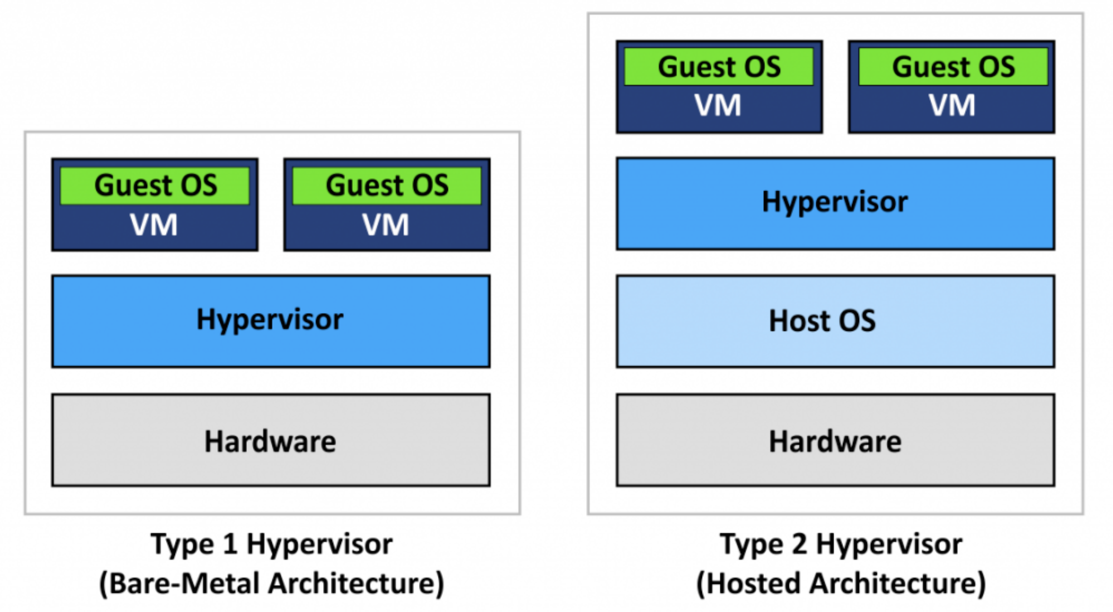
Containers 101

IBM **Developer**

Table of Content

What is a Virtual Machine (VM)?

VMs were developed in 1974 by Popek and Goldberg as a set of conditions to support system virtualization. System virtual machines virtualize the hardware resources, including processor, memory, storage and peripheral devices. The Virtual Machine Monitor (VMM or Hypervisor) abstracts the virtual machine. You can run a native Hypervisor directly on the hardware or a hosted Hypervisor on a host Operating System (OS).



What is a Container?

A container creates "VM-like” isolation but managed at the **process level.**

Three Linux kernel components are used to isolate applications running in containers and limit access to resources on the host:

- Linux namespaces
- Control groups (cgroups)
- SELinux contexts

achieved by a set of "**namespaces**" (isolated view):

- PID –isolated view of process IDs
- USER- user and group IDs
- UTS - hostname and domain name
- NS - mount points
- NET - Network devices, stacks, ports
- IPC - inter-process communications, message queues

And **cgroups** that control the limits and monitoring of resources. The `ns` subsystem in cgroups integrates namespaces and control groups.

The key statement: **A container is a process(es) running in isolation**

VM vs Container

Before containers in an OS environment:

- Entanglement between OS and App
- Entanglement between runtime environment and OS
- Version dependency and conflicts between Apps
- Breaking updates
- Updates require full app stop and downtime
- Deployment and Maintenance of HA system is complex
- Slow startup time

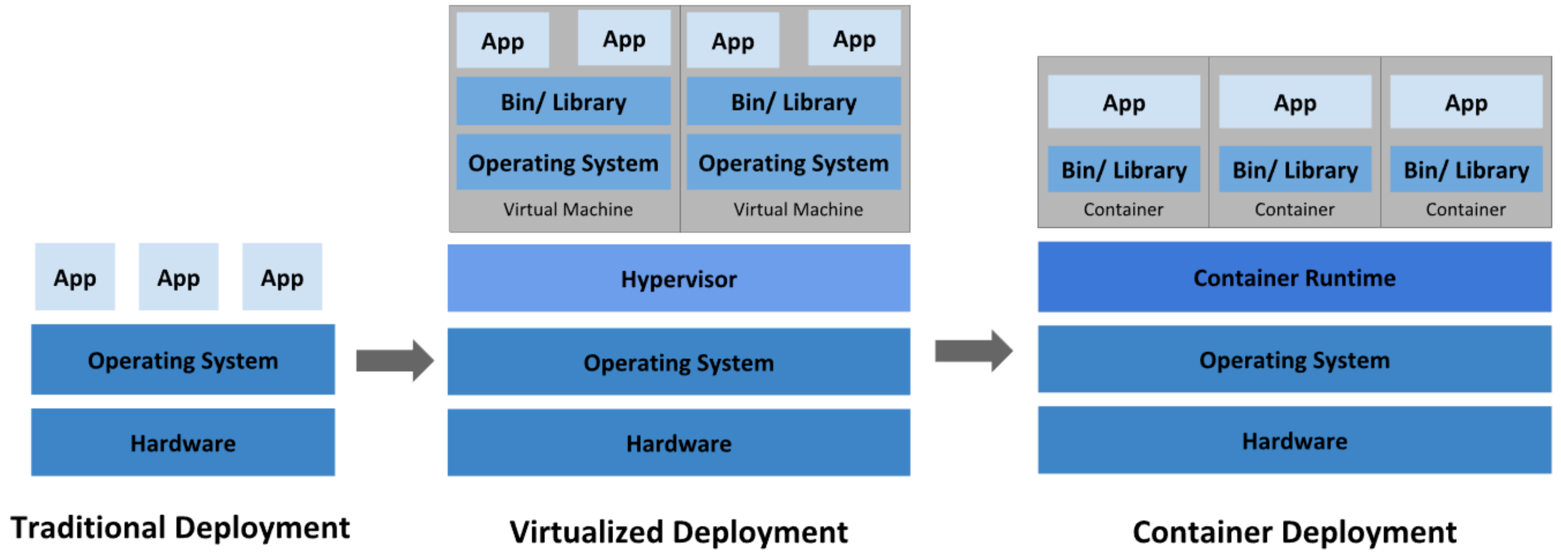


image source: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

VMs, Containers and Container Orchestration

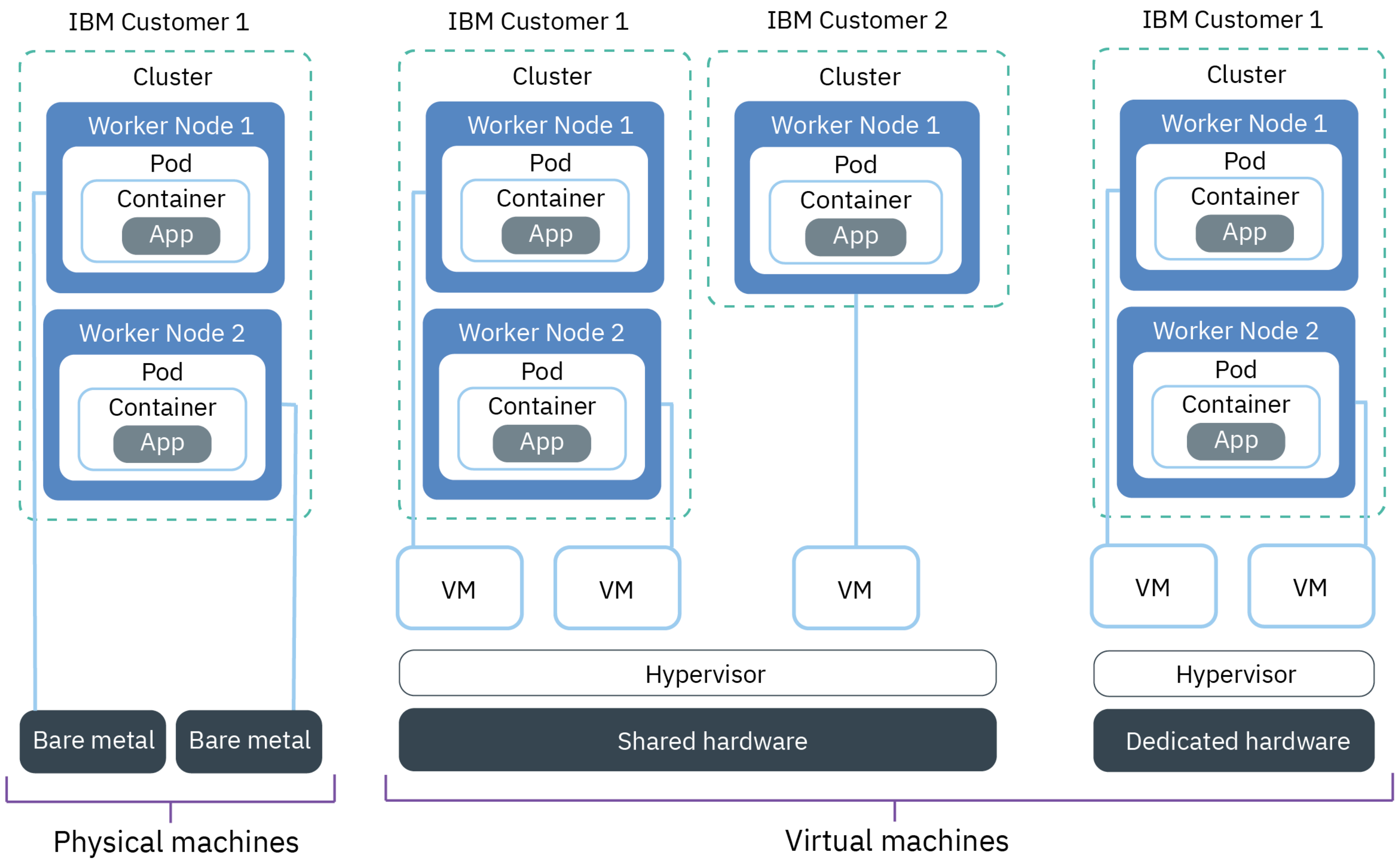
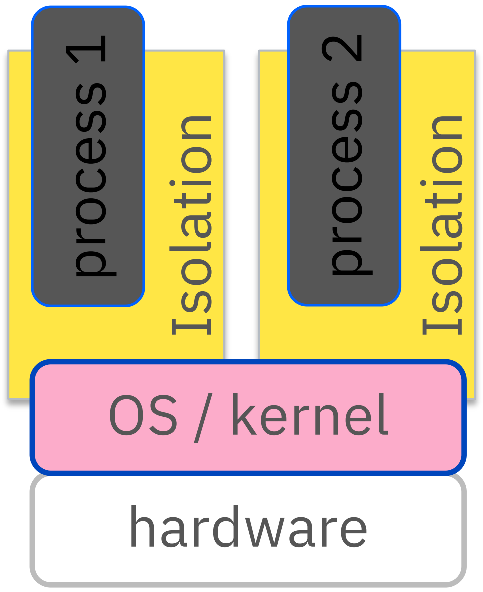


image source: https://cloud.ibm.com/docs/containers?topic=containers-planning_worker_nodes

A Short History of Containers

Containers – not a new idea

- (1979) `chroot` command, changes apparent root for running process to isolate system processes
- (1982) `chroot` added to Unix v7
- (1990s) `jail` command created
- (2000) `jail` added to FreeBSD to isolate filesystems, users, networks etc.
- (2001) Linux VServer is a jail mechanism that partitions resources (file systems, network addresses, memory)
- (2004) Solaris Containers using Solaris Zones, application has a full user, processes, and file system, and access to the system hardware, but can only see within its own zone.
- (2005) Open VZ (Virtuzzo), OS-level virtualization for Linux using a patched Linux kernel for virtualization, isolation, resource management and checkpointing
- (2006) Google launches `Process Containers` for limiting resource usage, renamed `cgroups` in 2007
- (2008) `cgroups` merged into Linux kernel 2.6.24, becoming LinuX Containers (LXC)
- (2009) Cloud Foundry developed by VMWare called Project B29 (Warden, Garden)
- (2011) Cloud Foundry started Warden
- (2013) Docker released as open source
- (2014) LXC replaced by `libcontainer`, Google contributes container stack `LMCTFY` (Let Me Contain That For You) to Docker project as `libcontainer` in 2015



Containers vs Docker

Containers is the technology, Docker is the **tooling** around containers

Without Docker, containers would be **hard to use** (for most people)

Docker **simplified** container technology

Added value: Lifecycle support, setup file system, etc

For extra confusion: **Docker Inc.** is a company, which is different than Docker the technology…

Docker’s ecosystem approach transformed the perception of containers,
- Building application-centric containers
- Mechanism for sharing images (Docker Registry)
- Open-source enabled

What is a Container Engine?

The container runtime packages, instantiates, and runs containers. There are many container engines and many cloud providers have their own container engines to consume OCI compliant Container images. Having a standard Container Image Format allows interoperability between all these platforms.

Typically, the container engine is responsible:

- Handling user input
- Handling input over an API often from a Container Orchestrator
- Pulling the Container Images from the Registry Server
- Expanding decompressing and expanding the container image on disk
- Preparing a container mount point
- Preparing metadata for container runtime to start container correctly (defaults from container image ex. Arch86, user input e.g. CMD or ENTRYPOINT)
- Calling the Container Runtime.

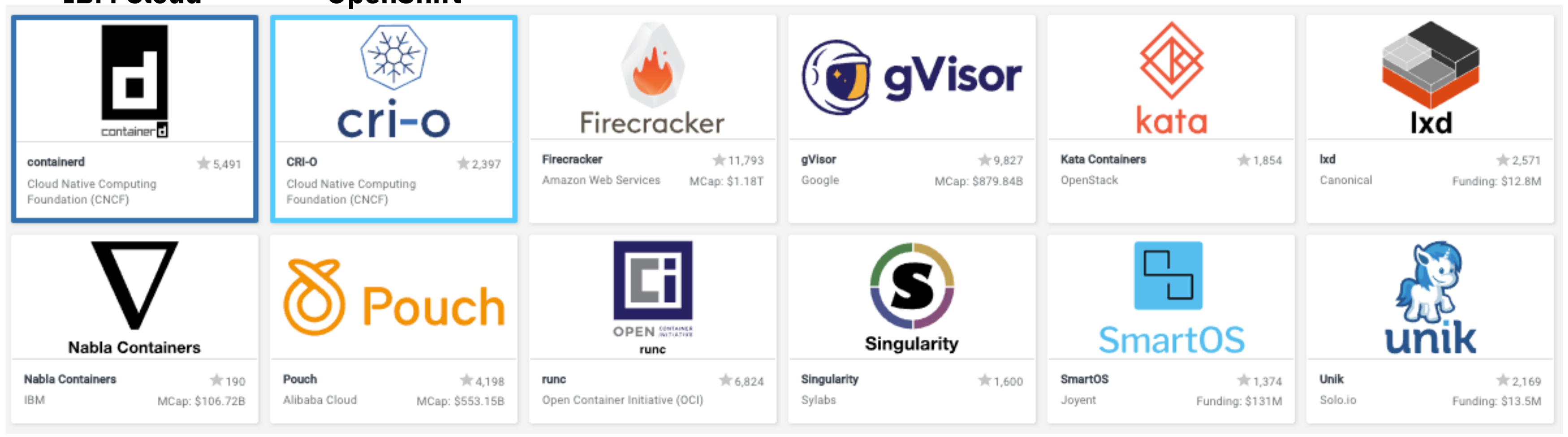
The Container Runtime is the lower-level component used in Container Engine. The OCI Runtime Standard reference implementation is `runc`. When Docker was first created it relied on LXC as the container runtime, later replaced by `libcontainer` to start containers. When OCI was created, Docker donated `libcontainer` to OCI and it became `runc`. At the lowest level this provides a consistent way to start containers, no matter the container engine.

Container Runtime

In December 2014, CoreOS (now RedHat) released “rkt” as an alternative to Docker and initiated *app container* (appc) and *application container image* (ACI) as independent committee-steered specifications, which developed into the Open Container Initiative (OCI). As the number of runtime providers increases and varies between providers, for End of Vendor Support (EOVS) or End of Life (EOL) reasons or with increase of hybrid cloud architectures, interoperability and interexchangeability is critical.

On June 22, 2015 the Open Container Initiative (OCI) was announced, formed under the Linux Foundation and launched by Docker, CoreOS and others. The OCI currently contains a Runtime Specification ([runtime-spec]) and an Image Specification ([image-spec]).

IBM Cloud **OpenShift**



containerd	★5,491	Cloud Native Computing Foundation (CNCF)	
CRI-O	★2,397	Cloud Native Computing Foundation (CNCF)	
Firecracker	★11,793	Amazon Web Services	MCap: $1.18T
gVisor	★9,827	Google	MCap: $879.84B
Kata Containers	★1,854	OpenStack	
lxd	★2,571	Canonical	Funding: $12.8M
Nabla Containers	★190	IBM	MCap: $106.72B
Pouch	★4,198	Alibaba Cloud	MCap: $553.15B
runc	★6,824	Open Container Initiative (OCI)	
Singularity	★1,600	Sylabs	
SmartOS	★1,374	Joyent	Funding: $131M
Unik	★2,169	Solo.io	Funding: $13.5M

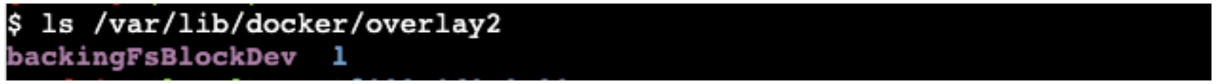
Our First Container

$ docker run ubuntu echo Hello World

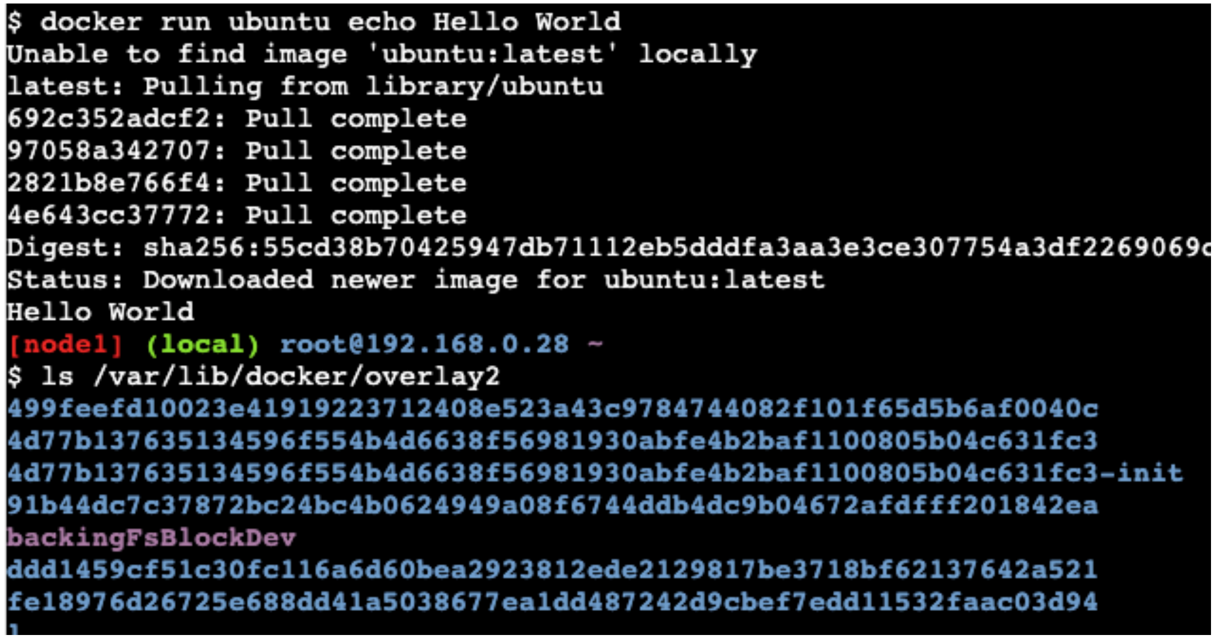
Hello World

What happened?

- Docker checked local repository for an image named \`ubuntu\`
- Docker downloaded an image \`ubuntu\` from https://registry.hub.docker.com/_/ubuntu
- Docker created a directory with an "ubuntu" filesystem (image)
- Docker created a new set of namespaces (lsns)
- Ran a new process: echo Hello World
  - Using the namespaces to isolate it from other processes
  - Using the new directory as the "root" of the filesystem (chroot)
- That's it!
  - Notice as a user I never installed "ubuntu"
- Run it again - notice how quickly it ran



```
$ ls /var/lib/docker/overlay2
backingFsBlockDev  l
```



```
$ docker run ubuntu echo Hello World
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
692c352adcf2: Pull complete
97058a342707: Pull complete
2821b8e766f4: Pull complete
4e643cc37772: Pull complete
Digest: sha256:55cd38b70425947db71112eb5dddfa3aa3e3ce307754a3df2269069
Status: Downloaded newer image for ubuntu:latest
Hello World
[node1] (local) root@192.168.0.28 ~
$ ls /var/lib/docker/overlay2
499feefd10023e41919223712408e523a43c9784744082f101f65d5b6af0040c
4d77b137635134596f554b4d6638f56981930abfe4b2baf1100805b04c631fc3
4d77b137635134596f554b4d6638f56981930abfe4b2baf1100805b04c631fc3-init
91b44dc7c37872bc24bc4b0624949a08f6744ddb4dc9b04672afdfff201842ea
backingFsBlockDev
ddd1459cf51c30fc116a6d60bea2923812ede2129817be3718bf62137642a521
fe18976d26725e688dd41a5038677ea1dd487242d9cbef7edd11532faac03d94
```

https://github.com/tianon/docker-brew-ubuntu-core/blob/5510699c08fd4c68c7fd05783ceedf15751b6cd9/bionic/Dockerfile

```
FROM scratch
ADD ubuntu-bionic-core-cloudimg-amd64-root.tar.gz /
RUN [ -z "$(apt-get indextargets)" ]
RUN set -xe \
  && ...
RUN mkdir -p /run/systemd && echo 'docker' > /run/systemd/container
CMD ["/bin/bash"]
```

"ssh-ing" into a container

```
$ docker run -ti ubuntu bash
root@62deec4411da:/# pwd
/
root@62deec4411da:/# exit
$
```

- Now the process is "bash" instead of "echo"
- But its still just a process
- Look around, mess around, its totally isolated
  - rm /etc/passwd – no worries!
  - MAKE SURE YOU'RE IN A CONTAINER!

A look under the covers

```
$ docker run ubuntu ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 14:33 ?        00:00:00 ps -ef
```

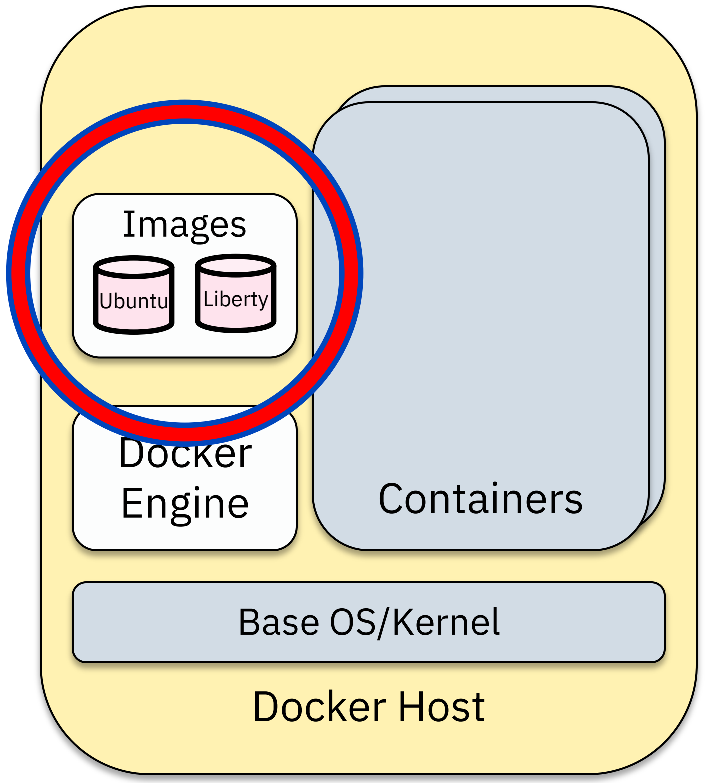
Things to notice with these examples:

- Each container only sees its own process(es)
- Each container only sees its own filesystem
- Running as "root"
- Running as PID 1

Docker Images

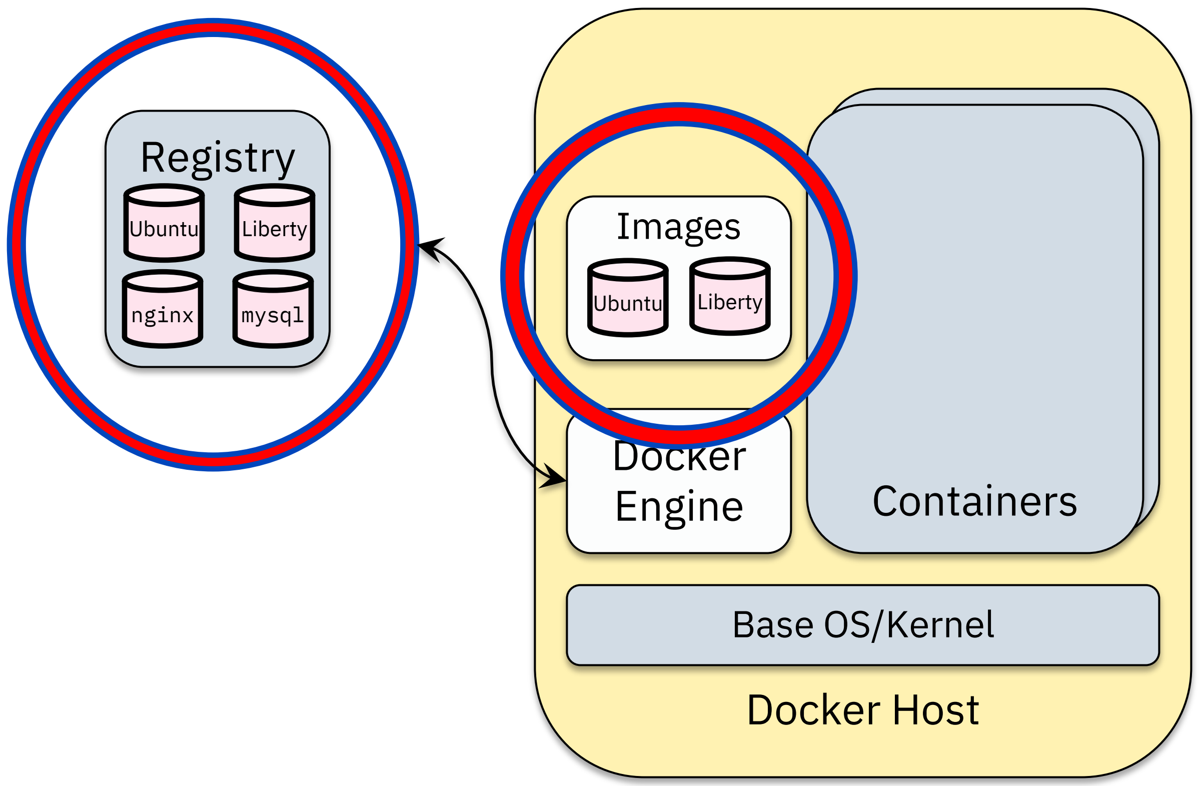
- Tar file containing a container's filesystem + metadata

- For sharing and redistribution
  - Global/public registry for sharing: DockerHub



Docker Registry

- DockerHub (https://hub.docker.com)
- Public registry of Docker Images
- The central place for sharing images with friends or coworkers!
- Also useful to find prebuilt images for web servers, databases, etc
- Enterprises will want to find a private registry to use (such as Artifactory)



Build your own image with a Dockerfile!

Step 1) Create Dockerfile to script how you want the image to be built

```
FROM java:8 # This might be an ubuntu or...
COPY *.jar app.jar
CMD java -jar app.jar
```

Step 2) **docker build** to build an image

Step 3) **docker push** to push to registry

Step 4) From another location, **docker pull** to download an image

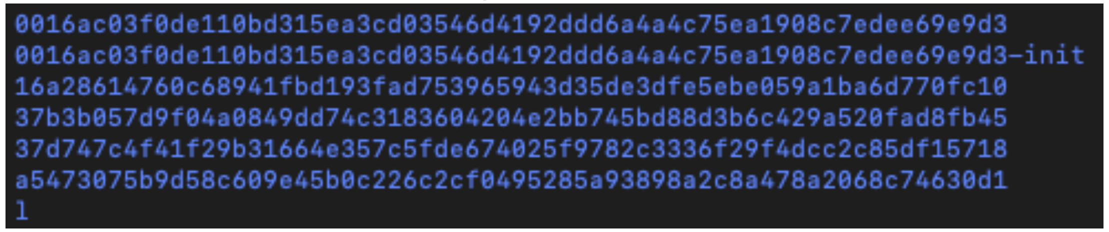
Shared/Layered/Union Filesystems

- Docker re-uses common layers between containers and images
- A single writeable layer is added on the top every time a new layer is created
- Layers are “smushed” with **union file system** (think transparencies on a projector)
- Files are copied up when writes need to be made (copy-on-write)

Bottom Line

- More containers per host
- Faster downloads and uploads
- Faster container startups

ls /var/lib/docker/overlay2



```
0016ac03f0de110bd315ea3cd03546d4192ddd6a4a4c75ea1908c7edee69e9d3
0016ac03f0de110bd315ea3cd03546d4192ddd6a4a4c75ea1908c7edee69e9d3-init
16a28614760c68941fbd193fad753965943d35de3dfe5ebe059a1ba6d770fc10
37b3b057d9f04a0849dd74c3183604204e2bb745bd88d3b6c429a520fad8fb45
37d747c4f41f29b31664e357c5fde674025f9782c3336f29f4dcc2c85df15718
a5473075b9d58c609e45b0c226c2cf0495285a93898a2c8a478a2068c74630d1
l
```

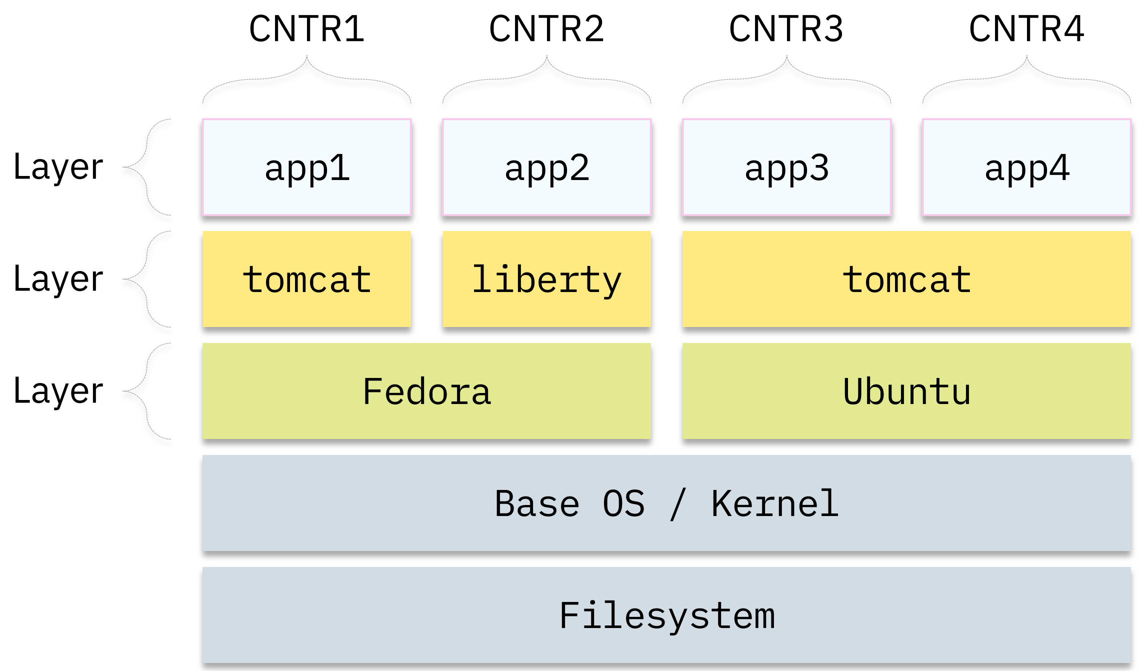
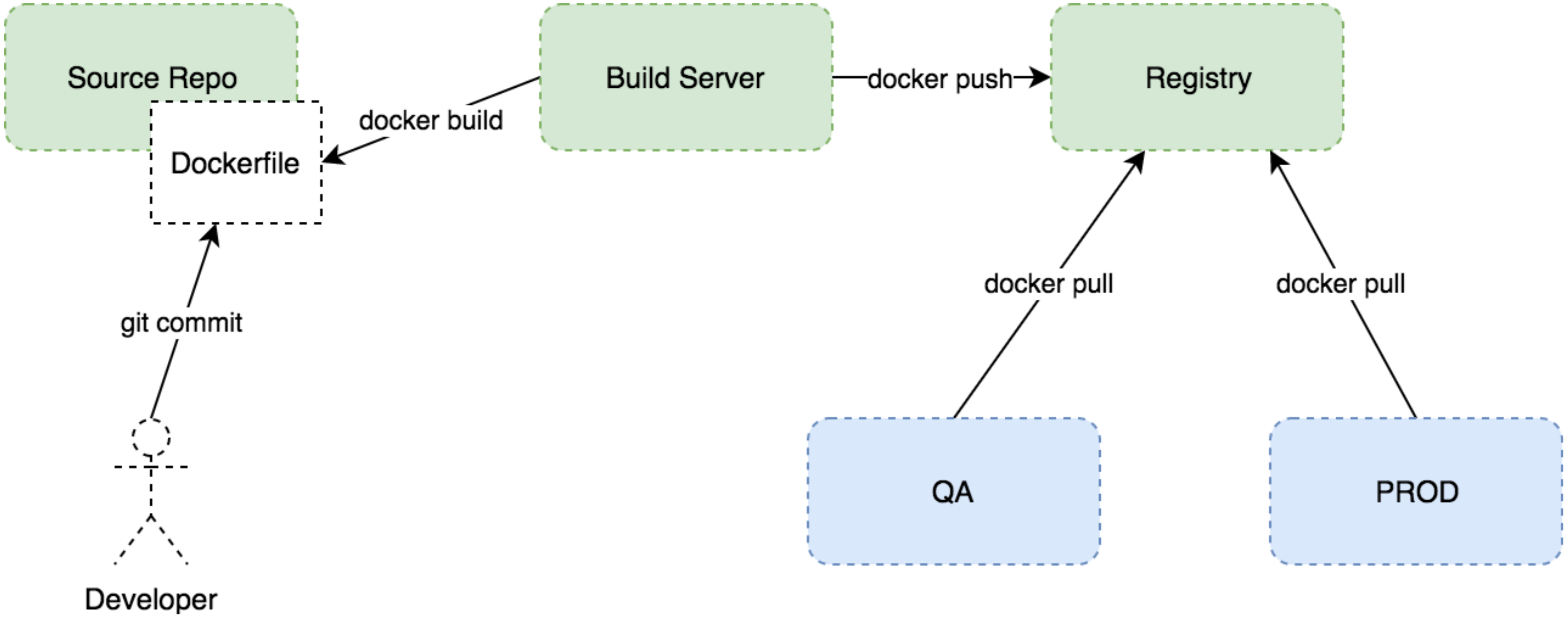


Image Optimization

- Use optimized Base Images, e.g. UBI
- Use Multi-Stage Builds, allows passing on artefacts
- Reduce number of layers
- Build custom base images
- Use production base images
- Avoid storing application data
- Design stateless containers without dependencies env and config
- Use .dockerignore when recursively using directories
- Add only required files
- Install required packages only
- Limit number of layers (RUN, COPY, ADD)
- Sort multi-line arguments
- Avoid using :latest
- etcetera

Container Security

Container Life-Cycle



Summary

Why? When compared to VMs:

- Low hardware footprint, better resource utilization (CPU, Memory, Disk – resources managed using namespaces and cgroups),
- Faster start-up times, no OS install or restart
- Quick deployment
- Better efficiency, elasticity, and reusability of the hosted applications
- Better portability of platform and containers
- Environment isolation, changes to host OS do not affect the container
- Multiple environment deployment
- Reusability
- Easier tooling/scripting

Docker value-add:

- User Experience
- Image layers
- Easily share images - DockerHub

Lab Time