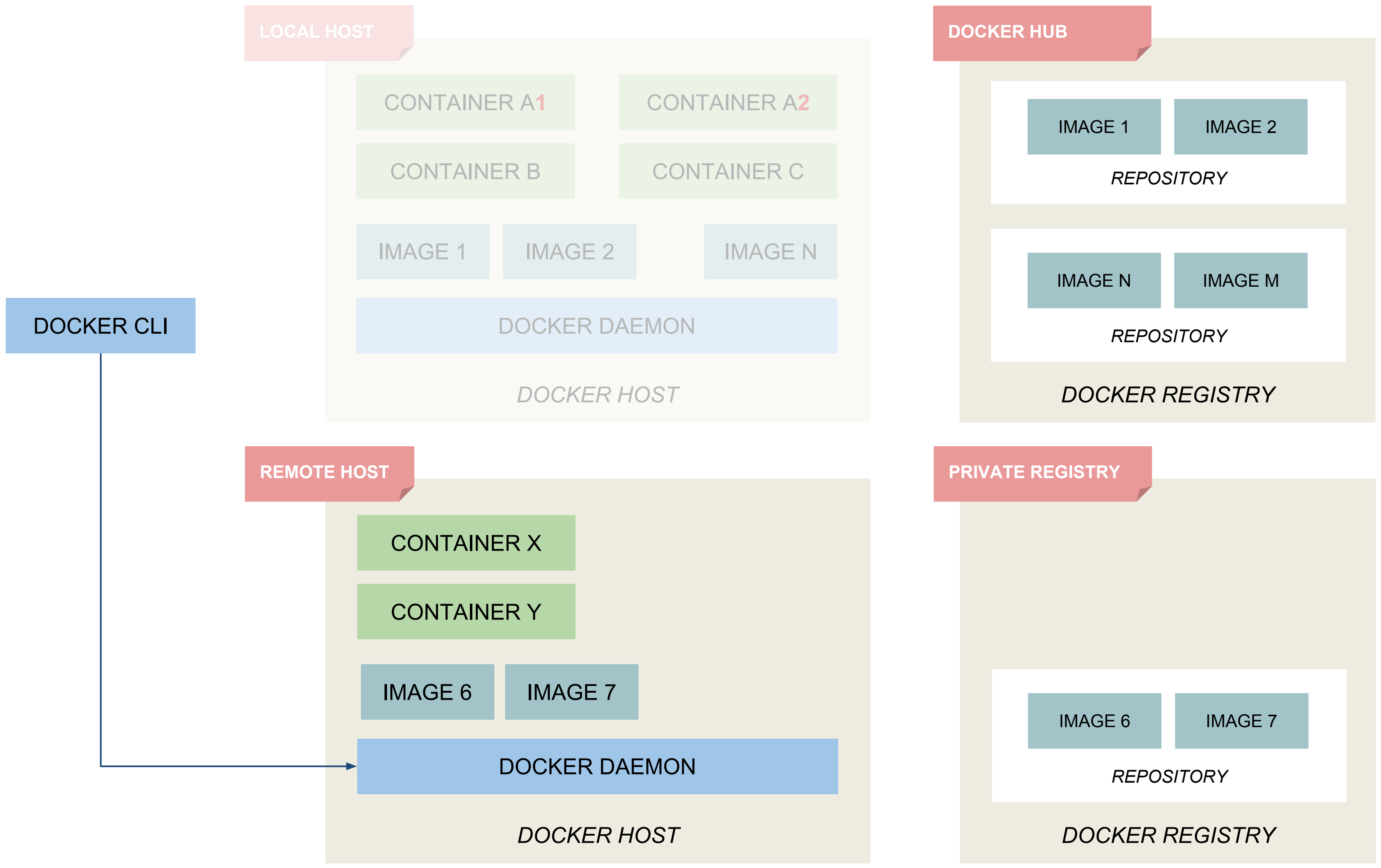
2. Docker for developers
    2.1. Integrating Maven and Docker - repeatable and scalable development/testing infrastructure
    2.2. Integrating Maven and Docker - hands on

3. Scaling to a (private, open-source) cloud

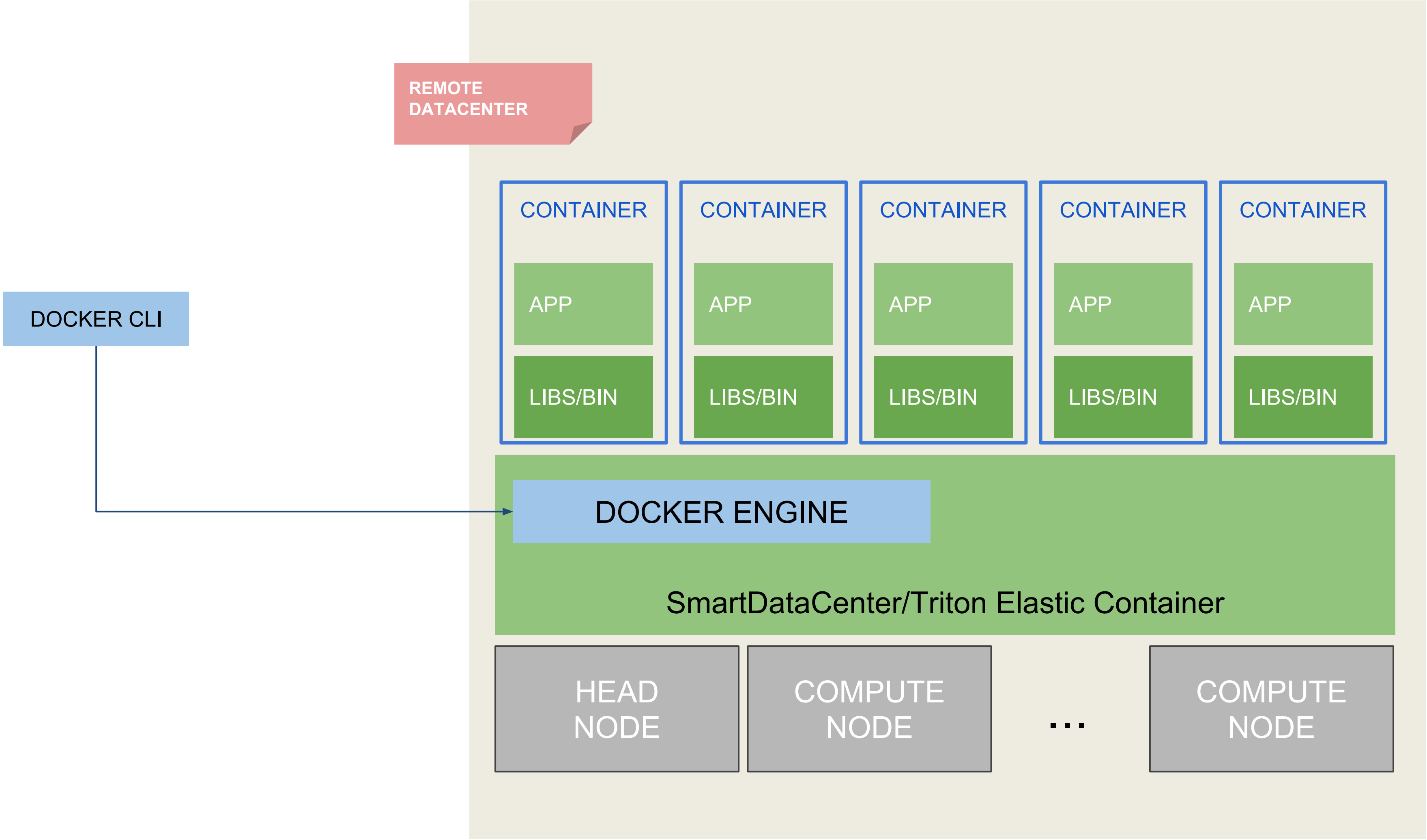
3.1 SmartDataCenter/Triton Elastic Container

- **Private (on-premise) cloud** platform based on SmartOS (a derivative of OpenSolaris)
- Native VM and Docker support
- Runs on bare metal and allows for flexible datacenter scaling
- Open-source
- Provided by Joyent Inc. (the company behind Node.js)
- Available as a public service

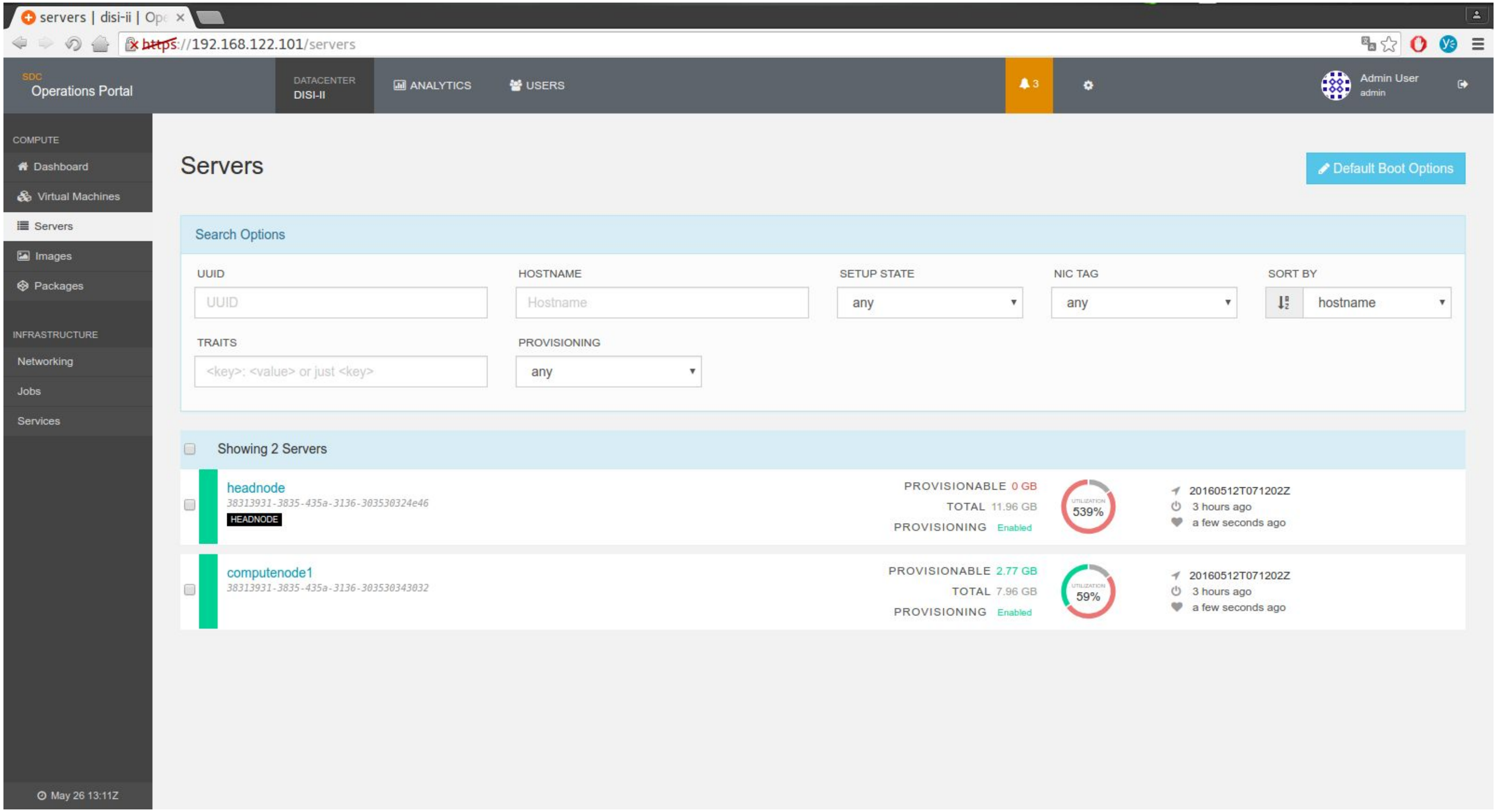
3.1 Docker components



3.1 SmartDataCenter/Triton Elastic Container



3.1 SmartDataCenter/Triton Elastic Container



Servers

Search Options

UUID: UUID

HOSTNAME: Hostname

SETUP STATE: any

NIC TAG: any

SORT BY: hostname

TRAITS: <key>: <value> or just <key>

PROVISIONING: any

□ Showing 2 Servers

□ headnode
38313931-3835-435a-3136-303530324e46
HEADNODE
PROVISIONABLE 0 GB
TOTAL 11.96 GB
PROVISIONING Enabled
UTILIZATION 539%
20160512T071202Z
3 hours ago
a few seconds ago

□ computenode1
38313931-3835-435a-3136-303530343032
PROVISIONABLE 2.77 GB
TOTAL 7.96 GB
PROVISIONING Enabled
UTILIZATION 59%
20160512T071202Z
3 hours ago
a few seconds ago

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**