Middleware Architectures 1

Lecture 4: Containers

doc. Ing. Tomáš Vitvar, Ph.D.
tomas@vitvar.com • @TomasVitvar • https://vitvar.com

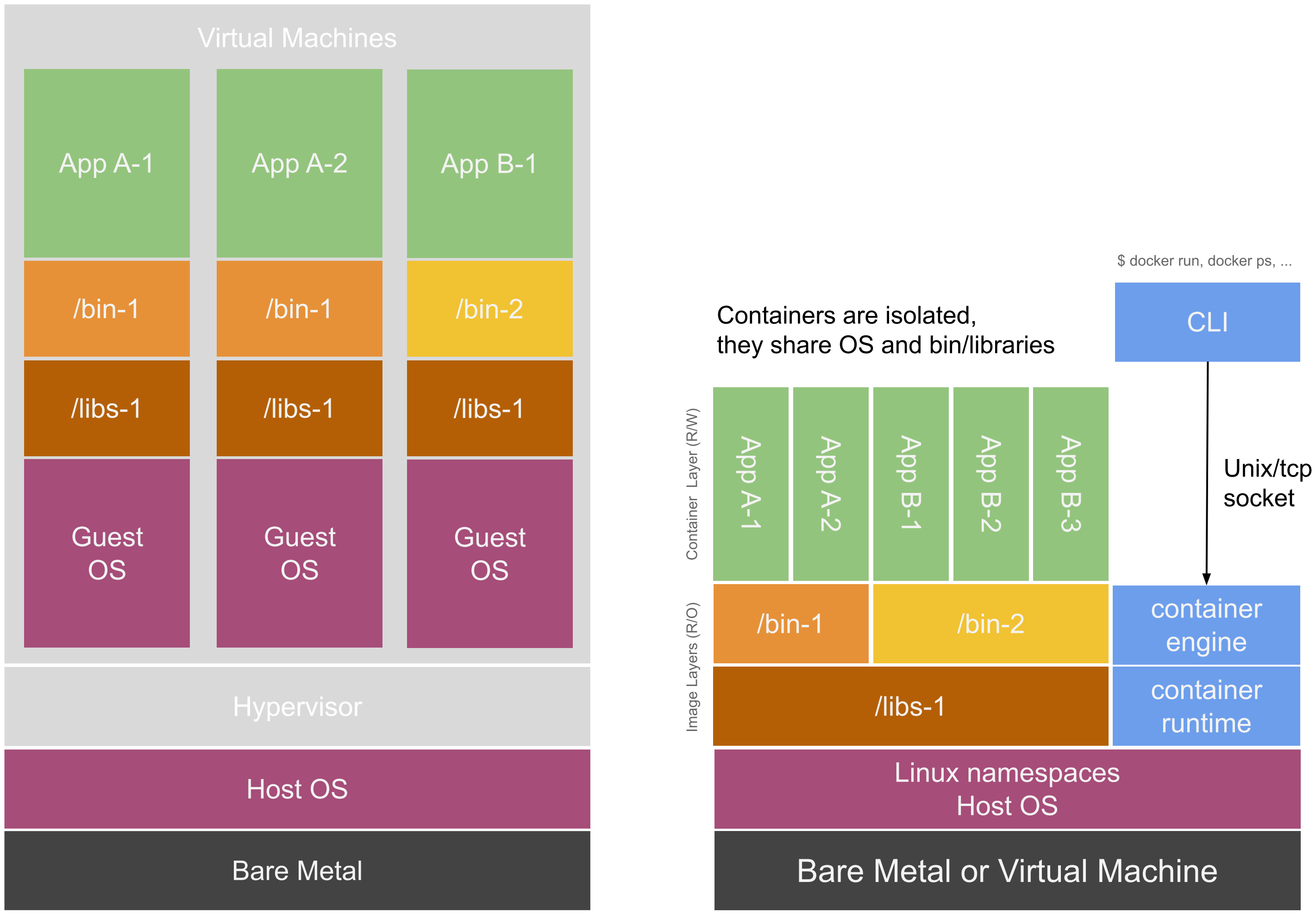
Czech Technical University in Prague
Faculty of Information Technologies • Software and Web Engineering • https://vitvar.com/lectures

Modified: Mon Oct 27 2025, 06:42:02
Humla v1.0

Overview

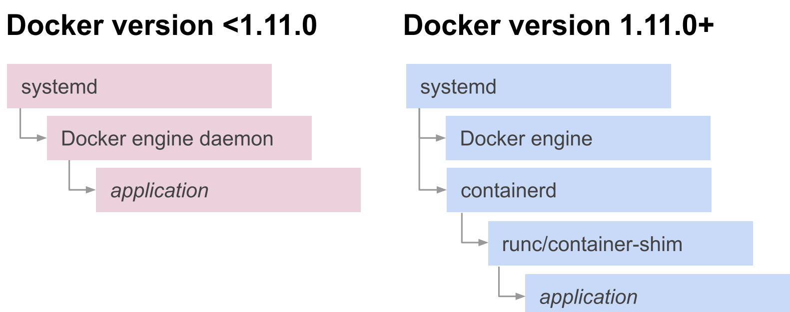
- Overview
- Linux Namespaces
- Container Image

Virtual Machines vs. Containers

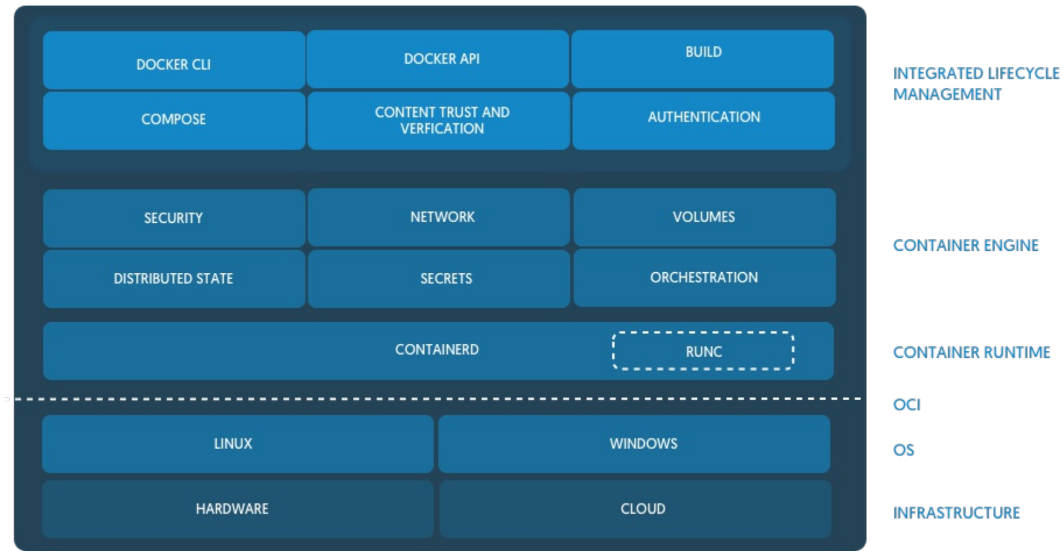


Overview

- Linux Containers
  - *Introduced in 2008*
  - *Allow to run a process tree in a isolated system-level "virtualization"*
  - *Use much less resources and disk space than traditional virtualization*
- Implementations
  - *LXC – default implementation in Linux*
  - *Docker Containers*
    - → *Builds on Linux namespaces and union file system (OverlayFS)*
    - → *A way to build, commit and share images*
    - → *Build images using a description file called Dockerfile*
    - → *Large number of available base and re-usable images*
- Monolithic design originally
  - *Now several layers*
  - *container runtime*
  - *container engine*



Containerd



- Container engine
  - *Accepts user inputs (via CLI or API), pulling images from registry, preparing metadata to be passed to container runtime*
- Container runtime
  - *Abstraction from syscalls or OS specific functionality to run containers on linux, windows, solaris, etc.*
  - *Uses* `runc` *and* `container-shim`
  - *Communicates with kernel to start containerized processes*

Terminology

- Image
  - *An image contains a union of layered filesystems stacked on top of each other*
  - *Immutable, it does not have state and it never changes*
- Container
  - *One or more processes running in one or more isolated namespaces in a filesystem provided by the image*
- Container Engine/Runtime
  - *The core processes providing container capabilities on a host*
- Client
  - *An app (e.g. CLI, custom app), communicates with a container engine by its API*
- Registry
  - *A hosted service containing repository of images*
  - *A registry provides a registry API to search, pull and push images*
  - *Docker Hub is the default Docker registry*
- Swarm
  - *A cluster of one or more docker engines*

Overview

- Overview
- Linux Namespaces
- Container Image

Linux Namespaces

- Isolation of Linux processes, there are **7 namespaces**
  - *Mount, UTS, IPC, PID, Network, User, Cgroup*
  - *By default, every process is a member of a default namespace of each type*
  - *In case no additional namespace configuration is in place, processes and all their direct children will reside in this exact namespace*
  - *Run* `lsns` *to check namespaces the process is in*

```
$ lsns
NS         TYPE   NPROCS   PID USER   COMMAND
4026531836 pid         2 30873 oracle -bash
4026531837 user      108  1636 oracle /bin/bash /u01/oracle/scripts/startWebLogicContainer.sh
4026531838 uts         2 30873 oracle -bash
4026531839 ipc         2 30873 oracle -bash
4026531840 mnt         2 30873 oracle -bash
4026531956 net       108  1636 oracle /bin/bash /u01/oracle/scripts/startWebLogicContainer.sh
4026532185 mnt        13 13542 oracle /bin/bash /u01/oracle/scripts/startNM_ohs.sh
4026532192 pid        13  2798 oracle /bin/bash /u01/oracle/scripts/startNM_ohs.sh
...
```

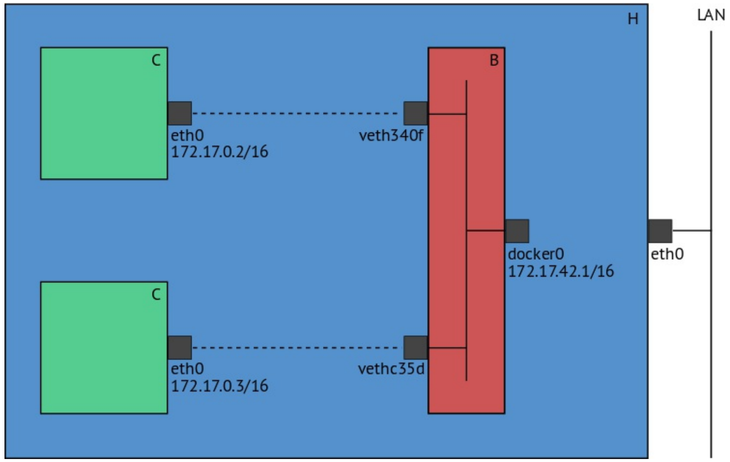
- Flexible configuration, for example:
  - *You can run two apps that only share the network namespace, e.g.* `4026531956`
  - *The apps can talk to each other*
  - *Any other app (not in this namespace) won't be able to talk to the apps*

Types: mnt, uts, ipc and pid

- `mnt` namespace
  - *Isolates filesystem mount points*
  - *Restricts the view of the global file hierarchy*
  - *Each namespace has its own set of mount points*
- `uts` namespace
  - *The value of the hostname is isolated between different UTS namespaces*
- `ipc` namespace
  - *Isolates interprocess communication resources*
  - *message queues, semaphore, and shared memory*
- `pid` namespace
  - *Isolates PID number space*
  - *A process ID number space gets isolated*
    → *Processes can have PIDs starting from the value 1*
    → *Real PIDs outside of the namespace of the same process is a different number*
  - *Containers have their own init processes with a PID value of 1*

Types: net

- `net` namespace
  - *Processes have their own private network stack (interfaces, routing tables, sockets)*
  - *Communication with external network stack is done by a virtual ethernet bridge*



  - *On the host there is a **userland proxy** or **NAT***
    - → *NAT is a prefered solution over userland proxy (`/usr/bin/docker-proxy`)*
    - → *Lack of NAT hairpinning may prevent to use NAT*
  - *Use case*
    - → *Multiple services binding to the same port on a single machine, e.g.* `tcp/80`
    - → *A port in the host is mapped to the port exposed by a process in the NS*

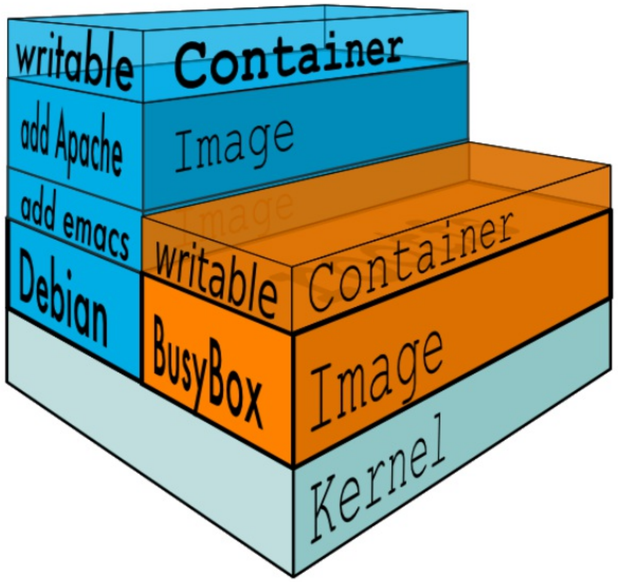
Types: User & Cgroup

- `user` namespace
  - *Isolates user and group IDs (UIDs/GIDs) between processes*
  - *Allows a process to have different privileges inside and outside the namespace*
  - *Enables **rootless containers** (process is non-root on host, but appears as root inside)*
  - *Example*
    - → *A process runs as UID 0 (root) inside the container, but maps to a regular UID on the host*
- `cgroup` namespace
  - ***cgroups** (control groups)*
    - → *Kernel feature to limit and measure process resource usage (CPU, memory, I/O)*
  - *cgroup namespace*
    - → *Isolates the view of the cgroup hierarchy for each process*
    - → *Prevents a container from seeing/modifying cgroups of the host/other containers*
    - → *Improves security by restricting what resource controls a container can observe*
    - → ***Example:** A container only sees its own CPU/memory usage limits, not the host's full cgroup tree*

Overview

- Overview
- Linux Namespaces
- Container Image

Container Images



– *Containers are made up of R/O layers via a storage driver (OverlayFS, AUFS, etc.)*
– *Containers are designed to support a single application*
– *Instances are ephemeral, persistent data is stored in bind mounts or data volume containers.*

Image Layering with OverlayFS

- OverlayFS
  - *A filesystem service implementing a **union mount** for other file systems.*
  - *Docker uses* `overlay` *and* `overlay2` *storage drivers to build and manage on-disk structures of images and containers.*
- Image Layering
  - *OverlayFS takes two directories on a single Linux host, layers one on top of the other, and provides a single unified view.*
  - *Only works for two layers, in multi-layered images hard links are used to reference data shared with lower layers.*

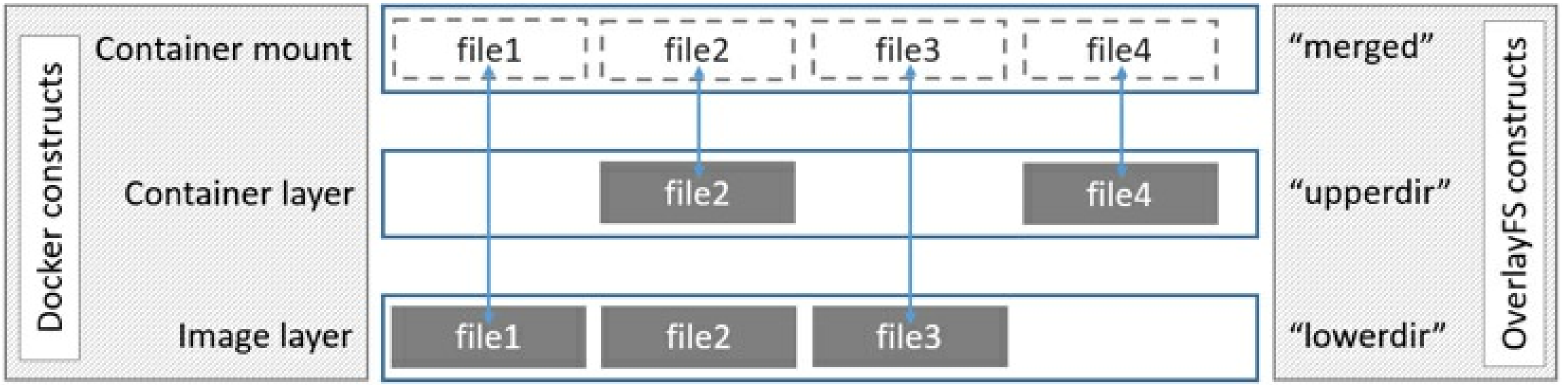


Image Layers Example

- Pulling out the image from the registry

```
$ docker pull ubuntu

Using default tag: latest
latest: Pulling from library/ubuntu

5ba4f30e5bea: Pull complete
9d7d19c9dc56: Pull complete
ac6ad7efd0f9: Pull complete
e7491a747824: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:46fb5d001b88ad904c5c732b086b596b92cfb4a4840a3abd0e35dbb6870585e4
Status: Downloaded newer image for ubuntu:latest
```

  - *Each image layer has its own directory under* `/var/lib/docker/overlay/`*.*
  - *This is where the contents of each image layer are stored.*

- Directories on the file system

```
$ ls -l /var/lib/docker/overlay/

total 20
drwx------ 3 root root 4096 Jun 20 16:11 38f3ed2eac129654acef11c32670b534670c3a06e483fce313d72e3e0a15baa
drwx------ 3 root root 4096 Jun 20 16:11 55f1e14c361b90570df46371b20ce6d480c434981cbda5fd68c6ff61aa0a535
drwx------ 3 root root 4096 Jun 20 16:11 824c8a961a4f5e8fe4f4243dab57c5be798e7fd195f6d88ab06aea92ba93165
drwx------ 3 root root 4096 Jun 20 16:11 ad0fe55125ebf599da124da175174a4b8c1878afe6907bf7c78570341f30846
drwx------ 3 root root 4096 Jun 20 16:11 edab9b5e5bf73f2997524eebeac1de4cf9c8b904fa8ad3ec43b3504196aa380
```

  - *The organization of files allows for efficient use of disk space.*
  - *There are* ***files unique to every layer*** *and* ***hard links to files*** *shared with lower layers*

Dockerfile

- Dockerfile is a script that creates a new image

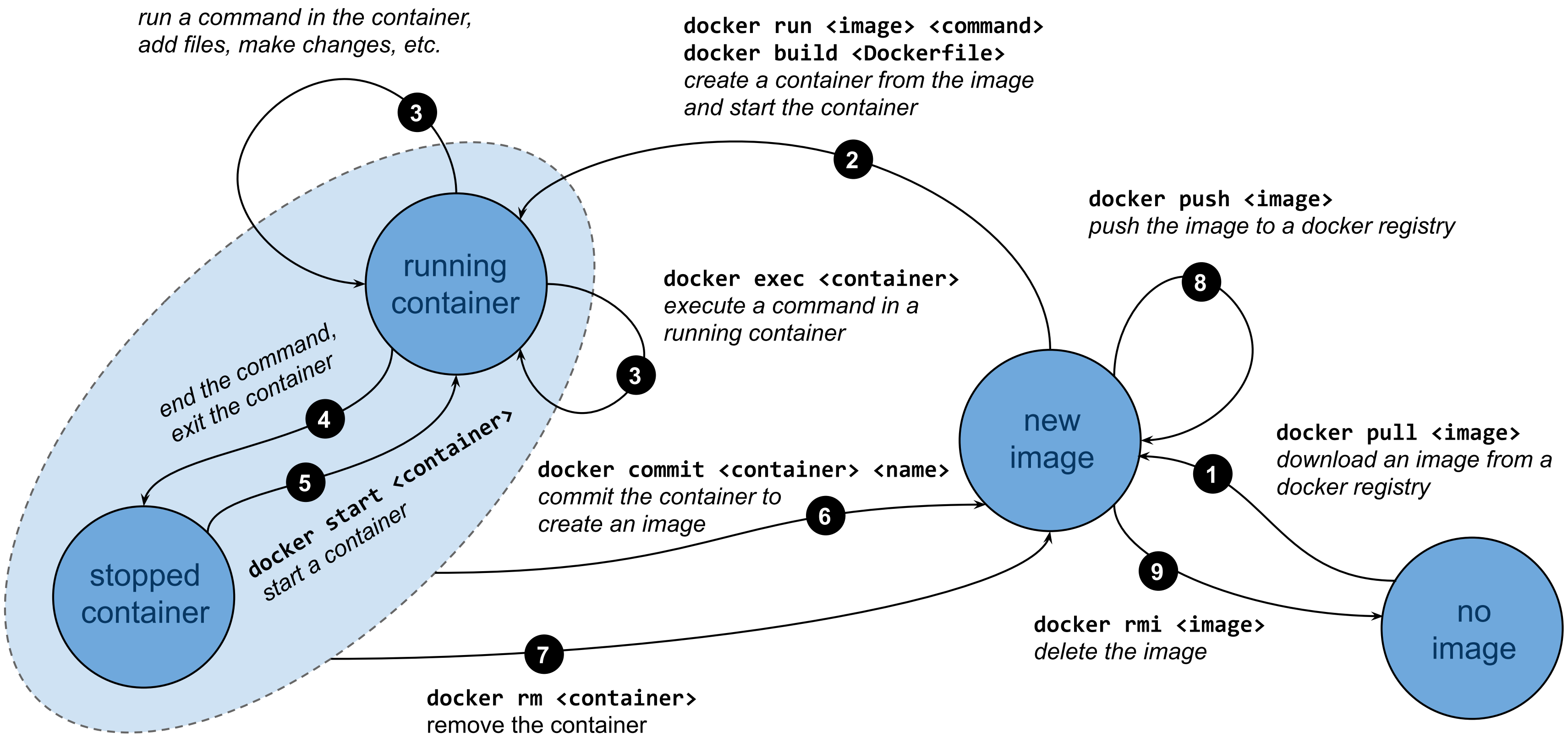
```
# This is a comment
FROM oraclelinux:7
MAINTAINER Tomas Vitvar <tomas@vitvar.com>
RUN yum install -q -y httpd
EXPOSE 80
CMD httpd -X
```

- A line in the Dockerfile will create an intermediary layer

```
$ docker build -t tomvit/httpd:v1 .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM oraclelinux:7
 ---> 4c357c6e421e
Step 2 : MAINTAINER Tomas Vitvar <tomas@vitvar.com>
 ---> Running in 35feebb2ffab
 ---> 95b35d5d793e
Removing intermediate container 35feebb2ffab
Step 3 : RUN yum install -q -y httpd
 ---> Running in 3b9aee3c3ef1
 ---> 888c49141af9
Removing intermediate container 3b9aee3c3ef1
Step 4 : EXPOSE 80
 ---> Running in 03e1ef9bf875
 ---> c28545e3580c
Removing intermediate container 03e1ef9bf875
Step 5 : CMD httpd -X
 ---> Running in 3c1c0273a1ef
```

If processing fails at some step, all preceeding steps will be loaded from the cache on the next run.

Docker Container State Diagram



1: There is no image in the local store; you pull an image a remote registry.
2: You run a new container on top a specified image.
3: You modify the container by adding a library/content in it; you can also run a command in the container from the host.
4: You stop a running container.
5: You start a stopped container.
6: You commit the container and create a new image from it.
7: You remove the container.
8: You push the image to the remote registry.
9: You can remove the image from the local store.