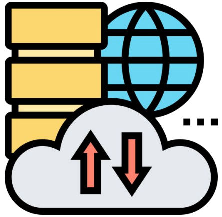
COEN 241
Introduction to Cloud Computing

Lecture 6 - Containers

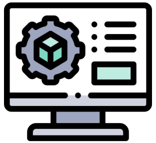


Lecture 5 Recap

- OS Virtualization
  - **Namespaces**
  - **Cgroups**
  - History of OS Virtualization

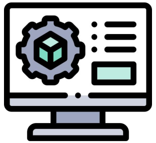
Motivations for Resource Isolation

- So far, the main motivation was **security**
  - VMs should not be able to see each others' data
- New motivation: **Software Manageability**
  - Applications that need specific, conflicting support software versions
    - Runtime environments may allow local installation, e.g., Python 'virtualenv's
  - Want to be able to cleanly install and remove sets of software
    - Linux distribution package managers can provide this support
  - Support **testing** of various versions of softwares



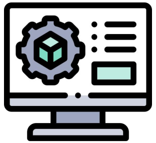
Two Features for Userspace Isolation

- Namespace
  - Allowing different view of the system for different process.
- Cgroup
  - Controls and isolates the resource usage (CPU, memory, disk I/O, network, etc.) for different processes.
- Both are LINUX Kernel features



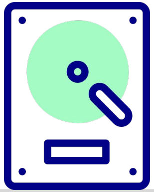
Kinds of Namespaces

- **mnt: Mount points, filesystems**
- **net: Virtualizing the network stack**
- pid: processes
- ipc: Interprocess communications
- uts: Unix Time Sharing for different hostname
- user: User identification
- cgroup: Identity of the control group (will cover this in later slides)
- time: Allow processes to see different times



“Old School” Chroot Jail

- Unix servers have to handle users that may be malign
  - Common historical example was running public FTP servers
    - Anonymous users could log into those servers
    - FTP as a protocol allows quite a lot of power over the server
    - Needed to cut down what anonymous users could do
- Solution: change the perceived root directory of the filesystem
  - i.e., a ‘chroot jail’: changes the set of available executables
  - The process and its children can’t access the files outside the new root.
  - Unix accesses binaries from /bin, libraries from /lib, etc.
  - Changing the meaning of / mitigates many vulnerabilities



Linux Cgroup

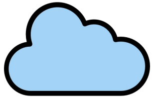
- This work was started by engineers at Google in 2006 under the name "process containers; in 2007, renamed to “Control Groups”.

- Defines parameters about the resource use of a set of processes, e.g.:
  - Limit total memory available to group of processes
  - Indicate non-even share of device input/output priority
  - Affect CPU scheduling to the group
  - cgroups also can assist accounting for resource use
  - cgroups can be applied hierarchically

- cgroups can facilitate starting / stopping processes
  - Important for snapshot functionality

Cgroup Subsystems

- Kernel modules that are used to control the access that cgroups have to various system resources
- Example 10 cgroup subsystems for Redhat
  - `blkio` — sets limits on input/output access to and from block devices such as physical drives
  - `cpu` — uses the scheduler to provide cgroup tasks access to the CPU.
  - `cpuacct` — generates automatic reports on CPU resources used by tasks in a cgroup.
  - `cpuset` — assigns individual CPUs (on a multicore system) and memory nodes to tasks in a cgroup.
  - `devices` — allows or denies access to devices by tasks in a cgroup.
  - `freezer` — suspends or resumes tasks in a cgroup.
  - `memory` — sets limits on memory use by tasks in a cgroup and generates reports on memory usage.
  - `net_cls` — tags network packets with a class identifier (classid) that allows the Linux traffic controller (`tc`) to identify packets originating from a particular cgroup task.
  - `net_prio` — provides a way to dynamically set the priority of network traffic per network interface.
  - `ns` — the *namespace* subsystem.
  - `perf_event` — identifies cgroup membership of tasks and can be used for performance analysis



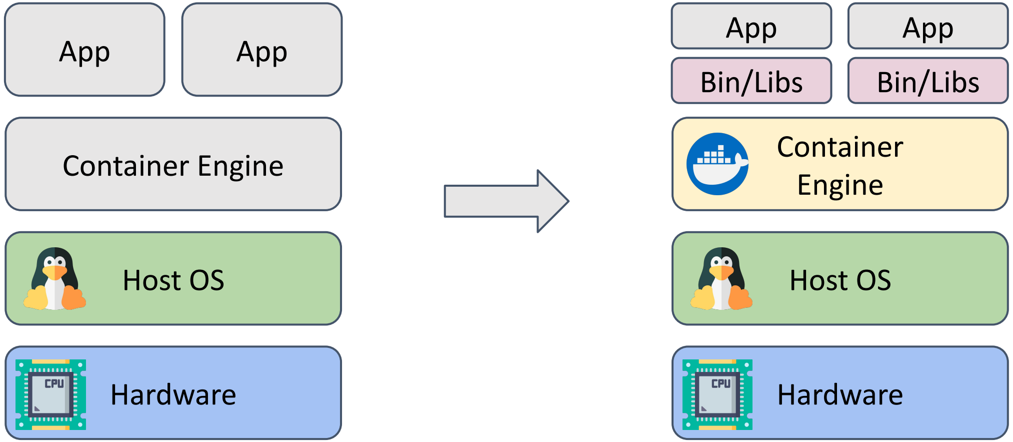
Agenda for Today

- Containers
  - What is it?
  - Pros and Cons
- Docker
  - Docker architecture
  - Docker demo
- Readings
  - Recommended: Demystifying Containers Part II, III
  - Optional
    - https://blog.ippon.tech/docker-engine-1-11-understanding-runc/
    - https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/

Container

What is a Container?

- A container is a standard unit of software that packages up **code** and **all its dependencies** so the application runs quickly and reliably from one computing environment to another



More about Containers

- Minimal requirements to run Containers
  - Container Image
  - Container Engine
  - Container Runtime (runc)
  - And a machine with a host OS to run on

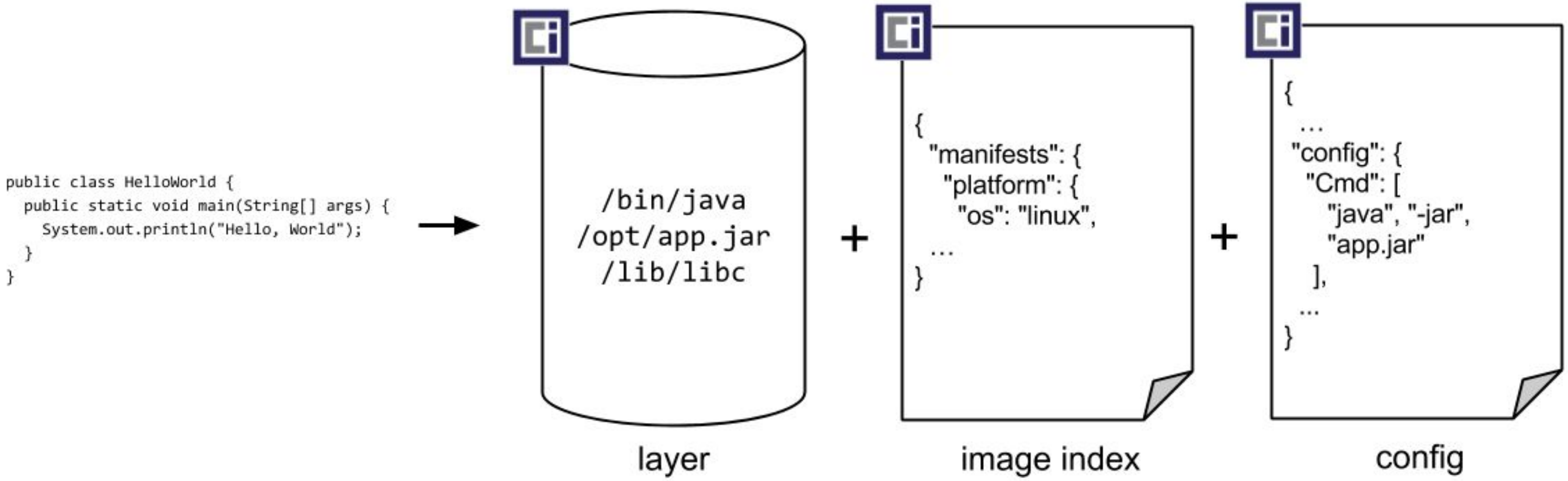
- Containers have standards
  - The Open Container Initiative (OCI) is a lightweight, open governance structure formed under the auspices of the Linux Foundation, for the express purpose of creating open industry standards around container formats and runtime.
  - OCI Image Specification covers how a container image is structured
  - All container image building tools are producing OCI images (e.g., Dockers)
  - With OCI images we can use the same image with different container engines

OCI Image Specification

- An OCI Image consists of:
  - Manifest: A document describing the components that make up a container image
  - Image layout: A filesystem layout representing the contents of an image
  - Image index (optional): An annotated index of image manifests
  - Set of filesystem layers: A changeset that describes a container's filesystem
  - Image configuration: A document determining layer ordering and configuration of the image suitable for translation into a runtime bundle

- https://github.com/opencontainers/image-spec/blob/main/spec.md

OCI Image Specification



The 5 Principles of Standard Containers

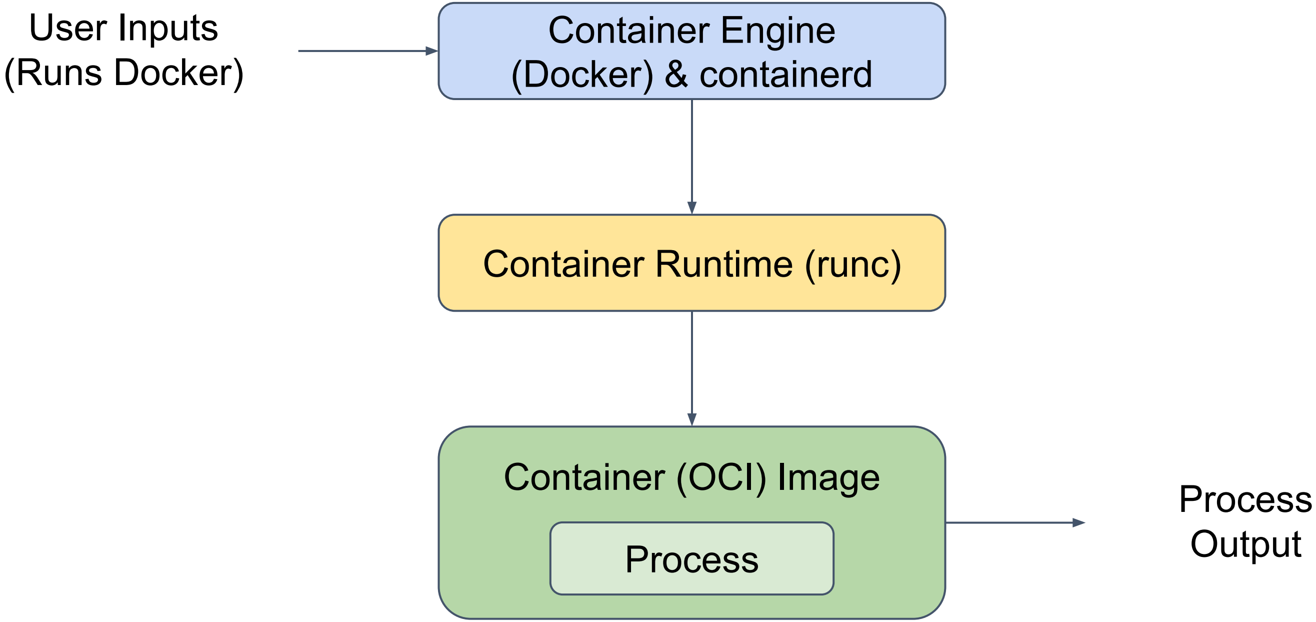
- A unit of software delivery is called a Standard Container via OCI

- A Standard Container must conform to the following 5 principles:
  1. Define a set of **Standard Operations**
  2. **Content-Agnostic:** All standard operations have the same effect regardless of the contents
  3. **Infrastructure-Agnostic:** Can be run in any OCI supported infrastructure
  4. **Designed for Automation**
  5. **Industrial-Grade (Software) Delivery**

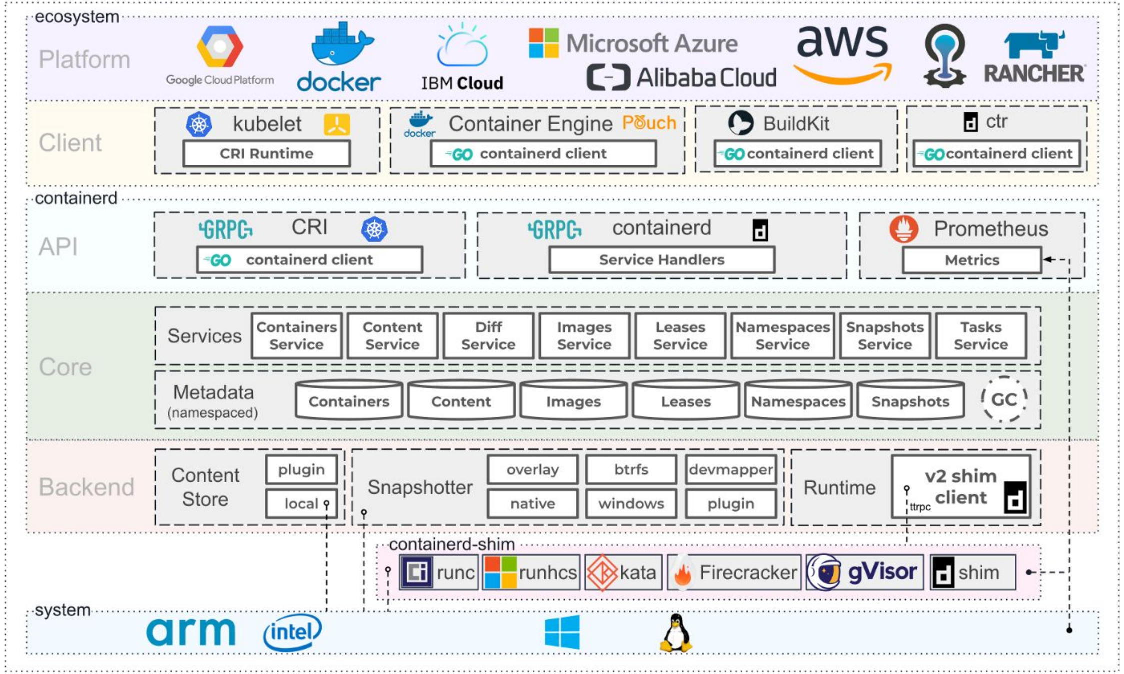
OCI Runtime Specification

- Specifies the state and lifecycle of a container
- When building a runtime, must conform to the specification
- See the following link for details
    - https://github.com/opencontainers/runtime-spec/blob/main/spec.md

Container Workflow



Container Ecosystem



State-of-the-art Container Solutions

- **Docker**: https://www.docker.com/
- **Kata container:** https://katacontainers.io/supporters/
- gVisor (google): https://github.com/google/gvisor
- Podman: https://podman.io/
- Singularity: https://sylabs.io/singularity/
- etc...

- Docker and gvisor container are built upon Namespaces and Cgroups
- Kata runs containers within VMs
- We will focus mostly on Docker & Kata containers

Docker

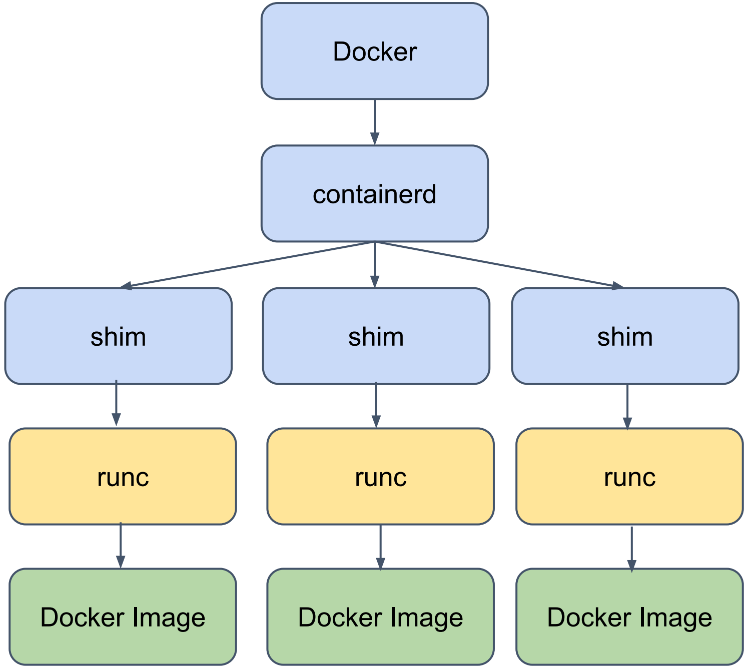
What is Docker?

- **Docker** is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers
- Founded in 2009, Released in 2013
- It is a software platform consisting of:
  - Docker Engine
  - Docker Hub
  - Docker Trusted Registry
  - Docker Machine
  - Docker Compose
  - Docker for Windows/Mac
  - Docker Datacenter

http://wiki.zenoss.org/download/core/drich_slides/DockerSlides.pdf

Docker Architecture

- **containerd**: a daemon process that manages and invokes container runtimes
- **“shim”**: facilitate communication and integration
  - Enables daemon-less container
- **runc**: low-level container runtime
  - actually creates and runs containers

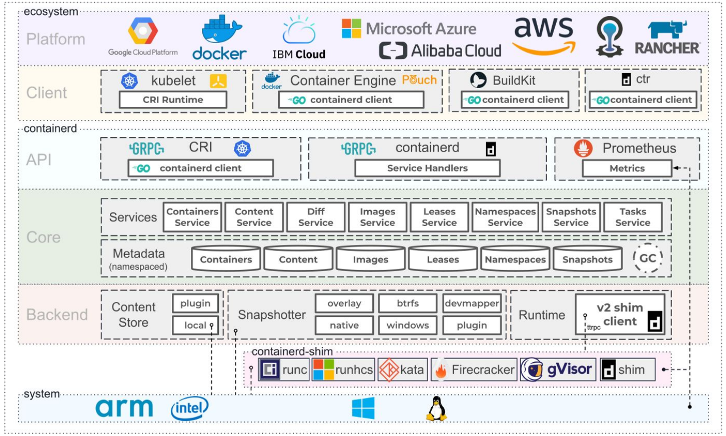


containerd

- A **container runtime** from Docker
  - P.s.: Runtime = environment and data structures that keep track of everything that's going along as your program runs

- Tasks
  - Pulls container images from registries & manages them
  - Hands them over to a lower-level runtime

- Alternatives
  - CRI-O



https://containerd.io/

runc: Tool that Really Runs your Containers

- **Lower-level runtime**
- Takes two inputs to start a container
  - a JSON configuration file
  - a (OCI) root filesystem bundle
  - You can try to run containers only with runc if interested!
    - https://mkdev.me/en/posts/the-tool-that-really-runs-your-containers-deep-dive-into-runc-and-oci-specifications
- Alternatives
  - **crun** a container runtime written in C (by contrast, runc is written in Go)
  - **kata-runtime** from the Katacontainers project, which implements the OCI specification as individual lightweight VMs (hardware virtualization)
  - **gVisor** from Google, which creates containers that have their own kernel
    - Implements OCI in its runtime called runsc

https://github.com/opencontainers/runc

Container Images

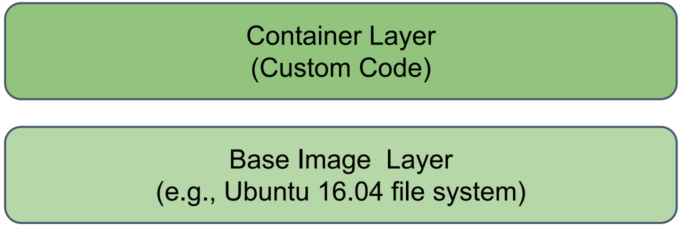
- A container is launched from a container image (a VM image)

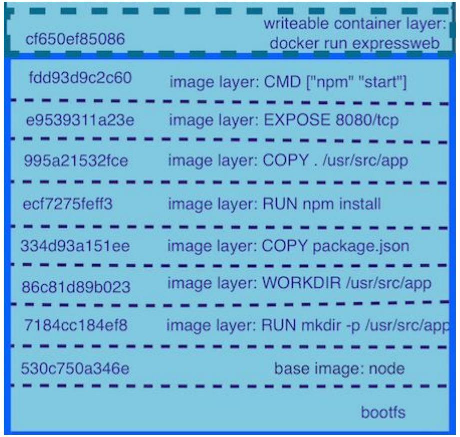
- A container image is a root file system that includes everything needed to run an application(s)
    - Contains the application code, a runtime, libraries, environment variables, and configuration files
    - Consists of folders and files just like a file system

- When we launch a container, a container instance is a runtime instance of an image
    - it's like binary code vs. processes

Docker Container Image

- A container originates from a base image layer, including a base file system (and applications)
- When you launch a container, another layer is created on top of the base image layer
- You can stack more container layers!



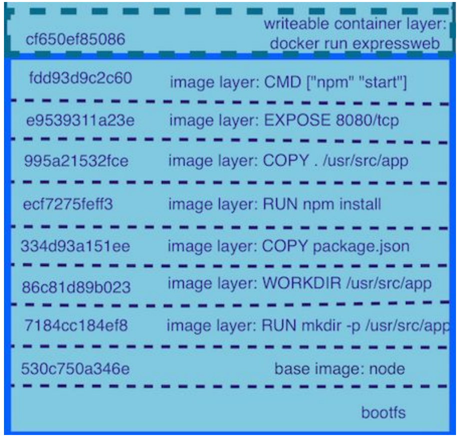


Read/Write Permissions

- Only the top level (container) layer has both read/write permission
- All base layers are read only
- **Merged view via file systems like AUFS**

Container Layer
(Custom Code)

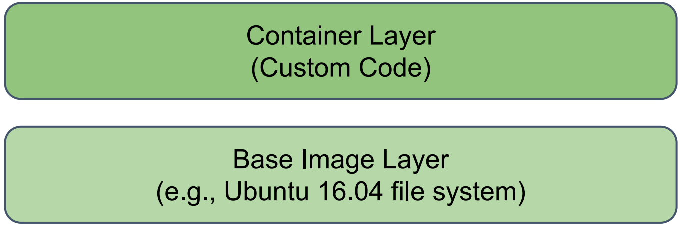
Base Image Layer
(e.g., Ubuntu 16.04 file system)



https://docs.docker.com/storage/storagedriver/aufs-driver/#how-the-aufs-storage-driver-works

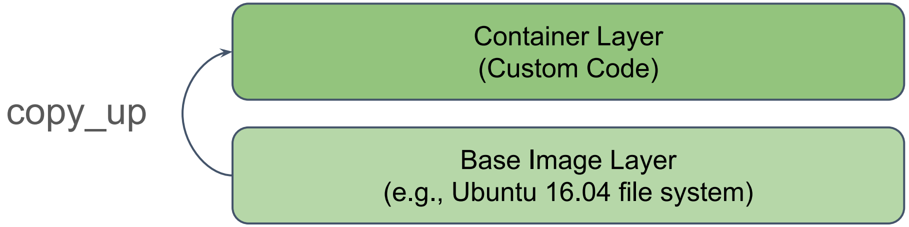
Read Policies of AUFS

- If the file only exist in base image layer, it is read from base image layer
- If the file only exist in container layer, it is read from container layer
- If the file exist in both layers, it is read from container
    - Files in the container layer obscure files with the same name in the image layer



Write Policies of AUFS

- Writing to a file for the first time (the file exists in the base image layer)
  - copy_up: copy files from the base image layer to the container layer, and write changes to it. Even copies the large entire file.
  - No changes to the files in the base layer.
- Deleting creates a whiteout file in the container layer
  - Prevents the to be available, since the file is still in the base image layer
- No rename

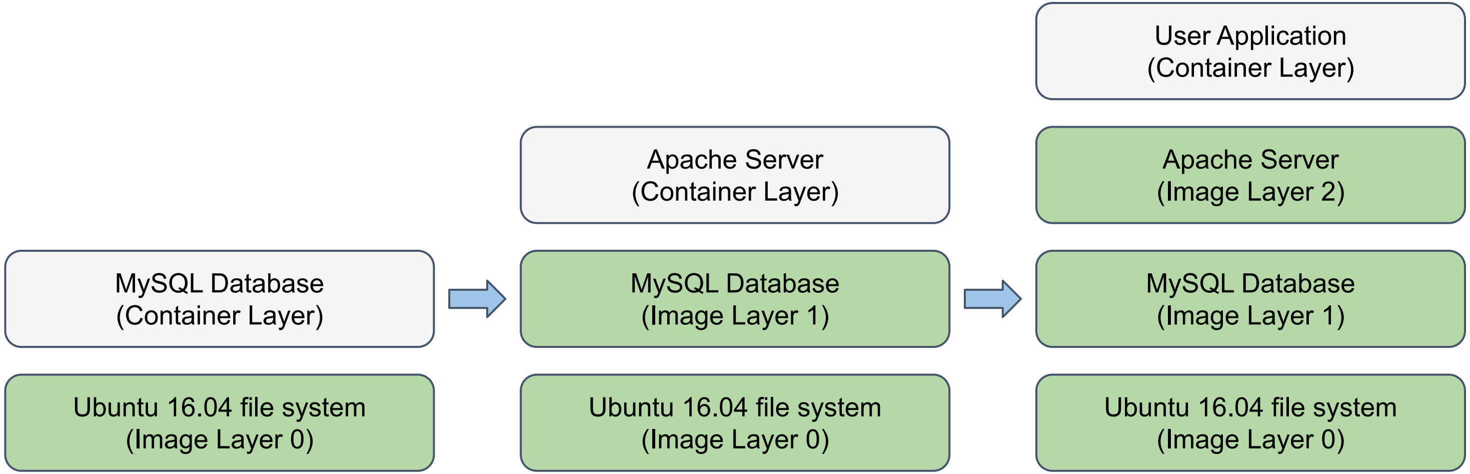


https://docs.docker.com/storage/storagedriver/aufs-driver/

Why use File Systems like AUFS?

- Many container instances share the same base images
- Saves space in the host
- Easy to build new images
- Downsides:
  - Overhead
  - Complex

Stackable Container Images



Docker Demo

Demo Overview

- How to install Docker
  - https://docs.docker.com/engine/install/ubuntu/
- How to start using a container
  - Start
  - Stop
  - Delete
- How to create a new container to upload
- How this relates to HW 1?

Docker Installation (Ubuntu)

```
# Remove old versions and add docker repo
$ sudo apt-get remove docker docker-engine docker.io containerd runc
$ sudo apt-get update
$ sudo apt-get install ca-certificates curl gnupg lsb-release
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg
--dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
$ echo "deb [arch=$(dpkg --print-architecture)
signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu  $(lsb_release -cs) stable" | sudo
tee /etc/apt/sources.list.d/docker.list > /dev/null
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Check Docker Installation

```
# Check if Docker is installed correctly
$ docker ps
$ docker run hello-world # Shows some output
$ docker images
$ docker rmi <image_id> # Delete new image
```

Running a Larger Image

```
# Running the docker shell, /bin/sh

$ docker images

$ docker run --rm -it --entrypoint /bin/sh ubuntu:20.04

# On a new window

$ docker ps
```

Creating a New Docker Image

```
# Create a new image from the base image
$ docker images
$ sudo docker run -it ubuntu:20.04
$ apt update; apt install iputils-ping;
# On another window
$ docker ps
$ docker commit <container_id> my_image_with_ping
$ docker images
$ docker history my_image_with_ping
```

Agenda for Today

- Containers
  - What is it?
  - Pros and Cons
- Docker
  - Docker architecture
  - Docker Demo
- Readings
  - Recommended: Demystifying Containers Part II, III
  - Optional
    - https://blog.ippon.tech/docker-engine-1-11-understanding-runc/
    - https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/

TODOs!

- HW 1!

Questions?