

Docker Fundamentals

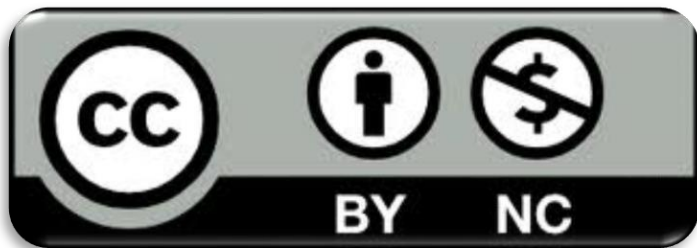


License & Disclaimer

2

License Information

This presentation is licensed under the
Creative Commons BY-NC License



To view a copy of the license, visit:

<http://creativecommons.org/licenses/by-nc/3.0/legalcode>

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Goal

3

- Present the difference between virtualized and container Deployment
- Present basic usage and command of Docker and Docker Compose

Prerequisites

4

➤ Lecture:

➤ *NS_0.1 – Network Fundamentals*

Outline

5

- Docker images and containers
- Docker networking
- Docker compose

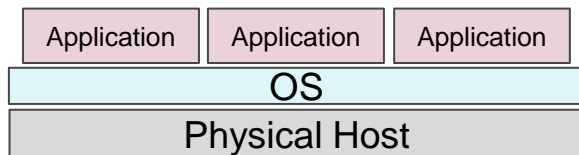
Outline

6

- Docker images and containers
- Docker networking
- Docker compose

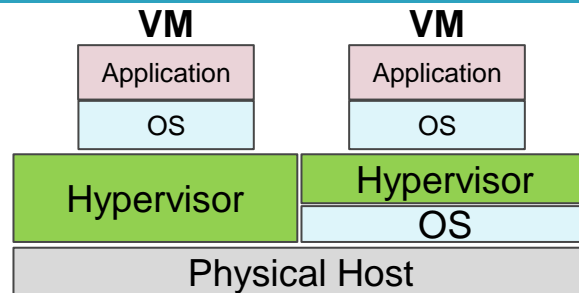
Traditional vs Virtualized Deployment

7



- Physical Hosts run an Operating System (e.g., Windows or Linux).
- Multiple applications run on the shared OS.

¹<https://www.virtualbox.org/>



- A special software, i.e., the **Hypervisor**, provides Virtual Machines.
- Examples of such technologies are *Virtualbox*¹ or *Linux KVM*².
- VM is a full machine running all the components, including its own Operating System, on top of the virtualized hardware

²<https://www.linux-kvm.org>

Traditional vs Virtualized Deployment

8

Traditional Deployment

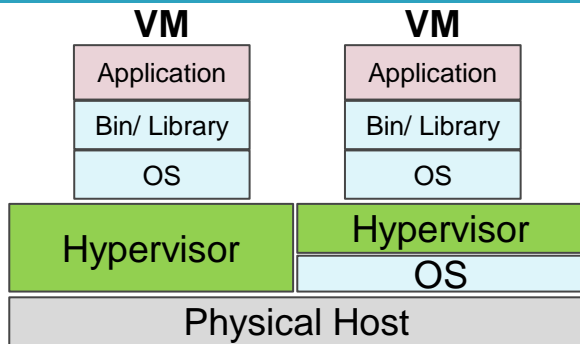
- No way to define resource boundaries for applications.
- Isolating applications requires running them on different physical servers (expensive and resources could be underutilized).

Virtualized Deployment

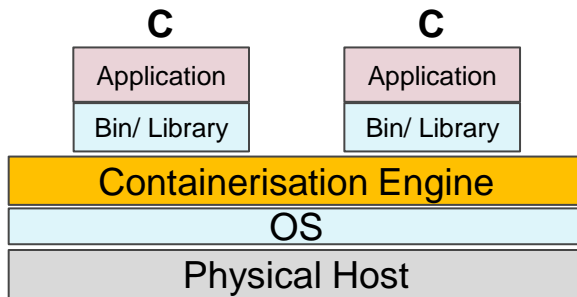
- Virtualization allows:
 - applications to be isolated between VMs
 - better utilization of resources in a physical server
 - better scalability.

Virtualized vs Container Deployment

9



- Virtual hardware
 - Each VM has an OS and Application
 - Share hardware resource from the Physical Host



- Virtual Operating Systems
 - Isolated environments, namely **containers**, sharing the same *real* operating system
 - Containers run from a distinct image that provides all files (Bin/, Library) necessary to support them
 - Examples of such technologies is *Docker*¹.

¹<https://www.docker.com/>

Virtualized vs Container Deployment

10

Virtualized Deployment

- Heavyweight: each VM relies on a full copy of an Operating System.
- Provides full isolation.
- Best suited for when you have applications that need to run on *different* Operating System flavors.

Container Deployment

- Lightweight: sharing OS resources significantly reduces the overhead required for running containers.
- Provides a (relaxed) process-level isolation.
- Best suited for when you have applications that need to run over a *single* Operating System kernel.

Docker images and containers

11

Image

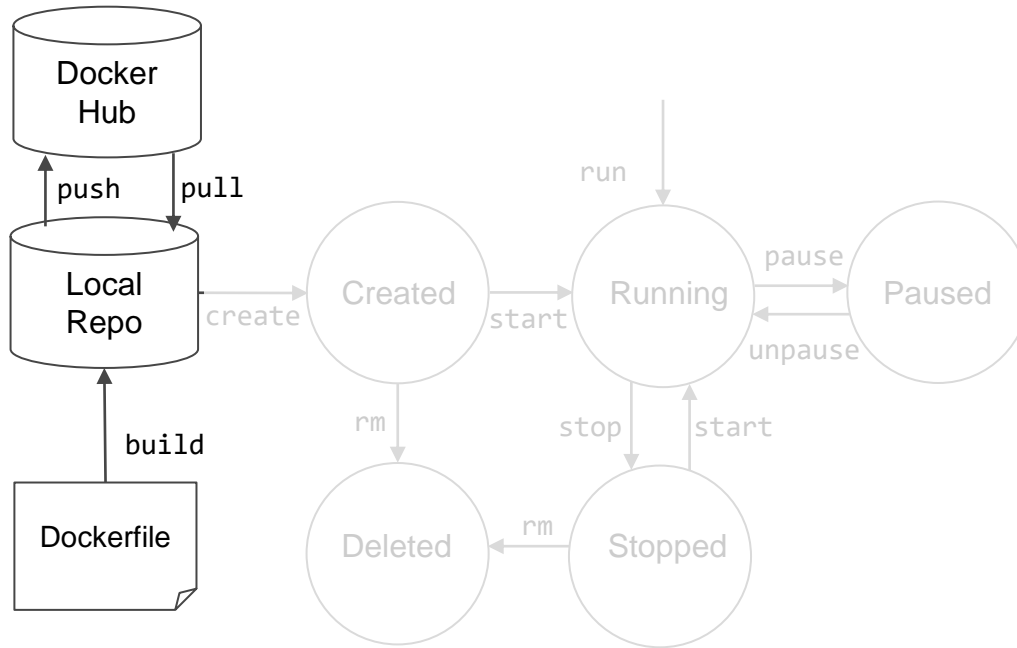
- **Immutable** template for containers
- Includes everything needed to run an application
 - code, runtime, system tools, system libraries, and settings

Container

- An instance of an image
- Add a **new writable layer** on top of the underlying image
 - all changes made to the running container (e.g., writing new files or modifying existing files) are written to this writable container layer

Docker container lifecycle

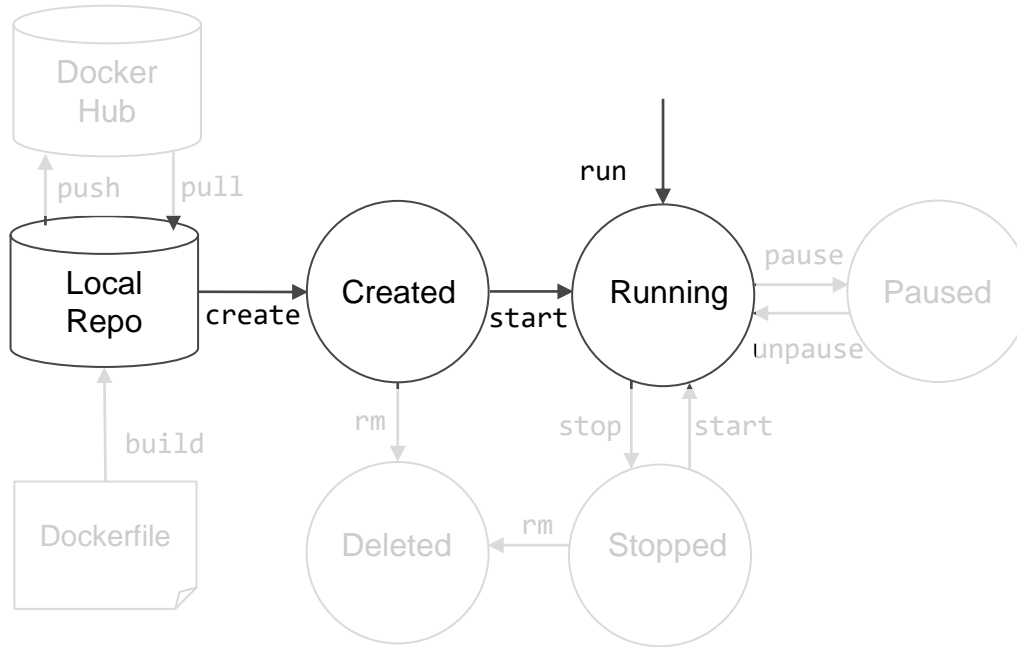
12



- Docker images can be pulled from a central repository (Docker Hub)
- Pulled images are saved in a local repository
- Custom images can be created and saved starting from a specific configuration file (Dockerfile)
- Custom images can be pulled to the central repository

Docker container lifecycle

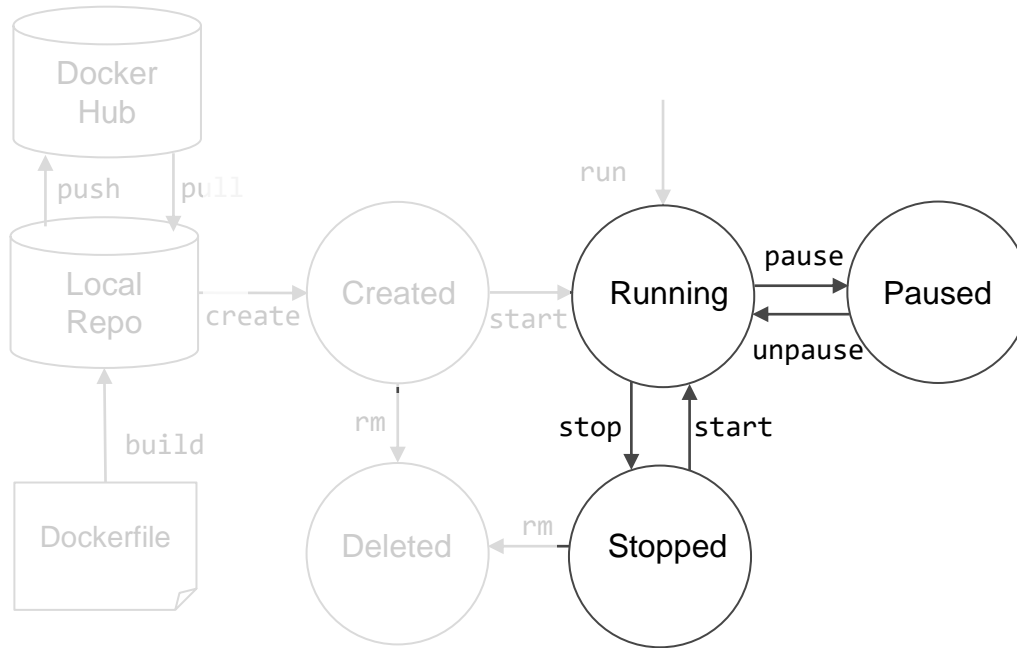
13



- A saved image can be used for creating a container (a writeable layer is added)
- Starting a container means running a default command contained in the image, namely the entrypoint (can be overridden)
- A container can be created and started using a single `run` command

Docker container lifecycle

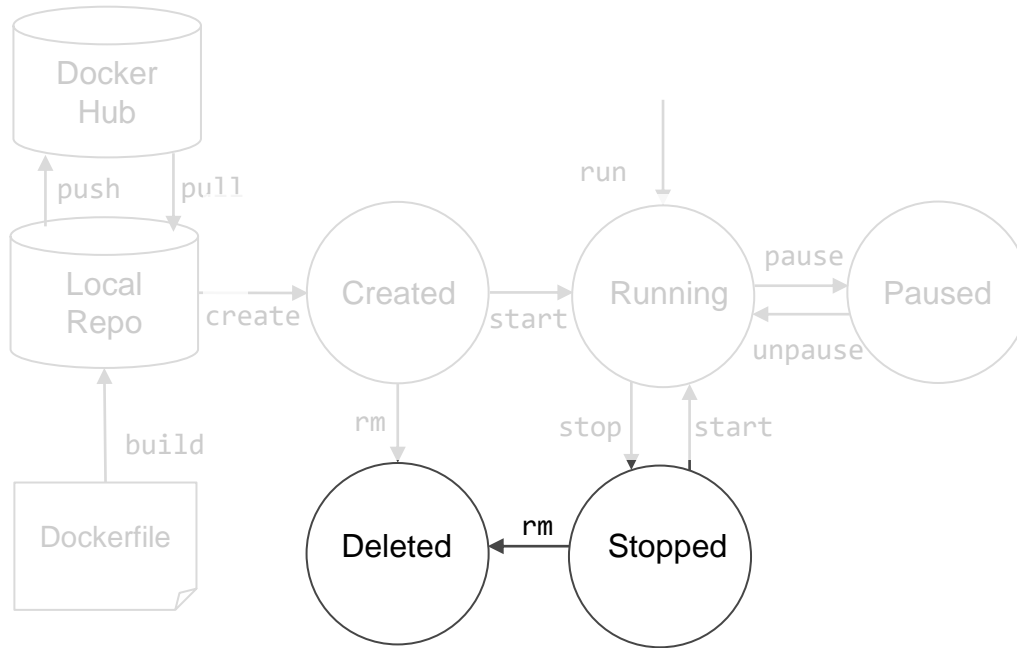
14



- After executing the entrypoint, the container stops
- A **foreground** process specified as the entrypoint can keep running the container
- A running container can be paused or stopped

Docker container lifecycle

15



- After a container is stopped, the writeable layer still exists
- Deleting a container permanently removes the associated writable layer

Docker lifecycle example: images

16

- Pull an image from the central hub
 - `docker pull <image>`
- Build an image from a Dockerfile
 - `docker build -t <image>`
- List images saved in the local repository
 - `docker images`
- Delete an image
 - `docker rmi <image>`

Docker lifecycle example: containers

17

- Start a container
 - `docker run <image>`
- Start a container overriding default entrypoint with `<newcmd>`
 - `docker run --entrypoint <newcmd> <image>`
- Execute a command `<cmd>` (e.g., `/bin/bash`) in a running container
 - `docker exec -it <containerID> <cmd>`
- Stop a container
 - `docker stop <containerID>`
- Remove a container
 - `docker rm <containerID>`
- List running and stopped containers
 - `docker ps -a`

Docker volumes

18

- Volumes can be used to save (persist) data and to share data between containers
- Volume is unrelated to the container layers: deleting a container does not involve deleting an associated volume
- A volume can be:
 - (anonymous/)named: managed internally by Docker itself
 - host: refers to a filesystem location of the host running Docker

Docker volumes: example

19

- Create a named volume
 - `docker volume create volumename`
- Run a container using the named volume
 - `docker run -v volumename:/path/in/container_filesystem`
- Running a container using a host volume
 - `docker run -v /path/on/host_filesystem:/path/in/container_filesystem`

Dockerfile

20

- Docker can build custom images automatically by reading the instructions from a Dockerfile
- Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image
- The *docker build* allows the execution of an automated build of an image starting from a Dockerfile

(Dockerfile reference: <https://docs.docker.com/engine/reference/builder/>)

Dockerfile example

21

Command	Description	Example
FROM <image>	Start building from this (base) image	FROM ubuntu
RUN <cmd>	Run the specified command	RUN apt install apache2
COPY <src> <dest>	Copy a file to the image fs	COPY vh.conf /etc/apache2/conf/
CMD ["exec", "param1", ...]	Configure the default command when container starts	CMD ["apache2", "-D", "FOREGROUND"]

Outline

22

- Docker images and containers
- **Docker networking**
- Docker compose

Docker networking

23

Docker supports different configurations, the two main ones being

➤ *Bridge* (default)

- isolated layer 3 networks enabling connected containers to communicate
- (can) allow the access to external networks masquerading connections with the host network configuration

➤ *Host*

- containers use the host network
- listening ports are exposed to the outside world

Docker networking: published ports

24

- A container connected to bridges is isolated and does not expose any of its ports to the outside world
- A published port can be made available to services running outside the container
- A published port is mapped to a port on the Docker host

Docker networking: examples

25

- Create network with a configured subnet and gateway address
 - `docker network create --driver bridge <networkname> --subnet=<ip/mask> --gateway=<ip/mask>`
- Connect a container to a network
 - `docker network connect <networkname> <containerid>`
- Run a container and expose a port
 - `docker run -h <host-name> -p <internal-port>:<exposed-port> --name <container-name> <image-name>`

Outline

26

- Docker images and containers
- Docker networking
- **Docker compose**

Docker compose

27

Compose is a tool for defining and running multi-container Docker applications

- A single file for providing configurations
- A single set of commands for configuring, building, and running all the containers

Docker compose configuration

28

- The Compose file (`docker-compose.yaml`) uses a standard, human-readable syntax, namely YAML* syntax. It defines
 - Services: configuration that is applied to each container (much like passing command-line parameters to *docker run*)
 - Networks (optional): define configuration of networks to be created
 - Volumes (optional): define configuration of volumes to be created

* <https://yaml.org/>

Docker compose configuration: example

29

services:

frontend container

app:

use a custom image

build:

directory containing Dockerfile

context: ./app

image name

image: custom_image

exposed ports (host:container)

ports:

- 8080:80

a mapped host volume

volumes:

- ./config/config.json:/etc/config.json

connected networks (defined in networks..)

networks:

ext:

ipv4_address: 192.168.100.100

int:

backend container

db:

pull an existing image

image: mariadb

a mapped named volume

volumes:

- db-content:/var/lib/mysql

networks:

int:

networks:

ext:

driver: bridge

ipam:

driver: default

config:

- subnet: 192.168.100/24

int:

driver: bridge

volumes:

db-content:

Docker compose: commands example

30

- (build images and) run services
 - `docker-compose up -d`
- stop services
 - `docker-compose stop`
- start services
 - `docker-compose start -d`
- stop and remove containers and networks
 - `docker-compose down`
- show logs (entrypoint output)
 - `docker-compose logs -f [service_name]`

Docker Fundamentals

