



Container-based virtualization: Docker

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2021/22

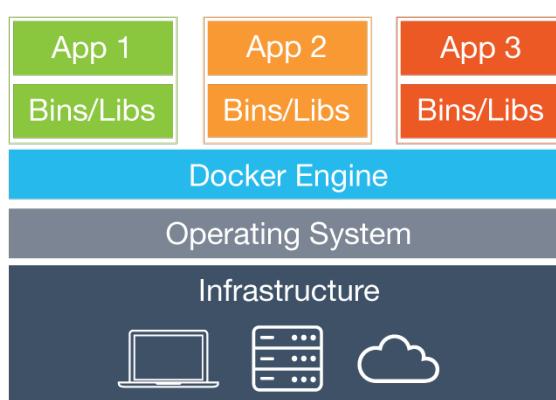
Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Case study: Docker



- Lightweight, open and secure container-based virtualization
 - Containers include the application and all of its dependencies, but share the OS kernel with other containers
 - Containers run as an isolated process in userspace on the host OS
 - Containers are also not tied to any specific infrastructure

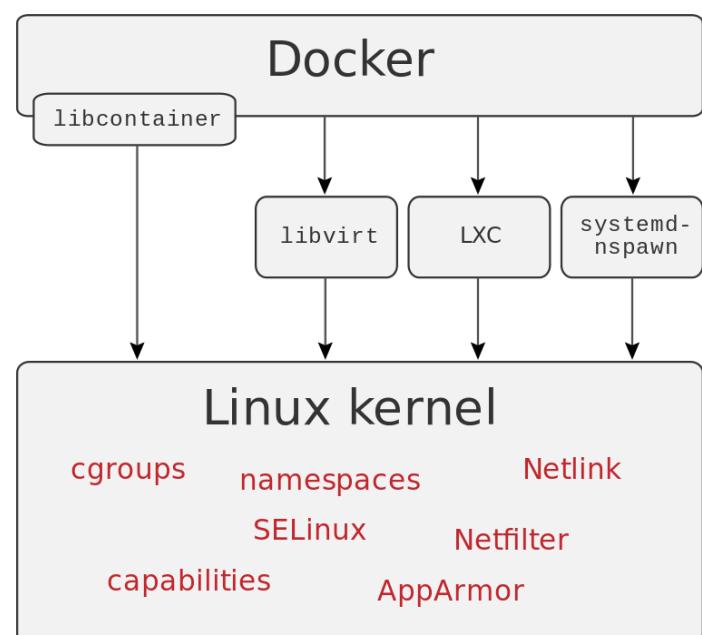


Docker internals

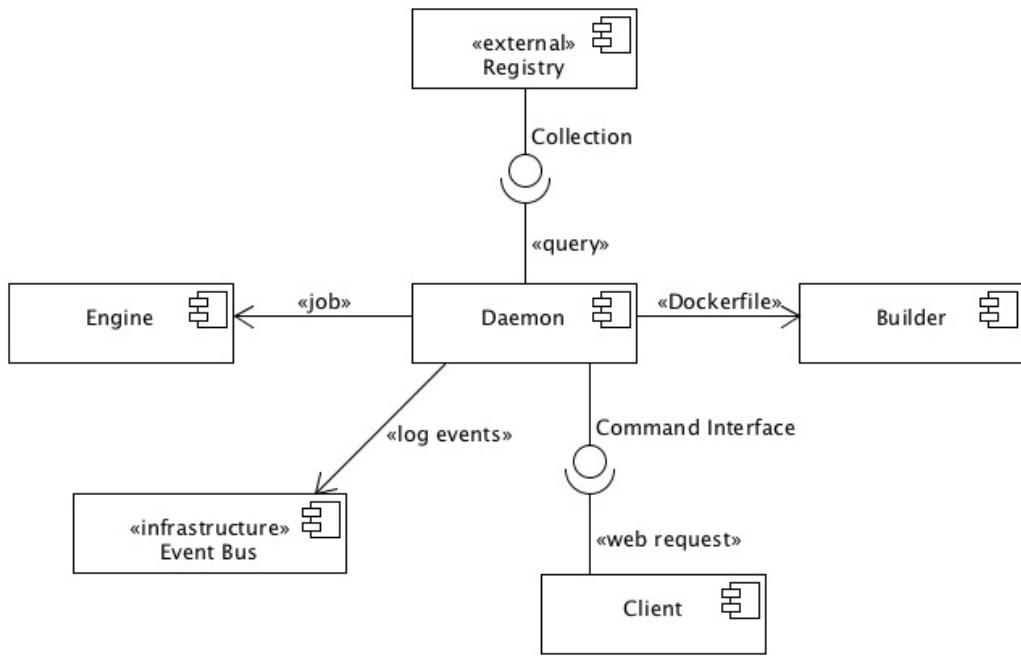
- Written in Go language
- With respect to other OS-level virtualization solutions, Docker is a higher-level platform that exploits Linux kernel mechanisms such as **cgroups** and **namespaces**
 - First versions were based on Linux Containers (LXC)
 - Then based on its own *libcontainer* runtime that uses Linux kernel namespaces and cgroups directly
- Docker adds to LXC
 - Portable deployment across machines
 - Versioning, i.e., git-like capabilities
 - Component reuse
 - Shared libraries, see Docker Hub hub.docker.com

Docker internals

- *libcontainer* (now included in *opencontainers/runc*): cross-system abstraction layer aimed to support a wide range of isolation technologies



Component diagram of Docker

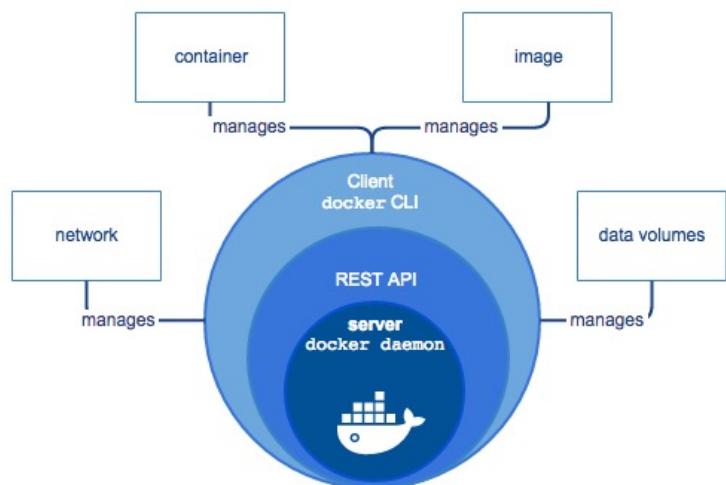


Valeria Cardellini - SDCC 2021/22

4

Docker engine

- **Docker Engine**: client-server application composed by:
 - Server, called Docker daemon
 - REST API which specifies interfaces that programs can use to control and interact with the daemon
 - Command line interface (CLI) client



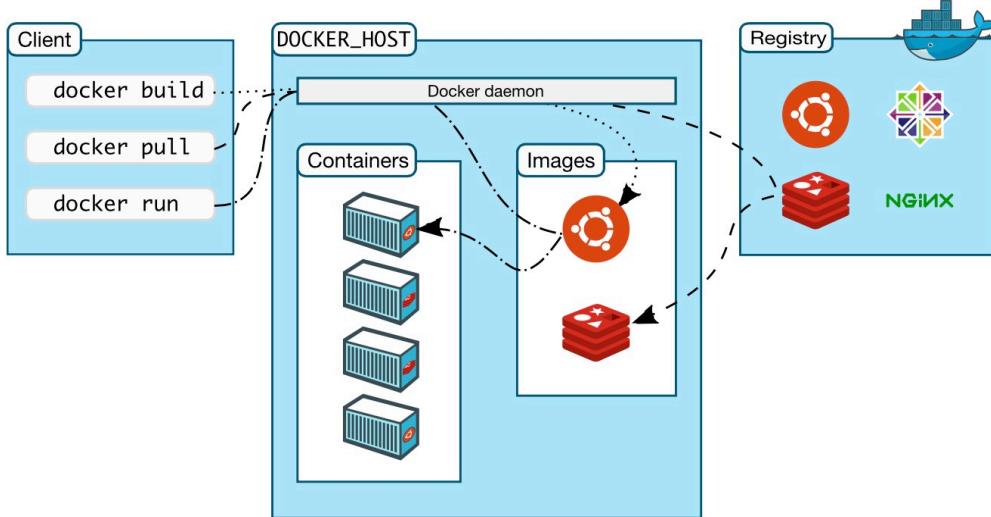
See <https://docs.docker.com/engine/docker-overview/>

Valeria Cardellini - SDCC 2021/22

5

Docker architecture

- Docker uses a client-server architecture
 - The Docker *client* talks to the Docker *daemon*, which **builds, runs, and distributes** Docker containers
 - Client and daemon communicate via sockets or REST API



Docker image

- Read-only template used to create a Docker container
- The **Build** component of Docker
 - Enables the distribution of apps with their runtime environment
 - Incorporates all the dependencies and configuration necessary to apps to run, eliminating the need to install packages and troubleshoot
 - Target machine must be Docker-enabled
- Docker can build images automatically by reading instructions from a **Dockerfile**
 - A text file with simple, well-defined syntax
- Images can be pulled and pushed towards a public/private registry
- Image name: **[registry/] [user/] name[:tag]**
 - Default for tag is **latest**

Docker image: Dockerfile

- Image can be created from a Dockerfile and a context
 - Dockerfile: instructions to assemble the image
 - Context: set of files (e.g., application, libraries)
 - Often, an image is based on a parent image (e.g., alpine)
- Dockerfile syntax

```
# Comment
INSTRUCTION arguments
```
- Instructions in a Dockerfile run in order
- Some instructions
 - FROM**: to specify parent image (mandatory)
 - RUN**: to execute any command in a new layer on top of current image and commit results
 - ENV**: to set environment variables
 - EXPOSE**: container listens on specified network ports at runtime
 - CMD**: to provide defaults for executing container

Docker image: Dockerfile

- Example of Dockerfile to build the image of a container that will run a simple todo list manager written in Node.js

```
# syntax=docker/dockerfile:1
FROM node:12-alpine
RUN apk add --no-cache python3 g++ make
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

See https://docs.docker.com/get-started/02_our_app/

Docker image: build

- Build image from Dockerfile

```
$ docker build [OPTIONS] PATH | URL | -
```

- E.g., to build the image for Node.js app (see Dockerfile in previous slide)

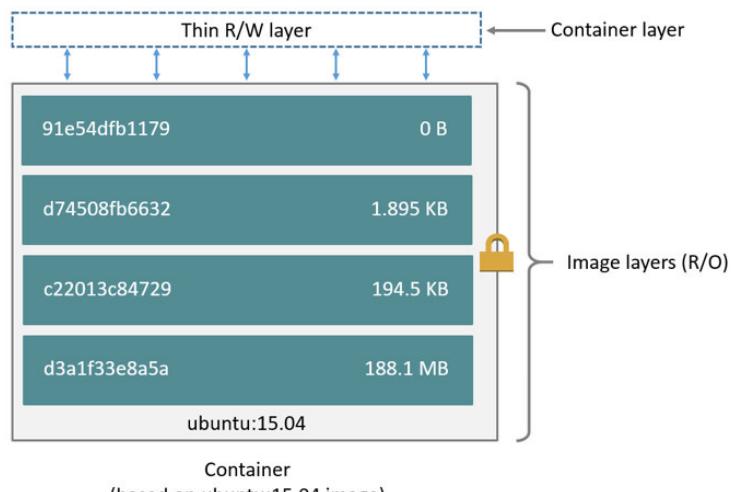
```
$ docker build -t getting-started .
```

```
# Use the official image as a parent image.  
FROM node:current-slim  
  
# Set the working directory.  
WORKDIR /usr/src/app  
  
# Copy the file from your host to your current location.  
COPY package.json .  
  
# Run the command inside your image filesystem.  
RUN npm install  
  
# Add metadata to the image to describe which port the container is listening on at runtime.  
EXPOSE 8080  
  
# Run the specified command within the container.  
CMD [ "npm", "start" ]  
  
# Copy the rest of your app's source code from your host to your image filesystem.  
COPY . .
```

10

Docker image: layers

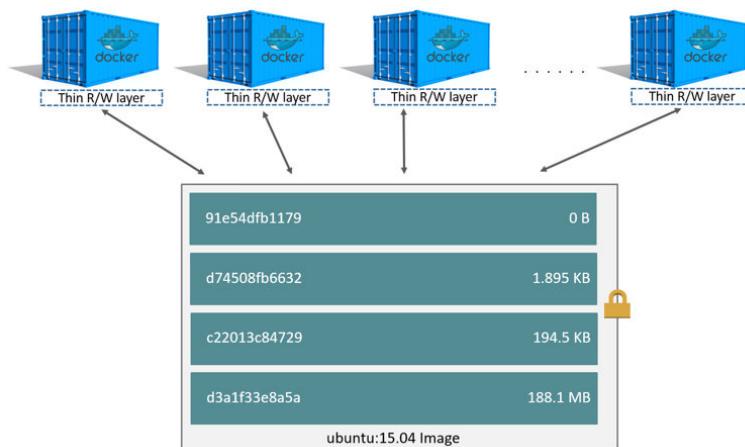
- Each image consists of a *series of layers*
- Docker uses *union file systems* to combine these layers into a single unified view
 - Layers are stacked on top of each other to form a base for a container's root file system
- Based on *copy-on-write* (COW) principle



Docker image: layers

- Layering pros
 - Enable layer sharing and reuse, installing common layers only once and saving bandwidth and storage space
 - Manage dependencies and separate concerns
 - Facilitate software specializations

See <https://docs.docker.com/storage/storagedriver/>



Docker image: layers and Dockerfile

- Each layer represents an instruction in the image's Dockerfile
 - CMD instruction specifies what command to run within the container: it only modifies the image's metadata and does not produce an image layer
- Each layer except the very last one is read-only
- Writable layer on top (aka *container layer*) is added when container is created
 - Changes made to running container (e.g., writing a file) are written to container layer
- To inspect an image, including image layers
 \$ `docker inspect imageid`

Docker image: storage

- Containers should be **stateless**. Ideally:
 - Very little data is written to container's writable layer
 - Best choice is to write data on **Docker volumes**
 - Nevertheless: some workloads require to write data to the container's writable layer

- The **storage driver** controls how *images* and *containers* are stored and managed on Docker host
- Multiple choices for storage driver
 - Including AuFS and **Overlay2** (at file level), Device Mapper, btrfs and zfs (at block level)
 - Storage driver's choice can affect the performance of containerized applications
 - See <https://dockr.ly/2FstUe6>

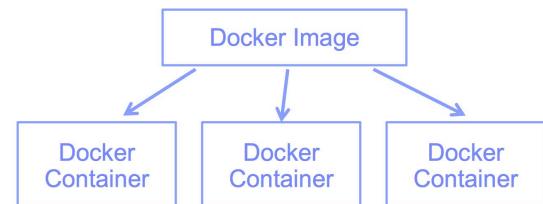
Valeria Cardellini - SDCC 2021/22

14

Docker container and registry

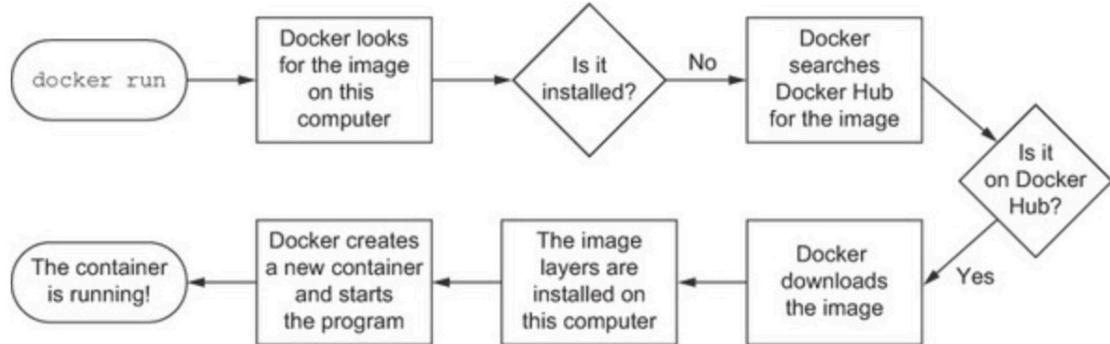
- **Docker container**: runnable instance of a Docker image
 - Run, start, stop, move, or delete a container using Docker API or CLI commands
 - The **Run** component of Docker
 - Docker containers are **stateless**: when a container is deleted, any data written not stored in a *data volume* is deleted along with the container

- **Docker registry**: stateless server-side application that stores and lets you distribute Docker images
 - Open library of images
 - The **Distribute** component of Docker
 - Docker-hosted registries: Docker Hub, Docker Store (open source and enterprise verified images)



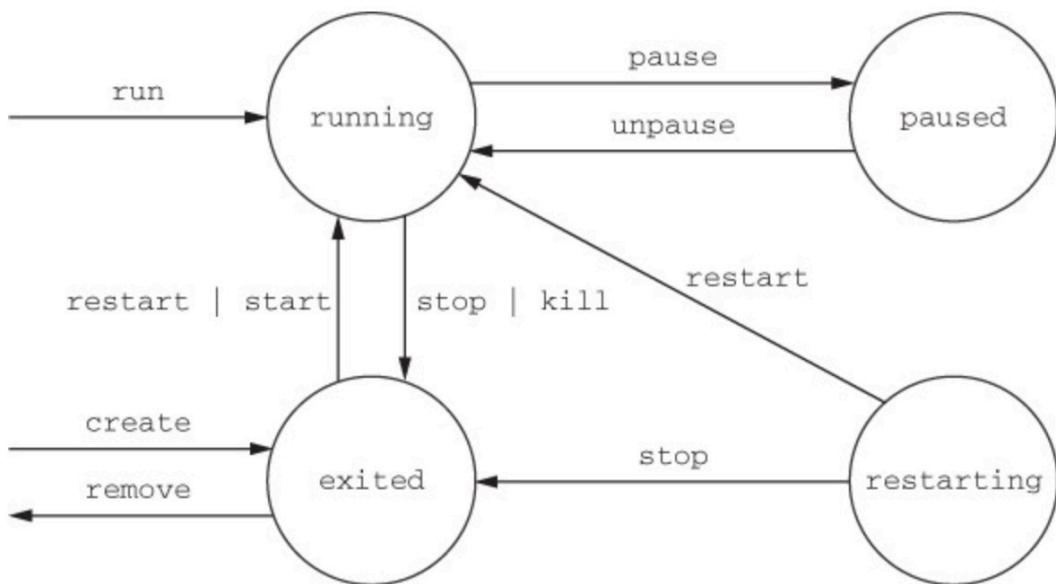
Docker: run command

- When you run a container whose image is not yet installed but is available on Docker Hub



Courtesy of “Docker in Action” by J. Nickoloff

State transitions of Docker containers



Courtesy of “Docker in Action” by J. Nickoloff

Commands: Docker info

- Obtain system-wide info on Docker installation

```
$ docker info
```

including:

- How many images, containers and their status
- Storage driver
- Operating system, architecture, total memory
- Docker registry

Commands: image handling

- List images on host (i.e., local repository)

```
$ docker images alternatively, $ docker image ls
```

- List every image, including intermediate image layers:

```
$ docker image ls -a
```

- Options to list images by name and tag, to list image digests (sha256), to filter images, to format the output

- E.g., to list untagged images (<none>) that have no relationship to any tagged images (no longer used but consume disk space)

```
$ docker images --filter "dangling=true"
```

- Remove an image

```
$ docker rmi imageid
```

Can also use [imagename](#) instead of [imageid](#)

Command: run

```
$ docker run [OPTIONS] IMAGE [COMMAND] [ARGS]
```

- Most common options

- name assign a name to the container
- d** detached mode (in background)
- i interactive (keep STDIN open even if not attached)
- t allocate a pseudo-tty
- expose expose a range of ports inside the container
- p publish a container's port or a range of ports to the host
- v bind and mount a volume
- e set environment variables
- link add link to other containers

Commands: containers management

- List containers
 - Only running containers: `$ docker ps`
 - Alternatively, `$ docker container ls`
 - All containers (including stopped or killed containers):
`$ docker ps -a`
- Manage container lifecycle
 - **Stop** running container
`$ docker stop containerid`
 - **Start** stopped container
`$ docker start containerid`
 - **Kill** running container
`$ docker kill containerid`
 - **Remove** container (need to stop it before attempting removal)
`$ docker rm containerid`

Can also use `containername` instead of `containerid`

Commands: containers management

- Stop and remove a running container

```
$ docker ps  
$ docker stop containerid  
$ docker ps -a  
$ docker rm containerid
```

- Stop all containers

```
$ for i in $(docker ps -q); do docker stop $i; done
```

Commands: containers management

- Inspect a container

- Most detailed view of the environment in which a container was launched

```
$ docker inspect containerid
```

- Copy files from and to container

```
$ docker cp containerid:path localpath  
$ docker cp localpath containerid:path
```

Docker volumes

- Preferred mechanism for persisting data generated by and used by Docker containers
 - New directory is created within Docker's storage directory on host machine, and Docker manages that directory's contents
 - Directory does not need to exist on host, it is created on demand if it does not yet exist
- To mount a volume, use -v or --mount flag
- More commands:
 - Create a volume: \$ docker volume create my-vol
 - List volumes: \$ docker volume ls
 - Inspect a volume: \$ docker volume inspect my-vol
 - Remove a volume: \$ docker volume rm my-vol

Docker volumes

- Example: start an nginx container with a volume

```
$ docker run -d \
--name devtest \
-v myvol2:/app
\ nginx:latest
```

Docker volumes

- Pros

- Completely managed by Docker
- Easy to back up or migrate
- Managed using Docker CLI commands or Docker API
- Work on both Linux and Windows containers.
- Can be more shared among multiple containers
- Their content can be encrypted
- Content can be pre-populated
- Better choice than persisting data in a container's writable layer: a volume does not increase the size of the containers using it, and its contents exist outside the container lifecycle

Hands-on Docker

- Download and install Docker
 - Available on multiple platforms
<https://docs.docker.com/get-docker/>
<https://docs.docker.com/get-started/>
- Test Docker version

```
$ docker --version
```
- Test Docker installation by running hello-world Docker image

```
$ docker run hello-world
```

Hands-on Docker

- Run the “Hello World” container

```
$ docker run alpine /bin/echo 'Hello world'
```

- alpine: lightweight Linux distro with reduced image size

- Use commands to:

- List containers and container images
- Remove containers and container images

Hands-on Docker

- Run a nginx Web server inside a container

- We bind the container to a specific port

```
$ docker run -dp 80:80 nginx --name web
```

Option -p: publish container port (80) to host port (80)

- Send HTTP request through Web browser

- First retrieve the hostname of the host machine (e.g., localhost if the browser is on the same machine)

- Send HTTP request through an interactive container using a **bridge network**

```
$ docker network create my_net
$ docker run -dp 80:80 --name web --net=my_net nginx
$ docker run -i -t --net=my_net --name web_test busybox
/ # wget -O - http://web:80/
/ # exit
```

Hands-on Docker

• Running Apache web server with minimal index page

1. Define container image with Dockerfile
 - Define image starting from Ubuntu, install and configure Apache
 - Incoming port set to 80 using EXPOSE instruction

```
FROM ubuntu:18.04

# Install dependencies
RUN apt-get update
RUN apt-get -y install apache2
# Install apache and write hello world message
RUN echo 'Hello World!' > /var/www/html/index.html
# Configure apache
RUN echo '. /etc/apache2/envvars' >> /root/run_apache.sh
RUN echo 'mkdir -p /var/run/apache2' >> /root/run_apache.sh
RUN echo 'mkdir -p /var/lock/apache2' >> /root/run_apache.sh
RUN echo '/usr/sbin/apache2 -D FOREGROUND' >> /root/run_apache.sh
RUN chmod 755 /root/run_apache.sh

EXPOSE 80
CMD /root/run_apache.sh
```

Hands-on Docker

2. Build container image from Dockerfile

```
$ docker build -t hello-apache .
```

3. Run container and bind

```
$ docker run -dp 80:80 hello-apache
```

- To reduce Docker image size let's improve the Dockerfile: avoid adding unnecessary layers
- E.g., update and install multiple packages in a single RUN instruction
 - To get a clear, comprehensive line, use \ to type out the command in multiple lines

Hands-on Docker

```
FROM ubuntu:18.04

# Install dependencies
RUN apt-get update && \
    apt-get -y install apache2

# Install apache and write hello world message
RUN echo 'Hello World!' > /var/www/html/index.html

# Configure apache
RUN echo '. /etc/apache2/envvars' > /root/run_apache.sh && \
    echo 'mkdir -p /var/run/apache2' >> /root/run_apache.sh && \
    echo 'mkdir -p /var/lock/apache2' >> /root/run_apache.sh && \
    echo '/usr/sbin/apache2 -D FOREGROUND' >> /root/run_apache.sh && \
    \
    chmod 755 /root/run_apache.sh

EXPOSE 80

CMD /root/run_apache.sh
```

Configuring container memory and CPU

- By default, a container has no resource constraints
 - Can use as much resource as host's kernel scheduler allows
- Docker provides ways to control how much memory or CPU a container can use by setting runtime configuration flags of docker run command
https://docs.docker.com/config/containers/resource_constraints/
 - Docker engine implements configuration changes by modifying settings for container's cgroup

Configuring container memory

- **Avoid running out of memory (OOM)**
 - Individual containers can be killed (Docker daemon has lower OOM priority, containers default one)
- **Docker can enforce hard or soft memory limits**
 - **Hard** limits: container cannot use more than a given amount of user or system memory; `--memory` flag
 - **Soft** limits: container can use as much memory as it needs unless certain conditions are met, such as when kernel detects contention or low memory on host machine
 - Example: limit container to use at most 500 MB of memory (hard limit) and specify also a soft limit
 - \$ `docker run -it --memory-reservation="300m" \ --memory="500m" ubuntu /bin/bash`

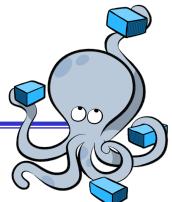
Configuring container CPU

- Various constraints to limit container usage of host machine's CPU cycles
- Some options
 - `--cpus=<value>`: limit how many CPU resources a container can use (hard limit)
 - `--cpu-quota=<value>`: set CPU Completely Fair Scheduler (CFS) quota on container
 - `--cpuset-cpus`: limit specific CPUs or cores a container can use
 - `--cpu-shares`: set to a value greater or less than default of 1024 to increase or reduce container's weight, and give it access to a greater or lesser proportion of CPU cycles (soft limit)
 - Example: limit container to use at most 50% of CPU every second
 - \$ `docker run -it --cpus=".5" ubuntu /bin/bash`
 - \$ `docker run -it --cpu-period=100000 \ --cpu-quota=50000 ubuntu /bin/bash`

Multi-container Docker applications

- How to run multi-container Docker apps?
- **Docker Compose**
 - Deployment only on single host
- Docker Swarm
 - Native orchestration tool for Docker
 - Deployment on multiple hosts
- Kubernetes
 - Deployment on multiple hosts
 - See next lesson

Docker Compose



- To coordinate the execution of multiple containers running on a single machine
<https://docs.docker.com/compose/>
 - Not bundled within Docker installation (on Linux)
<https://docs.docker.com/compose/install/>
- Allows to easily express the containers to be instantiated at once, and their relationships
- Runs the composition on a single Docker engine
 - Use Docker Swarm or Kubernetes if you need to deploy containers on multiple nodes

Docker Compose

- Specify how to compose containers in an easy-to-read YAML file named `docker-compose.yml`
- To start Docker composition (background -d):
\$ docker-compose up -d
- By default, Docker Compose looks for `docker-compose.yml` in current working directory
 - Specify different file using -f flag
- To stop Docker composition:
\$ docker-compose down

Docker Compose file

- Different versions of Docker Compose file format
<https://docs.docker.com/compose/compose-file/>

Latest: Docker Compose 1.27 implements the format defined by
[Compose Specification](#)

```
version: '3'

services:
  storm-nimbus:
    image: storm
    container_name: nimbus
    command: storm nimbus
    depends_on:
      - zookeeper
    links:
      - zookeeper
    ports:
      - "6627:6627"
  zookeeper:
    image: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
  worker1:
    image: storm
    command: storm supervisor
    depends_on:
      - storm-nimbus
      - zookeeper
    links:
      - storm-nimbus
      - zookeeper
```

Docker Compose: example

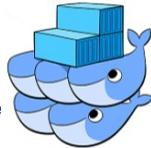
- Simple Python web app running on Docker Compose
 - Two containers: Python web app and Redis
 - Use Flask framework and maintain a hit counter in Redis
 - See <https://docs.docker.com/compose/gettingstarted/>
- Steps:
 1. Write Python app
 2. Define Python container image with Dockerfile
 3. Define services in docker-compose.yml file
 - Two services: web (image defined by Dockerfile) and redis (image pulled from Docker Hub)

```
version: "3.9"
services:
  web:
    build: .
    ports: - "5000:5000"
  redis:
    image: "redis:alpine"
```

Docker Compose: example

- Steps (cont'd):
 4. Build and run the app with Compose
\$ docker-compose up -d
 5. Send HTTP requests using curl or browser ([counter is increased](#))
 6. Stop Compose, bringing everything down
\$ docker-compose down
- Can also add a volume for the web service to keep the app code, so that code can be modified on the fly without rebuilding the image
- Many examples of Compose files
<https://github.com/docker/awesome-compose>

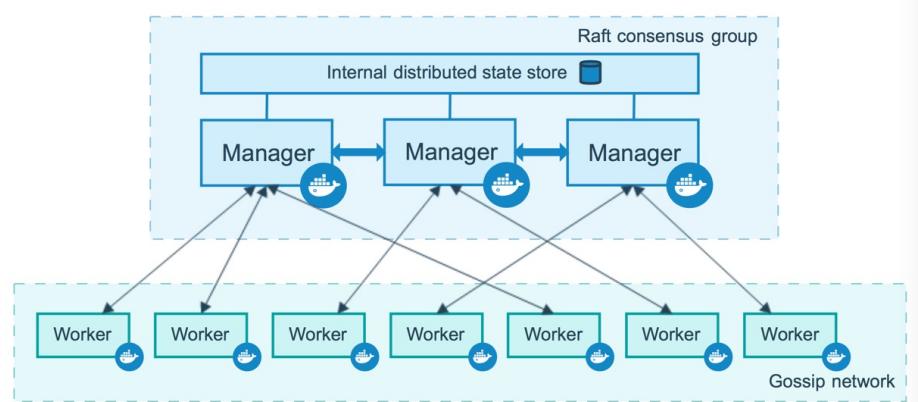
Docker Swarm



- Docker includes **swarm mode** for natively managing a **cluster of Docker Engines**, called **swarm**
<https://docs.docker.com/engine/swarm/>
- **Tasks**: containers running in a **service**
- Basic features of swarm mode:
 - **Scaling**: number of tasks for each service
 - **State reconciliation**: Swarm monitors cluster state and reconciles any differences w.r.t. desired state (e.g., replace containers after host failure)
 - **Multi-host networking**: to specify an overlay network among services
 - **Load balancing**: allows to expose the ports for services to an external load balancer. Internally, the swarm lets you specify how to distribute containers among nodes

Docker Swarm: architecture

- A swarm consists of multiple Docker engines which run in swarm mode
- Node: instance of Docker engine
 - **Manager node(s)**: handles cluster management, including scheduling tasks to worker nodes
 - Multiple managers to improve fault tolerance
 - Raft as consensus algorithm to manage global cluster state
 - **Worker nodes** execute tasks



Docker Swarm: Swarm cluster

- Create a swarm: manager node

```
$ docker swarm init --advertise-addr <MANAGER-IP>
Swarm initialized: current node (<nodeid>) is now a manager.
To add a worker to this swarm, run the following command:

  docker swarm join --token <token> <manager-ip>:port
```

- Create a swarm: worker node

```
$ docker swarm join --token <token> <manager-ip>:port
```

- Inspect status

```
$ docker info
```

```
$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
<nodeid> *	controller	Ready	Active	Leader
<nodeid>	storage	Ready	Active	

Docker Swarm: Swarm cluster

- Leave the swarm

- If the node is a manager node, warning about maintaining the quorum (to override warning, --force flag)

```
$ docker swarm leave
```

- After a node leaves the swarm, you can run docker node rm command on a manager node to remove the node from the node list

```
$ docker node rm node-id
```

Docker Swarm: manage services

- Deploy a service to the swarm (from manager node)

```
$ docker service create -d --replicas 1 \
--name helloworld alpine ping docker.com
```

- List running services

```
$ docker service ls
```

ID	NAME	MODE	REPLICAS	IMAGE	PORTS
<serviceid>	helloworld	replicated	1/1	alpine:latest	

Docker Swarm: manage services

- Inspect service

```
$ docker service inspect --pretty <SERVICE-ID>
$ docker service ps <SERVICE-ID>
```

ID	NAME	IMAGE	NODE	DESIRED ST	CURRENT ST	ERROR	PORTS
<cont.id1>	helloworld.1	alpine:latest	controller	Running	Running	...	
<cont.id2>	helloworld.2	alpine:latest	storage	Running	Running	...	

- Inspect container

```
$ docker ps <cont.id1>
```

Manager node

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	... NAMES
<cont.id1>	alpine:latest	"ping docker.com"	2 min ago	Up 2 min	helloworld.1.iuk1sj...

Worker node

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	... NAMES
<cont.id2>	alpine:latest	"ping docker.com"	2 min ago	Up 2 min	helloworld.2.skfos4...

Docker Swarm: manage services

- Scale service

```
$ docker service scale <SERVICE-ID>=<NUMBER-OF-TASKS>
```

Swarm manager will automatically enact the updates

- Apply rolling updates to a service

```
$ docker service update --limit-cpu 2 redis  
$ docker service update --replicas 2 helloworld
```

- Roll back an update

```
$ docker service rollback [OPTIONS] <SERVICE-ID>
```

- Remove a service

```
$ docker service rm <SERVICE-ID>
```

Torna alle slide sulla virtualizzazione.