DSC516: Cloud Computing

Part II: Cloud Building Blocks

Module 4: Virtualization, Containers and Resource Management

University of Cyprus
Department of Computer Science

Topic 7

Containers and Docker

University of Cyprus
Department of Computer Science

Lecture 18b

Containers and Docker

Learning Objectives

- Examine, understand and explain the concept of containers and the main techniques.
- Understand and explain how Docker containers manage resource controls.
- Understand and explain how is software installed on Docker containers
- Understand and explain how Docker containers manage storage.
- Understand and explain the key differences and comparison between containers and VMs.

Readings

J. Nickoloff and S. Kuenzli (2019), **"Docker in Action" 2nd Edition, Manning.**

Additional Readings

- P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, **“Containers and virtual machines at scale: A comparative study,”** in Proceedings of the 17th International Middleware Conference, Middleware 2016, 2016, pp. 1–13.
- Manco, F., Mendes, J., Yasukata, K., Lupu, C., Kuenzer, S., Raiciu, C., Schmidt, F., Sati, S., & Huici, F. (2017). **My VM is Lighter (and Safer) than your Container. SOSP 2017 -** Proceedings of the 26th ACM Symposium on Operating Systems Principles, 16, 218–233. https://doi.org/10.1145/3132747.3132763

University of Cyprus
Department of Computer Science

Containers and Docker

History and Introduction

Containers: History and Introduction

Linux Containers

Prior Techniques

POSIX Capabilities (mid-1990s)**:**

- Capabilities: a set of flags associated with a process or file, which determined whether a process was permitted to perform certain actions;
  - A process could execute a subprocess with a subset of its own capabilities; the specification attempted to support the principle of least privilege.
  - This feature was never adopted as a standard but formed the basis of the capabilities feature added to the Linux Kernel in 1999.

Namespaces and **resource usage controls** for process isolation:

- In 2000, **FreeBSD** added Jails, which isolated filesystem namespaces (using *chroot*), processes and network resources in such a way that a process might be granted root privileges inside the jail but blocked from performing operations that would affect anything outside the jail.
- In 2001 & 2006, the **Linux Kernel** was patched to add filesystem namespaces and user namespaces to support resource **usage limits** and **isolation** for filesystems, network addresses, memory, process IDs, IPC, network stack and user IDs.

Access control and **System Call Filtering** offering secure isolation of processes through restricted access to system calls

Resource sharing in large-scale cluster management: Borg, Mesos

Container techniques

In 2008, **Linux Containers (LXC)** combined cgroups, namespaces, and capabilities from the Linux Kernel into a **tool for building and launching low-level system containers**.

Cgroups: *Control groups* are a kernel mechanism for controlling the resource allocation to process groups.

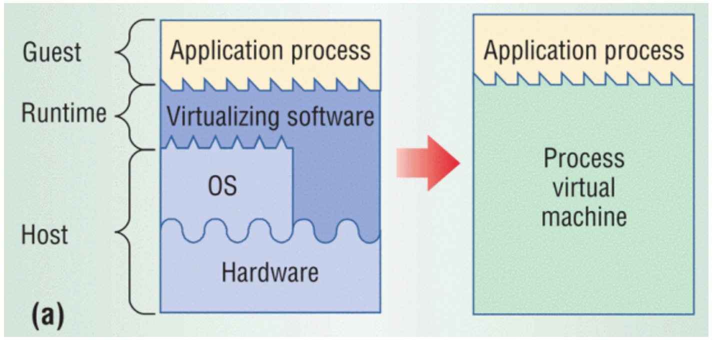
- Cgroups exist for each major resource type: CPU, memory, network, block-IO, and devices.
- The resource allocation for each of these can be controlled individually, allowing the complete resource limits for a process or a process group to be specified.

Namespaces. A namespace provides an abstraction for a kernel resource that makes it appear to a container that it has its own private, isolated instance of the resource.

- In Linux, there are namespaces for isolating: process IDs, user IDs, file system mount points, networking interfaces, IPC, and host names.

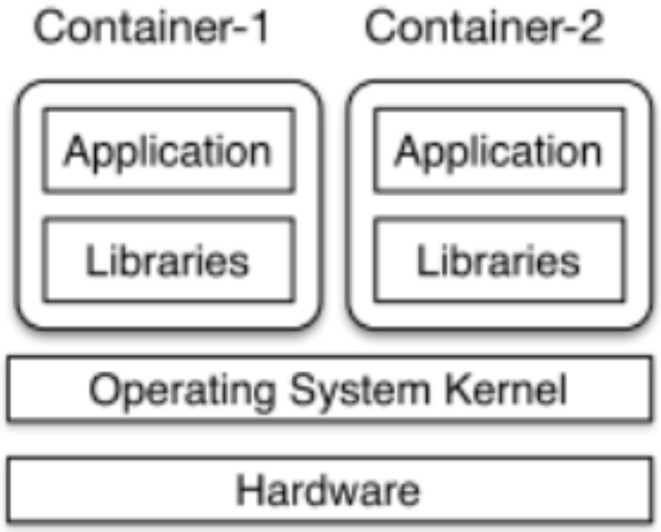
Linux Containers

- A Linux Container is a Linux process (or processes) that is a virtual environment with its own process network space.
- Linux Containers:
  - Offer lightweight process virtualization
  - Share portions of the host kernel
  - Use namespaces (per-process isolation of OS resources - filesystem, network and user ids) and cgroups (for resource management and accounting per process)
- Examples of container adoption in large-scale services:
  - https://www.netflix.com
  - https://www.dotcloud.com/
  - https://www.heroku.com/



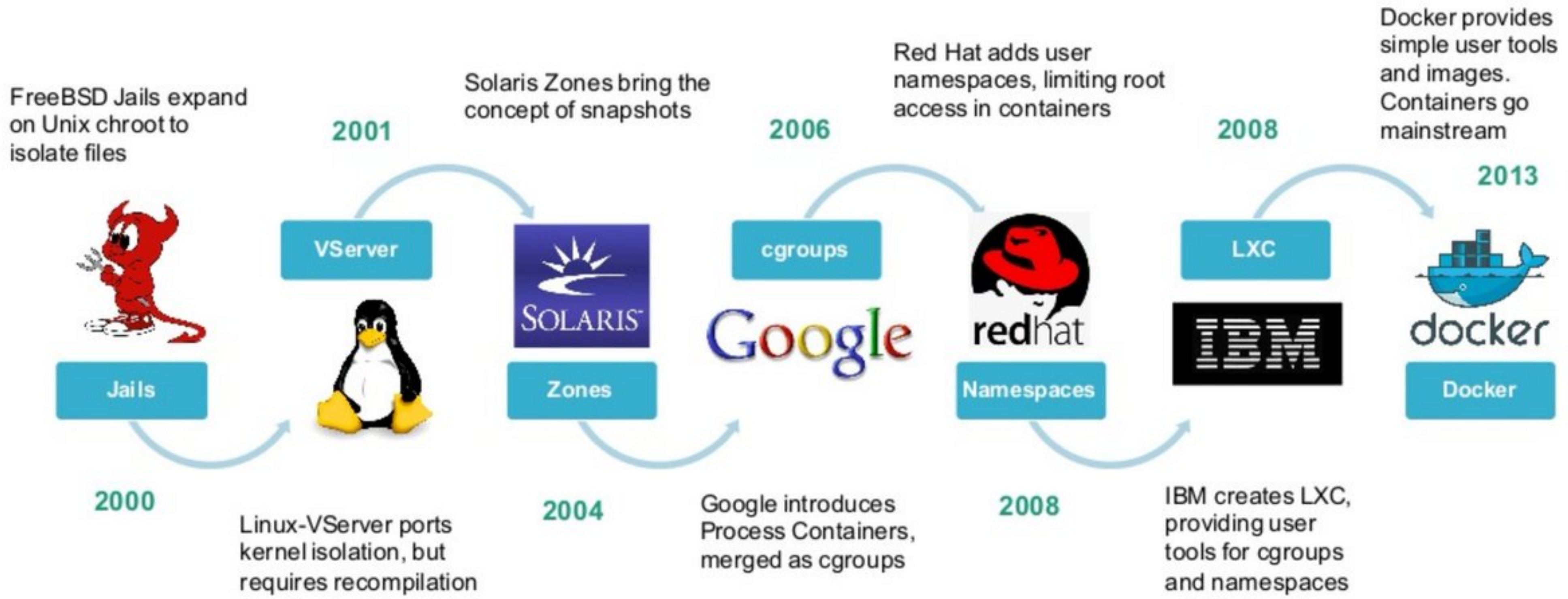
Container Abstraction

- A container encapsulates a group of processes that are isolated from other containers or processes in the system.
- The OS kernel is responsible for implementing the container abstraction:
  - It allocates CPU shares, memory and network I/O to each container
  - Can provide also file system isolation



Container behavior

- Containers may look like real computers from the point of view of programs running in them.
- However:
  - A computer program running on an ordinary operating system can see all resources of that computer (connected devices, files and folders, network shares, CPU power, quantifiable hardware capabilities).
  - Programs running inside of a container can only see the container's contents and devices assigned to the container.


FreeBSD Jails expand on Unix chroot to isolate files
2000
Jails
2001
VServer
Linux-VServer ports kernel isolation, but requires recompilation
Solaris Zones bring the concept of snapshots
SOLARIS
Zones
2004
2006
cgroups
Google
Google introduces Process Containers, merged as cgroups
Red Hat adds user namespaces, limiting root access in containers
redhat
Namespaces
2008
2008
LXC
IBM
IBM creates LXC, providing user tools for cgroups and namespaces
Docker provides simple user tools and images. Containers go mainstream
2013
docker
Docker

Containers: History and Introduction

Containers vs Virtual Machines

Containers vs VMs

- Hardware virtualization: predominant virtualization technology for deploying, packaging, and managing applications.
- Containers are increasingly filling that role due to the popularity of systems like Docker.
- Containers promise:
  - low-overhead virtualization since they do not run their own OS kernels, but instead rely on the underlying kernel for OS services
  - improved performance when compared to VMs.

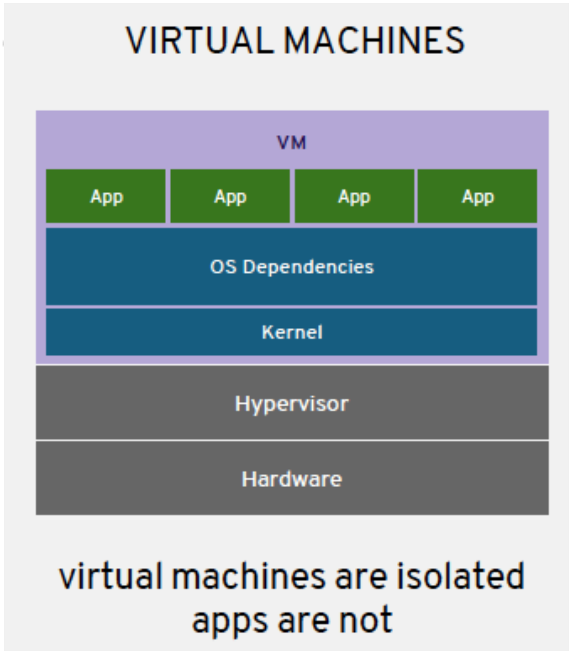
Virtual Machines VS Containers (Similarities)

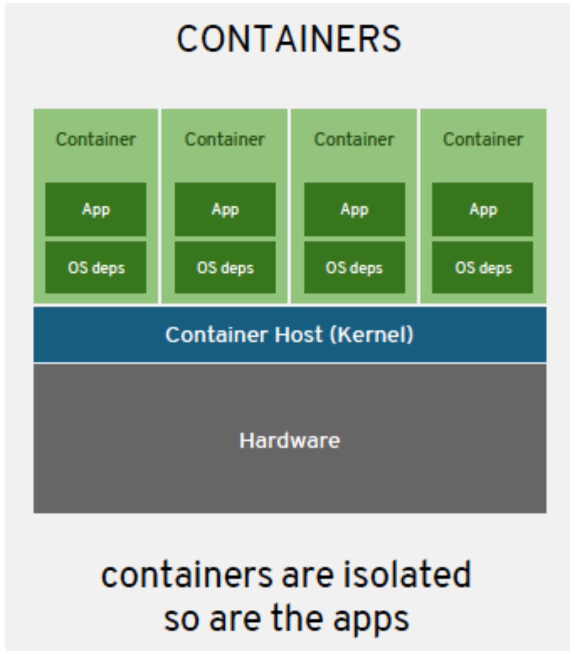
Virtual Machines	Containers
Process in one VM has not access to processes in other VMs	Process in one container has not access to processes in other containers
Each VM has own root filesystem	Each container has its own root file system (not Kernel)
Each VM gets its own virtual network adapter	Each container has its own virtual network adapter(s)
VMs run instances of physical files (.VMX and .VMDK)	Containers run instances of Images.
Host OS can be different from guest OS	Host OS distribution can be different from container OS distribution

Virtual Machines VS Containers (Differences)

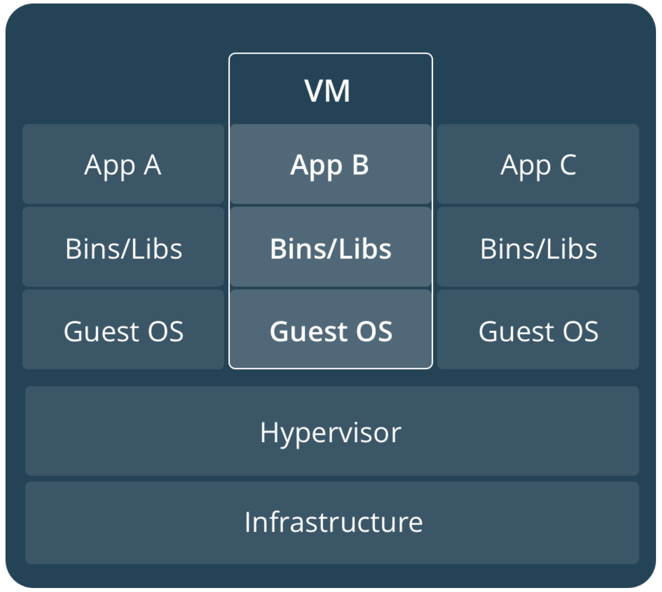
Virtual Machines	Containers
Each VM runs its own OS	All containers share the same Kernel of the host
Boot up time is in minutes	Containers instantiate in seconds
VMs snapshots are used sparingly	Images are built incrementally on top of another like layers. Lots of images/ snapshots
Not version controlled	Images can be diffed, version controlled and stored into repositories (Dockerhub).
Cannot run more than couple of VMs on a PC	Can run many containers on a PC

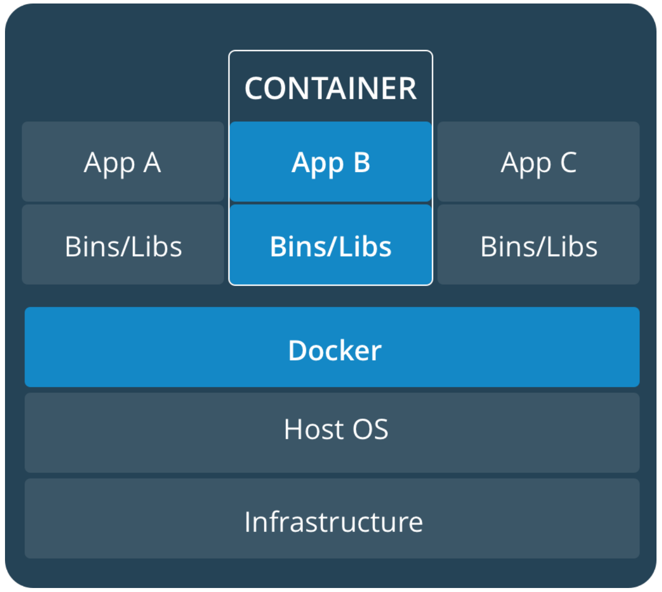
Containers vs VMs





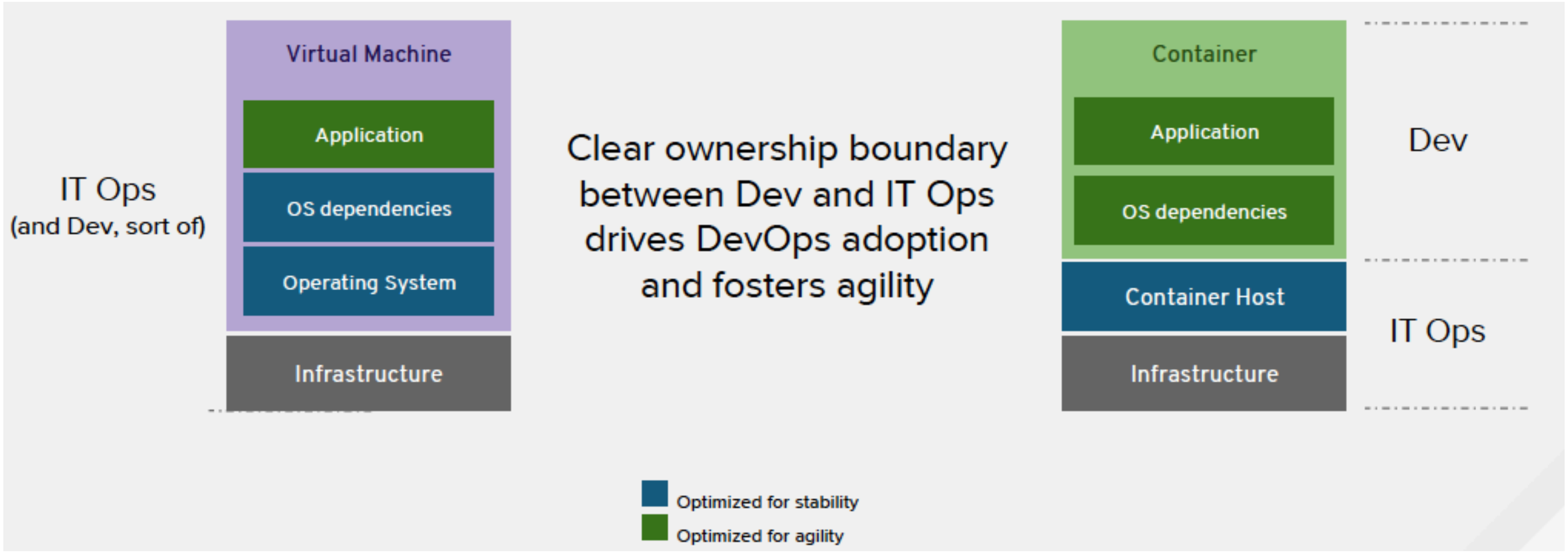
Virtual Machines VS Containers





- Each virtual machine (VM) includes the app, the necessary binaries and libraries and an entire guest operating system. Containers are NOT VMs because:
  - Use the host kernel
  - Can not boot a different OS (only if the host OS has pre-installed external kernel eg windows)
  - Do not have strict resource isolation (only on cgroups and namespace level)

Containers vs VMs: DevOps



Containers and Docker

Docker Overview

University of Cyprus
Department of Computer Science

What is Docker?

- Facilitates the building, management and use of containers.
- Most popular container solution:
  - Built initially on **LXC** with **namespaces** and **cgroups**.
  - Then replaced LXC with **libcontainer**, also using Linux Kernel **namespaces**, **cgroups**, and **capabilities**.
- Doesn't provide the container technology- it makes it simpler to use.
- Any software run with Docker is run inside a container.

Container isolation in Docker

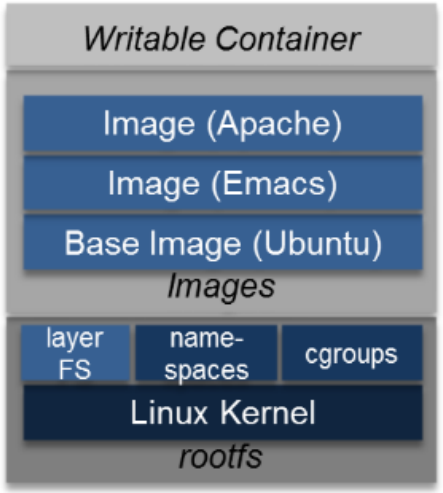
The containers that Docker builds are isolated with respect to eight aspects

1. PID namespace—Process identifiers and capabilities
2. UTS namespace—Host and domain name
3. MNT namespace—File system access and structure
4. IPC namespace—Process communication over shared memory
5. NET namespace—Network access and structure
6. USR namespace—User names and identifiers
7. chroot()—Controls the location of the file system root
8. cgroups—Resource protection

Container isolation in Docker

- **Namespace** isolation allows groups of processes to be separated. This ensures that **they cannot see resources in other groups**.
  - Different namespaces used for process isolation, network interfaces, access to inter-process communication, mount-points or for isolating kernel and version identifiers.
- **cgroups** (control groups) **manage and limit resource access** for process groups through limit enforcement, accounting and isolation, e.g., limiting the memory available to a specific container:
  - Enable better isolation between isolated applications on a host.
  - Restrict containers in multi-tenant host environments.
  - Allow sharing available hardware resources between containers while setting up **limits** and **constraints**.

Docker Image

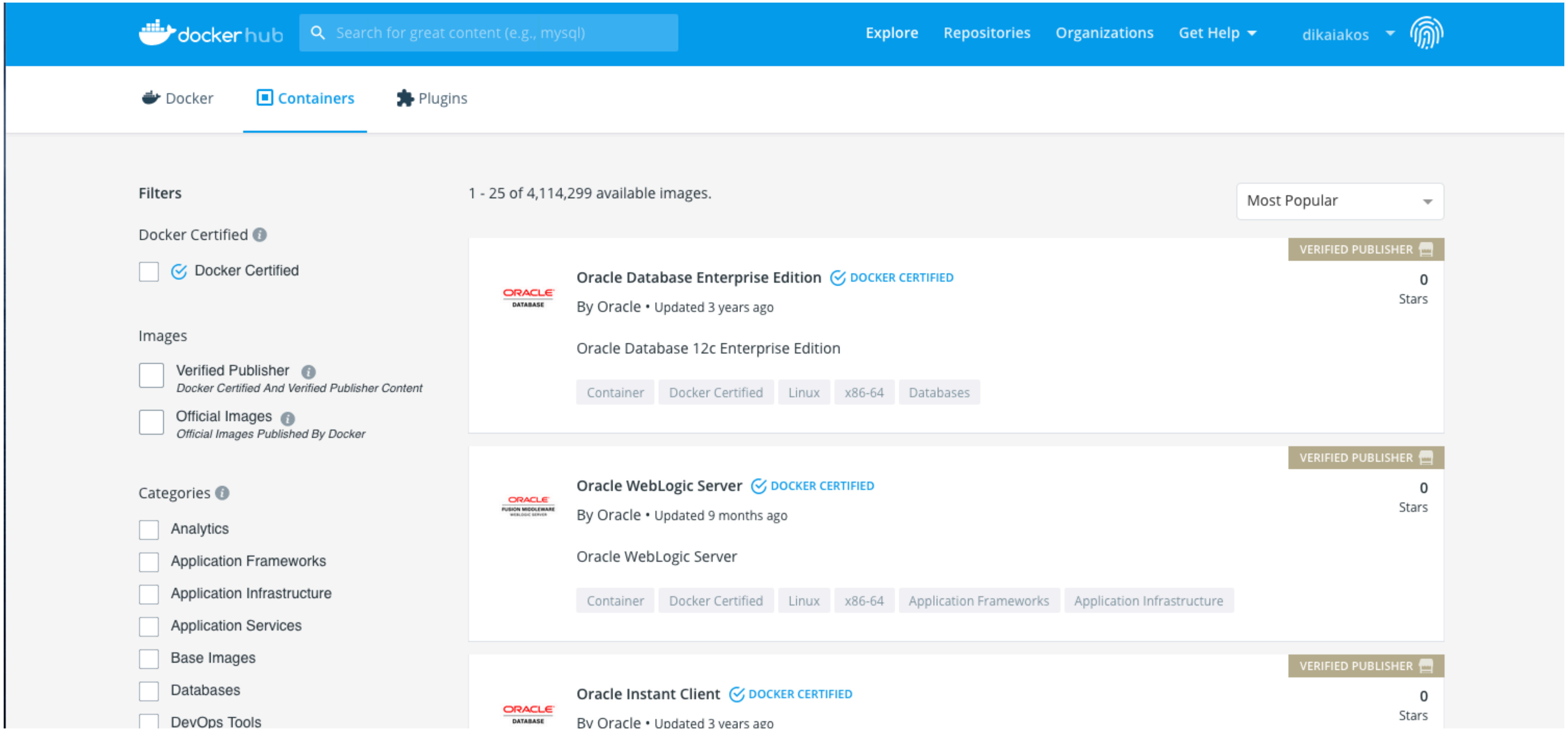


- Building block from which containers are launched.
- Bundled snapshot of all the files that should be available to a program running inside a container.
- Made up of file systems layered over each other.
- You can create as many containers as you want from an image.
- Containers started from the same image don’t share changes to their file systems.

Shipping Docker Images

- A Docker container is like a physical shipping container. It’s a box where you store and run an application and all of its dependencies.
- Docker can **run**, **copy**, and **distribute containers** with ease, including a way to package and distribute software.
- **Images** are the shippable units in the Docker ecosystem:
  - When you distribute software with Docker, you distribute Docker images, and the receiving computers create containers from them.
- Docker provides public infrastructure components that simplify distributing Docker images: registries and indexes.

Docker Hub



Filters

Docker Certified

☐ Docker Certified

Images

☐ Verified Publisher
Docker Certified And Verified Publisher Content

☐ Official Images
Official Images Published By Docker

Categories

☐ Analytics
☐ Application Frameworks
☐ Application Infrastructure
☐ Application Services
☐ Base Images
☐ Databases
☐ DevOps Tools

1 - 25 of 4,114,299 available images.

Most Popular

VERIFIED PUBLISHER

Oracle Database Enterprise Edition DOCKER CERTIFIED
By Oracle • Updated 3 years ago
Oracle Database 12c Enterprise Edition
Container | Docker Certified | Linux | x86-64 | Databases
0 Stars

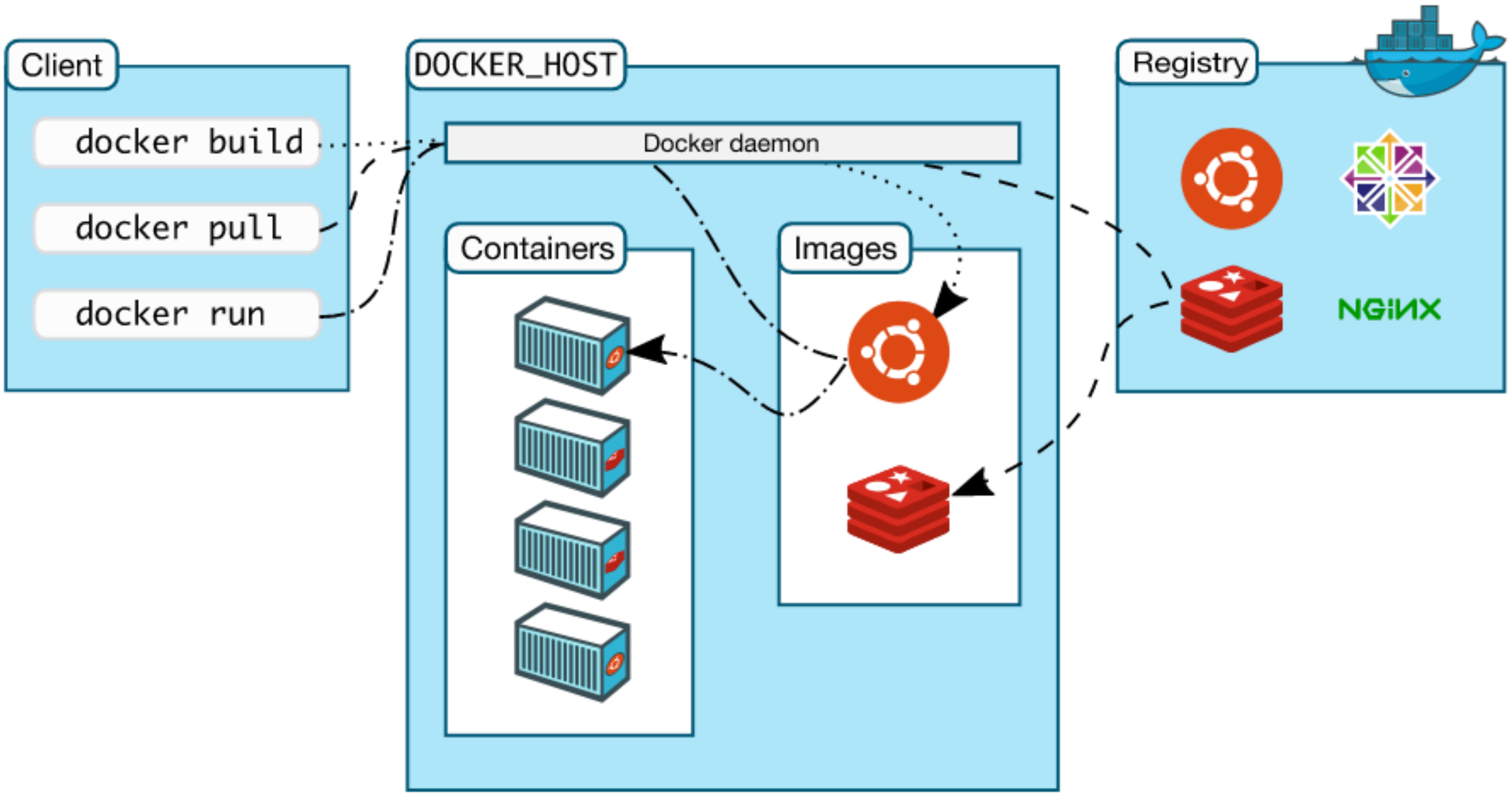
VERIFIED PUBLISHER

Oracle WebLogic Server DOCKER CERTIFIED
By Oracle • Updated 9 months ago
Oracle WebLogic Server
Container | Docker Certified | Linux | x86-64 | Application Frameworks | Application Infrastructure
0 Stars

VERIFIED PUBLISHER

Oracle Instant Client DOCKER CERTIFIED
By Oracle • Updated 3 years ago
0 Stars

High level Docker Architecture



Docker Overview

Running Docker

University of Cyprus
Department of Computer Science

Running Docker

- Natively on **Linux**
- Inside a single, small VM on **OS X** and **Windows**, where all its containers run:
    - small and fixed overhead of running the VM while the number of containers can scale up
- Convergence on Linux means that software running in Docker containers need only be written once against a consistent set of dependencies.

Running Docker

- Running two programs in user space:
  - Docker daemon: if installed properly, this process should always be running.
  - Docker CLI: the Docker program that users interact with.
    - If you want to start, stop, or install software, you'll issue a command using the Docker program.

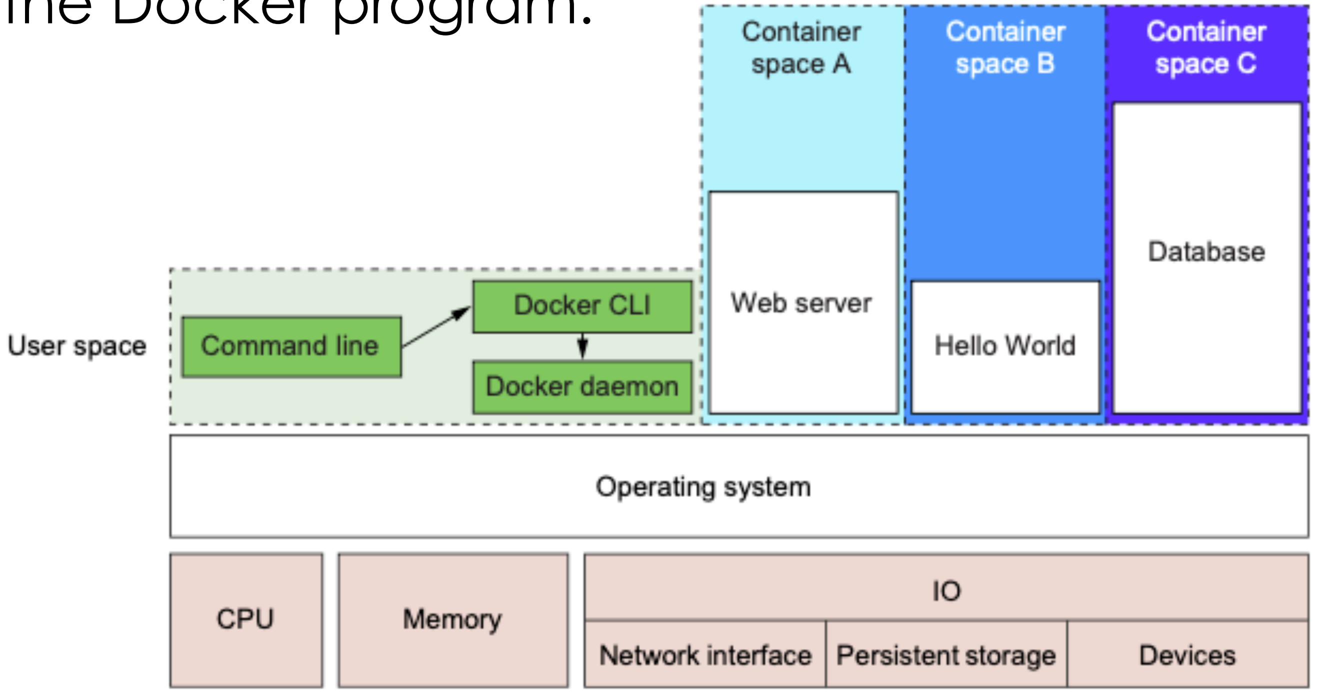
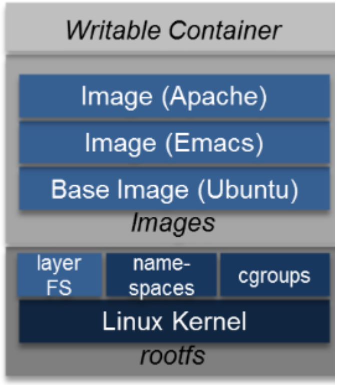


Figure 1.2 Docker running three containers on a basic Linux computer system

Running Docker (ctd')

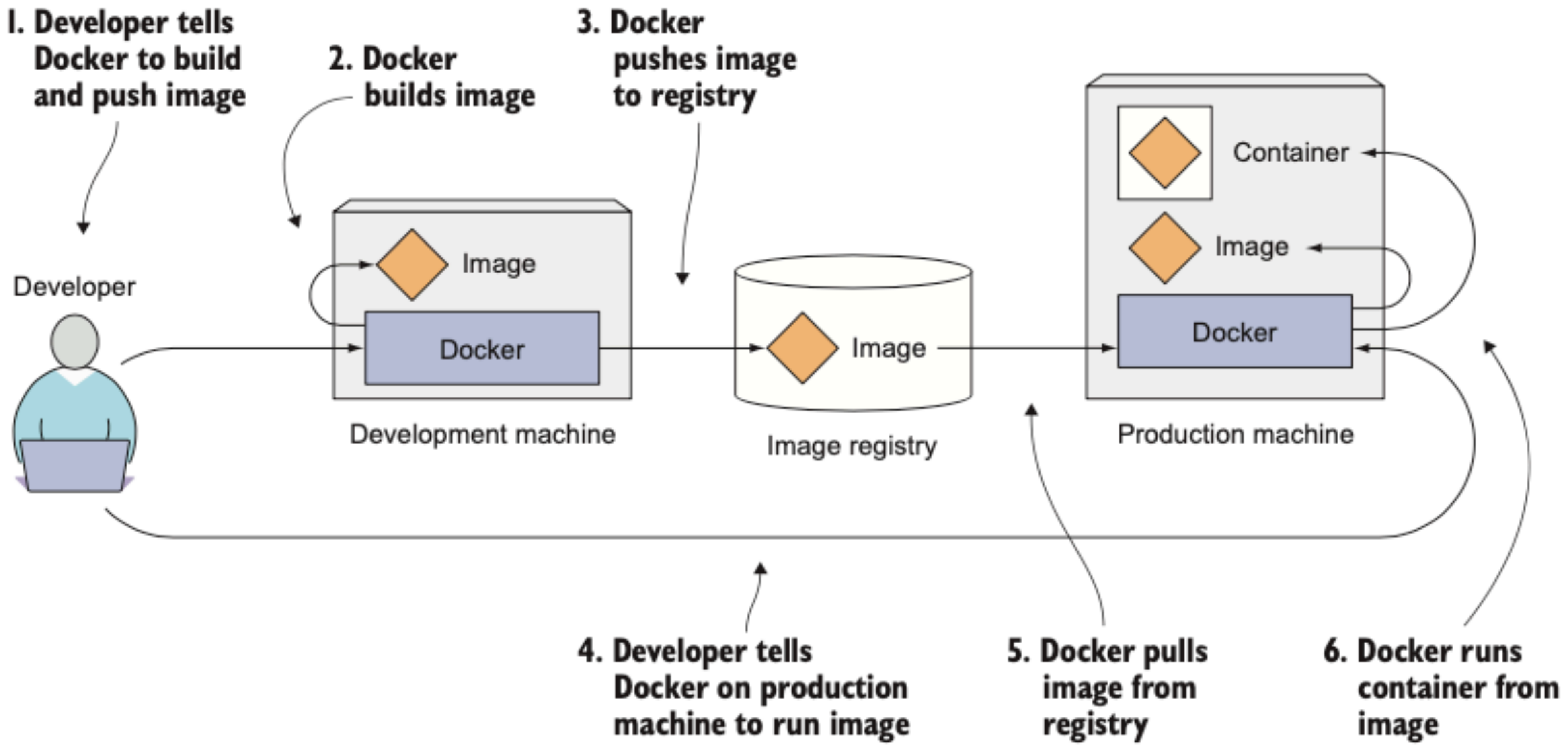


- systemd, a container-aware daemon starts containers as application processes.
  - It plays a key role as the root of the user's process tree.
- Boot process:
  - In a traditional Linux boot, the kernel first mounts the root file system as read-only, before checking its integrity. It then switches the rootfs volume to read-write mode.
  - Docker mounts the rootfs as read-only (as in a traditional Linux boot), but instead of changing the file system to read-write mode, it uses a union mount to add a writable file system on top of the read-only file system.
- Mounting: allows multiple read-only file systems to be stacked on top of each other.
  - This property can be used to create new images by building on top of base images.
  - Each of these file system layers is a separate image loaded by the container engine for execution.
- Container: Only the top layer is writable, which is the container itself.
  - The container can have state and is executable.
  - It is a kind of directory for everything needed for execution.
  - While they are normally stateful, containers can be made into stateless images to be reused in more complex builds.

Using Docker

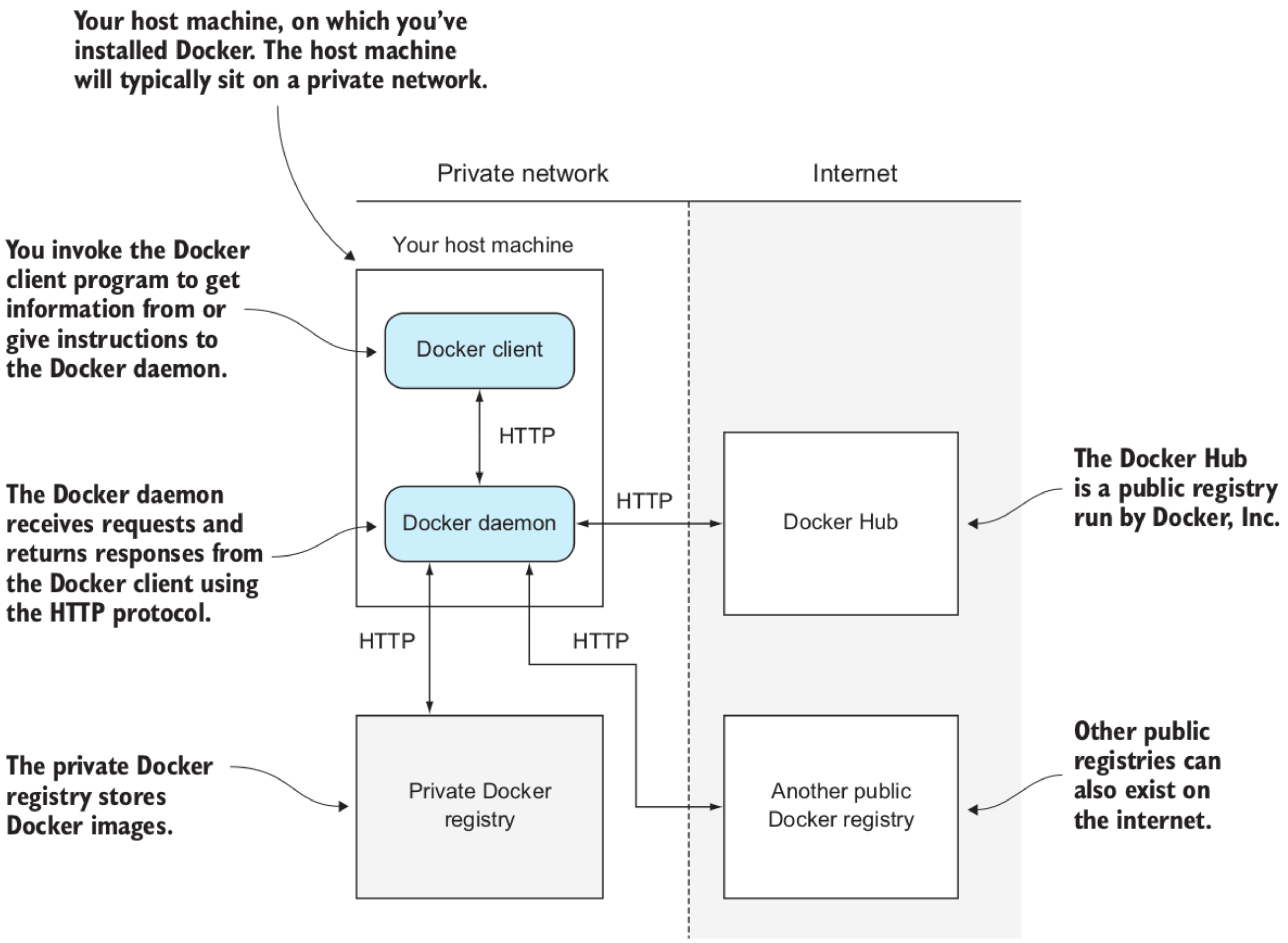
- Containers can be run with virtual terminals attached to the user's shell or in detached mode.
- By default, every Docker container has its own **PID namespace**, isolating process information for each container.
- Docker identifies every container by its generated container ID, abbreviated container ID, or its human-friendly name.
- All containers are in any one of four distinct states: **running**, **paused**, **restarting**, or **exited**.
- The docker **`exec`** command can be used to run additional processes inside a running container.
- A user can pass input or provide additional configuration to a process in a container by specifying environment variables at container-creation time.
- Using the **`--read-only`** flag at container-creation time will mount the container file system as read-only and prevent specialization of the container.
- A container restart policy, set with the **`--restart`** flag at container-creation time, will help your systems automatically recover in the event of a failure.
- Docker makes cleaning up containers with the **`docker rm`** command as simple as creating them.

Using Docker



I. Developer tells Docker to build and push image
2. Docker builds image
3. Docker pushes image to registry
Developer
Image
Docker
Development machine
Image
Image registry
Container
Image
Docker
Production machine
4. Developer tells Docker on production machine to run image
5. Docker pulls image from registry
6. Docker runs container from image

Figure 1.6 Docker images, registries, and containers


Your host machine, on which you've installed Docker. The host machine will typically sit on a private network.
Private network
Internet
Your host machine
You invoke the Docker client program to get information from or give instructions to the Docker daemon.
Docker client
HTTP
The Docker daemon receives requests and returns responses from the Docker client using the HTTP protocol.
Docker daemon
HTTP
Docker Hub
The Docker Hub is a public registry run by Docker, Inc.
HTTP
HTTP
The private Docker registry stores Docker images.
Private Docker registry
Another public Docker registry
Other public registries can also exist on the internet.

Docker Overview

Using Docker Containers

University of Cyprus
Department of Computer Science

WHAT HAPPENS WHEN YOU EXECUTE THE FOLLOWING COMMAND ON THE COMMAND LINE?

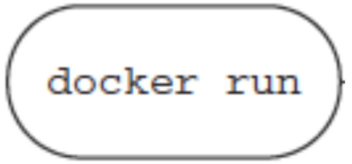
```
docker run --name hellow
   dockerinaction/hello_world
```

University of Cyprus
Department of Computer Science

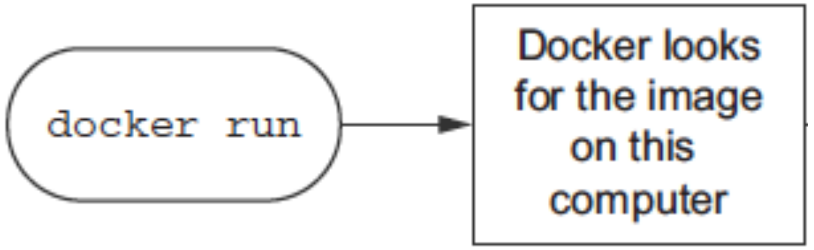
```
docker run --name hellow
    dockerinaction/hello_world
```

- `Docker run`: installs, runs, and stops a program inside a container.
- Assigns `hellow` as name of this container
- The program that you tell it to run in a container is:
  - `dockerinaction/hello_world`.
  - This is called the repository (or image) name.
- To learn more about this command, execute:
  - `docker help run`

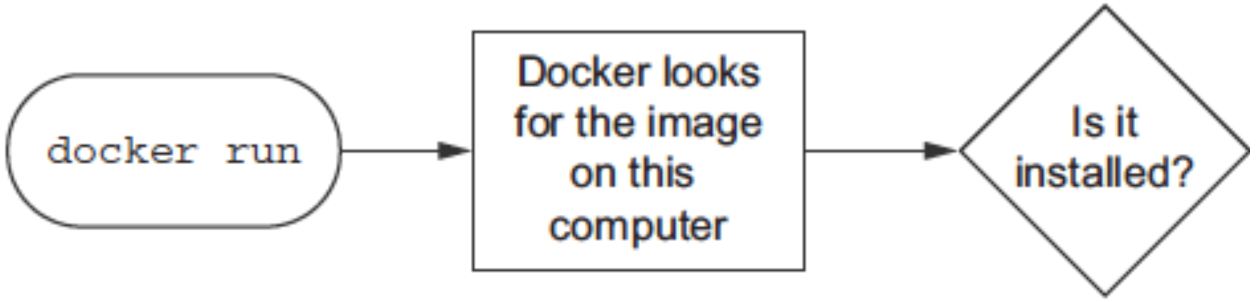
The lifecycle of "Hello World"



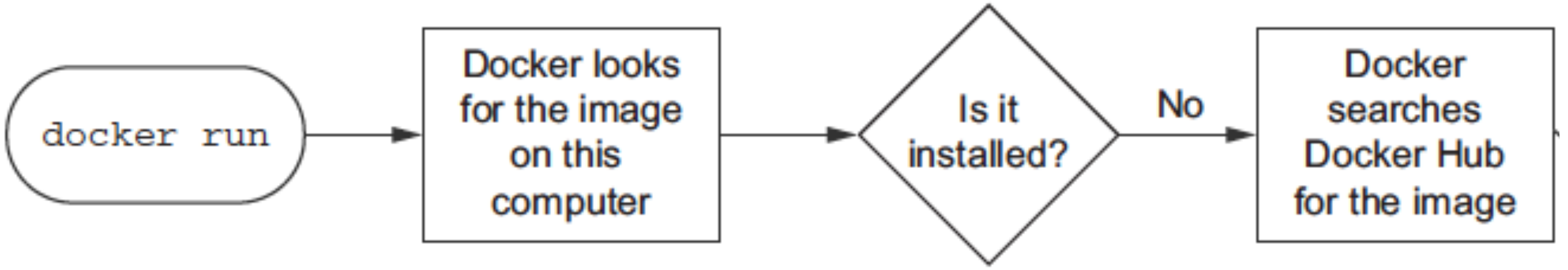
The lifecycle of "Hello World"



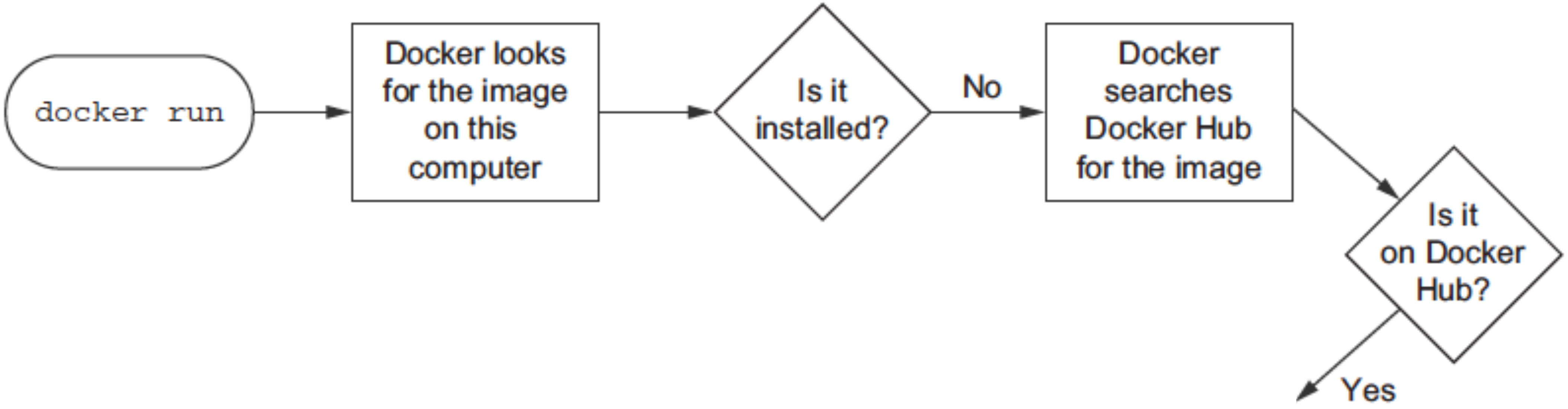
The lifecycle of "Hello World"



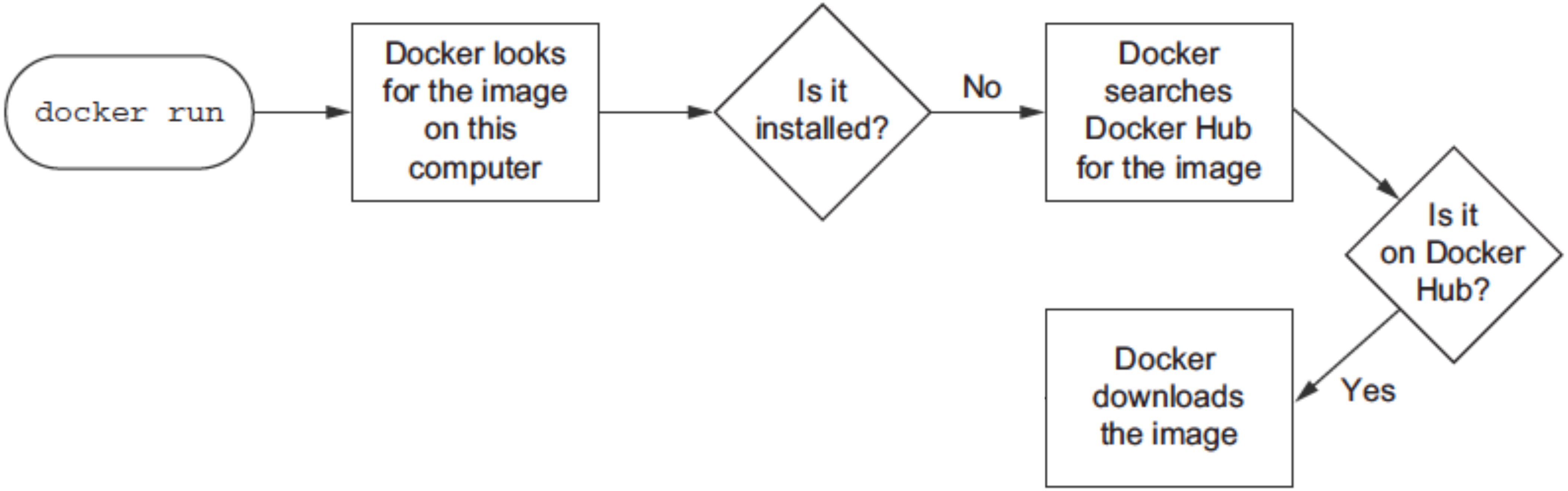
The lifecycle of “Hello World”


docker run
Docker looks for the image on this computer
Is it installed?
No
Docker searches Docker Hub for the image

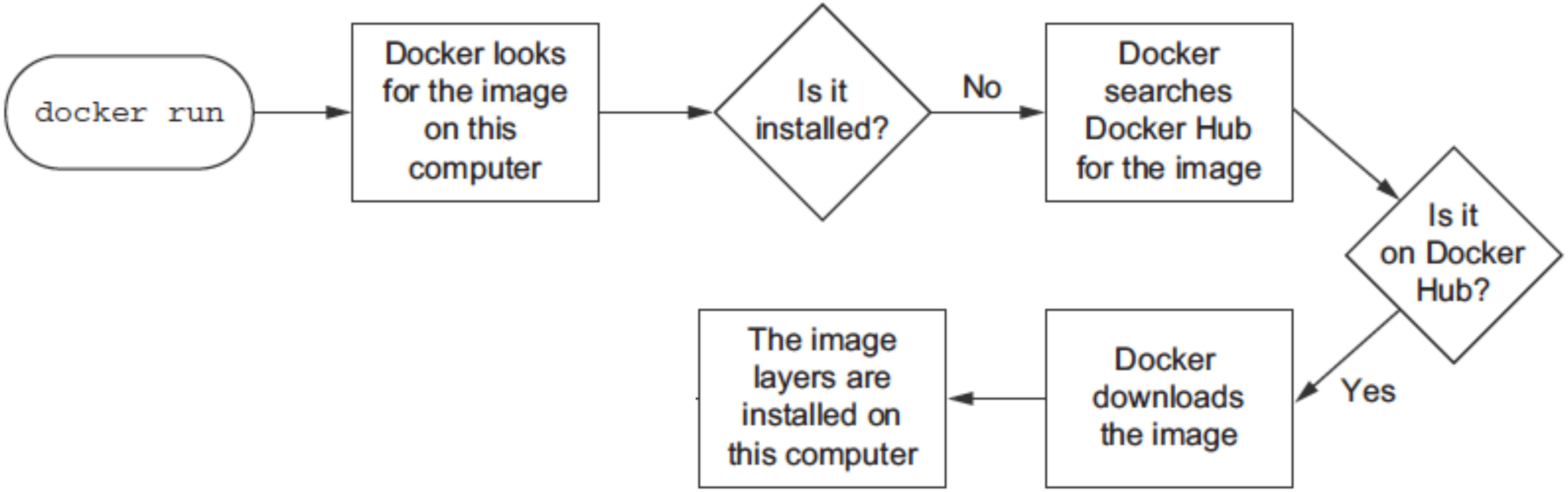
The lifecycle of "Hello World"


docker run
Docker looks for the image on this computer
Is it installed?
No
Docker searches Docker Hub for the image
Is it on Docker Hub?
Yes

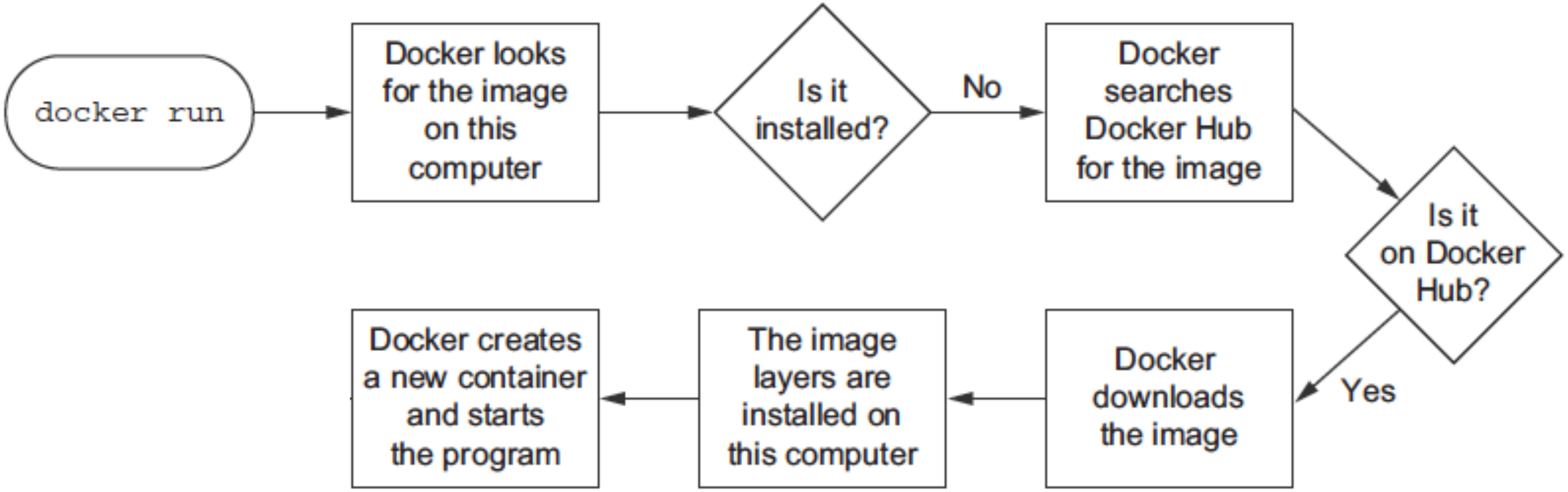
The lifecycle of “Hello World”


docker run
Docker looks for the image on this computer
Is it installed?
No
Docker searches Docker Hub for the image
Is it on Docker Hub?
Yes
Docker downloads the image

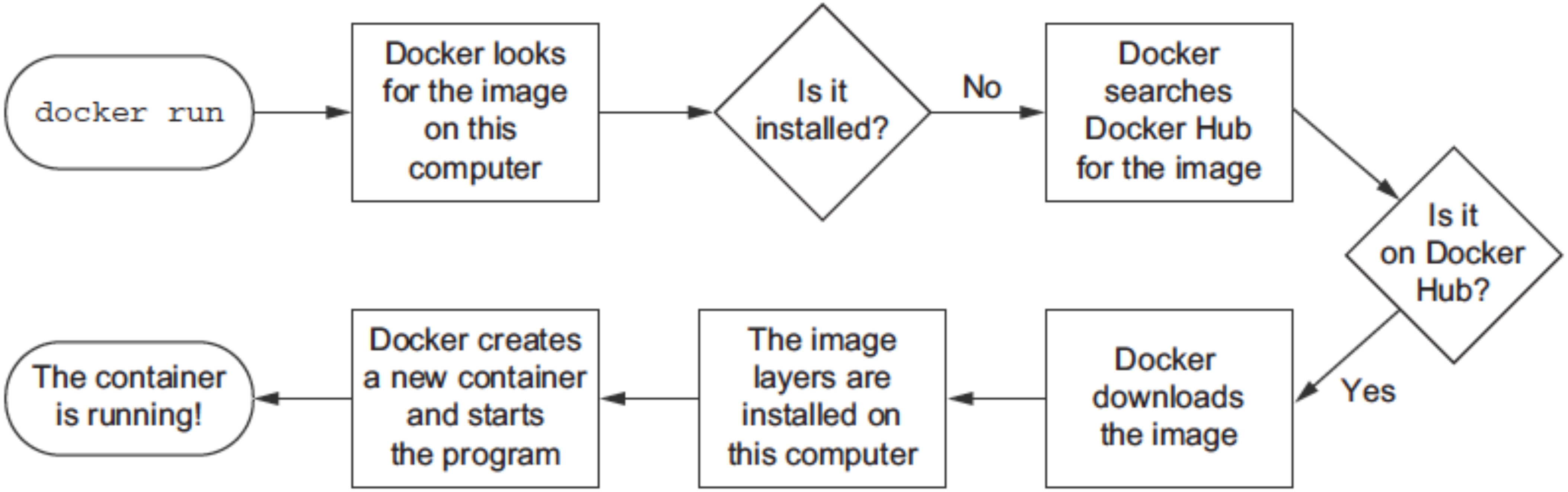
The lifecycle of "Hello World"


docker run
Docker looks for the image on this computer
Is it installed?
No
Docker searches Docker Hub for the image
Is it on Docker Hub?
Yes
Docker downloads the image
The image layers are installed on this computer

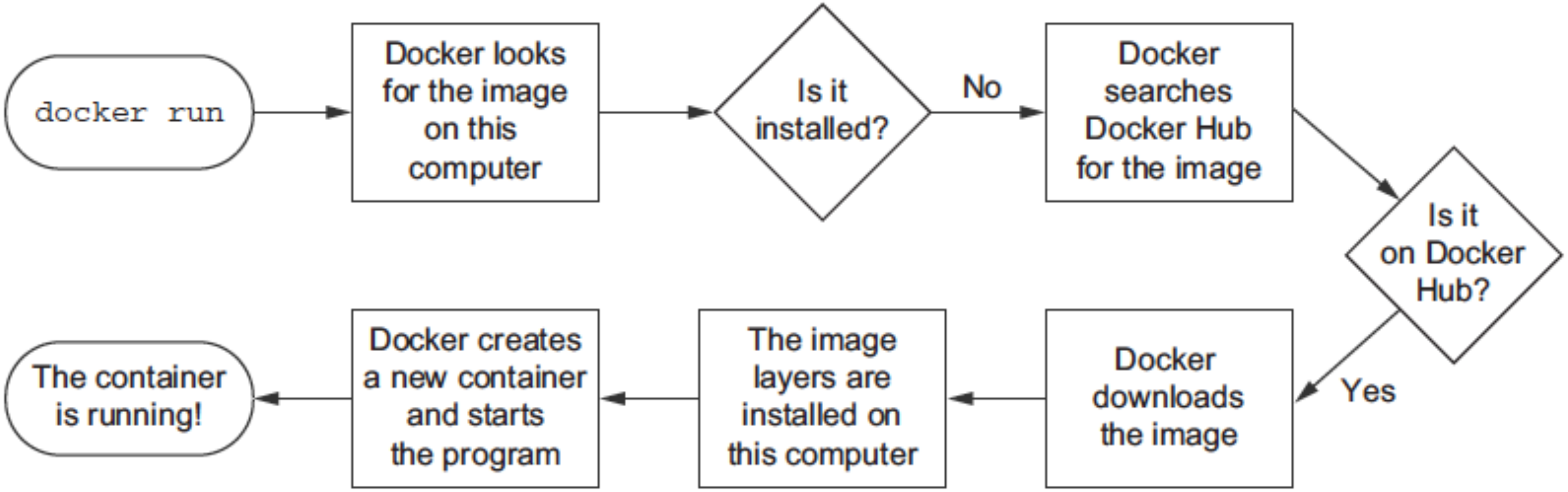
The lifecycle of “Hello World”


docker run
Docker looks for the image on this computer
Is it installed?
No
Docker searches Docker Hub for the image
Is it on Docker Hub?
Yes
Docker downloads the image
The image layers are installed on this computer
Docker creates a new container and starts the program

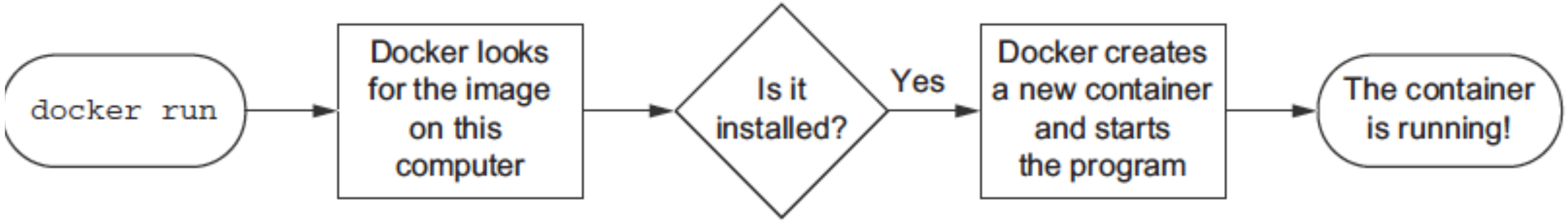
The lifecycle of “Hello World”


docker run
Docker looks for the image on this computer
Is it installed?
No
Docker searches Docker Hub for the image
Is it on Docker Hub?
Yes
Docker downloads the image
The image layers are installed on this computer
Docker creates a new container and starts the program
The container is running!

The lifecycle of “Hello World”


docker run
Docker looks for the image on this computer
Is it installed?
No
Docker searches Docker Hub for the image
Is it on Docker Hub?
Yes
Docker downloads the image
The image layers are installed on this computer
Docker creates a new container and starts the program
The container is running!

The lifecycle of “Hello World”


docker run
Docker looks for the image on this computer
Is it installed?
Yes
Docker creates a new container and starts the program
The container is running!

Pull and Run an Image

- **Problem**: You want download, save and run an image to your PC
- **Solution**: Execute the docker pull command to fetch the image and the docker run to execute the container
- Pull the image:
  - `docker pull busybox`
- Check if the image is downloaded:
  - `docker image ls`
- Run the busybox image:
  - `docker run busybox`
- Run the busybox image with parameters:
  - `docker run busybox echo "hello from busybox”`

Check your Containers

- **Problem**: You want to check your containers
- **Solution**: Execute the docker ps command
- Check the running containers:
  - `docker ps`
- Check all containers (even stopped):
  - `docker ps -a`
- Run the busybox image and connect to it:
  - `docker run -it busybox`
- Check again the running containers:
  - `docker ps`

Detaching without Stopping

- **Problem**: You want to detach from a container interaction without stopping it.
- **Solution**: Press Ctrl-P and then Ctrl-Q to detach.

Removing Containers

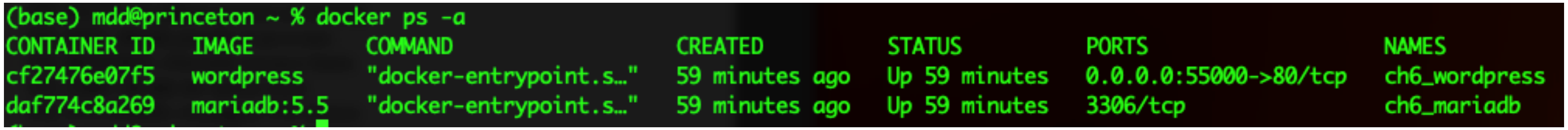
- **Problem**: You want to remove the not running container instances
- **Solution**: Execute the docker rm command
- Remove specific containers
  - `docker rm 305297d7a235 ff0a5c3750b9`
- Find the containers that have status exited
  - `docker ps -a -q -f status=exited`
- Remove all containers that have status exited
  - `docker rm $(docker ps -a -q -f status=exited)`
  - `docker container prune`
- Remove the container after run
  - `docker run --rm busybox`

Removing Docker Images

- **Problem**: You want to remove an image - You have to remove ALL CONTAINERS that use the image you want to remove
- **Solution**: Execute the docker rmi command
- Enumerate all docker images that are in your pc
  - `docker image ls`
- Select the correct image ID
- Remove specific containers
  - `docker rmi 305297d7a235 ff0a5c3750b9`
- Remove all docker relative items like containers, images, networks
  - `docker system prune -a`

Starting a Stopped Container

- **Problem**: You have closed a container and you want to restart
- **Solution**: Execute docker start command
- Check all closed containers: **`docker ps -a`**



```
(base) mdd@princeton ~ % docker ps -a
CONTAINER ID   IMAGE         COMMAND                   CREATED          STATUS          PORTS                    NAMES
cf27476e07f5   wordpress     "docker-entrypoint.s…"   59 minutes ago   Up 59 minutes   0.0.0.0:55000->80/tcp    ch6_wordpress
daf774c8a269   mariadb:5.5   "docker-entrypoint.s…"   59 minutes ago   Up 59 minutes   3306/tcp                 ch6_mariadb
```

- Start a specific container:
  - **`docker start db39...`**
- Run again docker ps in order to check if your container is running: **`docker ps`**

Connecting with Running Containers

- **Problem**: You want to connect with terminal or run commands on a running container
- **Solution**: Run the docker exec command
- Check all running containers: `docker ps`
- Find the ID of the specific running containe r: `3c3f8e3fb05d795`…
- Run the docker exec with the same ID as parameter and open an interactive terminal or run whatever command you like:
  - `docker exec -it 3c3f8e3fb05d795 sh`
  - `docker exec -it 3c3f8e3fb05d795 {command}`

Start a Container as Daemon

- **Problem**: You want to start a container as daemon
- **Solution**: Use the parameter -d at the run command
- Start the image in background
  - `docker run -d --name sleeper busybox sleep infinity`
- The previous command returns the ID of the new container `3c3f8e3fb05d795`...
- Run the docker exec with the same ID as parameter and open an interactive terminal
  - `docker exec -it 3c3f8e3fb05d795 sh`

Executing Commands on your Container

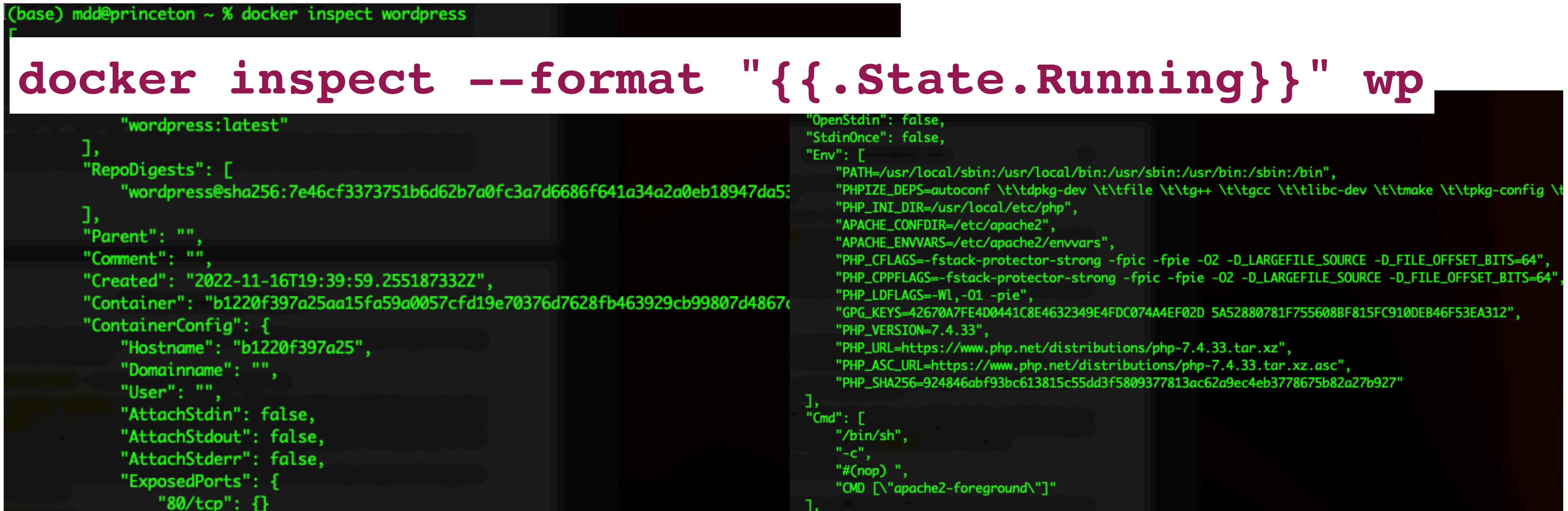
- **Problem**: You want to perform commands on a running container.
- **Solution**: Use the docker exec command.
- Run a container as daemon
  - `docker run -d --name sleeper busybox sleep infinity`
- Run an echo command from the container.
  - `docker exec sleeper echo "hello host from container”`
- Run in the container as background process.

```
docker exec -d sleeper \
   find / -ctime 7 -name '*log' -exec rm {} \;
```

- Run an interactive terminal in the container
  - `docker exec -i -t sleeper sh`

Inspecting a Container

- **Problem:** Find out all the information that Docker maintains regarding a container (its metadata)
- **Solution:** The docker inspect command will display all the metadata (a JSON document) that Docker maintains for a container.
  - The **`format`** option transforms that metadata
  - In this case it filters everything except for the field indicating the running state of the container.



```
(base) mdd@princeton ~ % docker inspect wordpress
[
```

```
docker inspect --format "{{.State.Running}}" wp
```

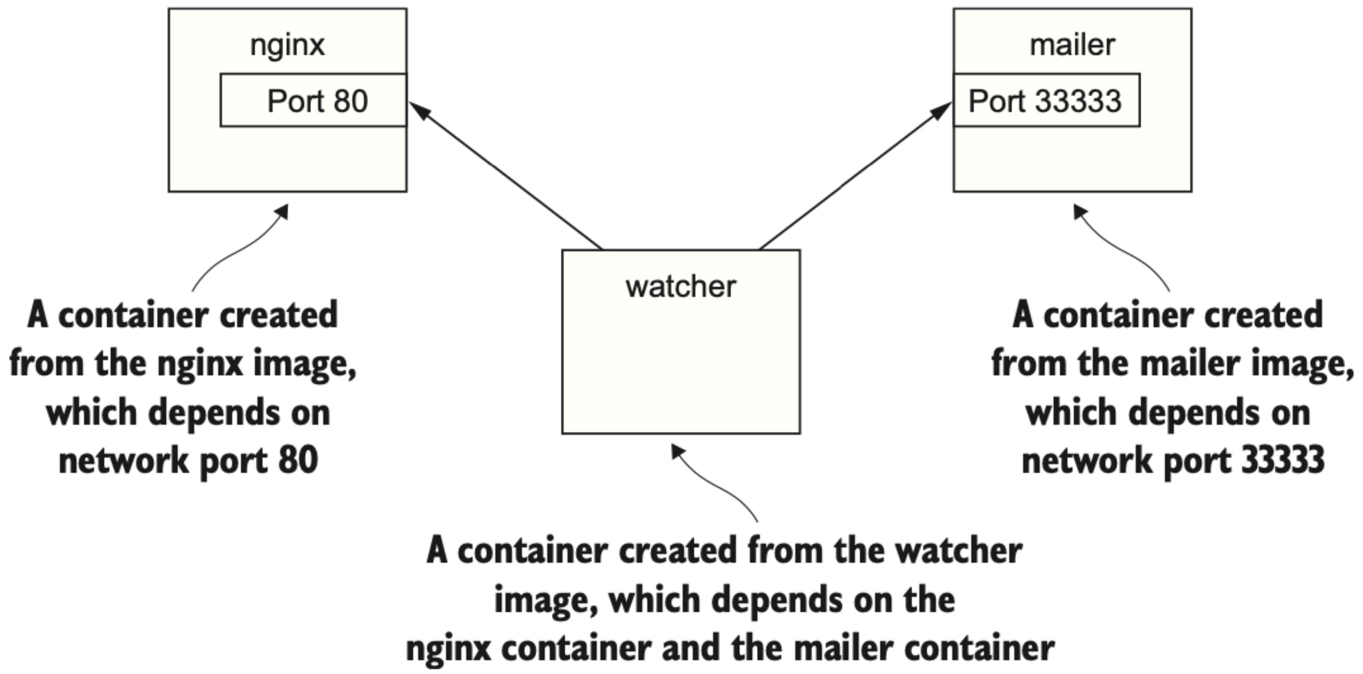
```
            "wordpress:latest"
        ],
        "RepoDigests": [
            "wordpress@sha256:7e46cf3373751b6d62b7a0fc3a7d6686f641a34a2a0eb18947da53
        ],
        "Parent": "",
        "Comment": "",
        "Created": "2022-11-16T19:39:59.255187332Z",
        "Container": "b1220f397a25aa15fa59a0057cfd19e70376d7628fb463929cb99807d4867
        "ContainerConfig": {
            "Hostname": "b1220f397a25",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "ExposedPorts": {
                "80/tcp": {}
```

```
"OpenStdin": false,
"StdinOnce": false,
"Env": [
    "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
    "PHPIZE_DEPS=autoconf \t\tdpkg-dev \t\tfile \t\tg++ \t\tgcc \t\tlibc-dev \t\tmake \t\tpkg-config \t
    "PHP_INI_DIR=/usr/local/etc/php",
    "APACHE_CONFDIR=/etc/apache2",
    "APACHE_ENVVARS=/etc/apache2/envvars",
    "PHP_CFLAGS=-fstack-protector-strong -fpic -fpie -O2 -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64",
    "PHP_CPPFLAGS=-fstack-protector-strong -fpic -fpie -O2 -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64",
    "PHP_LDFLAGS=-Wl,-O1 -pie",
    "GPG_KEYS=42670A7FE4D0441C8E4632349E4FDC074A4EF02D 5A52880781F755608BF815FC910DEB46F53EA312",
    "PHP_VERSION=7.4.33",
    "PHP_URL=https://www.php.net/distributions/php-7.4.33.tar.xz",
    "PHP_ASC_URL=https://www.php.net/distributions/php-7.4.33.tar.xz.asc",
    "PHP_SHA256=924846abf93bc613815c55dd3f5809377813ac62a9ec4eb3778675b82a27b927"
],
"Cmd": [
    "/bin/sh",
    "-c",
    "#(nop) ",
    "CMD [\"apache2-foreground\"]"
],
```

Docker CLI Basic Commands

Command	Purpose
`docker build`	Build a Docker image
`docker images / docker image ls`	List all images on a Docker host
`docker run {image}`	Run a Docker image as a container
`docker ps`	List all running (or stopped instances)
`docker commit {container}`	Commit a Docker container as an image
`docker tag {image}`	Tag a Docker image
`docker logs {container}`	Display the logs of an instance
`docker stop {container}`	Stop a running instance
`docker rm {container}`	Remove an instance
`docker rmi {image}`	Remove an image

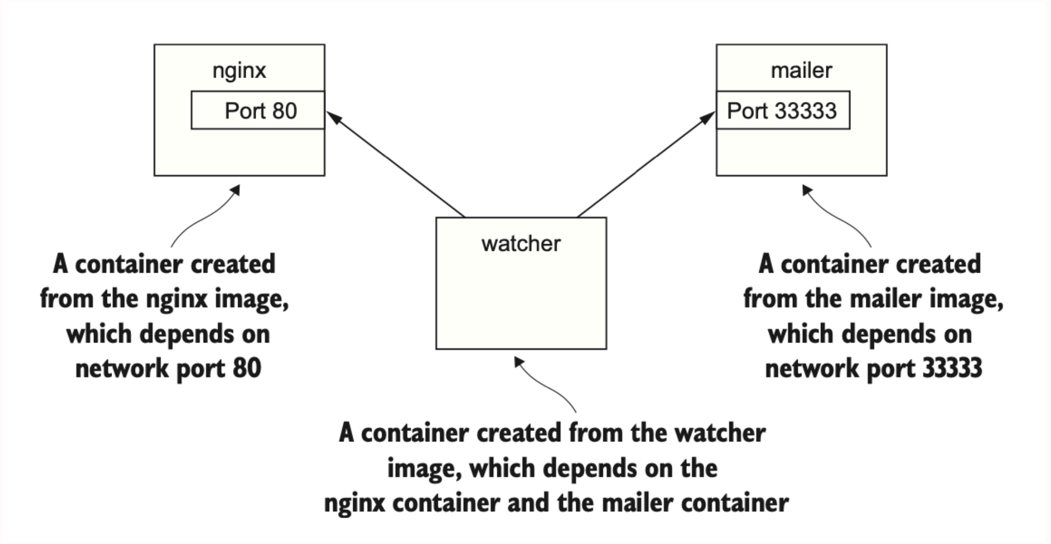
Web site monitoring


nginx
Port 80
mailer
Port 33333
watcher
A container created from the nginx image, which depends on network port 80
A container created from the mailer image, which depends on network port 33333
A container created from the watcher image, which depends on the nginx container and the mailer container

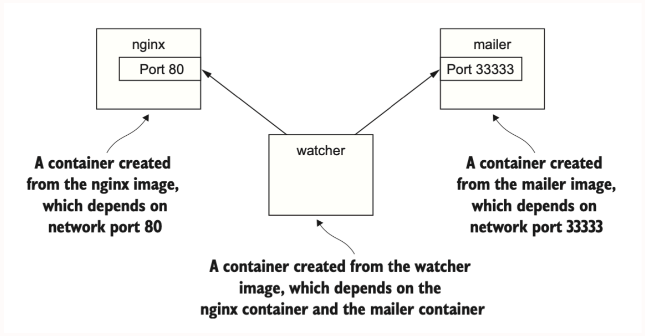
Three containers:

- The first will run NGINX;
- the second will run a program called a mailer.
  - Both of these will run as detached containers.
  - Detached means that the container will run in the background, without being attached to any input or output stream.
- A third program, called an agent, will run in an interactive container.

```
docker run --detach --name web nginx:latest

docker run -d --name mailer dockerinaction/ch2_mailer

docker run --interactive --tty \
    --link web:web \
    --name web_test \
    busybox:latest /bin/sh

wget -O - http://web:80

docker run -it \
    --name agent \
    --link web:insideweb \
    --link mailer:insidemailer \
    dockerinaction/ch2_agent

docker logs mailer
```


nginx
Port 80
mailer
Port 33333
watcher
A container created from the nginx image, which depends on network port 80
A container created from the mailer image, which depends on network port 33333
A container created from the watcher image, which depends on the nginx container and the mailer container

```
docker run --detach --name web nginx:latest

docker run -d --name mailer dockerinaction/ch2_mailer

docker run --interactive --tty \
    --link web:web \
    --name web_test \
    busybox:latest /bin/sh

wget -O - http://web:80

docker run -it \
    --name agent \
    --link web:insideweb \
    --link mailer:insidemailer \
    dockerinaction/ch2_agent

docker logs mailer
```


nginx
Port 80
mailer
Port 33333
watcher
A container created from the nginx image, which depends on network port 80
A container created from the mailer image, which depends on network port 33333
A container created from the watcher image, which depends on the nginx container and the mailer container

Linking Containers

- Network links is a legacy mechanism to connect containers:
  - Injects IP addresses into dependent *running* containers (containers that aren't running don't have IP addresses).
  - Links create a unidirectional network connection from one container to other containers on the same host.
- Bidirectional links can be created with user-defined networks.

Docker Overview

Namespaces in Docker

PID Namespaces

- Every running program—or process—on a Linux machine has a unique number called a **process identifier** (PID).
- A PID namespace is the set of possible numbers that identify processes.
- Linux provides facilities to create multiple PID namespaces - each namespace has a complete set of possible PIDs (1, 2, 3,…)
- Docker creates a new PID namespace for each container by default.
  - The container's PID namespace isolates processes in that container from processes in other containers.
- You can **optionally** create containers **without their own PID namespace** (e.g. to perform sysadmin tasks on a machine), using the `--pid host` flag to keep the host's namespace.
  - The following lists all processes running on the computer which runs Docker:

```
docker run --pid host busybox:latest ps
```

Naming Containers

```
docker run --detach --name web nginx:latest
```

- By default Docker assigns a unique (human-friendly) name to each container it creates.
    - The **`--name`** flag can override that.
- In systems with multiple containers, using fixed names like web can create conflicts.
- In addition to the name, Docker assigns each container with a unique 1024-bit identifier.
- To avoid conflict with fixed names and the complexity of long identifiers, Docker enables the handling of container IDs by assigning them to **environment variables**

```
CID=$(docker create nginx:latest)
```

or **files** (using the —cidfile flag):

```
docker create --cidfile /tmp/web.cid nginx
```

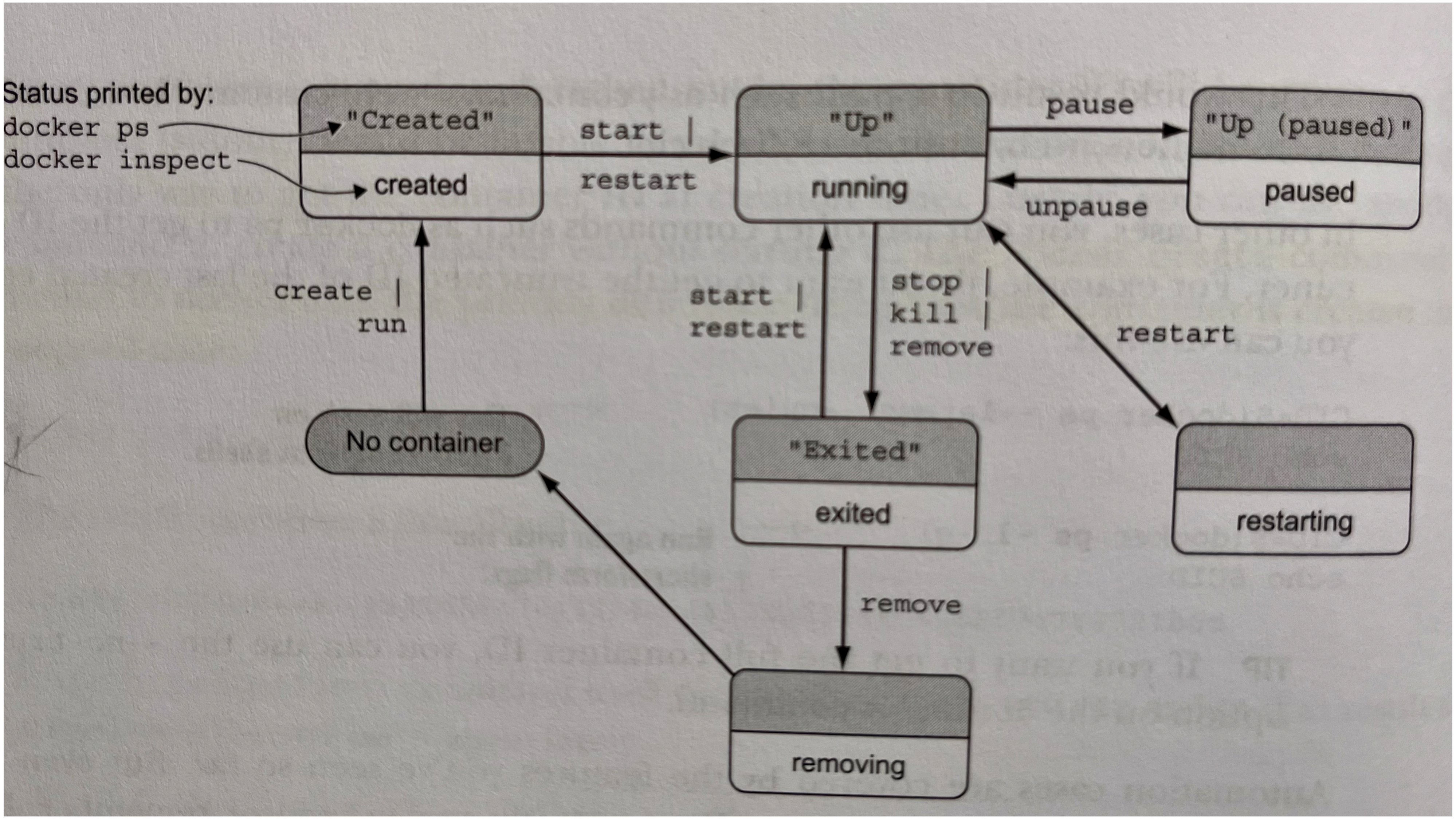
Docker Overview

Container State

Listing Containers

- To check which containers are currently running, use: **docker ps** This returns the following information for **running** containers:
  - The container ID
  - The image used
  - The command executed in the container
  - The time since the container was created
  - The duration that the container has been running
  - The network ports exposed by the container
  - The name of the container
- To see all the containers use: **docker ps -a**
- Note that whether you're using **docker run** or **docker create**, the resulting containers need to be started in the reverse order of their dependency chain, otherwise you get an error.
- A Docker container can be in one of four states: **Running**, **Paused**, **Restarting**, **Exited** (also used if the container has never been started)

Moving Between States



Docker Overview

Environment-agnostic Systems

Building Low-maintenance Systems

- If you build systems or software that know too much about their environment (addresses or fixed locations of dependency services) it's difficult to change that environment or reuse the software.
- Need to **minimize environment dependences**, namely *specializations* of the computing environment, such as:
  - Global-scoped dependencies - e.g., known host file system locations
  - Hard-coded deployment architectures- e.g. environment checks in code or configuration
  - Data locality- e.g. data stored on a particular computer outside the deployment architecture
- Building **low-maintenance systems** requires minimizing these aspects. To this end, docker provides:
  - Read-only file systems
  - Environment variable injection
  - Volumes

Read-only F/S in Docker

- With a container with a read-only filesystem:
  - The container won't be specialized from changes to the files it contains.
  - There is increased confidence that an attacker can't compromise files in the container.

Running WordPress

- **WordPress** is a popular open source content-management and blogging program (CMS).
- Each WordPress installation can be customized/ specialized based on the data/configuration parameters it works with.
- **Task**: run WordPress and integrate it with the monitoring infrastructure developed in the previous example!

- WordPress is published through Docker Hub in a repository named wordpress.

```
docker run -d --name wp --read-only wordpress
```

- Let’s see if it works (instead of using docker ps):

```
docker inspect --format "{{.State.Running}}" wp
docker logs wp
```

```
error: missing WORDPRESS_DB_HOST and MYSQL_PORT_3306_TCP environment variables
Did you forget to --link some_mysql_container:mysql or set an external db
with -e WORDPRESS_DB_HOST=hostname:port?
```

- WordPress has a dependency on a MySQL database
- Install MySQL:

```
docker run -d --name wpdb  -e MYSQL_ROOT_PASSWORD=ch2demo mysql
```

- Create a different WordPress container:

```
docker run -d --name wp2 --link  wpdb:mysql -p 80 \
     --read-only wordpress:4
docker inspect --format "{{.State.Running}}" wp2
docker logs wp2
```

- Fails again!
  - **Why?**
    - WordPress' Apache Web server cannot create a lock file to a specific location (part of Apache's standard config).
  - **Why it cannot create it?**
    - The container's f/s is **read-only**.
  - **What can you do?**
    - Find which part of the f/s should be made writeable
    - Create an exception to the read-only f/s.

```
docker run -d --name wp_writable wordpress
```

- Check where wordpress changes the container's filesystem:

```
docker container diff wp_writable
```

- Command reports:
  - `C /run`
  - `C /run/apache2`
  - `A /run/apache2/apache2.pid`
- Specify an exception to the read-only file system, using docker "volumes":

```
docker run -d --name wp3 \
   --link wpdb:mysql \
   -p 8000:80 \
   --read-only
   -v /run/apache2/ \
   --tmpfs /tmp \
   --read-only wordpress
```

Web site monitoring

Problems with previous approach:

- The database is running in a container on the same computer as the WordPress container.
- WordPress is using several default values for important settings like database name, administrative user, administrative password, database salt, and so on.
- To deal with this problem, you could create several versions of the WordPress software, each with a special configuration for each client.
- Doing so would turn your simple provisioning script into a monster that creates images and writes files.
- These problems can be simplified by:
  - Using environment variables
  - Running the database on a different computer; specify its hostname with an environment variable

University of Cyprus
Department of Computer Science

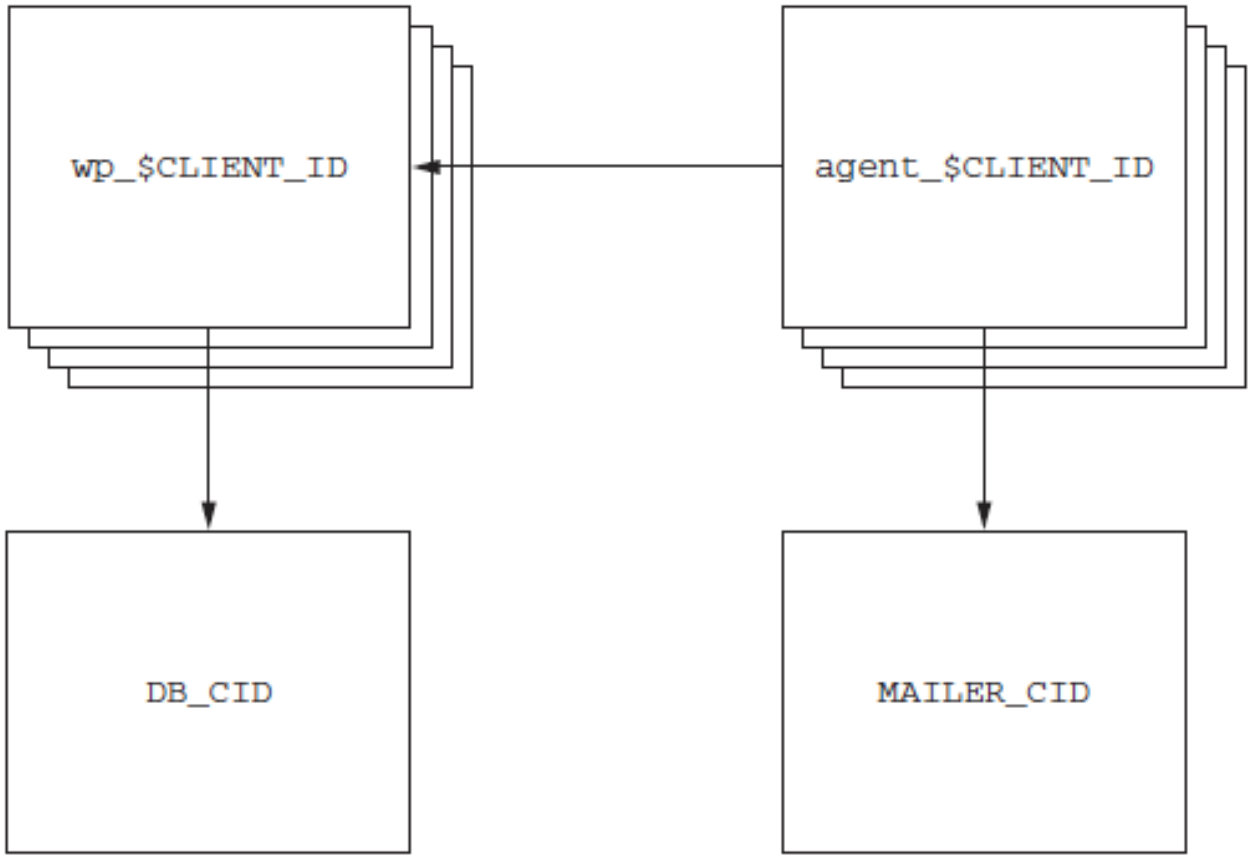
Environment variable injection

- Environment variables: **key-value pairs** that are made available to programs through their execution context.
  - They let you change a program's configuration without modifying any files or changing the command used to start the program.
- Docker uses environment variables to communicate information about:
  - dependent containers
  - the container's host name, and
  - other convenient information for programs running in containers.
- Docker provides mechanism to **inject** environment variables into a new container.
- Programs that know to expect important information through environment variables can be configured at container-creation time.

```
docker run --env MY_ENVIRONMENT_VAR="this is a test" busybox:latest env
```

Web site monitoring

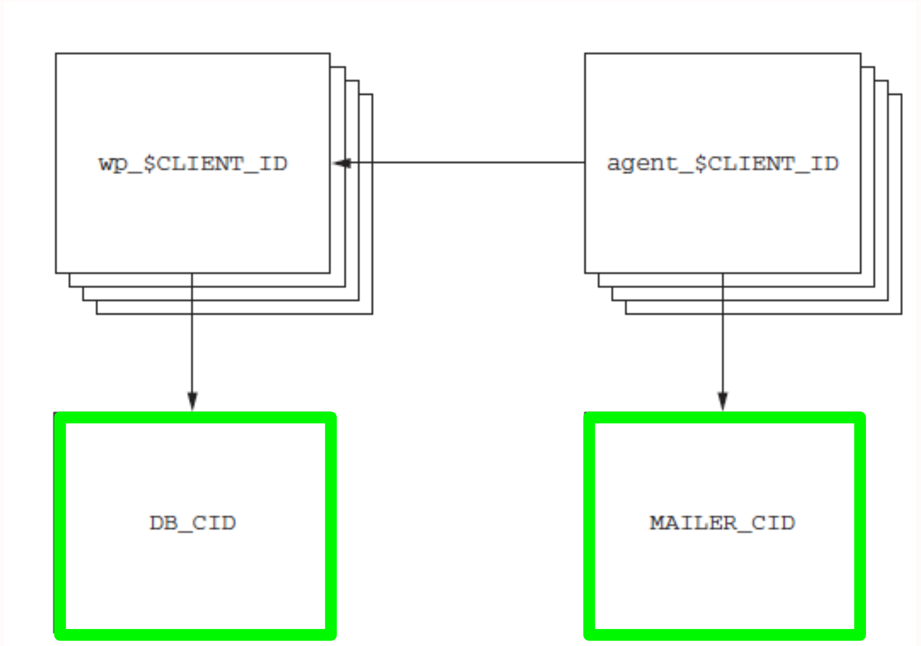
- We want to support multiple WordPress installations, using a common monitoring infrastructure, and a single database server with multiple hosted databases



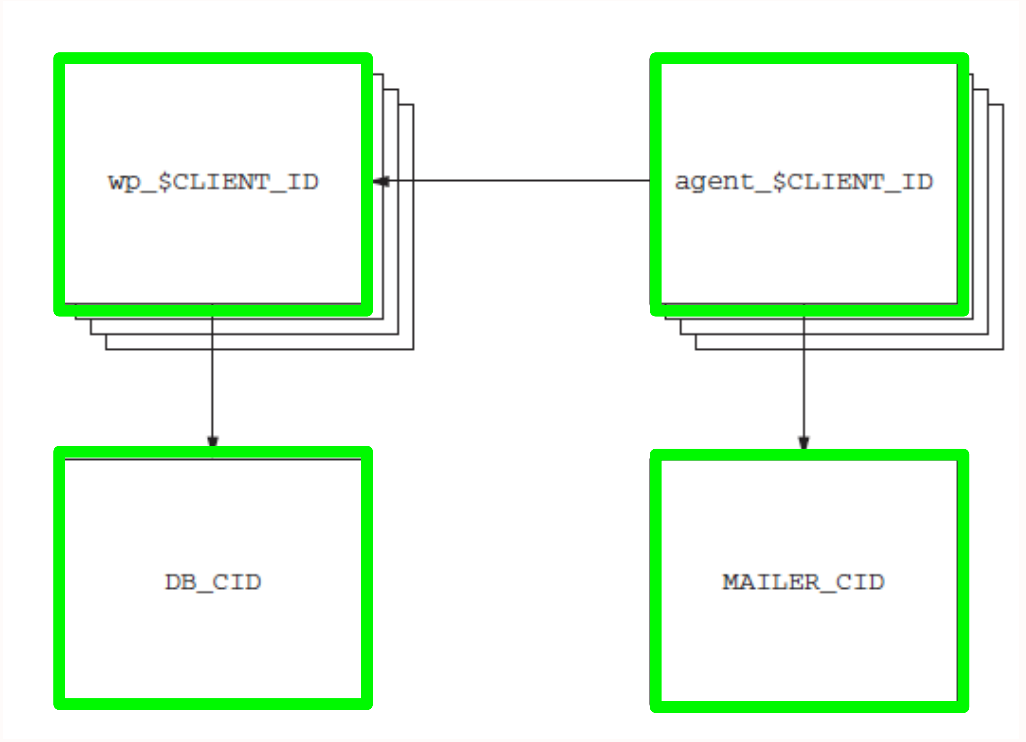
University of Cyprus
Department of Computer Science

Setup the database and the mailer that will be shared by the “clients” (Wordpress installations)

```
export DB_CID=$(docker run -d
    -e MYSQL_ROOT_PASSWORD=ch2demo
    mysql)
export MAILER_CID=$(docker run -d
    dockerinaction/ch2_mailer)
```



- Create and run a client site provisioning script, which:
  - Reads its client ID from an env variable
  - Reads the db and mailer container IDs
  - Launches the Wordpress container
  - Launches that container's monitoring agent



- Read client ID from an env variable
- Reads the db and mailer container IDs
- Launches the Wordpress container
- Launches that container's monitoring agent

```
#!/bin/sh

if [ ! -n "$CLIENT_ID" ]; then
        echo "Client ID not set"
        exit 1
fi

WP_CID=$(docker create \
      --link $DB_CID:mysql \
      --name wp_$CLIENT_ID \
      -p 8000:80\
      --read-only -v /run/apache2/ --tmpfs /tmp \
      -e WORDPRESS_DB_NAME=$CLIENT_ID \
      --read-only wordpress:5.0.0-php7.2-apache)

docker start $WP_CID

AGENT_CID=$(docker create \
   --name agent_$CLIENT_ID \
   --link $WP_CID:insideweb \
   --link $MAILER_CID:insidemailer \
  dockerinaction/ch2_agent)

docker start $AGENT_CID
```

Building Durable Containers

- Docker provides restart policies to help deal with failures: exponential backoff strategy for timing restart attempts

```
docker run -d --name backoff-detector --restart always
                    busybox date
```

- However, during backoff periods, the container isn't running.
- That means you can't do anything that requires the container to be in a running state, like execute additional commands in the container.
- To address this issue, you can adjust the supervisor process inside your container so that it deals with failures and restarts the way you want.

Supervisor Process

- A supervisor process, or init process, is a program that's used to launch and maintain the state of other programs.
  - On a Linux system, **PID #1** is an init process. It starts all the other system processes and restarts them in the event that they fail unexpectedly.
- Common practice to use a similar pattern inside containers to start and manage processes.
- Using a supervisor process inside your container will keep the container running in the event that the target process—a web server, for example—fails and is restarted.
- Popular supervisor programs for containers: **`init`**, `systemd`, `runit`, `upstart`, and `supervisord`.
- You can check the existence of this, as follows. First, run the lamp-test container:

```
docker run -d -p 80:80 --name lamp-test tutum/lamp
```

- Then, run the command below to kill the program inside the **lamp-test** container and tell the `apache2` process to shut down.

```
docker exec lamp-test ps

docker exec lamp-test kill <PID>
```

- When `apache2` stops, the `supervisord` process will log the event and restart the process.

Startup Scripts and Entrypoints

- A common alternative to just using `init` or `supervisor` programs:
  - Checking preconditions for successfully starting the contained software.
  - Sometimes used as the default command for the container.
- Docker containers run a **command** or **script** called an entrypoint before executing the default command:
  - Ideal place to put code that validates the preconditions of a container.
  - Docker allows to override or specifically set the entrypoint of a container on the command line.

See the entry point for the WordPress container

- Override the default command and use a command to view the contents of the startup script:

```
docker run wordpress \
  cat /usr/local/bin/docker-entrypoint.sh
```

- Define "cat" as the entrypoint and pass its location as argument to cat:
  - `docker run --entrypoint="cat" wordpress /usr/local/bin/docker-entrypoint.sh`

University of Cyprus
Department of Computer Science

- What is the output of:

```
docker run dockerinaction/hello_world
```

- What is the output of:

```
docker run –entrypoint “whoami”
       dockerinaction/hello_world
```

- What is the output of:

```
docker run dockerinaction/hello_world
       whoami
```

- What is the output of:

```
docker run  dockerinaction/hello_world ls
```

University of Cyprus
Department of Computer Science

Clean-up

- The isolation provided by containers simplifies the tasks of stopping processes and removing files.
- With Docker, you must first identify the container that you want to stop and/or remove and use the `docker rm` command. For example, to delete the stopped container named `wp` you'd run: `docker rm wp`
- The processes running in a container should be **stopped before the files in the container are removed** with the **`docker stop`** command or by using the **`-f`** flag on **`docker rm`**.
- You can avoid the cleanup burden by specifying `--rm` on the **`docker run`** command. Doing so will **automatically remove the container as soon as it enters the exited state**

Docker Overview

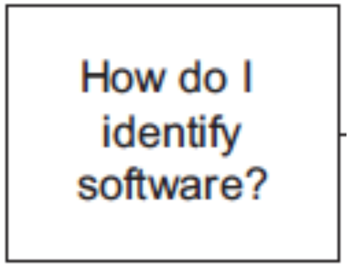
Software Installation & Images

WHAT DO YOU NEED TO DO TO INSTALL AND RUN SOFTWARE OF A CONTAINER?

University of Cyprus
Department of Computer Science

Software Installation Simplified

- Software is distributed using images.
- Need to tell Docker exactly which image to install and launch a container with it.
- Steps:
  1. Identify the software you want to install:
     - Name the program
     - Specify its version
     - Specify the source you want to install it from
  2. Discover the repository where the identified software image is located.
  3. Download the image containing the software required, install, built and run them isolated from other files in your system.

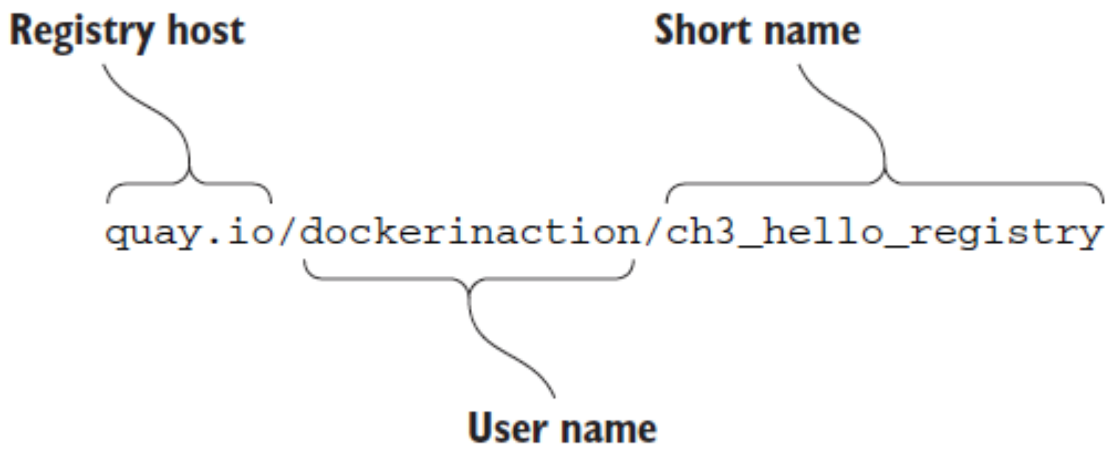


Software Installation & Images

Docker Repositories

Named Repositories

- A named repository is a named bucket of images (names are similar to URLs).
- Naming syntax:
  - name of the host where the image is located
  - the user account that owns the image
  - a short name.
- A repository can hold se image identified unique identify uniquely an imc aliases, etc.



Locating Repositories

- There are several public Docker indexes, where you can search for software images.
- Docker Hub is the default Docker registry: it is a registry and index with a website run by Docker Inc.

  You can find software on Docker Hub through either its **website** or the **docker command-line program**, e.g.:

  ```
  docker search Postgres
  ```

- To ensure that Docker is an open ecosystem, Docker Inc.:
  - Provides a public image to **run your own registry**, and
  - Allows to easily configure the docker command-line tool to **use alternative registries.**

Docker Registries

- The **Docker Hub** website allows registered users to start a repository and publish their images on Docker Hub. Typical approaches:
  - Use the command line to push images independently and on own system. Images pushed this way are considered to be less trustworthy: not clear how exactly they were built.
  - Make a Dockerfile publicly available and use Docker Hub's continuous build system.
    - **Dockerfiles** are **scripts for building images**.
    - Images created from dockerfiles are preferred because the Dockerfile is available for examination prior to installing the image.
- Working with private Docker Hub registries or pushing into registries that you control on Docker Hub requires that you authenticate.
- Closed source or proprietary projects may not want to risk publishing their software through a third party. There are three other ways to install software:
  - Use alternative repository registries or run your own registry:

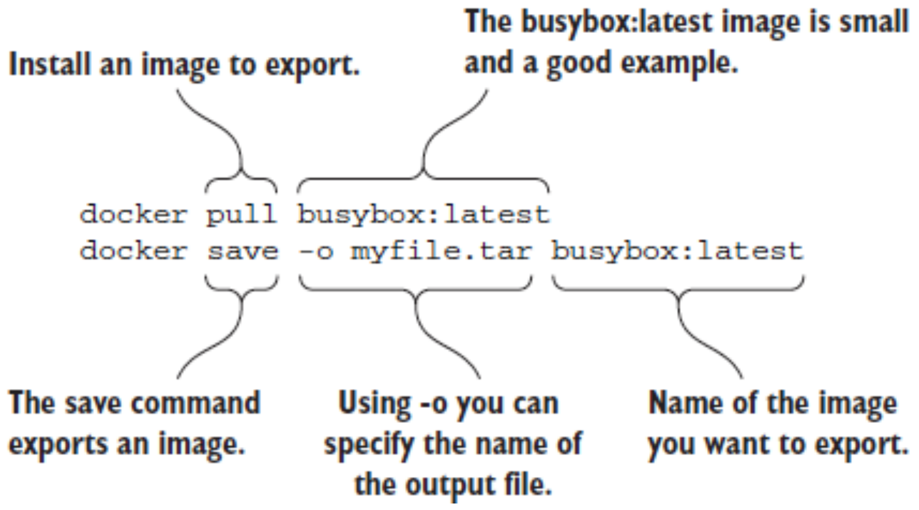
```
docker pull quay.io/dockerinaction/ch3_hello_registry:latest
```

  - You can manually load images from a file.
  - You can download a project from some other source and build an image using a provided Dockerfile.

```
docker rmi quay.io/dockerinaction/ch3_hello_registry
```

Working with Images as Files

- Docker provides the `docker load` command to load images into Docker from a file, and `docker save` to save an image to a file.
  - So, you can load images that you acquired through other channels.


Install an image to export.
The busybox:latest image is small and a good example.
The save command exports an image.
Using -o you can specify the name of the output file.
Name of the image you want to export.

```
docker pull busybox:latest
docker save -o myfile.tar busybox:latest
```

```
docker rmi busybox
```

```
docker load -i myfile.tar
```

Installing from a Dockerfile

- A dockerfile script describes the steps for Docker to take to build a new image.
  - Dockerfiles can be distributed along with software that the author wants to be put into an image.
- E.g.:

```
git clone https://github.com/dockerinaction/ch3_dockerfile.git
docker build -t dia_ch3/dockerfile:latest ch3_dockerfile
```

github.com/dockerinaction/ch3_dockerfile.git:

```
FROM busybox:latest
MAINTAINER dia@allingeek.com
ADD demo.sh /demo/
WORKDIR /demo/
CMD ./demo.sh
```

Installation Files and Isolation

Images and Image Hierarchies

- Most of the time, an **image** is actually a collection of image layers.
- A **layer is an image** that's related to at least one other image:
  - Images are usually related to other images in parent/child relationships.
- Installing an image means installing:
  - a target image and
  - each image layer in its lineage.

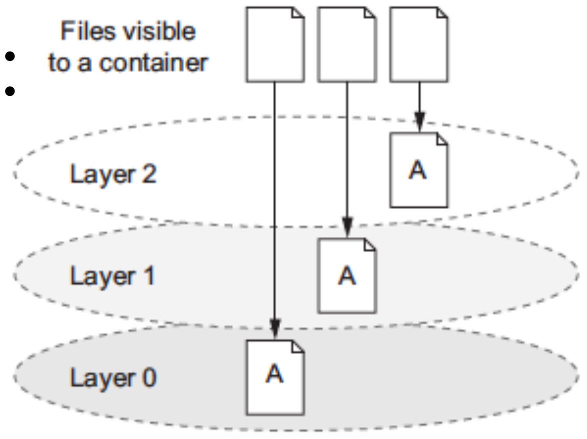
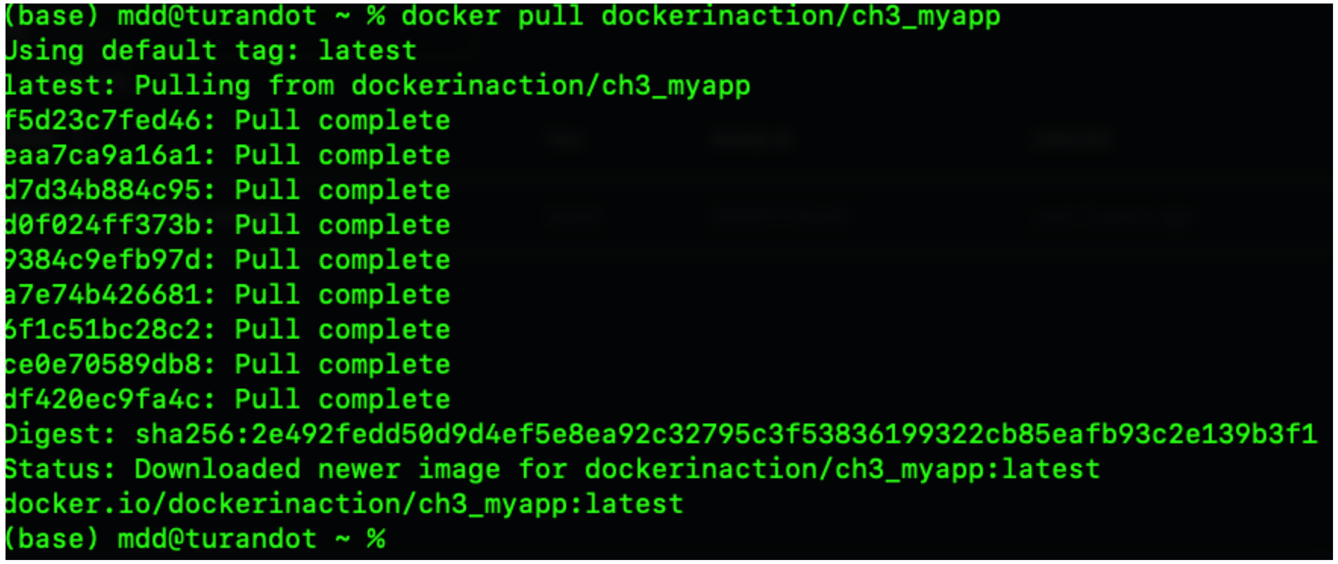


Figure 7.3 Reading files that are located on different layers

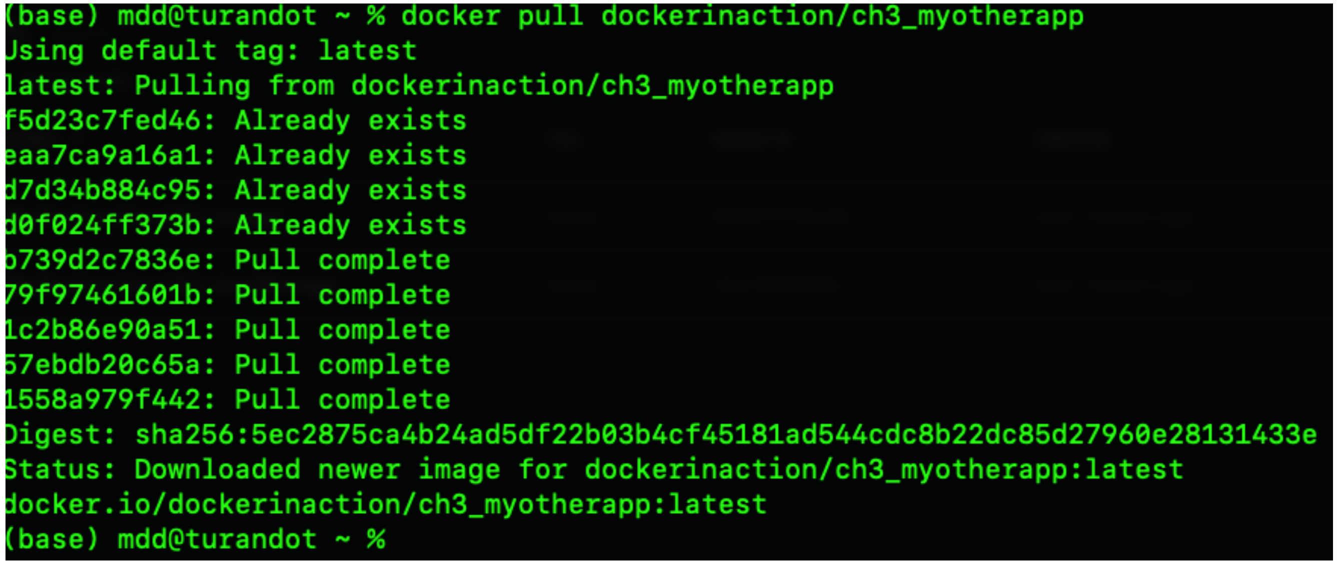
Images in Action

Install an image ch3_myapp (but don’t run it)



```
(base) mdd@turandot ~ % docker pull dockerinaction/ch3_myapp
Using default tag: latest
latest: Pulling from dockerinaction/ch3_myapp
f5d23c7fed46: Pull complete
eaa7ca9a16a1: Pull complete
d7d34b884c95: Pull complete
d0f024ff373b: Pull complete
9384c9efb97d: Pull complete
a7e74b426681: Pull complete
6f1c51bc28c2: Pull complete
ce0e70589db8: Pull complete
df420ec9fa4c: Pull complete
Digest: sha256:2e492fedd50d9d4ef5e8ea92c32795c3f53836199322cb85eafb93c2e139b3f1
Status: Downloaded newer image for dockerinaction/ch3_myapp:latest
docker.io/dockerinaction/ch3_myapp:latest
(base) mdd@turandot ~ %
```

Then, install another image ch3_myotherapp

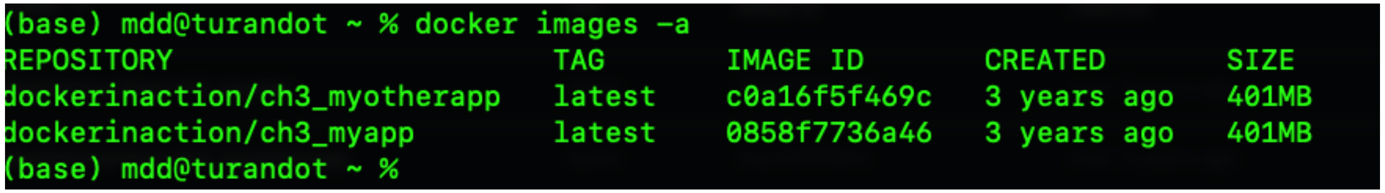


```
(base) mdd@turandot ~ % docker pull dockerinaction/ch3_myotherapp
Using default tag: latest
latest: Pulling from dockerinaction/ch3_myotherapp
f5d23c7fed46: Already exists
eaa7ca9a16a1: Already exists
d7d34b884c95: Already exists
d0f024ff373b: Already exists
b739d2c7836e: Pull complete
79f97461601b: Pull complete
1c2b86e90a51: Pull complete
57ebdb20c65a: Pull complete
1558a979f442: Pull complete
Digest: sha256:5ec2875ca4b24ad5df22b03b4cf45181ad544cdc8b22dc85d27960e28131433e
Status: Downloaded newer image for dockerinaction/ch3_myotherapp:latest
docker.io/dockerinaction/ch3_myotherapp:latest
(base) mdd@turandot ~ %
```

Layers already installed are not downloaded again

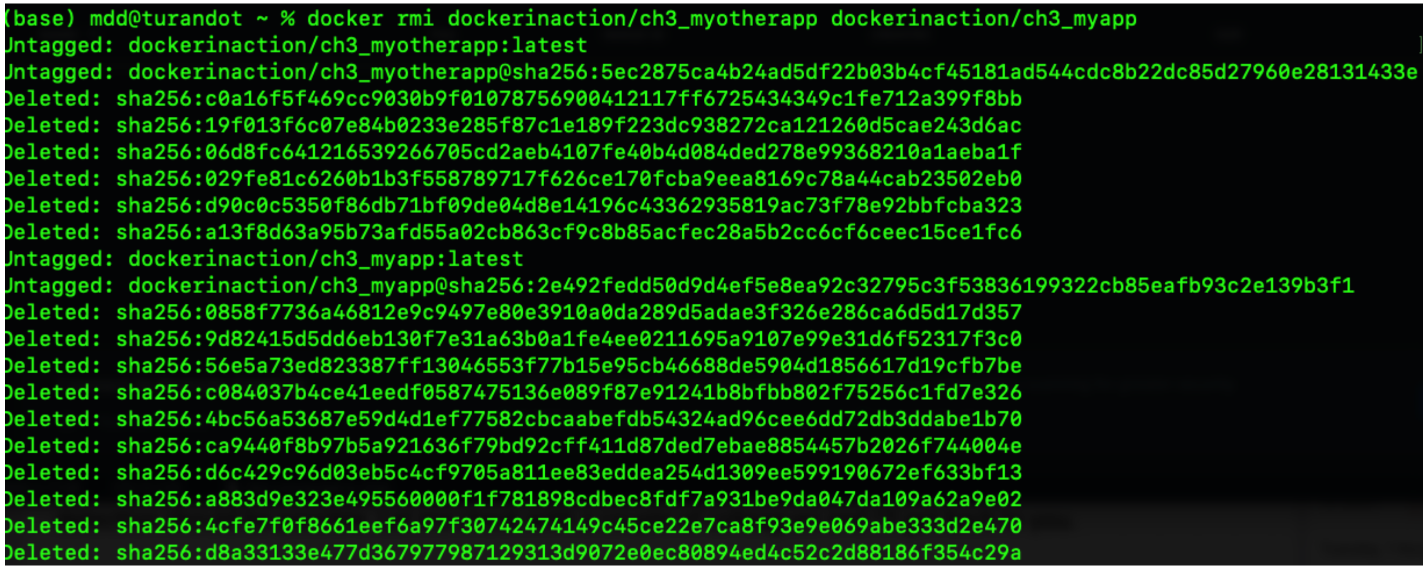
Images in Action

See which images you have installed:



```
(base) mdd@turandot ~ % docker images -a
REPOSITORY                        TAG       IMAGE ID       CREATED       SIZE
dockerinaction/ch3_myotherapp     latest    c0a16f5f469c   3 years ago   401MB
dockerinaction/ch3_myapp          latest    0858f7736a46   3 years ago   401MB
(base) mdd@turandot ~ %
```

Remove the images installed:



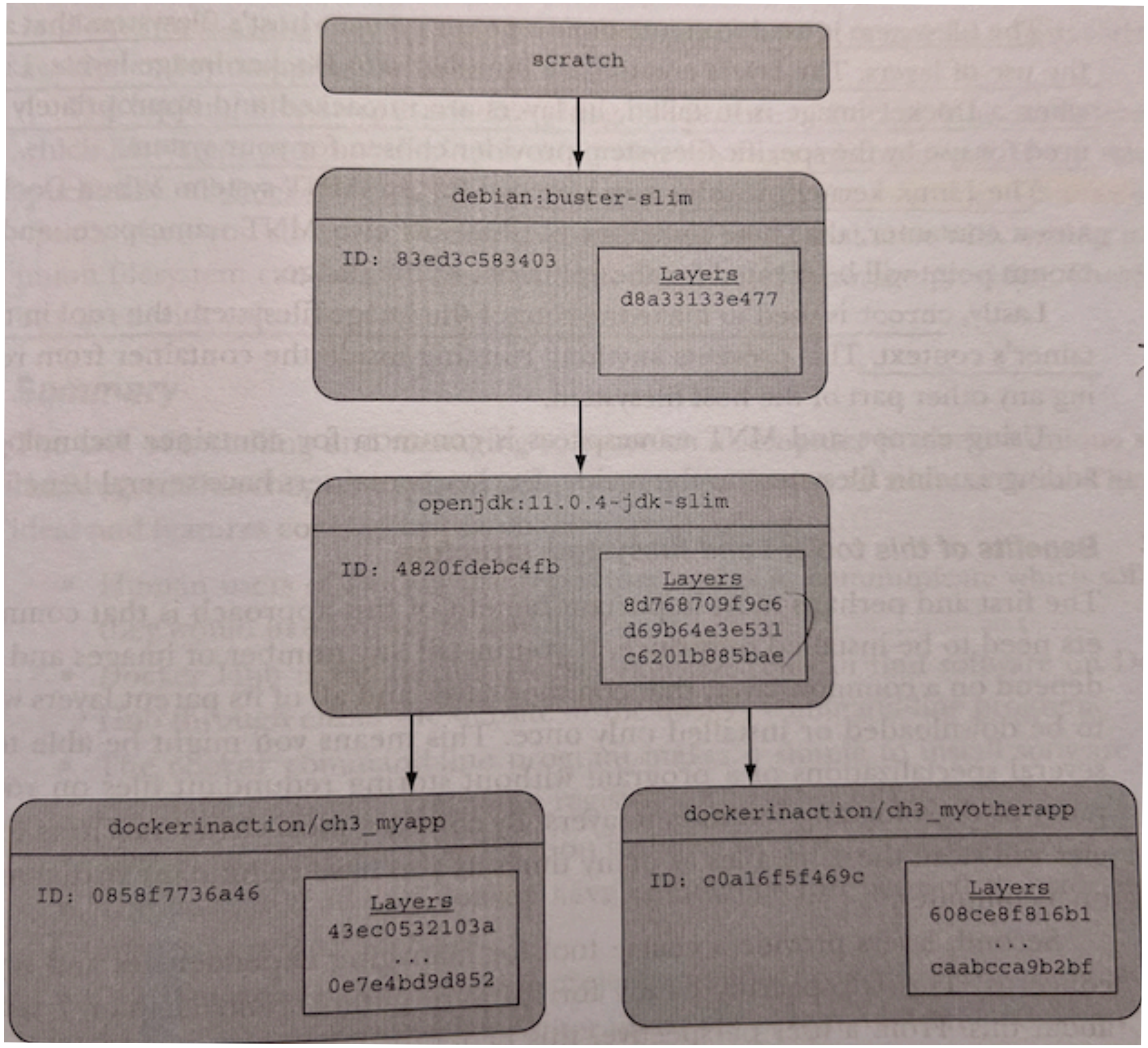
```
(base) mdd@turandot ~ % docker rmi dockerinaction/ch3_myotherapp dockerinaction/ch3_myapp
Untagged: dockerinaction/ch3_myotherapp:latest
Untagged: dockerinaction/ch3_myotherapp@sha256:5ec2875ca4b24ad5df22b03b4cf45181ad544cdc8b22dc85d27960e28131433e
Deleted: sha256:c0a16f5f469cc9030b9f01078756900412117ff6725434349c1fe712a399f8bb
Deleted: sha256:19f013f6c07e84b0233e285f87c1e189f223dc938272ca121260d5cae243d6ac
Deleted: sha256:06d8fc641216539266705cd2aeb4107fe40b4d084ded278e99368210a1aeba1f
Deleted: sha256:029fe81c6260b1b3f558789717f626ce170fcba9eea8169c78a44cab23502eb0
Deleted: sha256:d90c0c5350f86db71bf09de04d8e14196c43362935819ac73f78e92bbfcba323
Deleted: sha256:a13f8d63a95b73afd55a02cb863cf9c8b85acfec28a5b2cc6cf6ceec15ce1fc6
Untagged: dockerinaction/ch3_myapp:latest
Untagged: dockerinaction/ch3_myapp@sha256:2e492fedd50d9d4ef5e8ea92c32795c3f53836199322cb85eafb93c2e139b3f1
Deleted: sha256:0858f7736a46812e9c9497e80e3910a0da289d5adae3f326e286ca6d5d17d357
Deleted: sha256:9d82415d5dd6eb130f7e31a63b0a1fe4ee0211695a9107e99e31d6f52317f3c0
Deleted: sha256:56e5a73ed823387ff13046553f77b15e95cb46688de5904d1856617d19cfb7be
Deleted: sha256:c084037b4ce41eedf0587475136e089f87e91241b8bfbb802f75256c1fd7e326
Deleted: sha256:4bc56a53687e59d4d1ef77582cbcaabefdb54324ad96cee6dd72db3ddabe1b70
Deleted: sha256:ca9440f8b97b5a921636f79bd92cff411d87ded7ebae8854457b2026f744004e
Deleted: sha256:d6c429c96d03eb5c4cf9705a811ee83eddea254d1309ee599190672ef633bf13
Deleted: sha256:a883d9e323e495560000f1f781898cdbec8fdf7a931be9da047da109a62a9e02
Deleted: sha256:4cfe7f0f8661eef6a97f30742474149c45ce22e7ca8f93e9e069abe333d2e470
Deleted: sha256:d8a33133e477d367977987129313d9072e0ec80894ed4c52c2d88186f354c29a
```

Layers

- Layer (or intermediate image): a set of files and file metadata that is **packaged** and **distributed** as an atomic unit.
  - Internally, Docker **treats each layer like an image**.
  - A layer **can be promoted to an image** if it is tagged.
- Most layers are built upon a parent layer by applying filesystem changes to the parent (software updates, installations).
  - The resulting layer contains the combined set of files from the parent and the layer added.

Image Hierarchies

- **Images** maintain parent/child relationships:
  - In these relationships they build from their parents and form layers.
  - Images can have relationships with any other image, including images in different repositories with different owners.
- The **files available to a container** are the union of all of the layers in the lineage of the image the container was created from.
- An image is **named** when its author tags using the `docker tag` command and publishes it.
  - Until an image is tagged, the only way to refer to it is to use its 65 (base 16) digit unique identifier (UID) generated when the image was built.
    - Docker truncates the UID from to 65 to12 digits for the benefit of its human users.
    - Internally and through API access, Docker uses the full 65.


scratch
debian:buster-slim
ID: 83ed3c583403
Layers
d8a33133e477
openjdk:11.0.4-jdk-slim
ID: 4820fdebc4fb
Layers
8d768709f9c6
d69b64e3e531
c6201b885bae
dockerinaction/ch3_myapp
ID: 0858f7736a46
Layers
43ec0532103a
...
0e7e4bd9d852
dockerinaction/ch3_myotherapp
ID: c0a16f5f469c
Layers
608ce8f816b1
...
caabcca9b2bf

Images and Isolation

- A container image encapsulates almost all of an application's dependencies into a package that can be deployed into the container:
  - The only local external dependencies are on the Linux kernel system-call interface.
- Container images isolate applications from the heterogeneous OS on which they run.
- Containers abstract away from the application developer and the deployment infrastructure, many OS and machine details.

Images and Isolation

- **Programs** running inside containers know nothing about image layers.
  - From inside a container, the filesystem operates as though it's not running in a container or operating on an image.
  - From its perspective, the container has exclusive copies of the files provided by the image.
  - This is made possible with something called a **union file-system** (UFS).
- Docker uses a variety of union filesystems and will select the best fit for your system.
- The tools used by Docker to create effective filesystem isolation:
  - **Union filesystem**: used to create mount points on the host filesystem that abstract the use of layers.
  - **MNT namespaces**: the Linux kernel provides a namespace for the MNT system; when Docker creates a container, that new container has its own MNT namespace, and a new mount point will be created from the container to the image.
  - The **chroot system call**: used to make the root of the image filesystem the root in the container's context.

Image Hierarchies

- Benefits:
  - Layer reuse and bandwidth/storage savings.

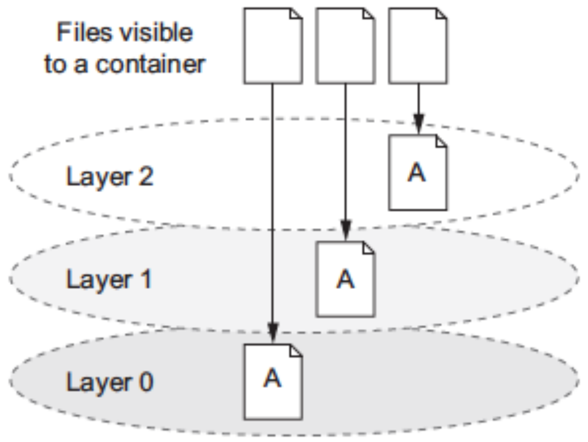
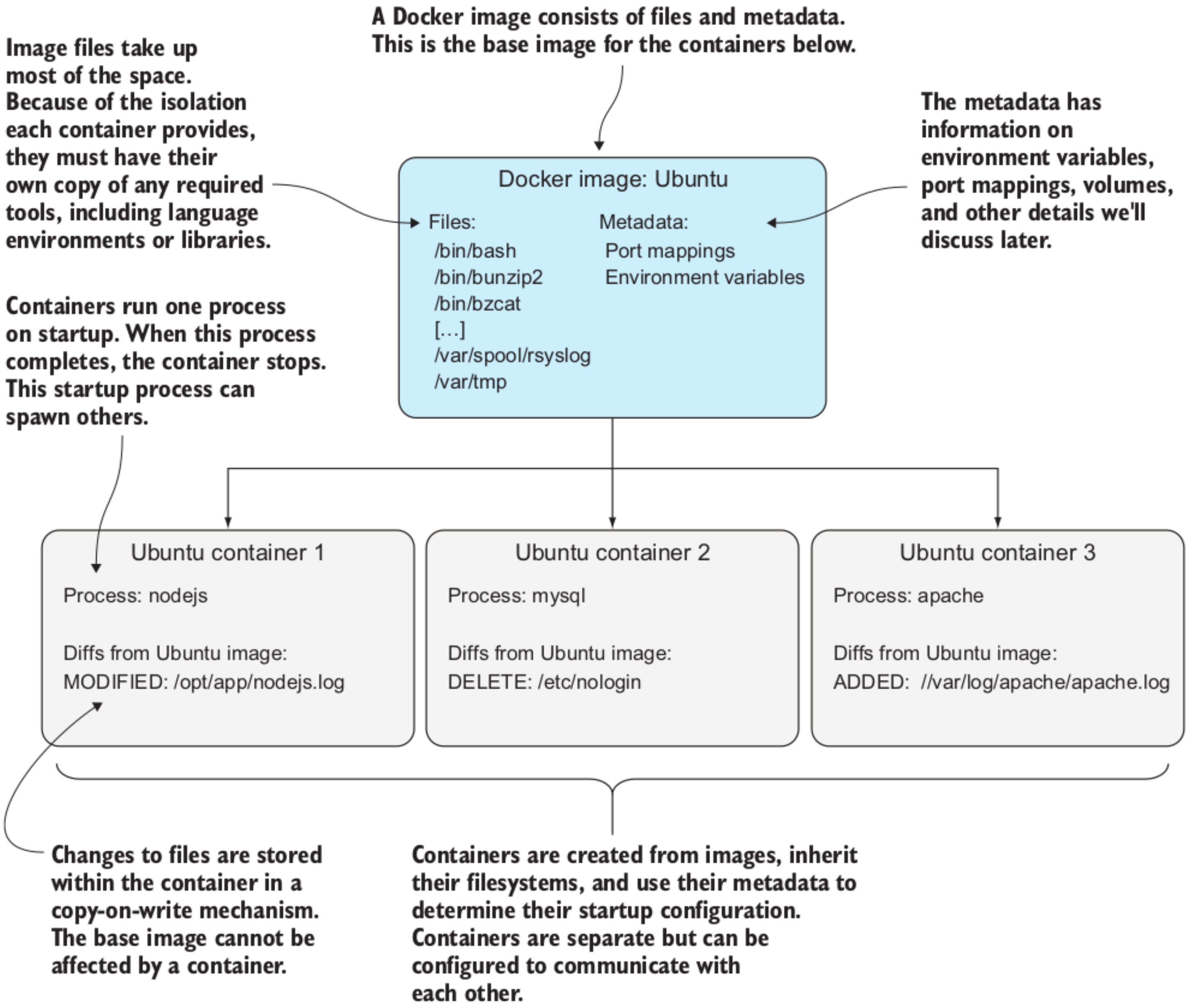


Figure 7.3 Reading files that are located on different layers

- Containers are an **OS virtualization** approach.
- Docker is the most popular solution for managing containers, introduced by Google. Docker containers were built initially on **LXC** with namespaces and cgroups. Later, they replaced LXC with libcontainer, also using Linux Kernel **namespaces**, **cgroups**, and **capabilities**.
- Containers can be run with virtual terminals attached to the user's shell or in detached mode.
- By default, every Docker container has its own **PID namespace**, isolating process information for each container.
- Docker **identifies** every container by its generated container ID, abbreviated container ID, or its human-friendly name.
- All containers are in any one of four distinct states: **running**, **paused**, **restarting**, or **exited**.
- The **docker exec** command can be used to run additional processes inside a running container.
- A user can **pass input** or provide **additional configuration** to a process in a container by specifying **environment variables** at container-creation time.
- Using the `--read-only` flag at container-creation time will mount the container file system as read-only and prevent specialization of the container.
- A container restart policy, set with the `--restart` flag at container-creation time, will help your systems automatically recover in the event of a failure.
- Docker makes cleaning up containers with the **docker rm** command as simple as creating them.

- Human Docker users use **repository names** to communicate which software they would like Docker to install.
- **Docker Hub** is the default Docker registry. You can find software on Docker Hub through either the website or the docker command-line program.
- The docker command-line program makes it simple to install software that's distributed through alternative registries or in other forms.
- The image repository specification includes a registry host field.
- The **docker load** and **docker save** commands can be used to load and save images from TAR archives.
- Distributing a **Dockerfile** with a project simplifies image builds on user machines.
- **Images** are usually related to other images in **parent/child relationships**. These relationships form **layers**. When we say that we have installed an image, we are saying that we have installed a target image and each image layer in its lineage.
- Structuring images with layers enables **layer reuse** and **saves bandwidth** during distribution and storage space on your computer.


A Docker image consists of files and metadata. This is the base image for the containers below.
Image files take up most of the space. Because of the isolation each container provides, they must have their own copy of any required tools, including language environments or libraries.
The metadata has information on environment variables, port mappings, volumes, and other details we'll discuss later.
Docker image: Ubuntu
Files:
/bin/bash
/bin/bunzip2
/bin/bzcat
[…]
/var/spool/rsyslog
/var/tmp
Metadata:
Port mappings
Environment variables
Containers run one process on startup. When this process completes, the container stops. This startup process can spawn others.
Ubuntu container 1
Process: nodejs
Diffs from Ubuntu image:
MODIFIED: /opt/app/nodejs.log
Ubuntu container 2
Process: mysql
Diffs from Ubuntu image:
DELETE: /etc/nologin
Ubuntu container 3
Process: apache
Diffs from Ubuntu image:
ADDED: //var/log/apache/apache.log
Changes to files are stored within the container in a copy-on-write mechanism. The base image cannot be affected by a container.
Containers are created from images, inherit their filesystems, and use their metadata to determine their startup configuration. Containers are separate but can be configured to communicate with each other.

Docker Overview

Storage and Volumes

SUPPOSE YOU LAUNCH A CONTAINER WITH A WEB DATABASE APPLICATION

WHEN PROGRAMS CONNECT TO THE DATABASE AND ENTER DATA, WHERE IS THAT DATA STORED? IS IT IN A FILE INSIDE THE CONTAINER?

WHAT HAPPENS TO THAT DATA WHEN YOU STOP THE CONTAINER OR REMOVE IT?

HOW WOULD YOU MOVE YOUR DATA IF YOU WANTED TO UPGRADE THE DATABASE PROGRAM?

WHERE WOULD YOU WRITE LOG FILES SO THAT THEY WILL OUTLIVE THE CONTAINER?

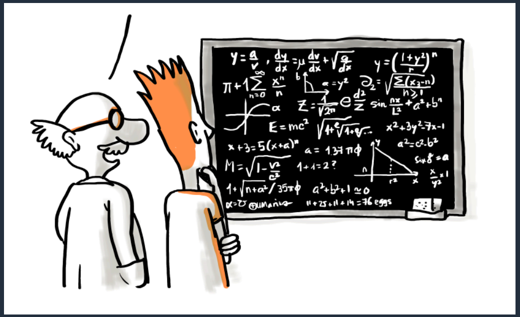
HOW WOULD YOU GET ACCESS TO THOSE LOGS TO TROUBLESHOOT A PROBLEM?

HOW CAN OTHER PROGRAMS SUCH AS LOG DIGEST TOOLS GET ACCESS TO THOSE FILES?

Union File System

- Without a storage solution, container users are limited to working with the union file system that provides image mounts.
- Although the **union file system** works for building and sharing images, it's **less than ideal** for working with **persistent** or **shared data**.

Mount points



- Linux unifies all storage into a single file system **tree**:
  - Storage devices such as disk partitions or USB disk partitions are attached to specific locations in that tree.
  - Those locations are called **mount points**.
- A **mount point** defines:
  - the location in the tree, the access properties to the data at that point (for example, writability), and
  - the source of the data mounted at that point (e.g., a specific hard disk, USB device, or memory-backed virtual disk).

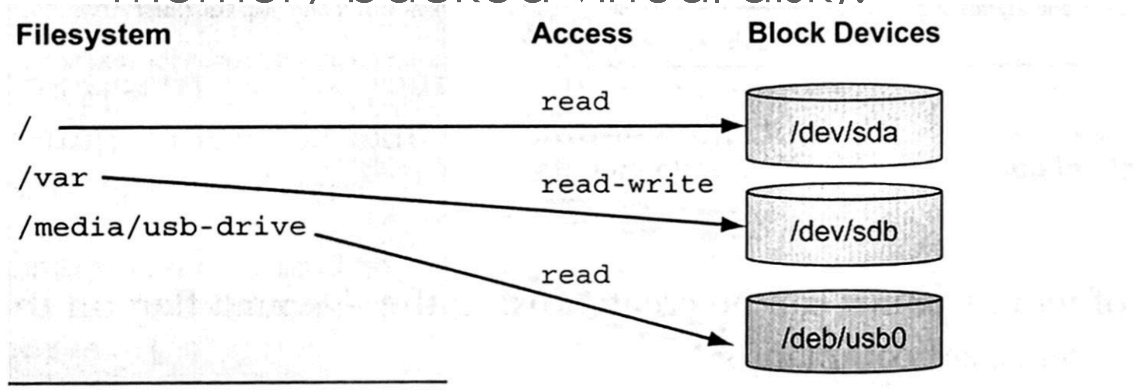


Figure 4.1 Storage devices attached to a filesystem tree at their mount point

ɔ use the
ːnowing
ecific
l in

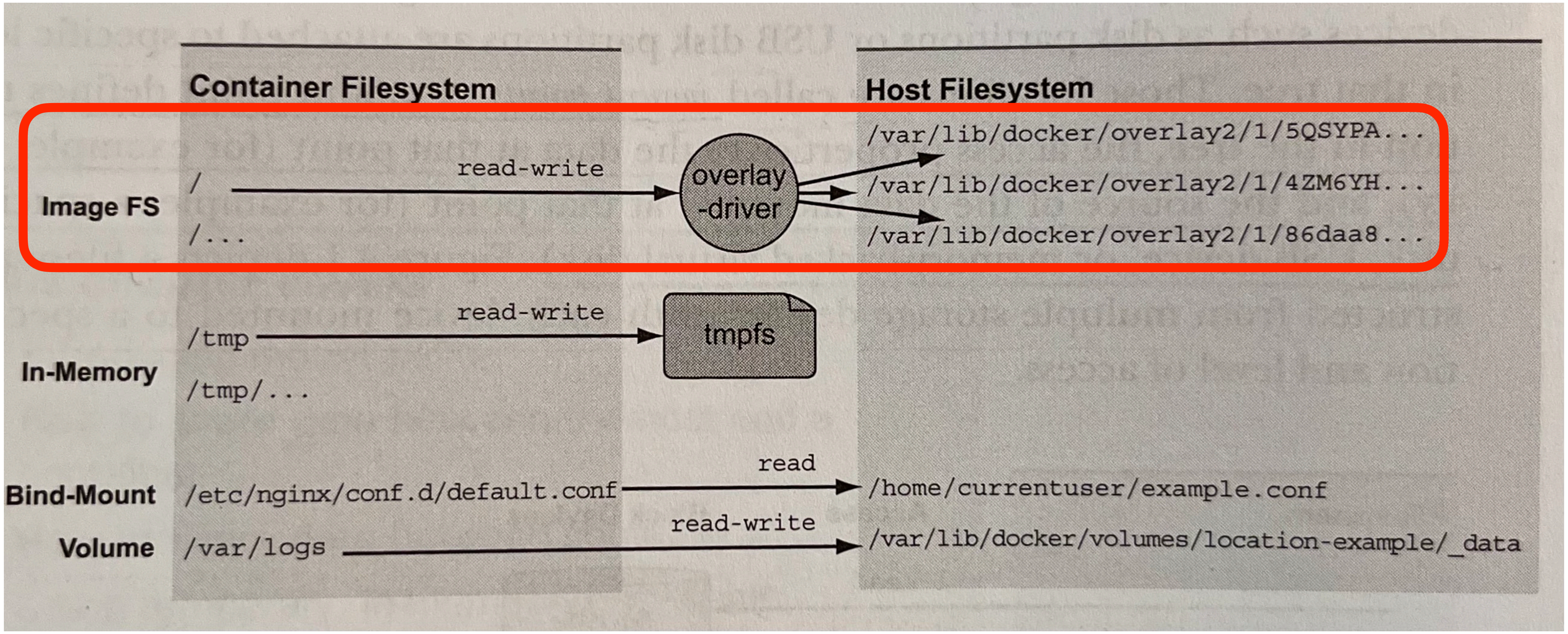
Mount Points and Containers

- Every container has something called a MNT namespace and a unique file tree root.
- The **image** that a container is created from is mounted at that container's file tree root- the / point
- Every container has a different set of mount points.
- Since different storage devices can be mounted at various points in a file tree, we can mount nonimage-related storage at other points in a container file tree. That is exactly how:
  - containers get access to storage on the host filesystem and
  - share storage between containers.

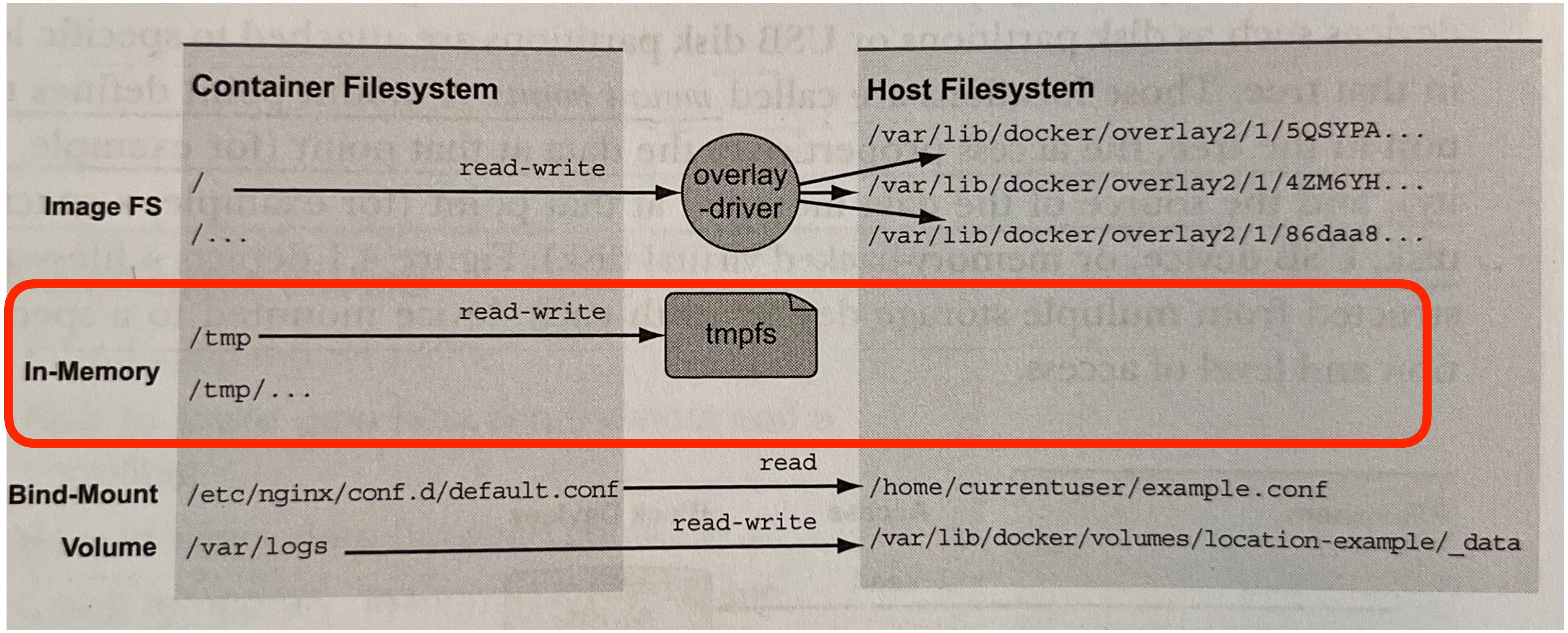
Container Storage

- There are three most common types of storage mounted into containers:
  - Bind mounts
  - In-memory storage
  - Docker volumes
- These types of mount points can be created using the `--mount` flag on the `docker run` and `docker create` subcommands.

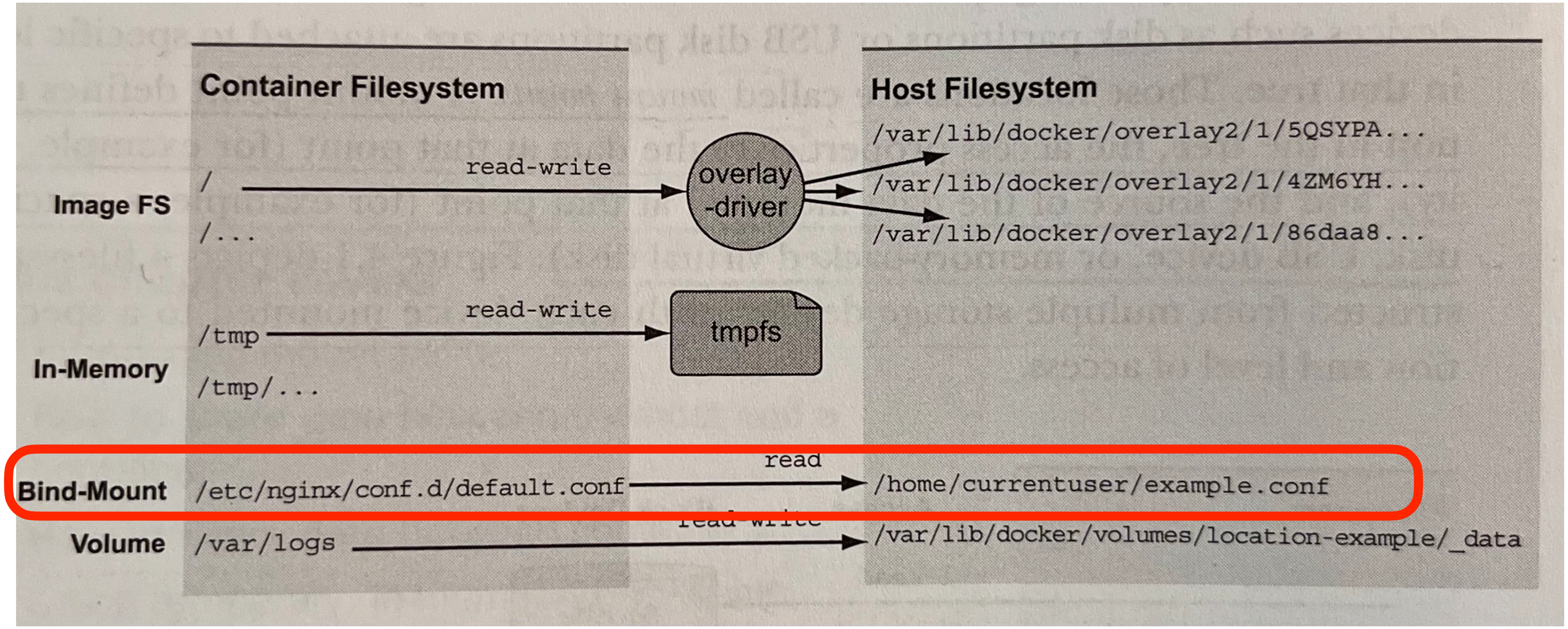
Container Storage


Container Filesystem
Host Filesystem
Image FS
/
/...
read-write
overlay -driver
/var/lib/docker/overlay2/1/5QSYPA...
/var/lib/docker/overlay2/1/4ZM6YH...
/var/lib/docker/overlay2/1/86daa8...
In-Memory
/tmp
/tmp/...
read-write
tmpfs
Bind-Mount
/etc/nginx/conf.d/default.conf
read
/home/currentuser/example.conf
Volume
/var/logs
read-write
/var/lib/docker/volumes/location-example/_data

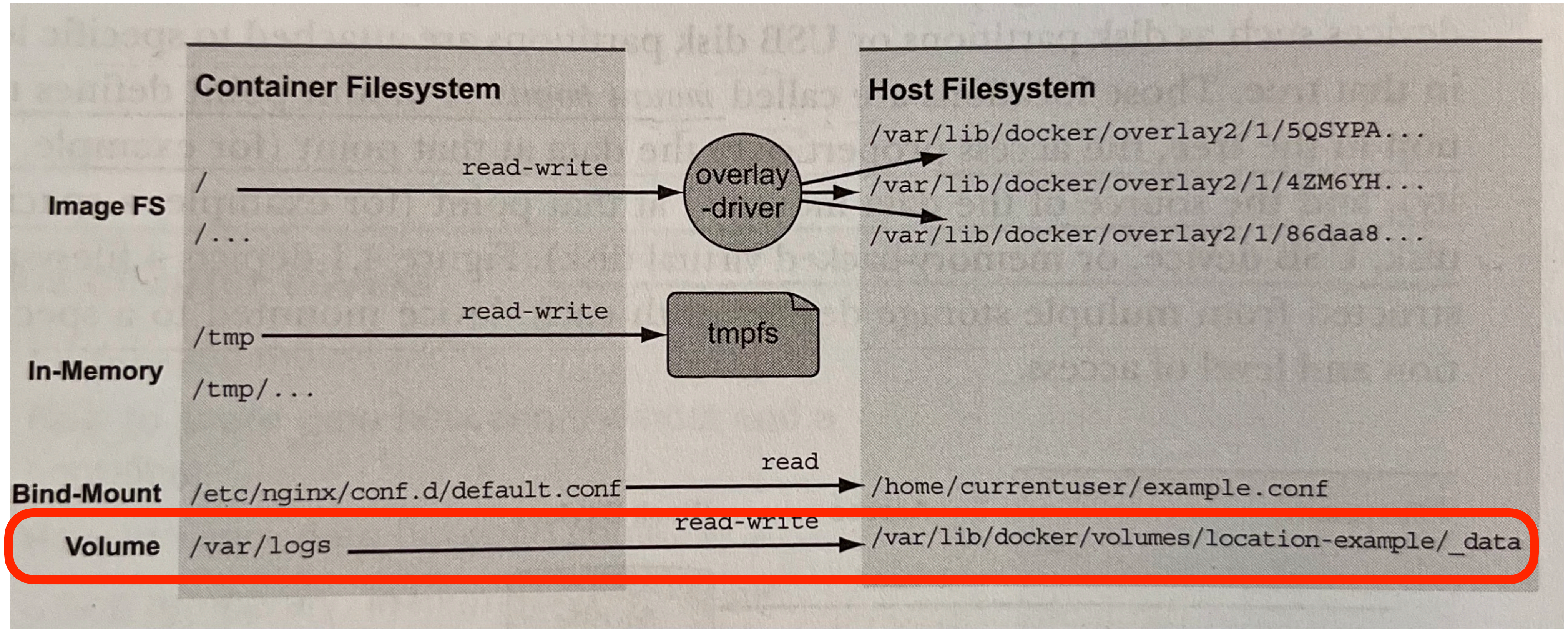
Container Storage


Container Filesystem
Host Filesystem
Image FS
/
/...
read-write
overlay -driver
/var/lib/docker/overlay2/1/5QSYPA...
/var/lib/docker/overlay2/1/4ZM6YH...
/var/lib/docker/overlay2/1/86daa8...
In-Memory
/tmp
/tmp/...
read-write
tmpfs
Bind-Mount
/etc/nginx/conf.d/default.conf
read
/home/currentuser/example.conf
Volume
/var/logs
read-write
/var/lib/docker/volumes/location-example/_data

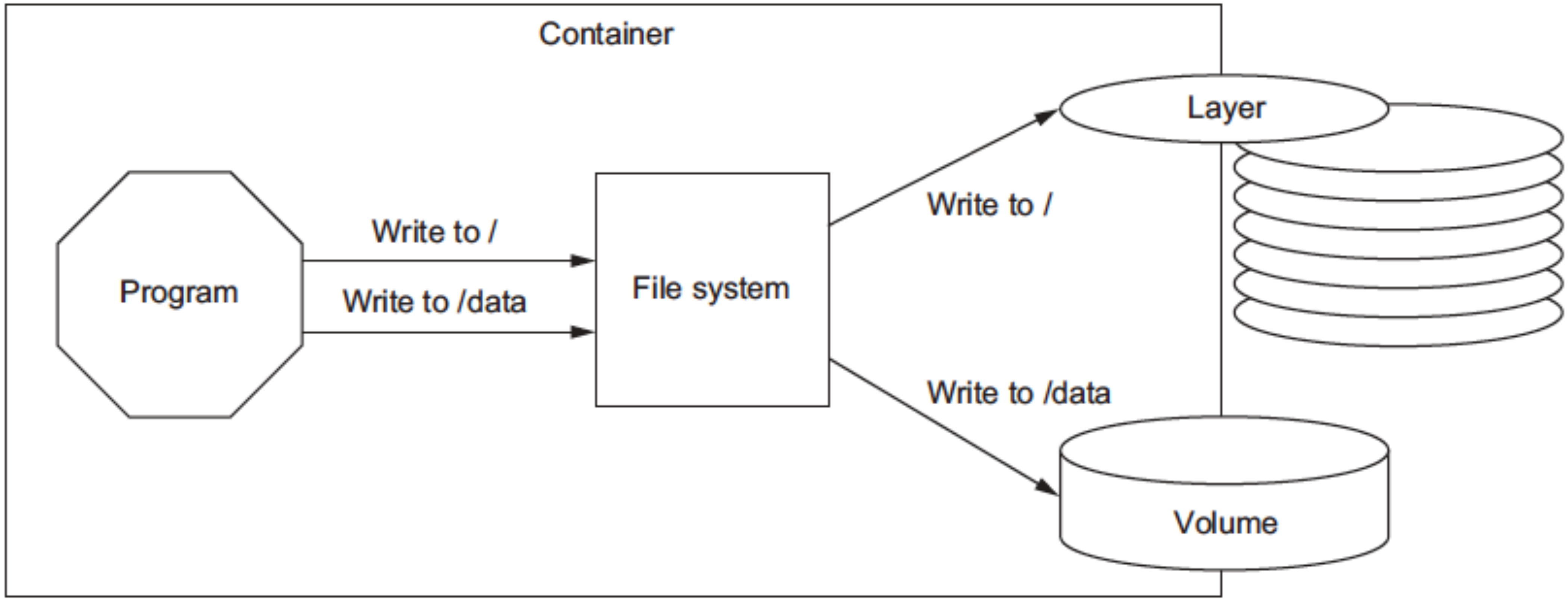
Container Storage


Container Filesystem
Host Filesystem
Image FS
/
/...
read-write
overlay -driver
/var/lib/docker/overlay2/1/5QSYPA...
/var/lib/docker/overlay2/1/4ZM6YH...
/var/lib/docker/overlay2/1/86daa8...
In-Memory
/tmp
/tmp/...
read-write
tmpfs
Bind-Mount
/etc/nginx/conf.d/default.conf
read
/home/currentuser/example.conf
Volume
/var/logs
read-write
/var/lib/docker/volumes/location-example/_data

Container Storage


Container Filesystem
Host Filesystem
Image FS
/
/...
read-write
overlay -driver
/var/lib/docker/overlay2/1/5QSYPA...
/var/lib/docker/overlay2/1/4ZM6YH...
/var/lib/docker/overlay2/1/86daa8...
In-Memory
/tmp
/tmp/...
read-write
tmpfs
Bind-Mount
/etc/nginx/conf.d/default.conf
read
/home/currentuser/example.conf
Volume
/var/logs
read-write
/var/lib/docker/volumes/location-example/_data

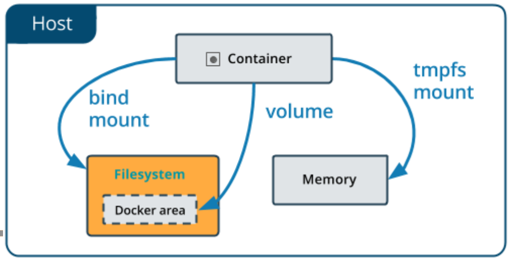
Container Storage



Images vs Volumes

- A **volume** is a tool for segmenting and sharing data that has a scope or life cycle that's independent of a single container.
- Volumes an important part of any containerized system design that shares or writes files:
  - Database software vs database data
  - Web application vs log data
  - Data processing application vs input and output data
  - Web server vs static content
- Volumes enable separation of concerns and create modularity for architectural components:
  - **Images**: appropriate for packaging/distributing relatively static files (e.g. programs)
  - **Volumes**: hold dynamic data or specializations.

Bind Mounts



- Bind mounts are mount points used to remount parts of a filesystem tree onto other locations.
- When you use a bind mount, a file or directory on the host machine (**source**) is mounted into a container, to a specific point in a container file tree (**destination**). The file or directory is referenced by its absolute path on the host machine.
- Bind mounts are useful when:
  - The host provides a file or directory that is needed by a program running in a Container
  - The containerized program produces a file or log that is processed by users or programs running outside containers.

Bind Mount Scenario

- Consider that you're running a NGINX web server that depends on sensitive configuration on the host and emits access logs that need to be forwarded by your log-shipping system.
- Use Docker to launch the web server in a container and bind-mount the locations where you want the web server to get the configuration and write the access-logs.

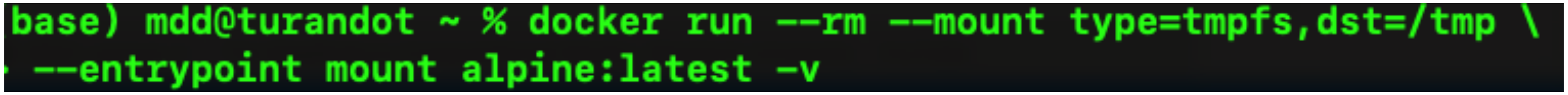
```
CONFSRC=~/example.conf
CONF_DST=/etc/nginx/conf.d/default.conf
LOGSRC=~/example.log
LOGDST=/var/1log/nginx/custom.host.access.log
docker run -d --name diaweb\
  --mount type=bind, src=${LOG_SRC},dst=${LOG DST) \
  --mount type=bind, src=$(CONF_SRC}, dst=${CONF_DST}, readonly=true \
  -p 80:80 \
nginx:latest
```

Bind Mount Cons and Pros

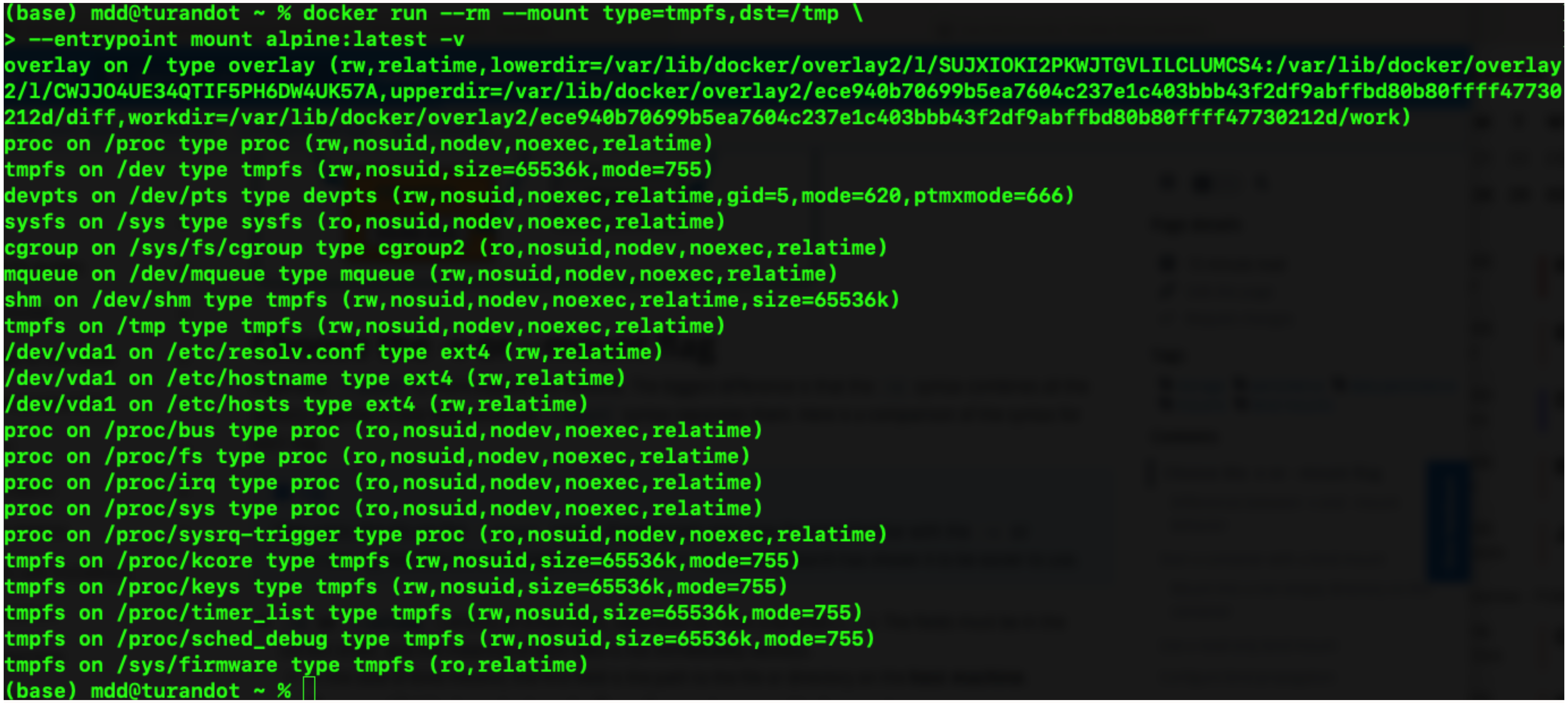
- Bind mounts tie otherwise portable container descriptions to the filesystem of a specific host.
  - ✗ If a container description depends on content at a specific location on the host file system, that description isn't portable to hosts where the content is unavailable or available in some other location.
- Bind mounts create an opportunity for conflict with other containers.
  - ✗ E.g. start multiple instances of a database that all use the same host location as a bind mount for data storage.
  - In that case, each of the instances would compete for the same set of files. Without other tools such as file locks, that would likely result in corruption of the database.
- ✓ Bind mounts are appropriate tools for workstations, machines with specialized concerns, or in systems combined with more traditional configuration management tooling.
- It's better to avoid these kinds of specific bindings in generalized platforms or hardware pools.

In-memory Storage

- Most service software and web applications use **private key** files, database **passwords**, **API key** files, or other sensitive configuration files, and need upload buffering space.
- In these cases, it is important that you never include those types of files in an image or write them to disk.
- Instead, you should use in-memory storage.
- You can add in-memory storage to containers with a special type of mount:



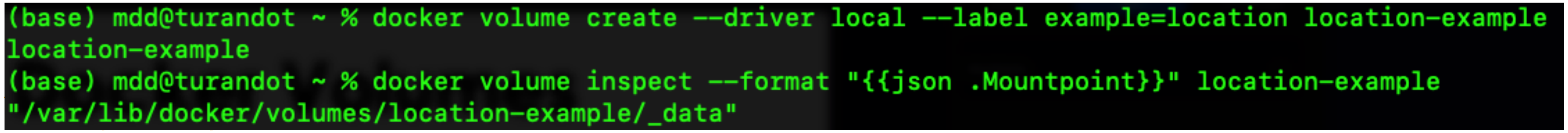
```
base) mdd@turandot ~ % docker run --rm --mount type=tmpfs,dst=/tmp \
 --entrypoint mount alpine:latest -v
```



```
(base) mdd@turandot ~ % docker run --rm --mount type=tmpfs,dst=/tmp \
> --entrypoint mount alpine:latest -v
overlay on / type overlay (rw,relatime,lowerdir=/var/lib/docker/overlay2/l/SUJXIOKI2PKWJTGVLILCLUMCS4:/var/lib/docker/overlay
2/l/CWJJO4UE34QTIF5PH6DW4UK57A,upperdir=/var/lib/docker/overlay2/ece940b70699b5ea7604c237e1c403bbb43f2df9abffbd80b80ffff47730
212d/diff,workdir=/var/lib/docker/overlay2/ece940b70699b5ea7604c237e1c403bbb43f2df9abffbd80b80ffff47730212d/work)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev type tmpfs (rw,nosuid,size=65536k,mode=755)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=666)
sysfs on /sys type sysfs (ro,nosuid,nodev,noexec,relatime)
cgroup on /sys/fs/cgroup type cgroup2 (ro,nosuid,nodev,noexec,relatime)
mqueue on /dev/mqueue type mqueue (rw,nosuid,nodev,noexec,relatime)
shm on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,size=65536k)
tmpfs on /tmp type tmpfs (rw,nosuid,nodev,noexec,relatime)
/dev/vda1 on /etc/resolv.conf type ext4 (rw,relatime)
/dev/vda1 on /etc/hostname type ext4 (rw,relatime)
/dev/vda1 on /etc/hosts type ext4 (rw,relatime)
proc on /proc/bus type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/fs type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/irq type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/sys type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/sysrq-trigger type proc (ro,nosuid,nodev,noexec,relatime)
tmpfs on /proc/kcore type tmpfs (rw,nosuid,size=65536k,mode=755)
tmpfs on /proc/keys type tmpfs (rw,nosuid,size=65536k,mode=755)
tmpfs on /proc/timer_list type tmpfs (rw,nosuid,size=65536k,mode=755)
tmpfs on /proc/sched_debug type tmpfs (rw,nosuid,size=65536k,mode=755)
tmpfs on /sys/firmware type tmpfs (ro,relatime)
(base) mdd@turandot ~ %
```

Docker Volumes

- Docker volumes are named filesystem trees managed by Docker. They can be implemented with:
  - disk **storage on the host** file system, or
  - as **cloud storage**.
- When using a volume, a new directory is created within Docker’s storage directory on the host machine, and Docker manages that directory’s contents.
- Using **volumes** decouples storage from specialized locations on the file system that you might specify with bind mounts: their life-cycle is independent of a single container’s.
- All operations on Docker volumes can be accomplished using the docker volume subcommand set: **`docker volume create`** and **`docker volume inspect`**.
- Volumes are an effective way to organize your data:
  - Using them decouples volumes from other potential concerns of the system.
  - When you're finished with a volume and you ask Docker to clean things up for you, Docker can confidently remove any directories or files that are no longer being used by a container.



```
(base) mdd@turandot ~ % docker volume create --driver local --label example=location location-example
location-example
(base) mdd@turandot ~ % docker volume inspect --format "{{json .Mountpoint}}" location-example
"/var/lib/docker/volumes/location-example/_data"
```

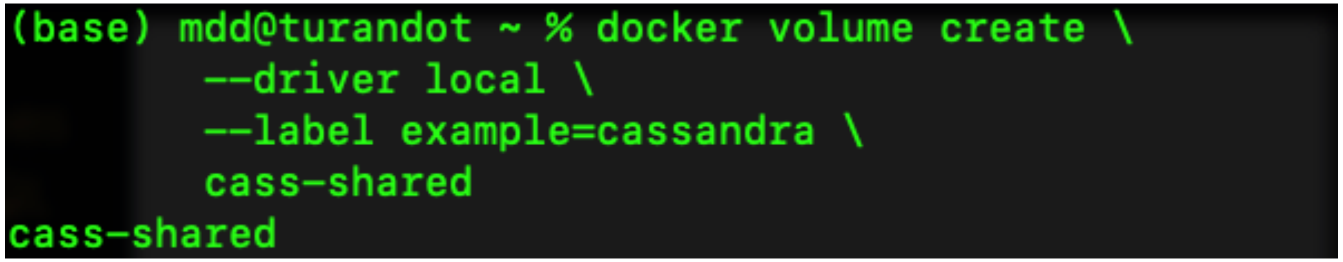
Using volumes with a NoSQL Database

Apache Cassandra is a popular, open source NoSQL distributed database offering linear scalability and fault-tolerance on commodity hardware or cloud infrastructure.

- Task: run a container with a Cassandra DB, using an external volume to store its data:
  - Create a docker volume
  - Run a container with Cassandra, linking it to the created volume
  - Run another container and connect through it to the DB, submitting instructions

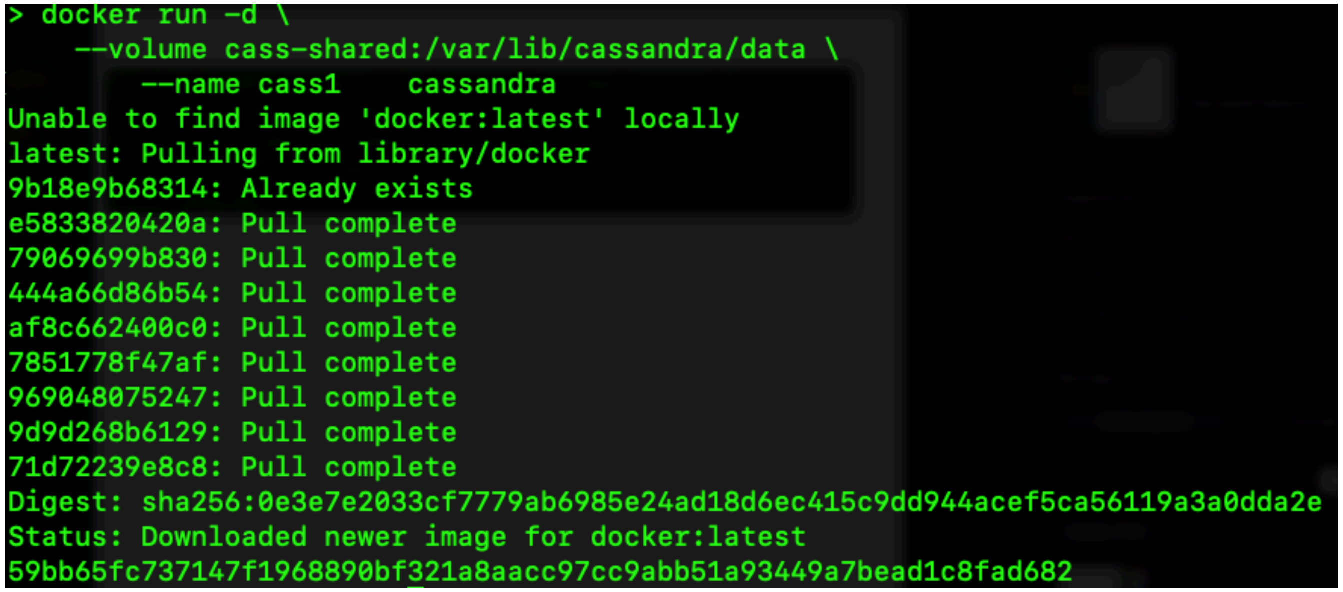
Using volumes with a NoSQL Database

Create volume on the disk space of the local machine and in a part managed by the Docker engine:



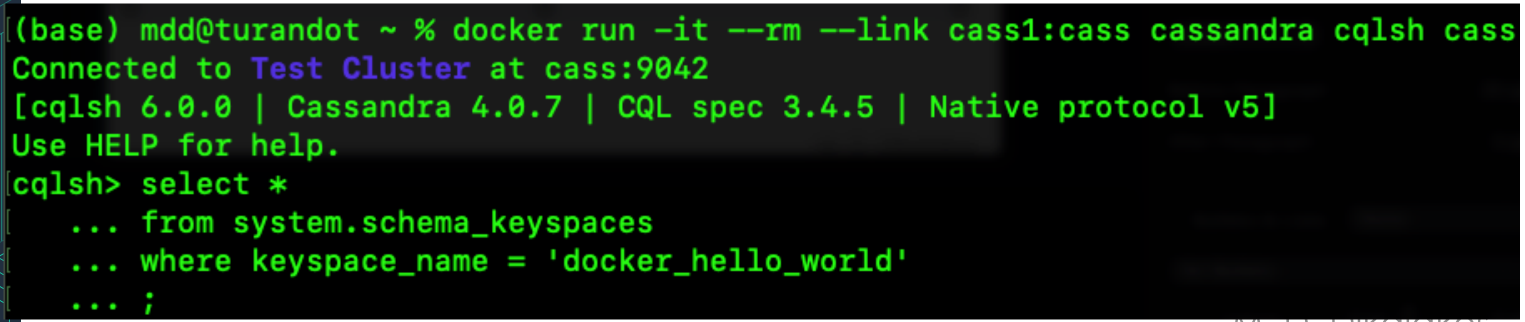
```
(base) mdd@turandot ~ % docker volume create \
        --driver local \
        --label example=cassandra \
        cass-shared
cass-shared
```

Launch a container running Cassandra, with the `cass-shared` volume mounted at /var/lib/cassandra/data



```
> docker run -d \
    --volume cass-shared:/var/lib/cassandra/data \
      --name cass1    cassandra
Unable to find image 'docker:latest' locally
latest: Pulling from library/docker
9b18e9b68314: Already exists
e5833820420a: Pull complete
79069699b830: Pull complete
444a66d86b54: Pull complete
af8c662400c0: Pull complete
7851778f47af: Pull complete
969048075247: Pull complete
9d9d268b6129: Pull complete
71d72239e8c8: Pull complete
Digest: sha256:0e3e7e2033cf7779ab6985e24ad18d6ec415c9dd944acef5ca56119a3a0dda2e
Status: Downloaded newer image for docker:latest
59bb65fc737147f1968890bf321a8aacc97cc9abb51a93449a7bead1c8fad682
```

Connect to Cassandra from another container & submit instructions



```
(base) mdd@turandot ~ % docker run -it --rm --link cass1:cass cassandra cqlsh cass
Connected to Test Cluster at cass:9042
[cqlsh 6.0.0 | Cassandra 4.0.7 | CQL spec 3.4.5 | Native protocol v5]
Use HELP for help.
cqlsh> select *
   ... from system.schema_keyspaces
   ... where keyspace_name = 'docker_hello_world'
   ... ;
```

Sharing Files between Containers

- Sharing access to the same set of files between multiple containers is where the **value of volumes** becomes most obvious.
- **Bind mounts** are the most obvious way to share disk space between containers.
- Unlike shares based on bind mounts, **named volumes** enable containers to share files without any knowledge of the underlying host file system.
    - Unless the volume needs to use specific settings or plugins, it does not have to exist before the first container mounts it.
    - Docker will automatically create volumes named in `run` or `create` commands by using the defaults.
- <u>Attention</u>: A named volume that exists on the host will be reused and shared by any other containers with the same volume dependency.
- Name conflicts can be avoided by using anonymous volumes and mountpoint definition inheritance between containers.

- **Volumes** allow containers to share files with the host or other containers:
- Volumes are parts of the host file system that Docker mounts into containers at specified locations. There are two types of volumes:
  - Docker-managed volumes that are located in the Docker part of the host file system
  - Bind mount volumes that are located anywhere on the host file system.
- Volumes have life-cycles that are independent of any specific container, but a user can only reference Docker-managed volumes with a container handle.
- Orphan volume problem can make disk space difficult to recover.
  - Use the `docker rm -v` option.
- A number of patterns can be followed to provide for volume organization, storage efficiency on the host, static content distribution, maximing reuse of storage etc
  - Volume container pattern
  - Data-packed volume container pattern: useful for distributing static content for other containers
  - The polymorphic container pattern: a way to compose minimal functional components and maximize reuse

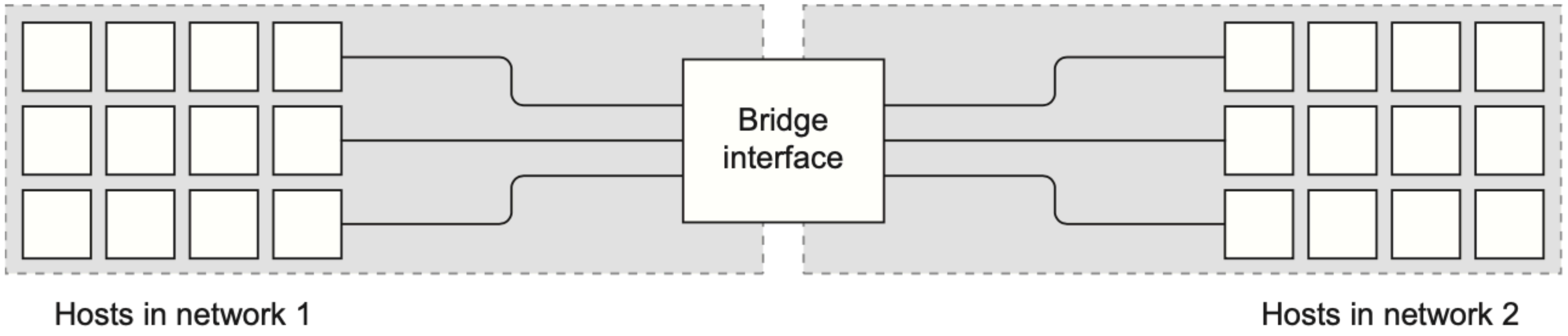
- Mount points allow many f/s from many devices to be attached to a single file tree.
- Every container has its own file tree.
- Containers can use bind mounts to attach parts of the host f/s into a container.
- In-memory filesystems can be attached to a container file tree so that sensitive or temporary data is not written to disk.
- Docker provides anonymous or named storage references called volumes.
- Volumes can be created, listed, and deleted using the appropriate docker volume subcommand.
- Volumes are parts of the host filesystem that Docker mounts into containers at specified locations.
- Volumes have life cycles of their own and might need to be periodically cleaned up.
- Docker can provide volumes backed by network storage or other more sophisticated tools if the appropriate volume plugin is installed.

Docker Overview

Single-host networking

Docker Container Networking

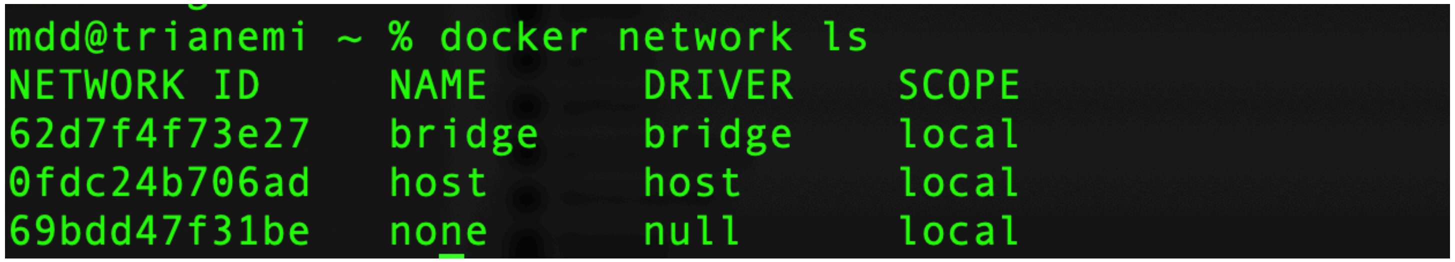
- There are two specific networks of interest when examining the networking capabilities of Docker containers running on a server:
  - The first network is the one that the server is connected to.
  - The second is a virtual network that Docker creates to connect all of the running containers to the network that the computer is connected to.
- The second network is called a **bridge**. The bridge is an interface that connects multiple networks so that they can function as a single network.
  - Bridges work by selectively forwarding traffic between the connected networks based on another type of network address.



Docker Networks

- Docker abstracts the underlying host-attached network from containers.
- A container attached to a Docker network will get a unique IP address that is routable from other containers attached to the same Docker network.
- Docker treats networks as first-class entities: they have their own lifecycle and are not bound to any other objects.
- They can be defined and managed with the **`docker network`** subcommands.

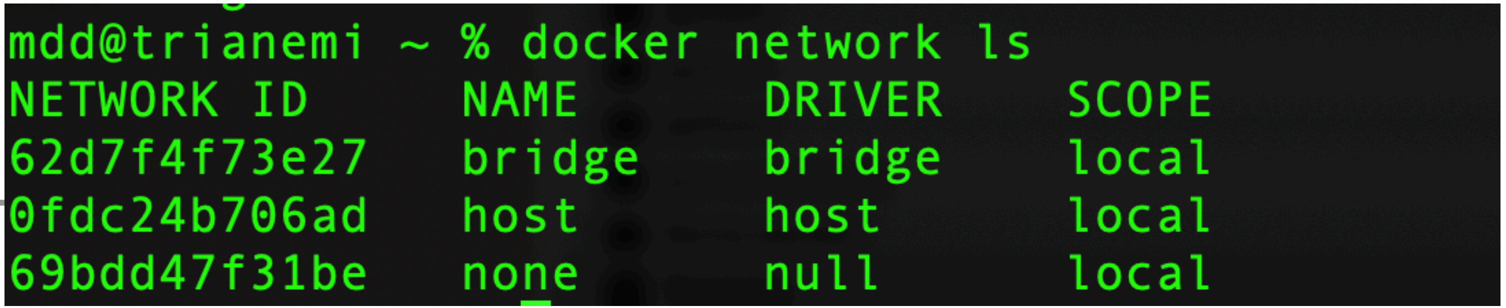
WHICH NETWORKS ARE AVAILABLE BY DEFAULT WITH EVERY DOCKER INSTALLATION?



```
mdd@trianemi ~ % docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
62d7f4f73e27   bridge    bridge    local
0fdc24b706ad   host      host      local
69bdd47f31be   none      null      local
```

Docker Default Networks

- By default, Docker includes three networks, each provided by a different driver.
- **Bridge** is the default network, provided by a *bridge driver*.
  - This offers inter-container connectivity for all containers running on the same machine.
  - Included to maintain compatibility with legacy Docker and cannot take advantage of modern Docker features including service discovery or load balancing - so if you need a bridge network, you have to provide your own.
- The **host network** is provided by a *host driver*, which instructs Docker not to create any special networking namespace or resources for attached containers.
  - The containers on the **host network** interact with the host's network stack like uncontained processes.
- The **none network** uses the *null driver*.
  - Containers attached to the none network will not have any network connectivity outside themselves.



```
mdd@trianemi ~ % docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
62d7f4f73e27   bridge    bridge    local
0fdc24b706ad   host      host      local
69bdd47f31be   none      null      local
```

Docker Network Scope

- The scope of a network can take three values:
  - **Local**: the network is constrained to the machine where the network exists
  - **Global**: should be created on every node in a cluster but not route between them
  - **Swarm**: seamlessly spans all of the hosts participating in a Docker swarm (multi-host or cluster-wide)
- All of the default networks have the local scope, and will **not be able to directly route traffic between containers** running on **different machines**.

Bridge Network

- Containers have their own private **loopback interface** and a separate **virtual Ethernet interface linked to another virtual interface in the host's namespace.**
  - These two linked interfaces form a link between the host's network and the container.
- Each container is assigned a unique private IP address that's not directly reachable from the external network.
- Connections are routed through another Docker network that routes traffic between containers and may connect to the host's network to form a bridge.

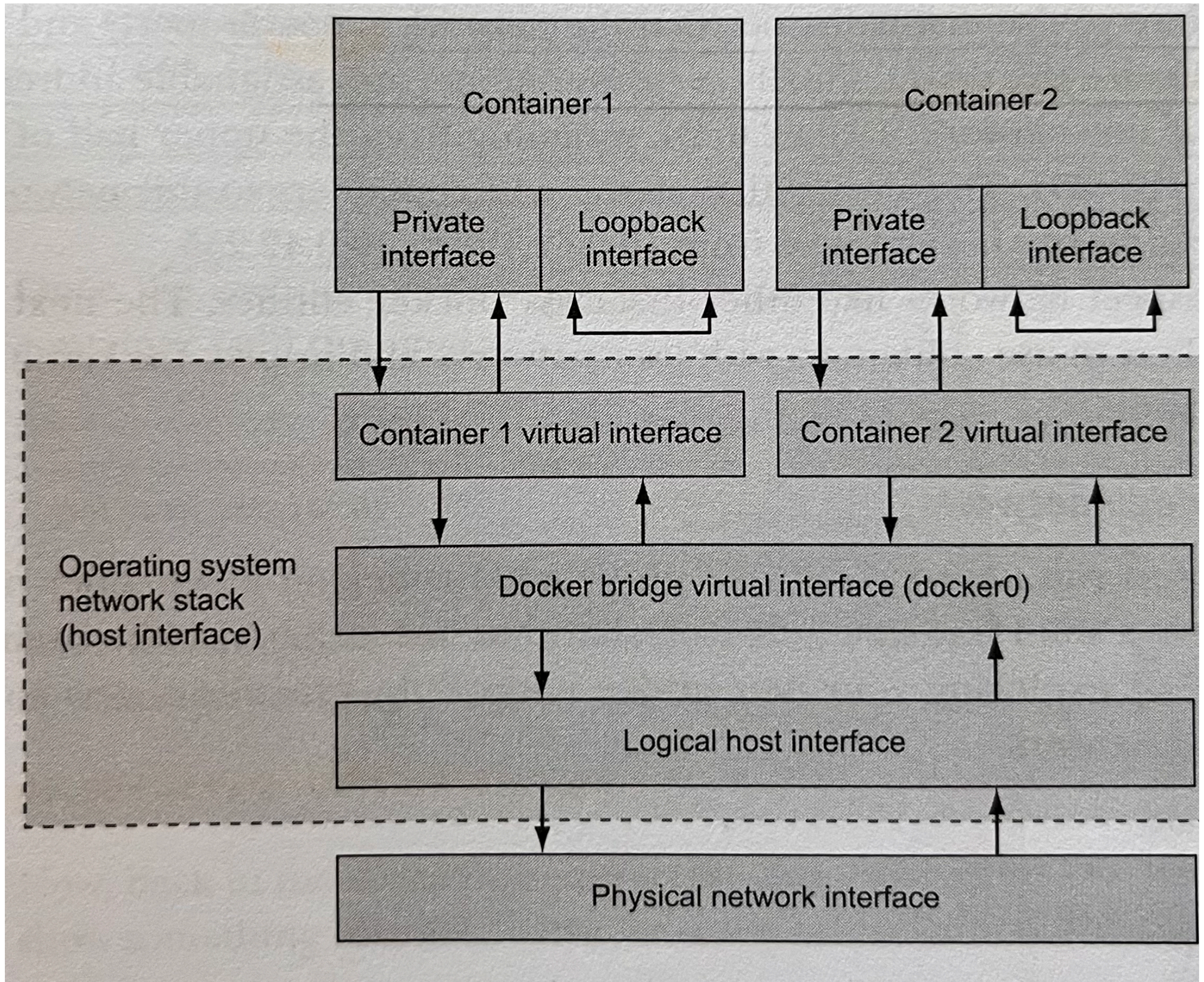

Container 1
Private interface
Loopback interface
Container 2
Private interface
Loopback interface
Container 1 virtual interface
Container 2 virtual interface
Operating system network stack (host interface)
Docker bridge virtual interface (docker0)
Logical host interface
Physical network interface

Figure 5.4 The default local Docker network topology and two attached containers

User-defined Bridge Networks

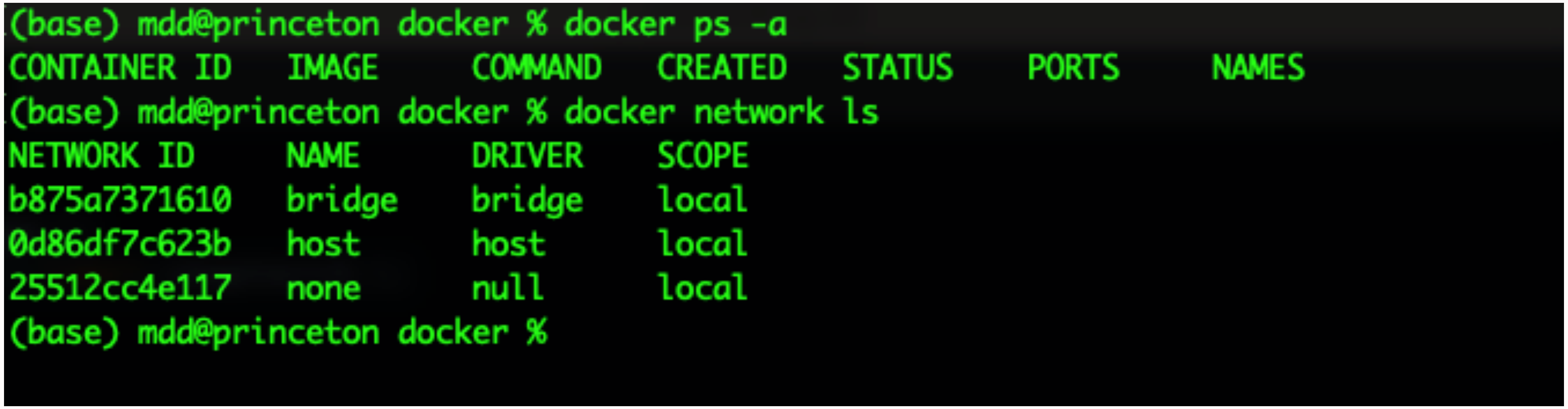
- Docker allows you to create your own specific and customizable virtual network topology, using the Docker bridge network driver, which relies on: Linux namespaces, virtual Ethernet devices, and the Linux firewall.
- The resulting virtual network (the **bridge**):
  - Is local to the machine where Docker is installed.
  - Creates routes between participating containers and the wider network where the host is attached.

Exploring Docker Networking

- Let's see how to use containers with user networks and inspect the resulting network configuration.
- Scenario:
  - Create 2 bridge networks, **`user-network`** and **`user-network2`**.
  - Create 2 containers, **`network-explorer`** and **`lighthouse`**, and connect them to the bridge networks.
  - Examine how these steps are implemented and how the networks are configured.

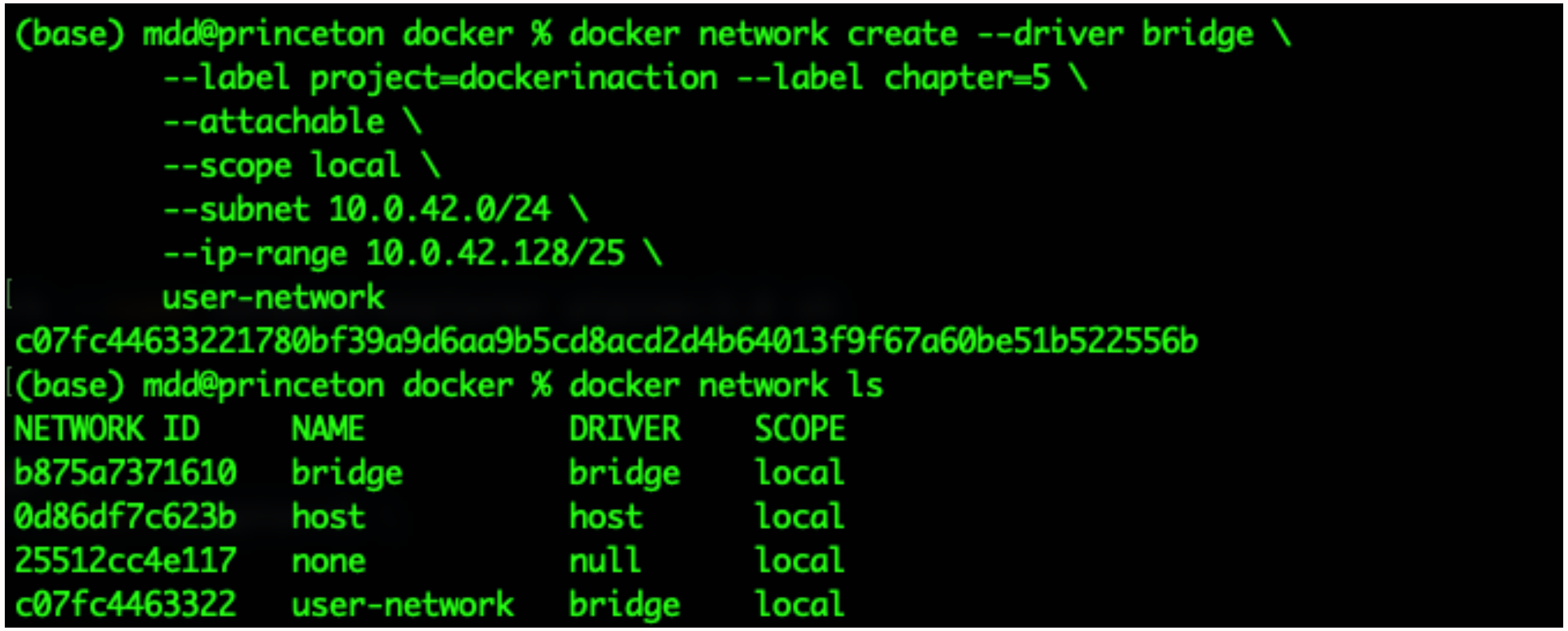
University of Cyprus
Department of Computer Science

- Before starting any containers, check which networks are available



```
(base) mdd@princeton docker % docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
(base) mdd@princeton docker % docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
b875a7371610   bridge    bridge    local
0d86df7c623b   host      host      local
25512cc4e117   none      null      local
(base) mdd@princeton docker %
```

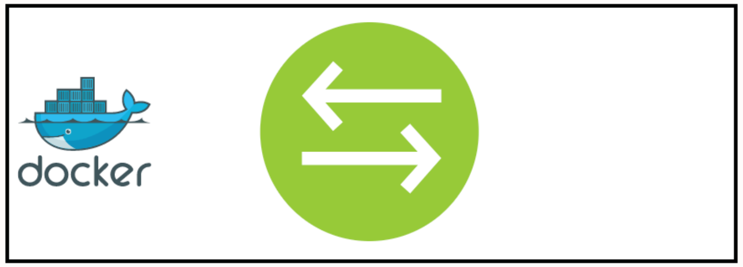
- Create a bridge network, **`user-network`**, and check what happens:



```
(base) mdd@princeton docker % docker network create --driver bridge \
        --label project=dockerinaction --label chapter=5 \
        --attachable \
        --scope local \
        --subnet 10.0.42.0/24 \
        --ip-range 10.0.42.128/25 \
        user-network
c07fc44633221780bf39a9d6aa9b5cd8acd2d4b64013f9f67a60be51b522556b
(base) mdd@princeton docker % docker network ls
NETWORK ID     NAME           DRIVER    SCOPE
b875a7371610   bridge         bridge    local
0d86df7c623b   host           host      local
25512cc4e117   none           null      local
c07fc4463322   user-network   bridge    local
```

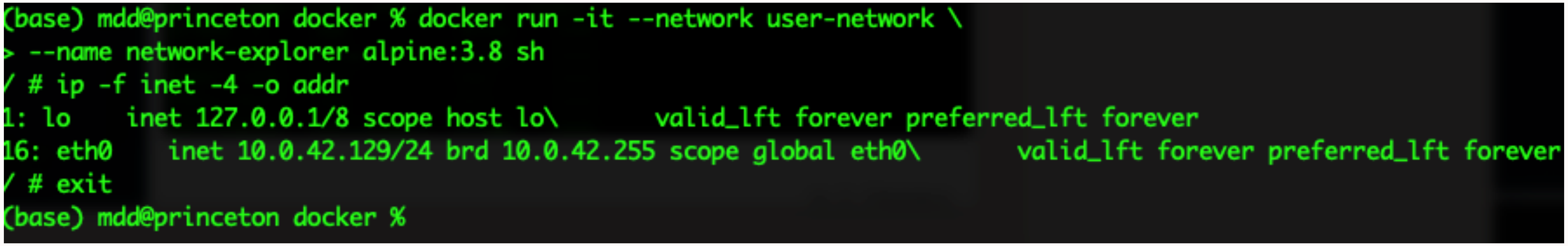
- Creates a new local bridge network named **`user-network`**.
- Assigns the network with label metadata, to help identifying it later.
- Marks the network as attachable to allow attaching and detaching containers to it at any time.
- Sets the network scope property to the default value for its driver.
- Creates a custom subnet **`10.0.42.0/24`**, and an assignable address range for this network's upper half of the last octet (**`10.0.42.128/25`**).
  - This means that as you add containers to this network, they will receive IP addresses in the range from **`10.0.42.128`** to **`10.0.42.255`**.

Configuration



user-network

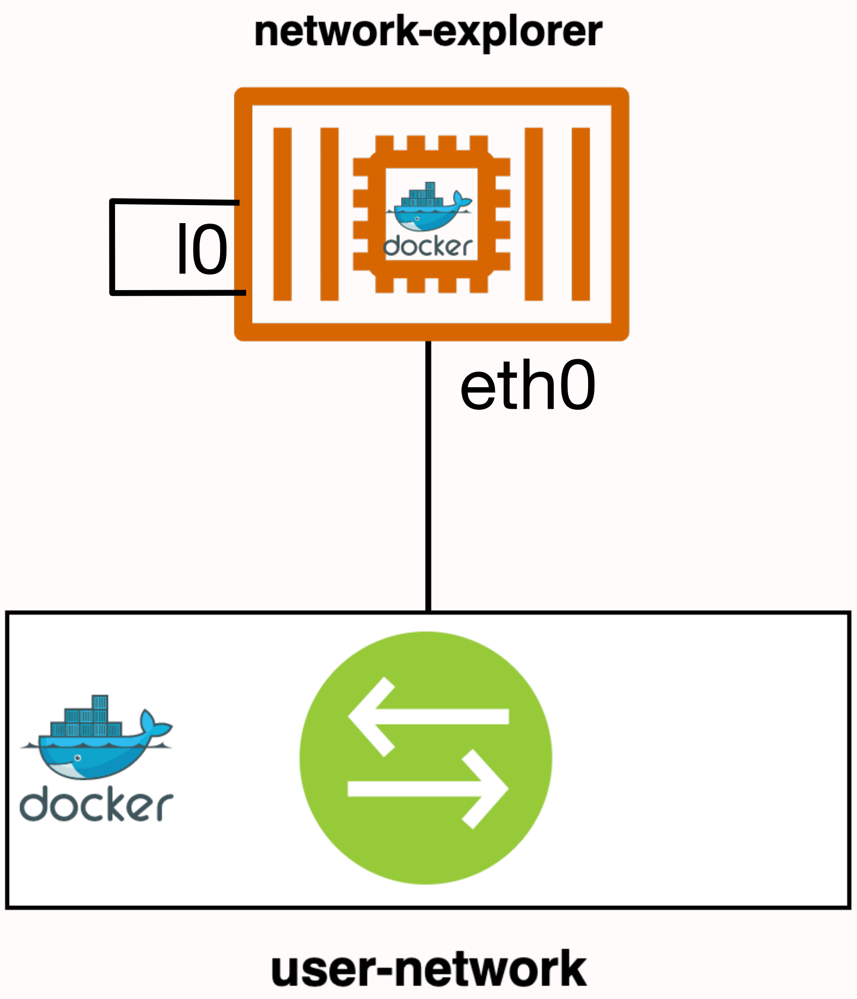
- Launch a container, attach it to the created **`user-network`**, and run **`sh`** on the container.
- From the shell, check the network interfaces available on the container.



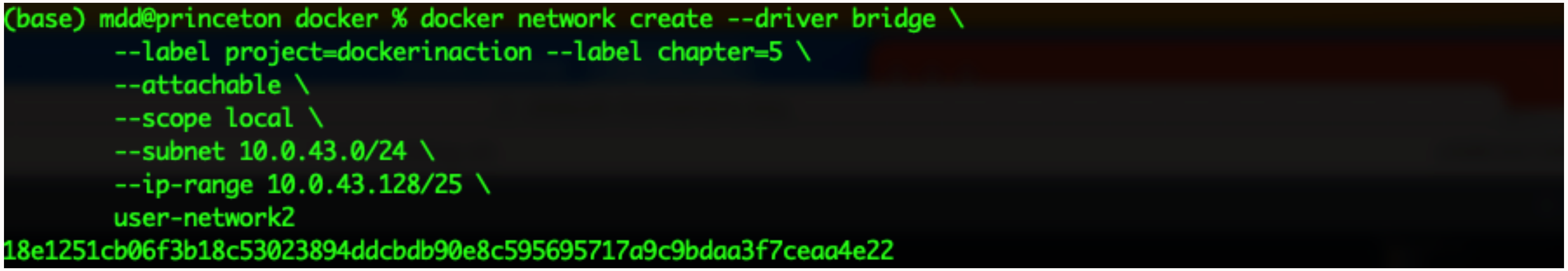
```
(base) mdd@princeton docker % docker run -it --network user-network \
> --name network-explorer alpine:3.8 sh
/ # ip -f inet -4 -o addr
1: lo    inet 127.0.0.1/8 scope host lo\       valid_lft forever preferred_lft forever
16: eth0    inet 10.0.42.129/24 brd 10.0.42.255 scope global eth0\       valid_lft forever preferred_lft forever
/ # exit
(base) mdd@princeton docker %
```

- Two network devices are available, with IPv4 addresses:
  - **`lo`** (loopback)
  - **`eth0`** (virtual ethernet device): has an IP address between **`10.0.42.128`** to **`10.0.42.255`**
- That IP address is the one that any other container on this bridge network would use to communicate with services you run in this container.
- The loopback interface can be used only for communication within the same container.

Configuration

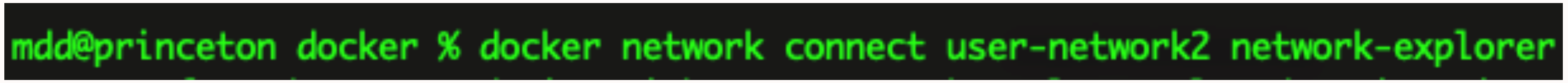


- Create another network named **`user-network2`**



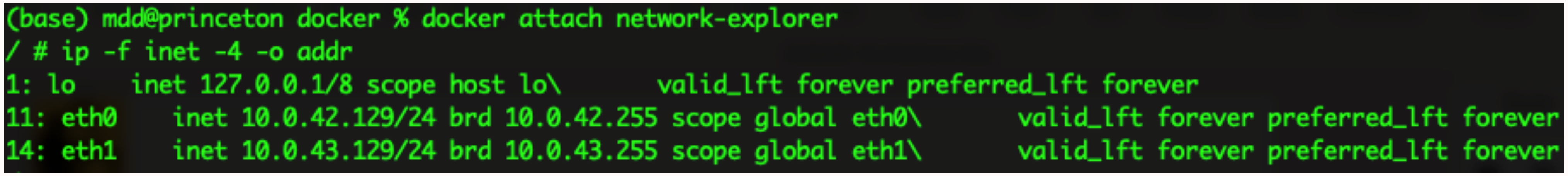
```
(base) mdd@princeton docker % docker network create --driver bridge \
        --label project=dockerinaction --label chapter=5 \
        --attachable \
        --scope local \
        --subnet 10.0.43.0/24 \
        --ip-range 10.0.43.128/25 \
        user-network2
18e1251cb06f3b18c53023894ddcbdb90e8c595695717a9c9bdaa3f7ceaa4e22
```

- Connect your container to the new network:



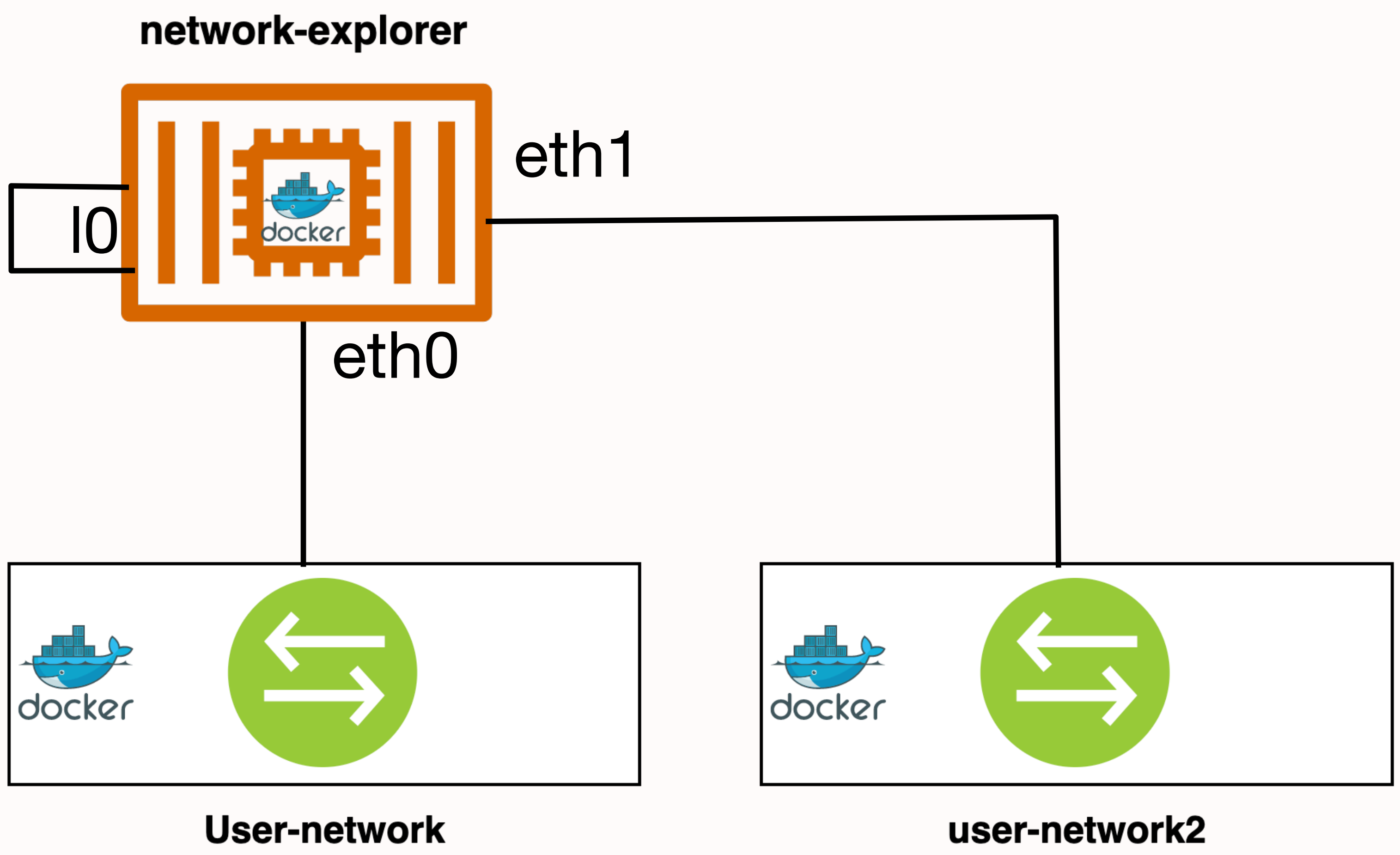
```
mdd@princeton docker % docker network connect user-network2 network-explorer
```

- Attach your terminal to the container and check its network interfaces: the **`network-explorer`** container is attached to both user-defined bridge networks.

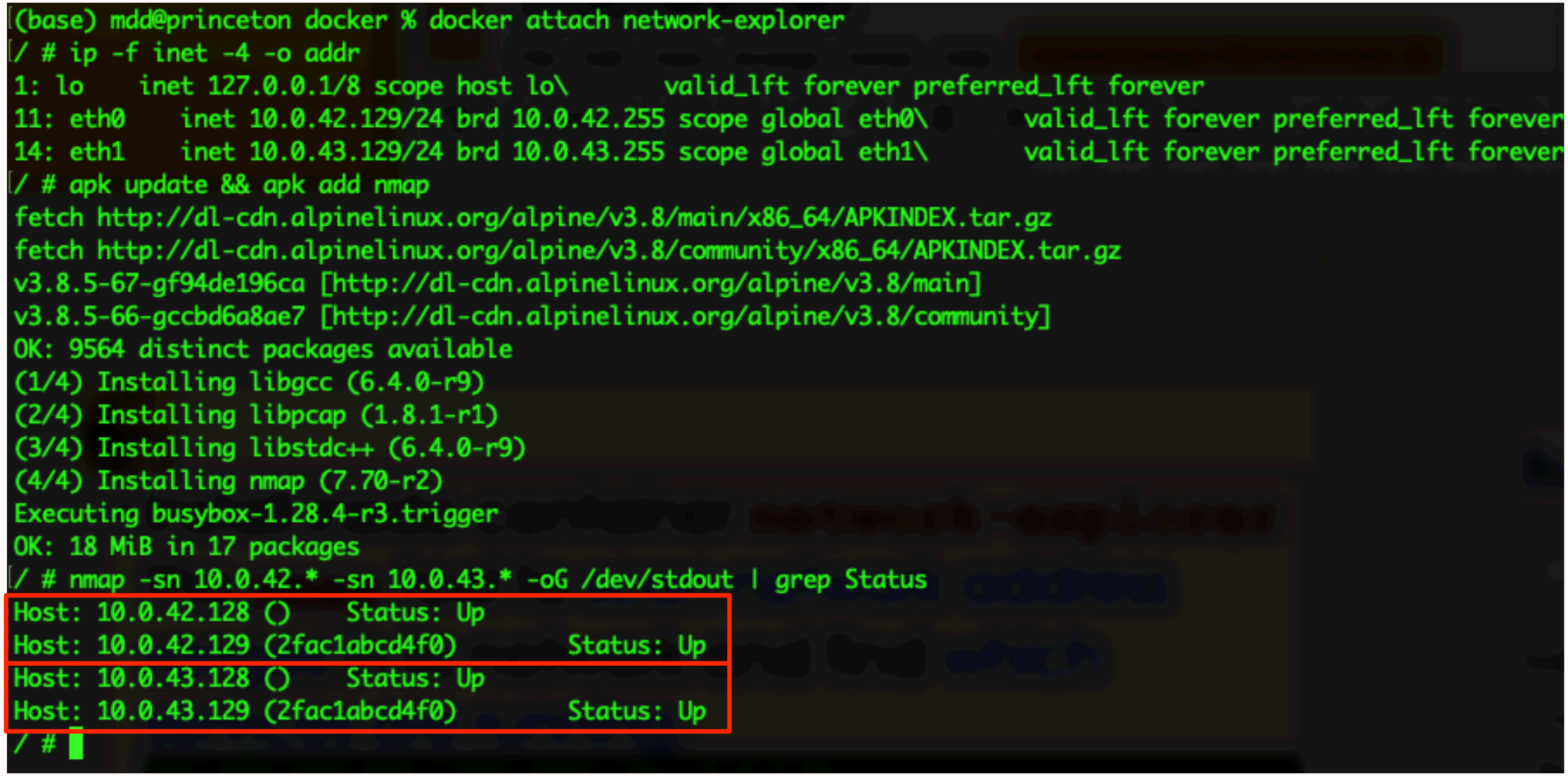


```
(base) mdd@princeton docker % docker attach network-explorer
/ # ip -f inet -4 -o addr
1: lo    inet 127.0.0.1/8 scope host lo\       valid_lft forever preferred_lft forever
11: eth0    inet 10.0.42.129/24 brd 10.0.42.255 scope global eth0\       valid_lft forever preferred_lft forever
14: eth1    inet 10.0.43.129/24 brd 10.0.43.255 scope global eth1\       valid_lft forever preferred_lft forever
```

Configuration



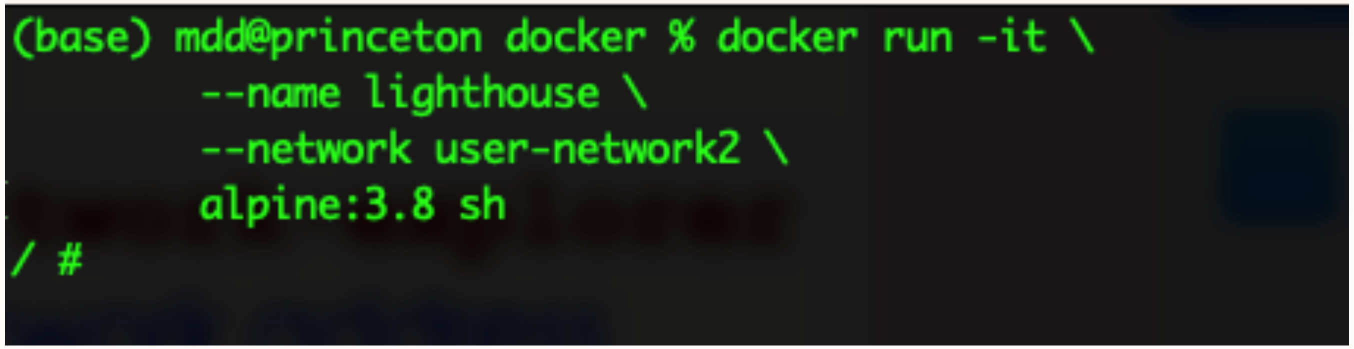
- Install inside container **network-explorer** the **nmap** tool to scan network address ranges in its network and find which services are running.



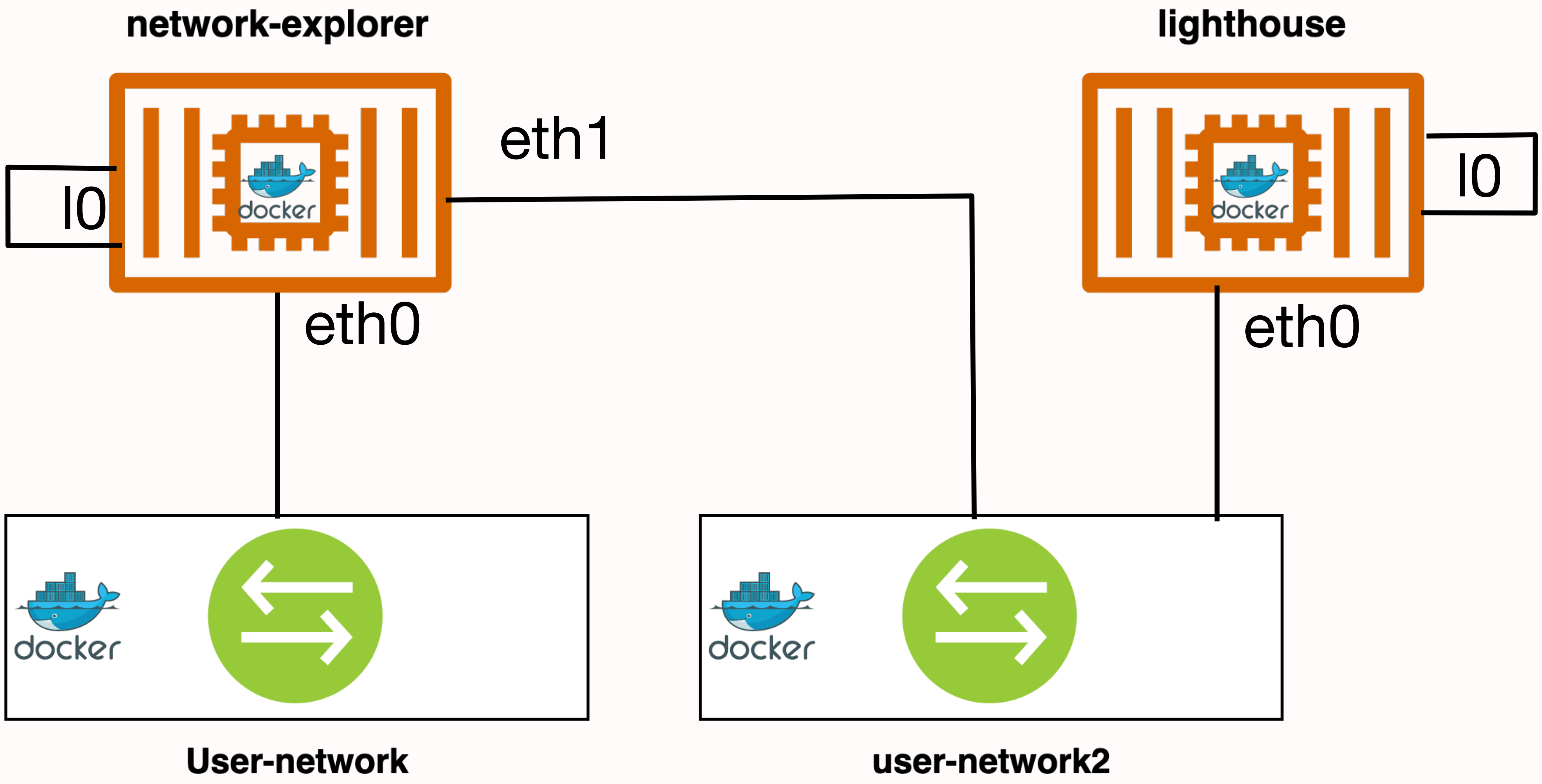
```
(base) mdd@princeton docker % docker attach network-explorer
/ # ip -f inet -4 -o addr
1: lo    inet 127.0.0.1/8 scope host lo\       valid_lft forever preferred_lft forever
11: eth0    inet 10.0.42.129/24 brd 10.0.42.255 scope global eth0\       valid_lft forever preferred_lft forever
14: eth1    inet 10.0.43.129/24 brd 10.0.43.255 scope global eth1\       valid_lft forever preferred_lft forever
/ # apk update && apk add nmap
fetch http://dl-cdn.alpinelinux.org/alpine/v3.8/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.8/community/x86_64/APKINDEX.tar.gz
v3.8.5-67-gf94de196ca [http://dl-cdn.alpinelinux.org/alpine/v3.8/main]
v3.8.5-66-gccbd6a8ae7 [http://dl-cdn.alpinelinux.org/alpine/v3.8/community]
OK: 9564 distinct packages available
(1/4) Installing libgcc (6.4.0-r9)
(2/4) Installing libpcap (1.8.1-r1)
(3/4) Installing libstdc++ (6.4.0-r9)
(4/4) Installing nmap (7.70-r2)
Executing busybox-1.28.4-r3.trigger
OK: 18 MiB in 17 packages
/ # nmap -sn 10.0.42.* -sn 10.0.43.* -oG /dev/stdout | grep Status
Host: 10.0.42.128 ()    Status: Up
Host: 10.0.42.129 (2fac1abcd4f0)        Status: Up
Host: 10.0.43.128 ()    Status: Up
Host: 10.0.43.129 (2fac1abcd4f0)        Status: Up
/ # 
```

- **nmap** finds that only two devices are attached to each of the bridge networks:
  - The gateway drivers created by the bridge network driver
  - The running container

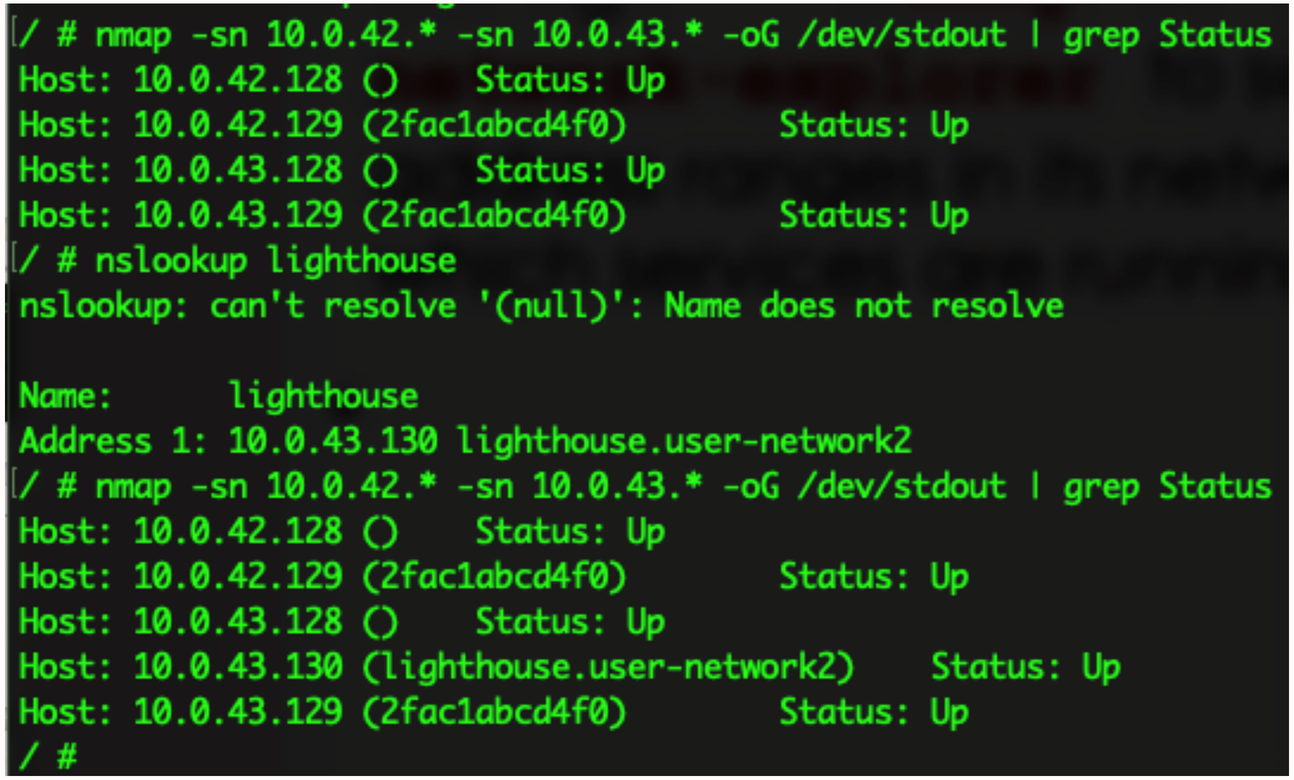
Create a second container named **`lighthouse`** and attach it to **`user-network2`**.



```
(base) mdd@princeton docker % docker run -it \
        --name lighthouse \
        --network user-network2 \
        alpine:3.8 sh
/ #
```



- Use again the **nmap** tool to from container **network-explorer** to scan network address ranges in its network and find which services are running.



```
[/ # nmap -sn 10.0.42.* -sn 10.0.43.* -oG /dev/stdout | grep Status
Host: 10.0.42.128 ()    Status: Up
Host: 10.0.42.129 (2fac1abcd4f0)        Status: Up
Host: 10.0.43.128 ()    Status: Up
Host: 10.0.43.129 (2fac1abcd4f0)        Status: Up
[/ # nslookup lighthouse
nslookup: can't resolve '(null)': Name does not resolve

Name:      lighthouse
Address 1: 10.0.43.130 lighthouse.user-network2
[/ # nmap -sn 10.0.42.* -sn 10.0.43.* -oG /dev/stdout | grep Status
Host: 10.0.42.128 ()    Status: Up
Host: 10.0.42.129 (2fac1abcd4f0)        Status: Up
Host: 10.0.43.128 ()    Status: Up
Host: 10.0.43.130 (lighthouse.user-network2)    Status: Up
Host: 10.0.43.129 (2fac1abcd4f0)        Status: Up
/ #
```

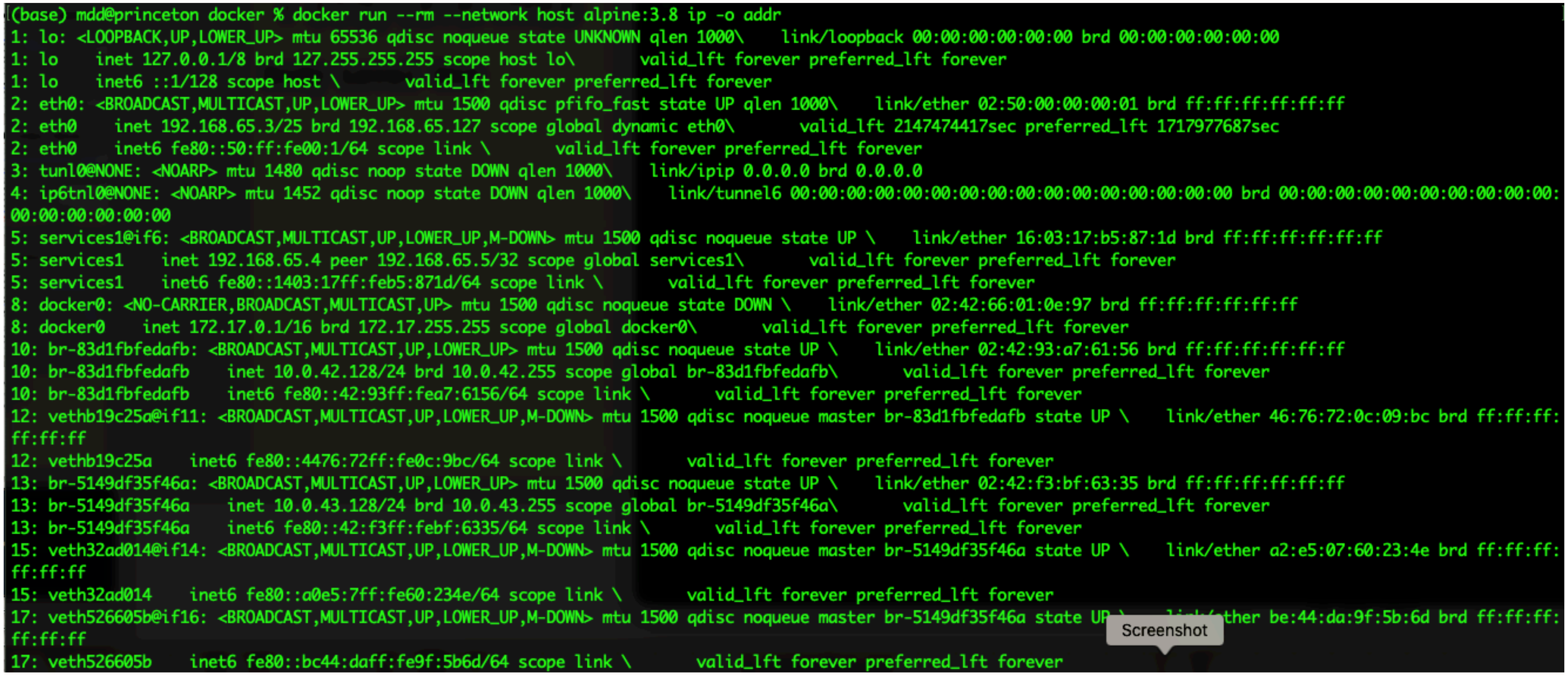
- The results show that the **lighthouse** container is up and running, and accessible from the **network-explorer** container via its attachment to **user-network2**.
  - So network attachment works as expected. DNS-based service discovery system works.
  - By scanning the network, you discover the new node by its IP address, and **nmap** is able to resolve that IP address to a name.
- Your code can discover individual containers on the network based on their name.

Beyond bridge networks

- Bridge networks work on only a single machine: They are not cluster-aware; The container IP addresses are not routable from outside that machine.
  - Useful for single-server deployments, e.g. for a LAMP stack running a CMS or for local development work.
- For Docker networking in multi-server environments, options are:
  - **Underlay networks** - (w. Docker on Linux and w. control of the host network), using drivers like macvlan or ipvlan:
    - Create first-class network addresses for each container
    - Addresses are discoverable & routable from the same network where the host is attached - each container looks like an independent node of the network.
  - **Overlay networks**: similar in construction to bridge networks but the logical bridge component is multi-host aware and can route inter container connections between every node in a swarm (swarm mode must be enabled).
    - Containers on overlay network not routable outside the cluster.

Special Container Networks: `Host`

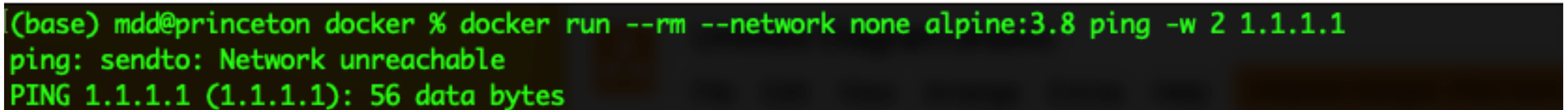
- When you specify the `--network host` option on a `docker run` command, you are telling Docker to:
  - create a new container without any special network adapters or network namespace.
  - Whatever software is running inside the resulting container will have the same degree of access to the host network as it would running outside the container.
  - All of the kernel tools for tuning the network stack are available for modification (as long as the modifying process has access to do so).
- Containers running on the host network are able to:
  - access host services running on localhost
  - see and bind to any of the host network interfaces.
- Running on the host network
  - Is useful for system services or other infrastructure components.
  - Is not appropriate in multi-tenant environments and should be disallowed for third-party containers.



```
(base) mdd@princeton docker % docker run --rm --network host alpine:3.8 ip -o addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000\    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
1: lo    inet 127.0.0.1/8 brd 127.255.255.255 scope host lo\       valid_lft forever preferred_lft forever
1: lo    inet6 ::1/128 scope host \       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000\    link/ether 02:50:00:00:00:01 brd ff:ff:ff:ff:ff:ff
2: eth0    inet 192.168.65.3/25 brd 192.168.65.127 scope global dynamic eth0\       valid_lft 2147474417sec preferred_lft 1717977687sec
2: eth0    inet6 fe80::50:ff:fe00:1/64 scope link \       valid_lft forever preferred_lft forever
3: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN qlen 1000\    link/ipip 0.0.0.0 brd 0.0.0.0
4: ip6tnl0@NONE: <NOARP> mtu 1452 qdisc noop state DOWN qlen 1000\    link/tunnel6 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00 brd 00:00:00:00:00:00:00:00:00:
00:00:00:00:00:00
5: services1@if6: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP \    link/ether 16:03:17:b5:87:1d brd ff:ff:ff:ff:ff:ff
5: services1    inet 192.168.65.4 peer 192.168.65.5/32 scope global services1\       valid_lft forever preferred_lft forever
5: services1    inet6 fe80::1403:17ff:feb5:871d/64 scope link \       valid_lft forever preferred_lft forever
8: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN \    link/ether 02:42:66:01:0e:97 brd ff:ff:ff:ff:ff:ff
8: docker0    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0\       valid_lft forever preferred_lft forever
10: br-83d1fbfedafb: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP \    link/ether 02:42:93:a7:61:56 brd ff:ff:ff:ff:ff:ff
10: br-83d1fbfedafb    inet 10.0.42.128/24 brd 10.0.42.255 scope global br-83d1fbfedafb\       valid_lft forever preferred_lft forever
10: br-83d1fbfedafb    inet6 fe80::42:93ff:fea7:6156/64 scope link \       valid_lft forever preferred_lft forever
12: vethb19c25a@if11: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue master br-83d1fbfedafb state UP \    link/ether 46:76:72:0c:09:bc brd ff:ff:ff:
ff:ff:ff
12: vethb19c25a    inet6 fe80::4476:72ff:fe0c:9bc/64 scope link \       valid_lft forever preferred_lft forever
13: br-5149df35f46a: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP \    link/ether 02:42:f3:bf:63:35 brd ff:ff:ff:ff:ff:ff
13: br-5149df35f46a    inet 10.0.43.128/24 brd 10.0.43.255 scope global br-5149df35f46a\       valid_lft forever preferred_lft forever
13: br-5149df35f46a    inet6 fe80::42:f3ff:febf:6335/64 scope link \       valid_lft forever preferred_lft forever
15: veth32ad014@if14: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue master br-5149df35f46a state UP \    link/ether a2:e5:07:60:23:4e brd ff:ff:ff:
ff:ff:ff
15: veth32ad014    inet6 fe80::a0e5:7ff:fe60:234e/64 scope link \       valid_lft forever preferred_lft forever
17: veth526605b@if16: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue master br-5149df35f46a state UP \    link/ether be:44:da:9f:5b:6d brd ff:ff:ff:
ff:ff:ff
17: veth526605b    inet6 fe80::bc44:daff:fe9f:5b6d/64 scope link \       valid_lft forever preferred_lft forever
```

Special Container Networks: `None`

- To create a container that cannot be attached to a network when building systems of least privilege, you should use the `none` network whenever possible.
- Creating a container on the `none` network:
  - Instructs Docker not to provision any connected virtual Ethernet adapters for the new container.
  - Gives the container its own network namespace and so it will be isolated: without adapters connected across the namespace boundary, it will not be able to use the network to communicate outside the container.
- Containers configured this way will still have their own loopback interface, and so multiprocess containers can still use connections to `localhost` for interprocess communication:
  - Any program running in the container can connect to or wait for connections on the `localhost` interface.
  - Nothing outside the container can connect to that interface.
  - No program running inside that container can reach anything outside the container.



```
(base) mdd@princeton docker % docker run --rm --network none alpine:3.8 ping -w 2 1.1.1.1
ping: sendto: Network unreachable
PING 1.1.1.1 (1.1.1.1): 56 data bytes
```

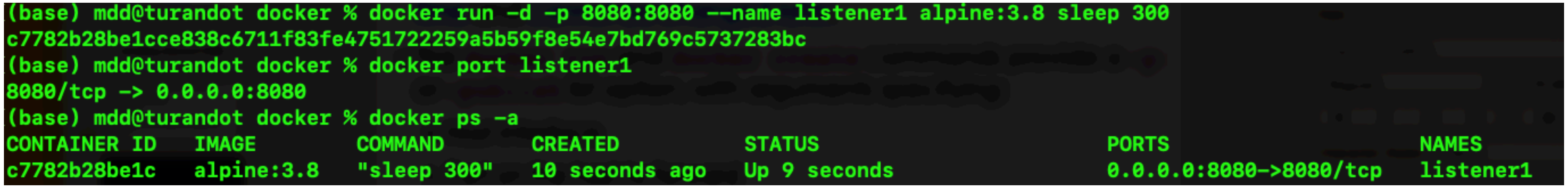
Handling Inbound Traffic

- Bridge networks use network address translation (NAT):
  - All outbound container traffic with destinations outside the bridge network look like they are coming from the host itself.
  - The service software you have running in containers is isolated from the rest of the world, where most of your clients and customers are located.
- For **inbound traffic** to reach a container from external network interfaces, you have to specifically tell Docker how to forward traffic to the container, specifying:
  - A TCP or UDP port on the host interface and
  - A target container and container port

  similar to forwarding traffic through a NAT barrier on your home network.

Handling Inbound Traffic

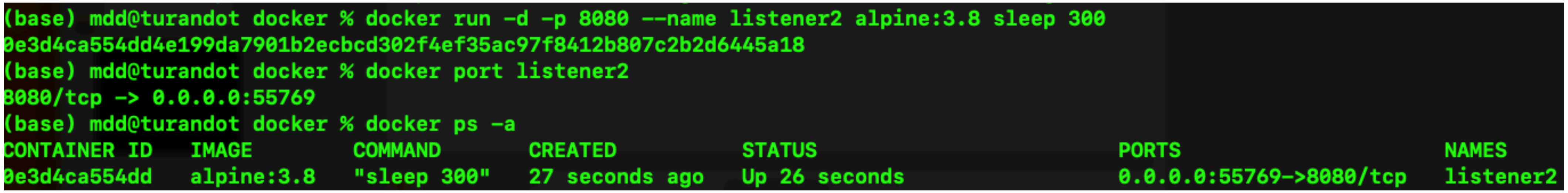
- Port publication configuration is provided at container creation time and cannot be changed later.
- The `docker run` and `docker create` commands provide a `-p` or `–publish` list option, with arguments specifying:
  - Host interface
  - Port on the host to forward
  - Target port
  - Port protocol
- Map port `8080` of the host interface to port `8080` of the `container listener1`:



```
(base) mdd@turandot docker % docker run -d -p 8080:8080 --name listener1 alpine:3.8 sleep 300
c7782b28be1cce838c6711f83fe4751722259a5b59f8e54e7bd769c5737283bc
(base) mdd@turandot docker % docker port listener1
8080/tcp -> 0.0.0.0:8080
(base) mdd@turandot docker % docker ps -a
CONTAINER ID   IMAGE        COMMAND       CREATED          STATUS                        PORTS                      NAMES
c7782b28be1c   alpine:3.8   "sleep 300"   10 seconds ago   Up 9 seconds                  0.0.0.0:8080->8080/tcp     listener1
```

Handling Inbound Traffic

- Map some (randomly chosen by the host operating system) port of the host interface to port `8080` of the `container listener2`:

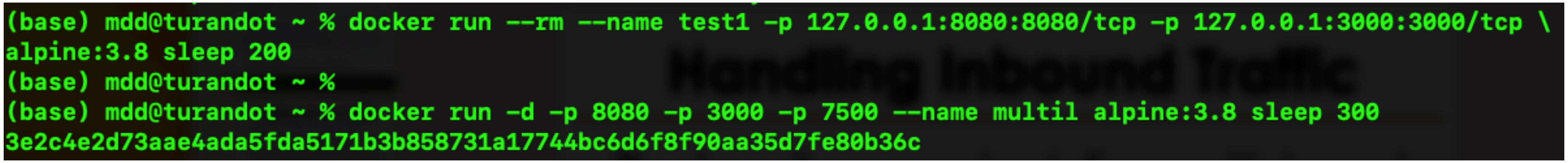


```
(base) mdd@turandot docker % docker run -d -p 8080 --name listener2 alpine:3.8 sleep 300
0e3d4ca554dd4e199da7901b2ecbcd302f4ef35ac97f8412b807c2b2d6445a18
(base) mdd@turandot docker % docker port listener2
8080/tcp -> 0.0.0.0:55769
(base) mdd@turandot docker % docker ps -a
CONTAINER ID   IMAGE        COMMAND       CREATED          STATUS                        PORTS                     NAMES
0e3d4ca554dd   alpine:3.8   "sleep 300"   27 seconds ago   Up 26 seconds                 0.0.0.0:55769->8080/tcp   listener2
```

- **Ports are scarce resources** - choosing a random port avoids potential conflicts.
- Note: programs running inside a container have no way of knowing that they are:
  - running inside a container
  - bound to a container network,
  - which port is being forwarded from the host

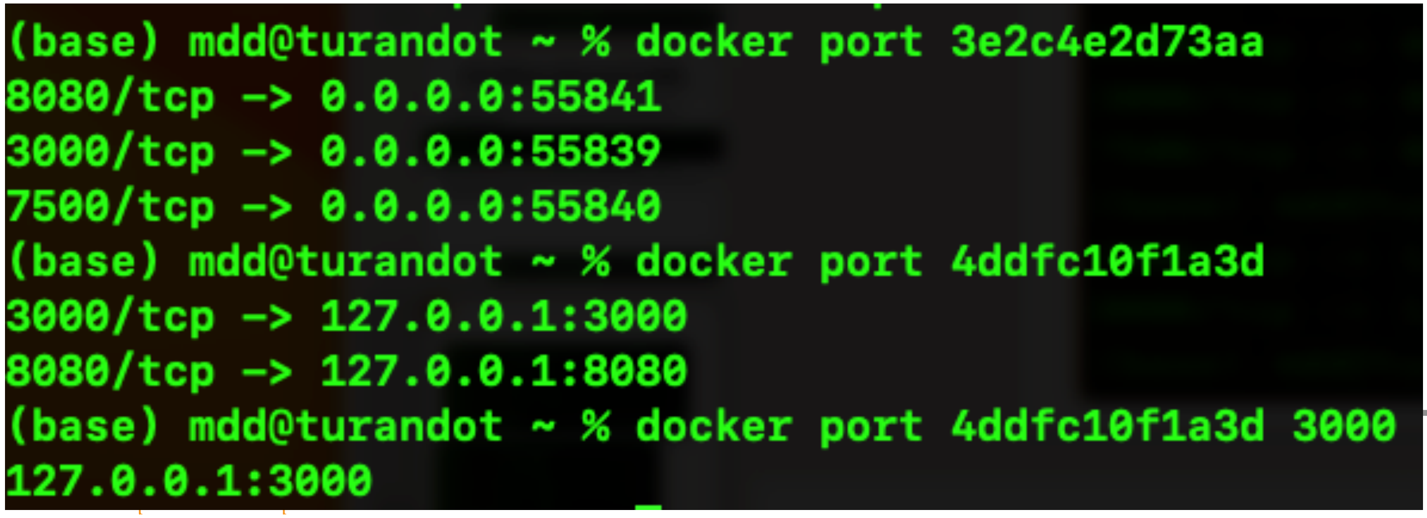
Handling Inbound Traffic

- Docker allows you to define multiple port mappings:



```
(base) mdd@turandot ~ % docker run --rm --name test1 -p 127.0.0.1:8080:8080/tcp -p 127.0.0.1:3000:3000/tcp \
alpine:3.8 sleep 200
(base) mdd@turandot ~ %
(base) mdd@turandot ~ % docker run -d -p 8080 -p 3000 -p 7500 --name multil alpine:3.8 sleep 300
3e2c4e2d73aae4ada5fda5171b3b858731a17744bc6d6f8f90aa35d7fe80b36c
```

- With `docker port` subcommand you can specify the specific port of your container for which you are looking for its mapping on a port of its host machine:



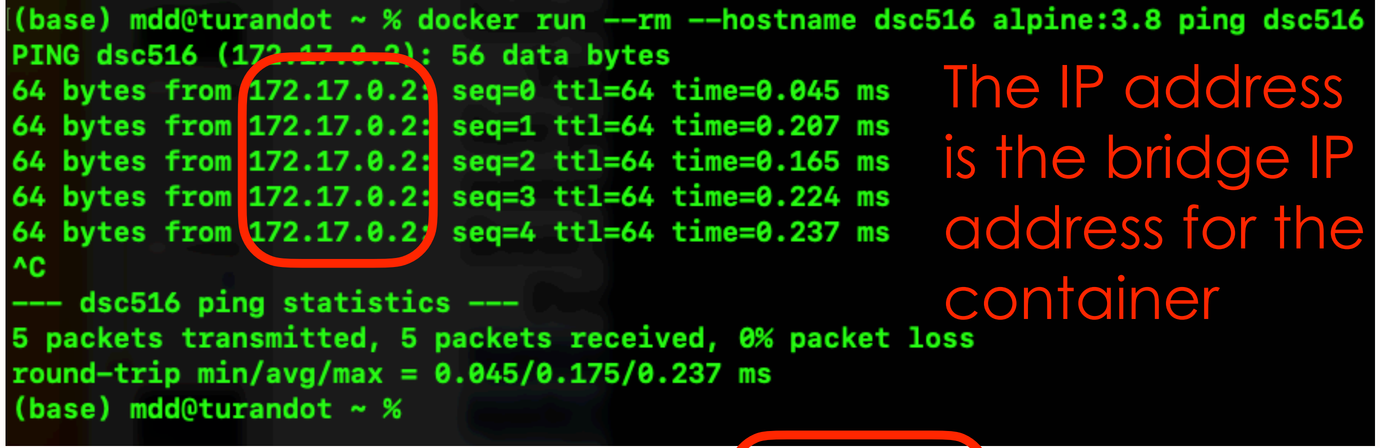
```
(base) mdd@turandot ~ % docker port 3e2c4e2d73aa
8080/tcp -> 0.0.0.0:55841
3000/tcp -> 0.0.0.0:55839
7500/tcp -> 0.0.0.0:55840
(base) mdd@turandot ~ % docker port 4ddfc10f1a3d
3000/tcp -> 127.0.0.1:3000
8080/tcp -> 127.0.0.1:8080
(base) mdd@turandot ~ % docker port 4ddfc10f1a3d 3000
127.0.0.1:3000
```

Firewalls in Docker

- Docker networking follows the **namespace model**, wherein containers in the same container network know each other's names and through that knowledge they can communicate.
  - Resource access-control problems are transformed into addressability problems.
  - Containers on the same container network will have mutual (bidirectional) unrestricted network access.
- However, different applications carry different vulnerabilities and might be running in containers with different security postures.
- Consequently:
  - A firewall will **not protect** you against a compromised application running in a container of your network.
  - Nothing short of application-level authentication and authorization can protect containers from each other on the same network.
  - Always deploy containers with appropriate application-level access-control mechanisms.

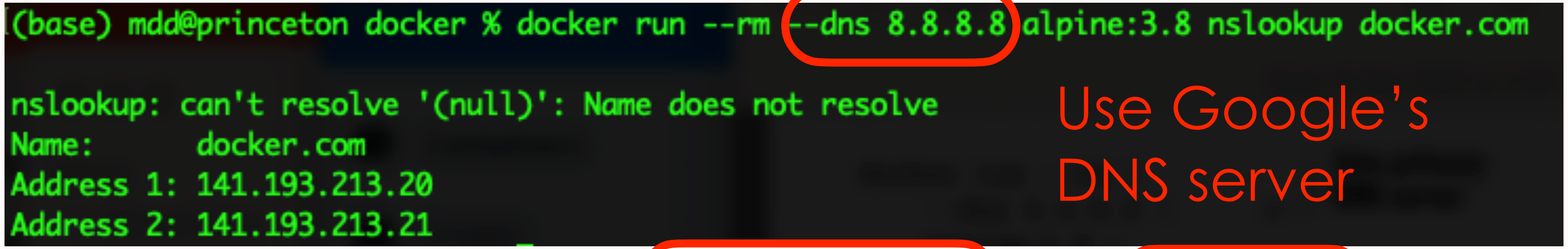
Containers and DNS

- Typically, containers on the bridge network and other computers on your network have private IP addresses that aren't publicly routable:
  - unless you're running your own DNS server, you can't refer to them by a name.
- Options for customizing the DNS configuration for a new container, use docker `run` with:
- The `--hostname` flag to set the hostname of the new container: adds **this entry to the DNS override system** inside the container.
  - The entry maps the provided hostname to the container's bridge.
  - Setting the hostname of a container is useful when programs running inside a container need to look up their own IP address or must self-identify. However, other containers **don't know this hostname:** its uses are limited.
  - If you use an external DNS server, you can share those hostnames.
- The `--dns` flag to specify **one or more DNS servers** to use.
- The `--dns-search` flag allows you to specify a **DNS search domain**, which is like a **default hostname suffix**.
  - With a DNS search domain set, any hostnames that don't have a known top-level domain (for example, `.com` or `.net`) will be searched for with the specified suffix appended.
- The `--add-host` flag allows overriding the DNS system, providing a custom mapping for an IP address and hostname pair.



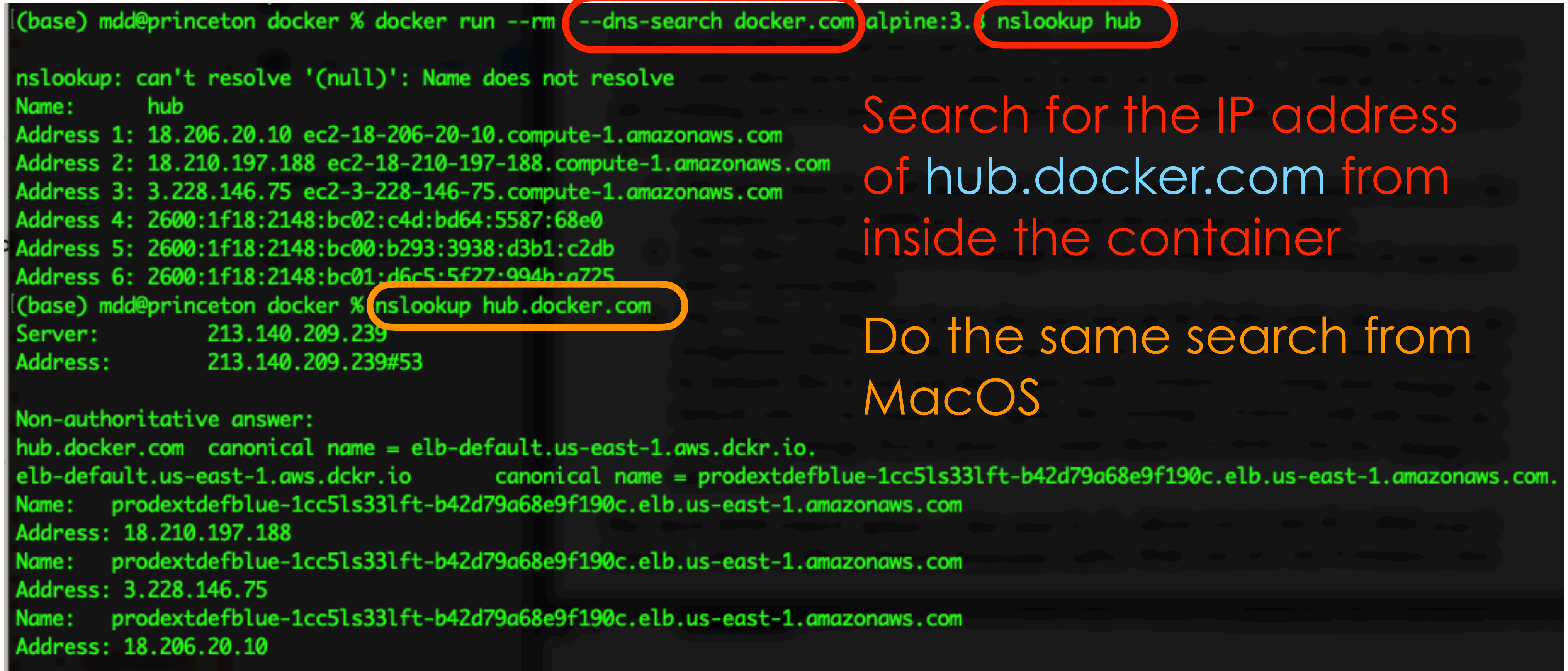
```
(base) mdd@turandot ~ % docker run --rm --hostname dsc516 alpine:3.8 ping dsc516
PING dsc516 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.045 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.207 ms
64 bytes from 172.17.0.2: seq=2 ttl=64 time=0.165 ms
64 bytes from 172.17.0.2: seq=3 ttl=64 time=0.224 ms
64 bytes from 172.17.0.2: seq=4 ttl=64 time=0.237 ms
^C
--- dsc516 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.045/0.175/0.237 ms
(base) mdd@turandot ~ %
```

The IP address is the bridge IP address for the container



```
(base) mdd@princeton docker % docker run --rm --dns 8.8.8.8 alpine:3.8 nslookup docker.com

nslookup: can't resolve '(null)': Name does not resolve
Name:      docker.com
Address 1: 141.193.213.20
Address 2: 141.193.213.21
```

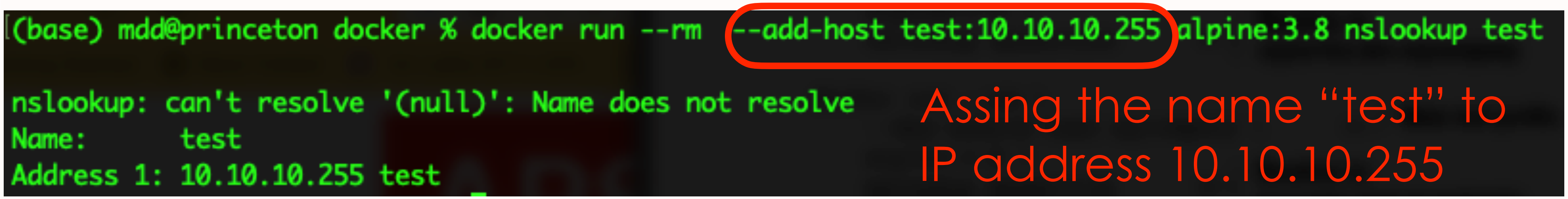
Use Google's DNS server



```
(base) mdd@princeton docker % docker run --rm --dns-search docker.com alpine:3.8 nslookup hub

nslookup: can't resolve '(null)': Name does not resolve
Name:      hub
Address 1: 18.206.20.10 ec2-18-206-20-10.compute-1.amazonaws.com
Address 2: 18.210.197.188 ec2-18-210-197-188.compute-1.amazonaws.com
Address 3: 3.228.146.75 ec2-3-228-146-75.compute-1.amazonaws.com
Address 4: 2600:1f18:2148:bc02:c4d:bd64:5587:68e0
Address 5: 2600:1f18:2148:bc00:b293:3938:d3b1:c2db
Address 6: 2600:1f18:2148:bc01:d6c5:5f27:994b:a725
(base) mdd@princeton docker % nslookup hub.docker.com
Server:         213.140.209.239
Address:        213.140.209.239#53

Non-authoritative answer:
hub.docker.com  canonical name = elb-default.us-east-1.aws.dckr.io.
elb-default.us-east-1.aws.dckr.io       canonical name = prodextdefblue-1cc5ls33lft-b42d79a68e9f190c.elb.us-east-1.amazonaws.com.
Name:   prodextdefblue-1cc5ls33lft-b42d79a68e9f190c.elb.us-east-1.amazonaws.com
Address: 18.210.197.188
Name:   prodextdefblue-1cc5ls33lft-b42d79a68e9f190c.elb.us-east-1.amazonaws.com
Address: 3.228.146.75
Name:   prodextdefblue-1cc5ls33lft-b42d79a68e9f190c.elb.us-east-1.amazonaws.com
Address: 18.206.20.10
```

Search for the IP address of hub.docker.com from inside the container

Do the same search from MacOS



```
(base) mdd@princeton docker % docker run --rm  --add-host test:10.10.10.255 alpine:3.8 nslookup test

nslookup: can't resolve '(null)': Name does not resolve
Name:      test
Address 1: 10.10.10.255 test
```

Externalizing Network Management

- Some organizations, infrastructures, or products require direct management of container network configuration, service discovery, and other network-related resources.
- In those cases, you or the container orchestrator you are using will create containers by using the Docker `none` network.
- Then use some other container-aware tooling to create and manage the container network interfaces, manage NodePort publishing, register containers with service-discovery systems, and integrate with upstream load-balancing systems.
- When you externalize network management, Docker is still responsible for creating the network namespace for the container, but it will not create or manage any of the network interfaces.
- You will not be able to use any of the Docker tooling to inspect the network configuration or port mapping.

- Docker networks are first-class entities that can be created, listed, and removed just like containers, volumes, and images.
- **Bridge networks** are a special kind of network that allows direct inter-container network communication with builtin container name resolution.
- Docker provides two other **special networks** by default: **host** and **none**.
- Networks created with the **none driver** will **isolate attached containers** from the network.
- A container on a host network will have **full access** to the network facilities and interfaces **on the host**.
- **Forward network traffic** to a host port into a target container and port with NodePort publishing.
- Docker bridge networks do not provide **any network firewall** or **access-control** functionality.
- The **network name-resolution stack can be customized** for each container. Custom DNS servers, search domains, and static hosts can be defined.
- Network management can be **externalized** with third-party tooling and by using the Docker none network.

Docker Overview

Controlling Resources

Controlling Resources

Resource Limits (CPU, Memory)

Overview

- If the resource consumption of processes on a computer **exceeds the available physical resources**, the processes will experience performance issues and may stop running:
  - Container systems that provide **strong isolation** include providing resource allowances / setting limits on resource use on individual containers.
- By default, Docker containers may use **unlimited** CPU, memory, and device I/0 resources.
- However, Docker allows the management of resources provided to its containers, upon their creation or launch.

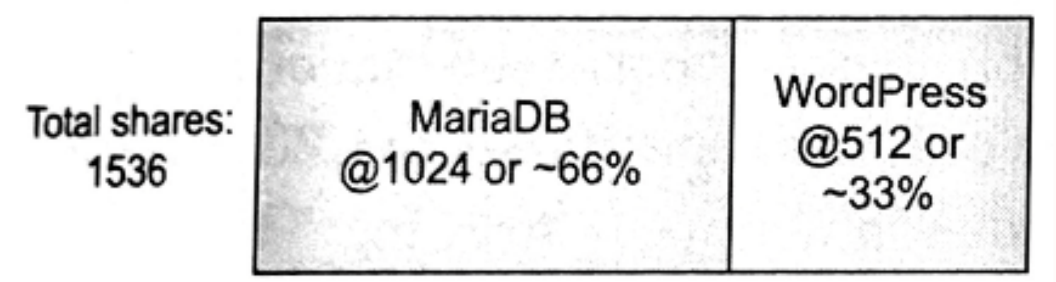
Resource Limits: Memory

- Memory limits restrict the amount of main memory that can be used by processes inside a container.
- Memory limits ensure that one container **can't allocate all of the system's memory**, starving other programs for the memory they need.
- Memory limits are not reservations: They **don't guarantee** that the specified amount of memory will be available: They're only a protection from overconsumption.
- Memory limit enforcement by the Linux kernel is very efficient: its runtime **overhead is minimal**.
- To see memory consumption of a container: **`docker stats`**

```
docker container run -d --name ch6 mariadb \
  --memory 256m \
  --cpu-shares1024 \
  --cap-drop net_raw \
  --e MYSQL_ROOT_PASSWORD=test \
  mariadb:5.5

docker container run -d -P --name ch6 wordpress \
 --memory 512m
 --cpu-shares 512 \
 --cap-drop net_raw \
 --link ch6mariadb:mysql\
 -e WORDPRES_DB_PASSWORD=test \
wordpress:5.0.0-php7.2-apache
```

Set a memory constraint of 256 MB

Set relative CPU shares



```
docker container run -d -P --name ch6_wordpress \
 --memory 512m \
 --cpus 0.75 \
 --cap-drop net_raw \
 --link ch6_mariadb:mysqi \
 -e WORDPRESS_DB_PASSWORD=test \
wordpress:5.0.0-php7.2-apache
```

Use 3/5 of the available CPUs

Resource Limits: CPU

- Starvation of a process' processing time results in performance degradation:
  - A process waiting for time on the CPU is still working correctly, but
  - A slow process may be worse than a failing one, if it is running a latency-sensitive program.
- Docker can **limit a container's CPU resources** by limiting:
  - The sum of the computing cycles of all processors available to the container, relatively to other containers (relative weight of the container).
  - The total number of CPU cores used by a container (cpus option).
- Linux uses the **relative weight** to determine the percentage of CPU time the container should use relatively to other running containers.
  - CPU shares enforced **only when there is contention** for time on the CPU.
- The cpus option allocates a **quota of CPU resources** the container may use by configuring the Linux Completely Fair Scheduler (CES).
  - CPU quota allocated, enforced, refreshed **every 100ms** by default

Resource Access Control: Devices

- Controlling access to devices refers to providing (or not) a container with access to a host's device (cameras, microphones, etc).
- More like a resource-authorization control than a limit.

Mount video0

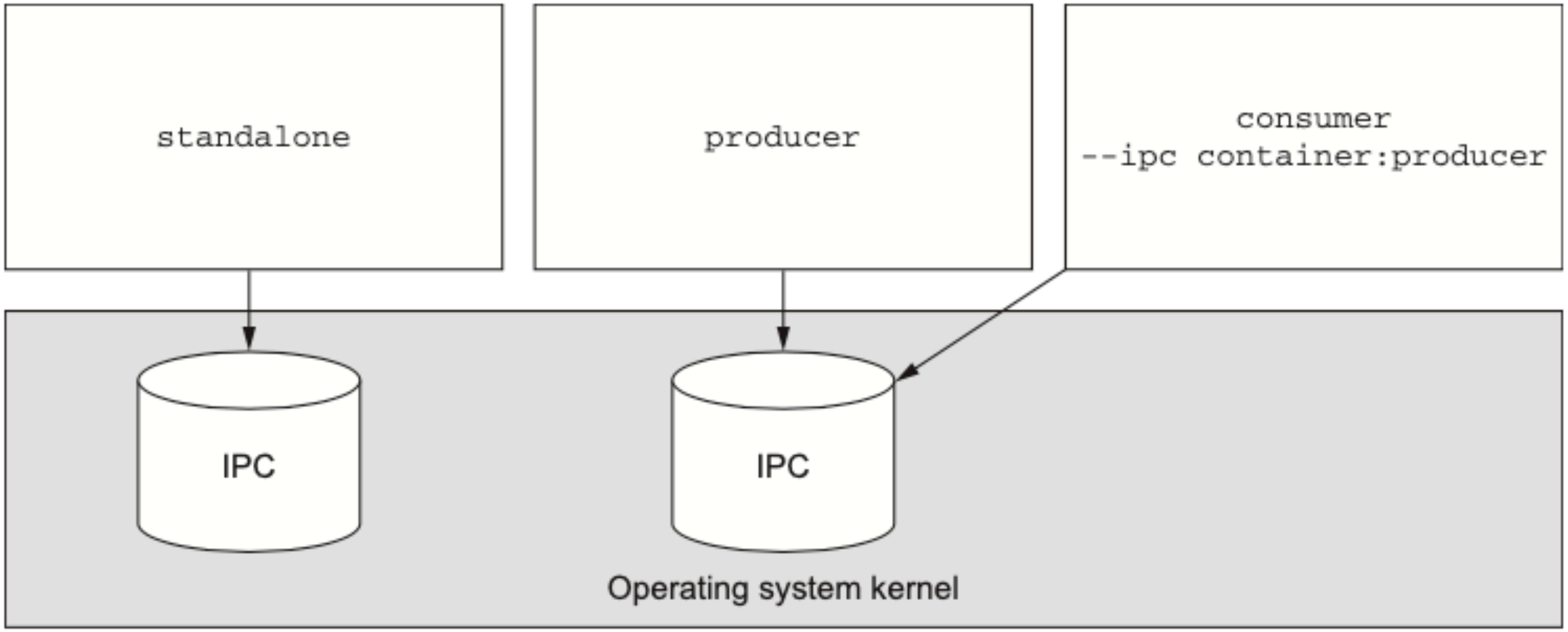
```
docker -it --rm \
    --device /dev/video0:/dev/video0 \
    ubuntu:latest ls -al /dev
```

Controlling Resources

Sharing Memory

Shared Memory

- Linux provides a tools for sharing memory between processes running on the same computer.
- Sharing memory between processes enable inter-process communication (IPC) performed at memory speed.
- Docker creates a unique IPC namespace for each container by default.
  - The Linux IPC namespace partitions, share memory primitives such as **named shared memory blocks, semaphores**, and **message queues**.
  - In Docker, the IPC namespace **prevents** processes in one container from accessing the memory on the host or in other containers.



- To enable shared memory, the consumer needs to join the IPC namespace of the producer, at run time:

```
docker -d --name ch6_ipc_consumer --ipc container:ch6_ipc_producer \
 dockerinaction/ch6_ipc -consumer
```

Open Memory Containers

- If a container needs to operate in the **same memory namespace** as the rest of the **host**, it can be launched as open memory container.
- To enable this feature specify `host` on the `--ipc` flag.
- Open memory containers are a risk, but it's better to use them than to run those processes outside a container.

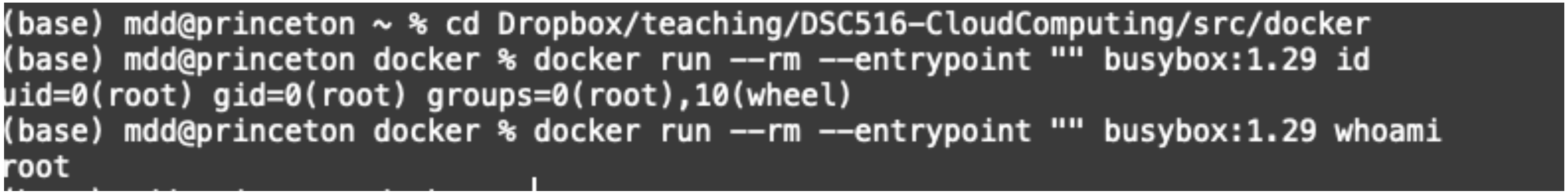
```
docker -d --name ch6_ipc_producer --ipc host \
    dockerinaction/ch6_ipc –producer
docker -d --name ch6_ipc_consumer --ipc host \
    dockerinaction/ch6_ipc -consumer
```

Controlling Resources

Users and User Namespaces

Default Docker User: root

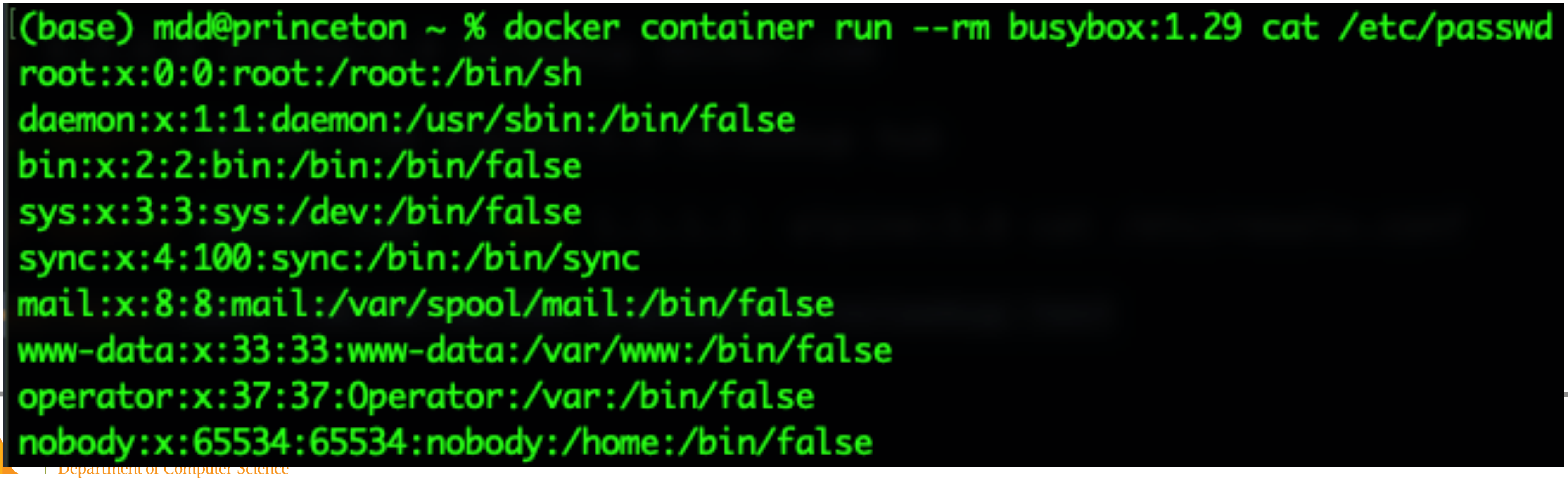
- Docker starts containers as the user that is specified by the image metadata by default, which is often the **root** user.
- The root user has almost full privileged access to the state of the container.
- Any processes running as that user inherit those permissions.



```
(base) mdd@princeton ~ % cd Dropbox/teaching/DSC516-CloudComputing/src/docker
(base) mdd@princeton docker % docker run --rm --entrypoint "" busybox:1.29 id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
(base) mdd@princeton docker % docker run --rm --entrypoint "" busybox:1.29 whoami
root
```

Avoiding Root

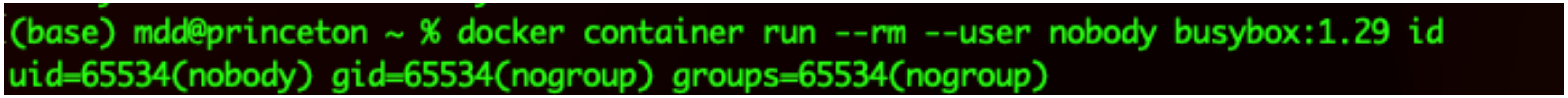
- You can entirely avoid the default user problem if you change the run-as user when you create the container.
  - However, the username must exist on the image.
  - Note that different Linux distributions ship with different users pre-defined.
  - You can get a list of available users in an image with the following command:



```
(base) mdd@princeton ~ % docker container run --rm busybox:1.29 cat /etc/passwd
root:x:0:0:root:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/false
bin:x:2:2:bin:/bin:/bin/false
sys:x:3:3:sys:/dev:/bin/false
sync:x:4:100:sync:/bin:/bin/sync
mail:x:8:8:mail:/var/spool/mail:/bin/false
www-data:x:33:33:www-data:/var/www:/bin/false
operator:x:37:37:Operator:/var:/bin/false
nobody:x:65534:65534:nobody:/home:/bin/false
```

Setting Run-as User

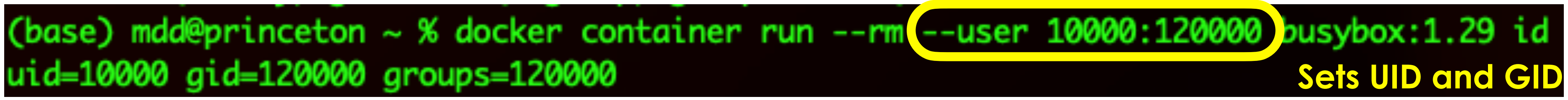
- Once you've identified the user you want to use, you can create a new container with a specific run-as user:
  - Docker provides the **`--user`** or **`--u`** flag on **`container run`** and **`docker container create`** for setting the user.
- The **`--user`** flag can accept any user or group pair: name or UID.
  - When you specify a user by name, that name is resolved to the user ID (UID) specified in the container's passwd file.



```
(base) mdd@princeton ~ % docker container run --rm --user nobody busybox:1.29 id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
```

Setting Run-as User

- Problem: How can you run software in a container as a user that **does not exist** in its underlying Linux distribution.
- Solution: Using the `--user` flag to set the run-as a user (UID) and group (GID) that **do not exist** in the container.
  - When that happens, the IDs won't resolve to a user or group name, but all file permissions will work as if the user and group did exist.
  - Depending on how the software packaged in the container is configured, changing the run-as user may cause problems.



```
(base) mdd@princeton ~ % docker container run --rm --user 10000:120000 busybox:1.29 id
uid=10000 gid=120000 groups=120000
```

Linux User Namespaces

- Linux's user (USR) namespaces can map users in one namespace to users in another
- By default, Docker containers **do not use the USR namespace**, so containers and their hosts share the same user ID space:
  - A container running with a user ID (number, not name) that is the same as a user on the host machine has the same host file permissions as that user.
  - However, the filesystem available inside a container has been mounted so that **changes made inside that container will stay inside that container's filesystem**.
  - But this does **impact volumes** in which files are shared between containers or with the host.

Users and Volumes

- File permissions set by the host for directories mounted as volumes on a container, are respected inside the container.
- Unless you want a file to be accessible to a container, **don't mount it into that container with a volume**.

USR Namespaces & Containers

- When a user namespace is **enabled for a container**, the container's UIDS are re-mapped to a range of unprivileged UIDS on the host.
  - Operators activate user namespace remapping by defining **`subuid`** and **`subgid`** maps for the host in Linux and configuring the Docker daemon's **`userns-remap`** option.
- The mappings determine how user IDs on the host correspond to user IDs in a container namespace.
- E.g.:
  - UID remapping could be configured to map container UIDS to the host starting with host UID 5000 and a range of 1000 UIDS.
  - The result is that UID 0 in containers would be mapped to host UID 5000, container UID 1 to host UID 5001, and so on for 1000 UIDS.
  - Since UID 5000 is an unprivileged user from Linux' perspective and doesn't have permissions to modify the host system files, **the risk of running with uid=0 in the container is greatly reduced**.
  - Even if a containerized process gets ahold of a file or other resource from the host, the containerized process will be running as a remapped UID without privileges to do anything with that resource unless an operator specifically gave it permissions to do so.

Reducing Container Capabilities

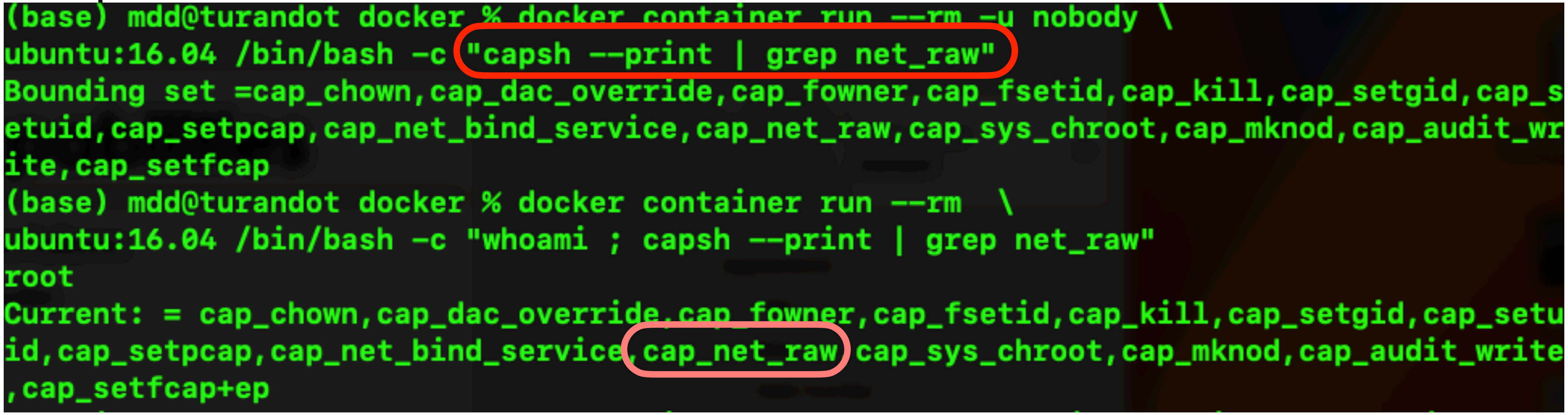
- **Capabilities**: a set of flags associated with a process or file, which determine whether a process was permitted to perform certain actions.
- Docker drops all capabilities for new containers, except an explicit list of capabilities that are necessary and safe to run most applications.
- Purpose: further isolates the running process from the administrative functions of the operating system.

Dropped Capabilities

- A sample of the 37 **dropped** capabilities follows:
- **SYS_MODULE** Insert/remove kernel modules
- **SYS_RA WIO** Modify kernel memory
- **SYS_NICE** Modify priority of processes
- **SYS_RESOURCE** Override resource limits
- **SYS_TIME** Modify the system clock
- **AUDIT_CONTROL** Configure audit subsystem
- **MAC ADMIN** Configure MAC configuration
- **SYSLOG** Modify kernel print behavior
- **NET_ADMIN** Configure the network
- **SYS_ADMIN** Catchall for administrative functions

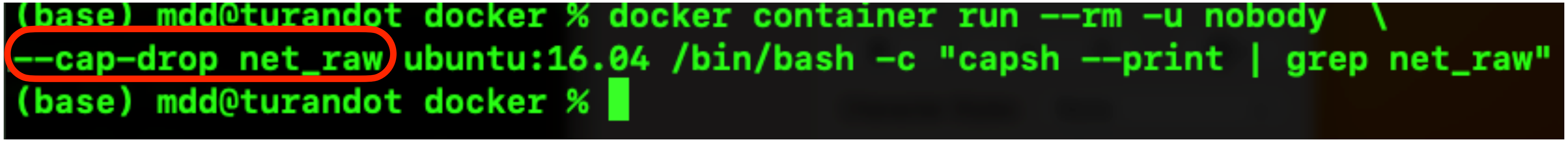
Removing Extra Capabilities

- The default capabilities of a containerized process can be viewed as follows:



```
(base) mdd@turandot docker % docker container run --rm -u nobody \
ubuntu:16.04 /bin/bash -c "capsh --print | grep net_raw"
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_s
etuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_wr
ite,cap_setfcap
(base) mdd@turandot docker % docker container run --rm  \
ubuntu:16.04 /bin/bash -c "whoami ; capsh --print | grep net_raw"
root
Current: = cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setu
id,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write
,cap_setfcap+ep
```

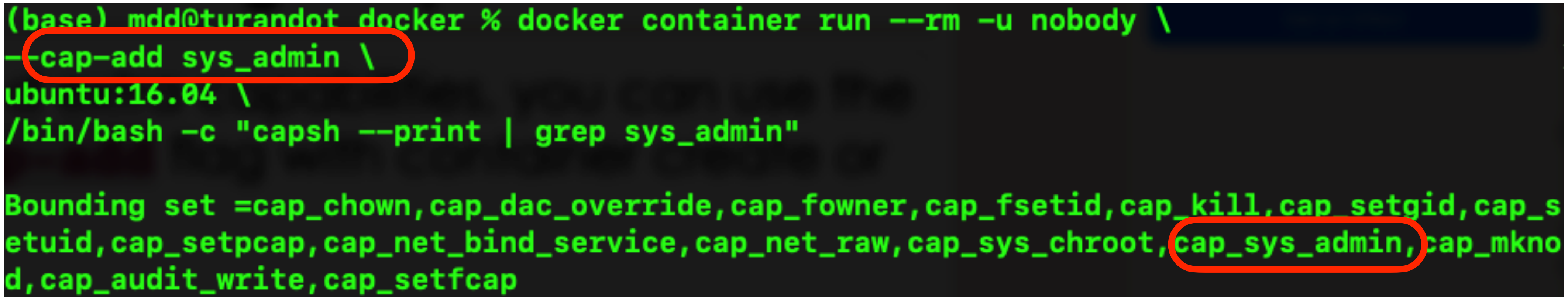
- To drop an extra capability, you can use the **`--cap-drop`** flag with container **`create`** or **`run`**:



```
(base) mdd@turandot docker % docker container run --rm -u nobody  \
--cap-drop net_raw ubuntu:16.04 /bin/bash -c "capsh --print | grep net_raw"
(base) mdd@turandot docker %
```

Adding Capabilities

- To add extra capabilities, you can use the **`--cap-add`** flag with container create or run:



```
(base) mdd@turandot docker % docker container run --rm -u nobody \
--cap-add sys_admin \
ubuntu:16.04 \
/bin/bash -c "capsh --print | grep sys_admin"

Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_s
etuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_sys_admin,cap_mkno
d,cap_audit_write,cap_setfcap
```

- These flags can be used to build containers that will let a process perform **exactly and only what is required for proper operation**.
- E.g. you might be able to run a network management daemon as the **`nobody`** user and give it the **`NET_ADMIN`** capability instead of running it as **`root`** directly on the host or as a privileged container.

Docker API

- The docker command-line program interacts with the Docker daemon almost entirely via the Docker API.
- Any program that can read and write to the Docker API can do anything docker can do, subject to Docker's Authorization plugin system.
  - The Docker daemon API is accessible **via a UNIX domain socket** located on the host at `/var/run/docker.sock`.
  - The **domain socket is protected with filesystem permissions** ensuring that only the **root user** and **members of the docker group** may send commands or retrieve data from the Docker daemon.
  - Some programs are built to interact directly with the Docker daemon API and know how to send commands to inspect or run containers.
- Be careful about which users or programs on your systems can control your Docker daemon:
  - If a user or program controls your Docker daemon, it effectively controls the root account on your host and can run any program or delete any file.

- Docker uses **`cgroups`**, which let a user set memory limits, CPU weight, limits, and core restrictions as well as restrict access to specific devices.
- Docker containers each have their own **IPC namespace** that can be shared with other containers or the host in order to facilitate communication over shared memory.
- Docker supports isolating the **USR namespace**. By default, user and group IDs inside a container are equivalent to the same IDs on the host machine. When the user namespace is enabled, user and group IDs in the container are remapped to IDs that do not exist on the host.
- You can and should use the **`-u`** option on **`docker container run`** and **`docker container create`** to run containers as non-root users.
- Avoid running containers in privileged mode whenever possible.
- Linux capabilities provide operating system feature authorization. Docker drops certain capabilities in order to provide reasonably isolating defaults.
- The capabilities granted to any container can be set with the **`--cap-add`** and **`--cap-drop`** flags.
- Docker provides tooling for integrating easily with enhanced isolation technologies such as seccomp, SELinux, and AppArmor. These are powerful tools that security-conscious Docker adopters should investigate.

Packaging Software for Distribution

Packaging Software in Images

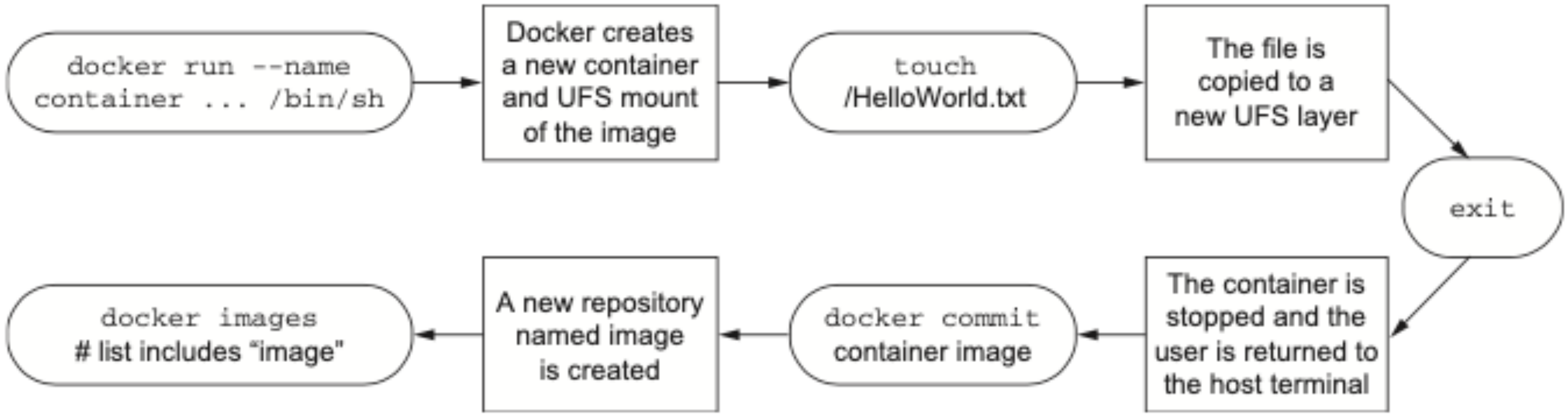
Creating Docker Images

- Alternatives:
  - Modify an existing image inside a container
  - Define and execute a build script called a Dockerfile.

Building Images from Containers

1. Create a container from an existing image.
   - Choose the image based on what you want to be included with the new finished image and the tools you will need to make the changes.
2. Modify the file system of created container.
   - Changes will be written to a new layer on the union file system for the container.
3. Commit those changes.
   - Once the changes are committed, you'll be able to create new containers from the resulting image.

Building Images from Containers


docker run --name container ... /bin/sh
Docker creates a new container and UFS mount of the image
touch /HelloWorld.txt
The file is copied to a new UFS layer
exit
The container is stopped and the user is returned to the host terminal
docker commit container image
A new repository named image is created
docker images # list includes "image"

```
docker commit [OPTIONS] CONTAINER-NAME [REPOSITORY-NAME[:TAG]]
```

File System Changes

- Use the **diff** subcommand to review changes made inside a container's file system, before committing it into a new image:

```
docker container diff image-dev
```

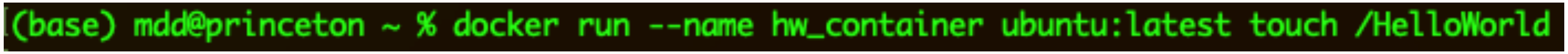
  - Returns large list of directories and files with prefixes A (added), C (changed) or D (deleted)

- When committing the container into a new image, these changes will comprise a new UFS layer added to the new image.

Attribute Changes

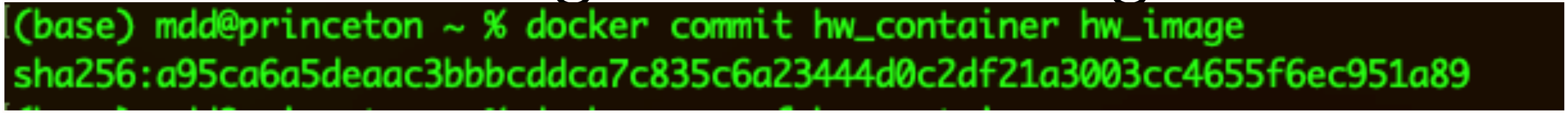
- Besides the new layer added to an image with `docker commit`, a new image carries forward from the container used to commit the new image:
  - All environment variables
  - The working directory
  - The set of exposed ports
  - All volume definitions
  - The container entrypoint
  - Command and arguments
- If these values were not specifically set for the container, the values will be inherited from the original image.

- Create a new container, UFS mount the image, copy a new file to a new UFS layer:



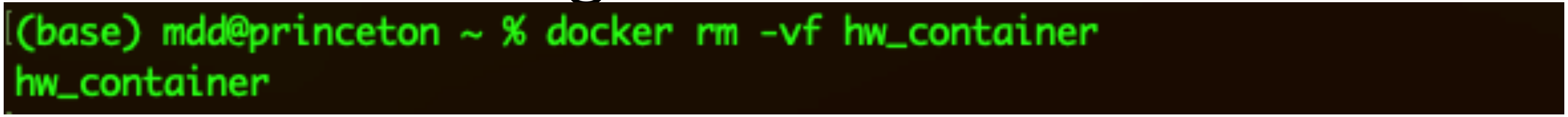
```
(base) mdd@princeton ~ % docker run --name hw_container ubuntu:latest touch /HelloWorld
```

- Commit change to a new image:



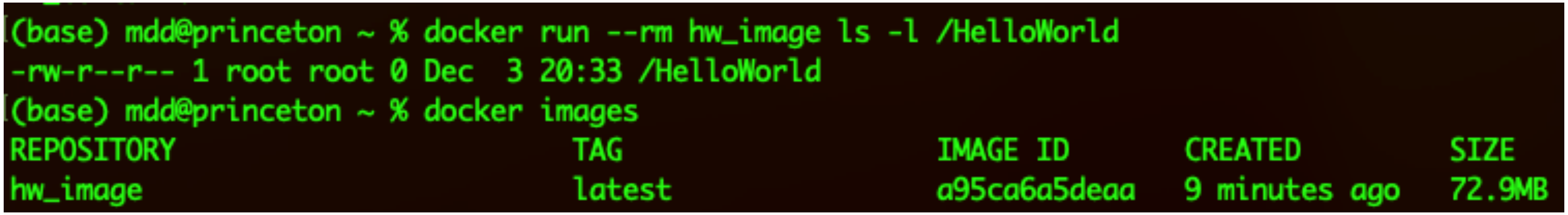
```
(base) mdd@princeton ~ % docker commit hw_container hw_image
sha256:a95ca6a5deaac3bbbcddca7c835c6a23444d0c2df21a3003cc4655f6ec951a89
```

- Remove changed container:



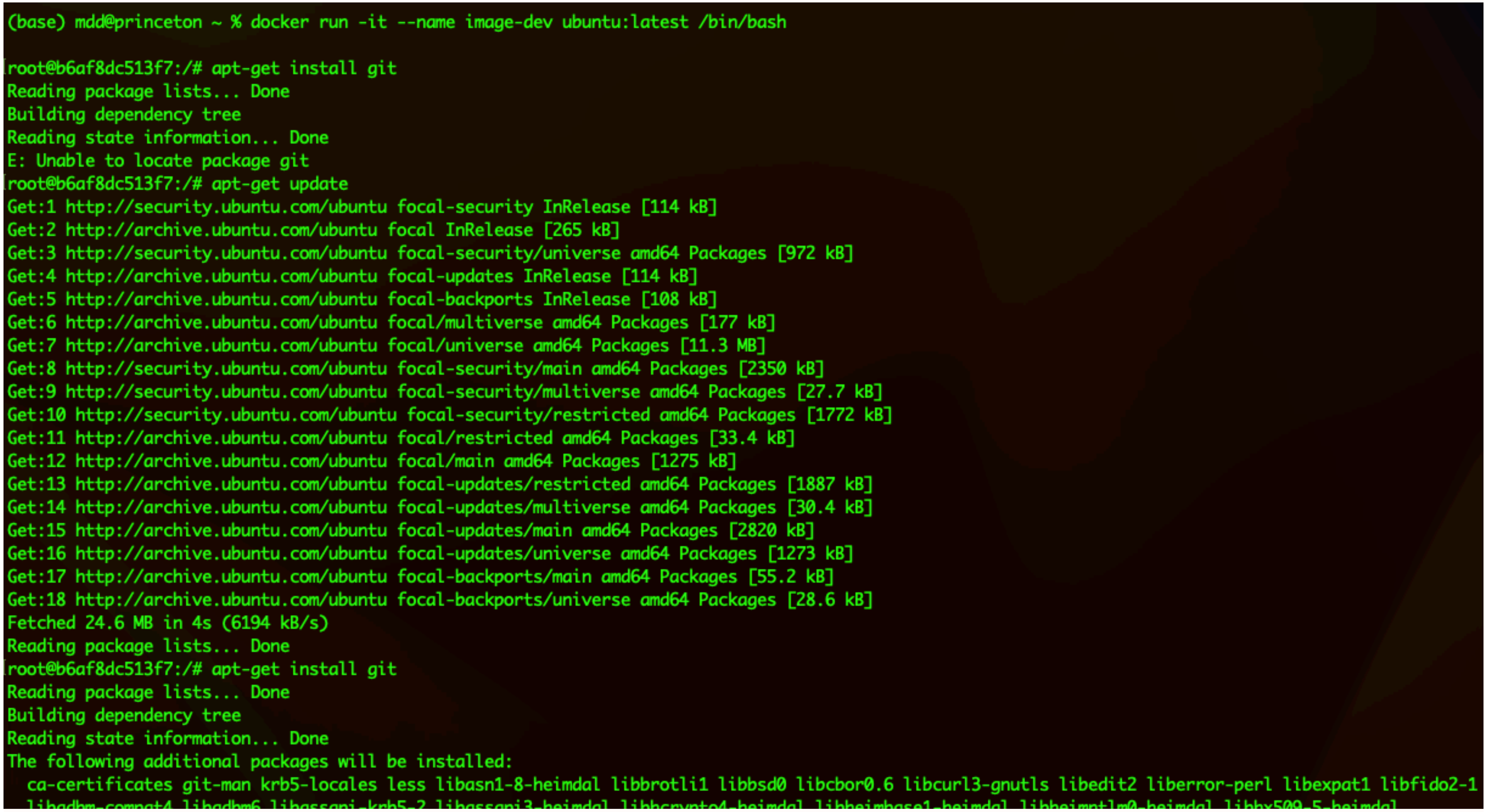
```
(base) mdd@princeton ~ % docker rm -vf hw_container
hw_container
```

- Launch and examine new container:



```
(base) mdd@princeton ~ % docker run --rm hw_image ls -l /HelloWorld
-rw-r--r-- 1 root root 0 Dec  3 20:33 /HelloWorld
(base) mdd@princeton ~ % docker images
REPOSITORY                  TAG                  IMAGE ID        CREATED          SIZE
hw_image                    latest               a95ca6a5deaa    9 minutes ago    72.9MB
```

- Create an image that contains Linux Ubuntu and has installed inside Git:



```
(base) mdd@princeton ~ % docker run -it --name image-dev ubuntu:latest /bin/bash

[root@b6af8dc513f7:/# apt-get install git
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package git
[root@b6af8dc513f7:/# apt-get update
Get:1 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Get:2 http://archive.ubuntu.com/ubuntu focal InRelease [265 kB]
Get:3 http://security.ubuntu.com/ubuntu focal-security/universe amd64 Packages [972 kB]
Get:4 http://archive.ubuntu.com/ubuntu focal-updates InRelease [114 kB]
Get:5 http://archive.ubuntu.com/ubuntu focal-backports InRelease [108 kB]
Get:6 http://archive.ubuntu.com/ubuntu focal/multiverse amd64 Packages [177 kB]
Get:7 http://archive.ubuntu.com/ubuntu focal/universe amd64 Packages [11.3 MB]
Get:8 http://security.ubuntu.com/ubuntu focal-security/main amd64 Packages [2350 kB]
Get:9 http://security.ubuntu.com/ubuntu focal-security/multiverse amd64 Packages [27.7 kB]
Get:10 http://security.ubuntu.com/ubuntu focal-security/restricted amd64 Packages [1772 kB]
Get:11 http://archive.ubuntu.com/ubuntu focal/restricted amd64 Packages [33.4 kB]
Get:12 http://archive.ubuntu.com/ubuntu focal/main amd64 Packages [1275 kB]
Get:13 http://archive.ubuntu.com/ubuntu focal-updates/restricted amd64 Packages [1887 kB]
Get:14 http://archive.ubuntu.com/ubuntu focal-updates/multiverse amd64 Packages [30.4 kB]
Get:15 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 Packages [2820 kB]
Get:16 http://archive.ubuntu.com/ubuntu focal-updates/universe amd64 Packages [1273 kB]
Get:17 http://archive.ubuntu.com/ubuntu focal-backports/main amd64 Packages [55.2 kB]
Get:18 http://archive.ubuntu.com/ubuntu focal-backports/universe amd64 Packages [28.6 kB]
Fetched 24.6 MB in 4s (6194 kB/s)
Reading package lists... Done
[root@b6af8dc513f7:/# apt-get install git
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  ca-certificates git-man krb5-locales less libasn1-8-heimdal libbrotli1 libbsd0 libcbor0.6 libcurl3-gnutls libedit2 liberror-perl libexpat1 libfido2-1
  libgdbm-compat4 libgdbm6 libgssapi-krb5-2 libgssapi3-heimdal libhcrypto4-heimdal libheimbase1-heimdal libheimntlm0-heimdal libhx509-5-heimdal
```

Create an image that contains Linux Ubuntu and has installed inside Git

1. Launch a container named `image-dev`, which runs the `ubuntu:latest` image; inside the container run the bash shell

```
docker run -it --name image-dev ubuntu:latest /bin/bash
```

2. From inside the bash shell, install git:

```
root@b82409d3679a:/# apt-get update; apt-get -y install git
```

3. Commit the container w git in a new image named `ubuntu-git`:

```
docker container commit -m "added git" image-dev ubuntu-git
```

4. Launch the image `ubuntu-git` and test that git runs:

```
% docker run --rm ubuntu-git  git version
git version 2.34.1
```

5. Launch new container `cmd-git` and replace its default command to be executed at launch time with a call to `git --version`.

```
docker run --name cmd-git  --entrypoint git ubuntu-git --version
```

6. Commit the updated container into a new image with the same name ubuntu-git:

```
docker commit -m "Set CMD git"  -a "@dockerinaction" cmd-git ubuntu-git
```

- To create an image from a container, use the docker `commit` command:
  - This commits a new layer to the image of the container.
- The new image inherits as default starting command, the one used by the original container:
  - This will be executed when launching a container from the new image.
- Best practices:
  - use `-a` flag that signs the image with an author string
  - use `-m` flag, which sets a commit message
  - To set a different entrypoint for the new image, create a new container with the `--entrypoint` flag properly set, and then create a new image from that container.

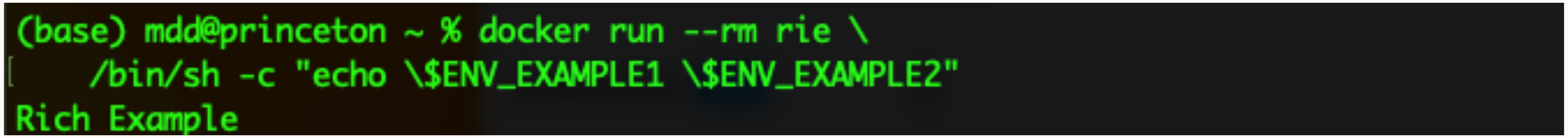
- Create environment variable specialization for container created with busybox image:

```
mdd@princeton ~ % docker run --name rich-image-example \
    -e ENV_EXAMPLE1=Rich -e ENV_EXAMPLE2=Example \
    busybox:latest
```

- Commit new image - no files changed from busybox, just variables:

```
mdd@princeton ~ % docker commit rich-image-example rie
sha256:7ceb042ef889b7612552e71314fa672da4d470599ca647224793111dd48b4e8
```

- Launch container with the new image and check if it has the variables defined:



```
(base) mdd@princeton ~ % docker run --rm rie \
[    /bin/sh -c "echo \$ENV_EXAMPLE1 \$ENV_EXAMPLE2"
Rich Example
```

- Next, consider a container that introduces an entrypoint and command specialization as a new layer on top of the previous example:

```
% docker run --name rich-image-example-2 \
    --entrypoint "/bin/sh" rie -c "echo \$ENV_EXAMPLE1 \$ENV_EXAMPLE2"
Rich Example
% docker commit rich-image-example-2 rie
sha256:c872a1a4f0d6719c5d670919cad41a6a3f32f51d8cd539d38c0b6648c7911079
% docker run --rm rie
Rich Example
```

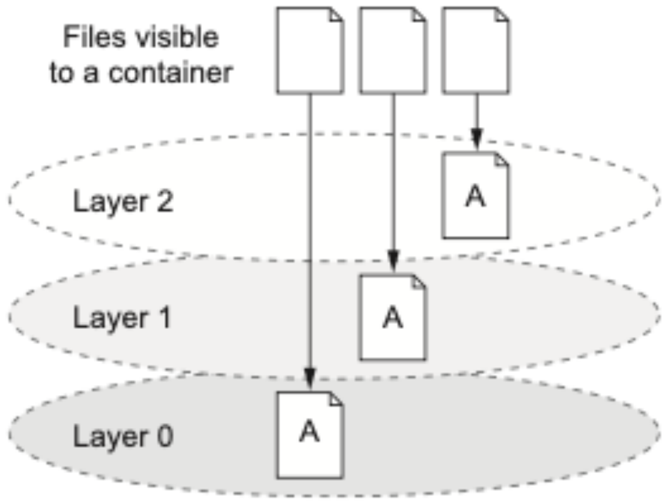
- The two commits build two additional layers on top of BusyBox.
- In neither case are files changed, but the behavior changes because the context metadata has been altered.
- These changes include:
  - Two new environment variables in the first new layer, inherited by the second new layer too.
  - The entrypoint and default command to display the environment variables' values.
- The last command uses the final image without specifying any alternative behavior, but it's clear that the previous defined behavior has been inherited.

Packaging Software in Images

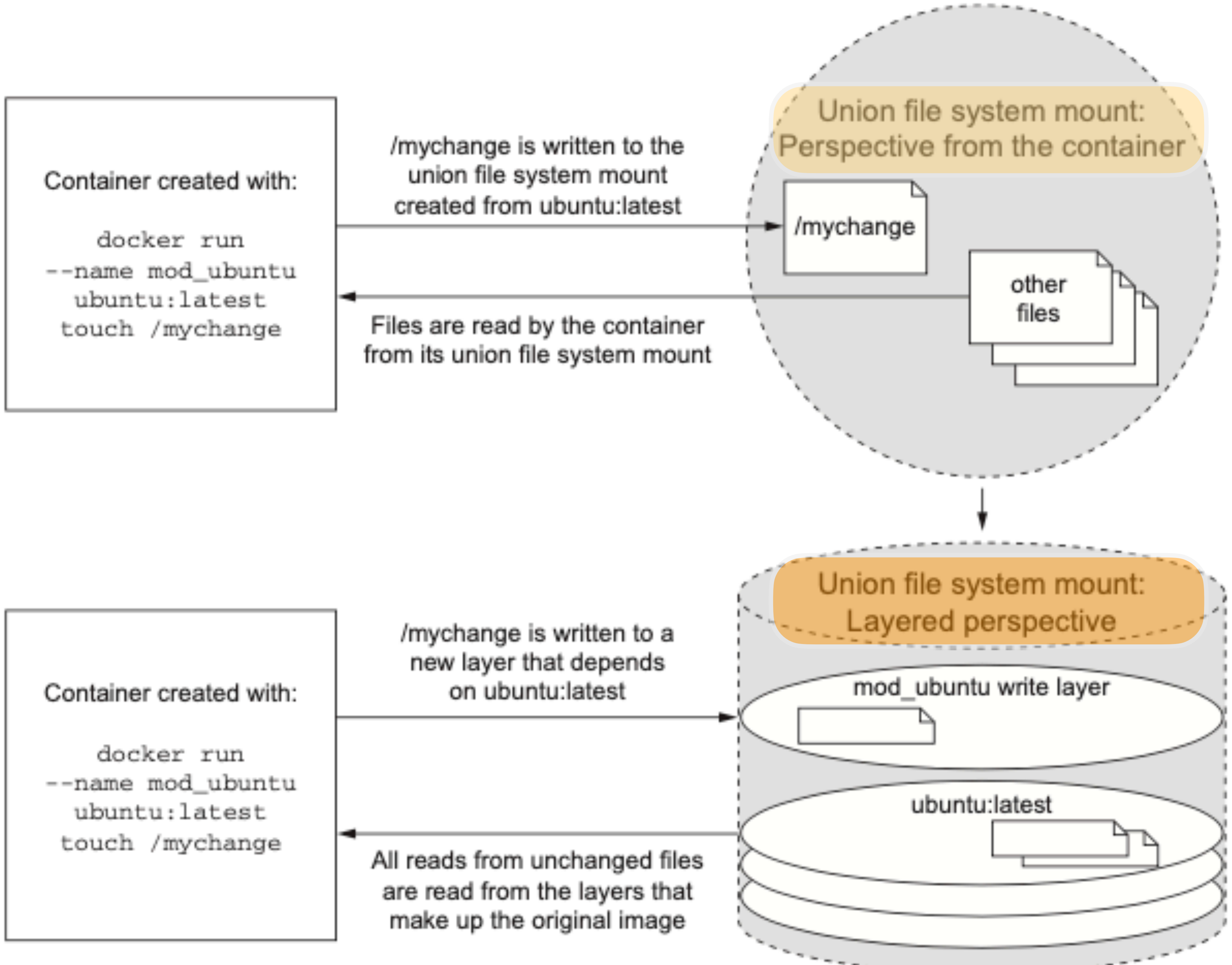
Exploring Union Filesystems

Revisiting the Union File System

- The union file system is made up of layers.
- Each time a change is made to a UFS, that change is recorded on a new layer on top of all of the others.
- The "union" of all of those layers, or top-down view, is what the container (and user) sees when accessing the file system.
- When you read a file from UFS, that file will be read from the top-most layer where it exists.
    - If a file **was not created or changed on the top layer**, the read will fall through the layers until it reaches a layer where that file does exist.
- All this layer functionality is hidden by the UFS. No special actions need to be taken by the software running in a container to take advantage of these features.



A simple file write example on a union file system from two perspectives

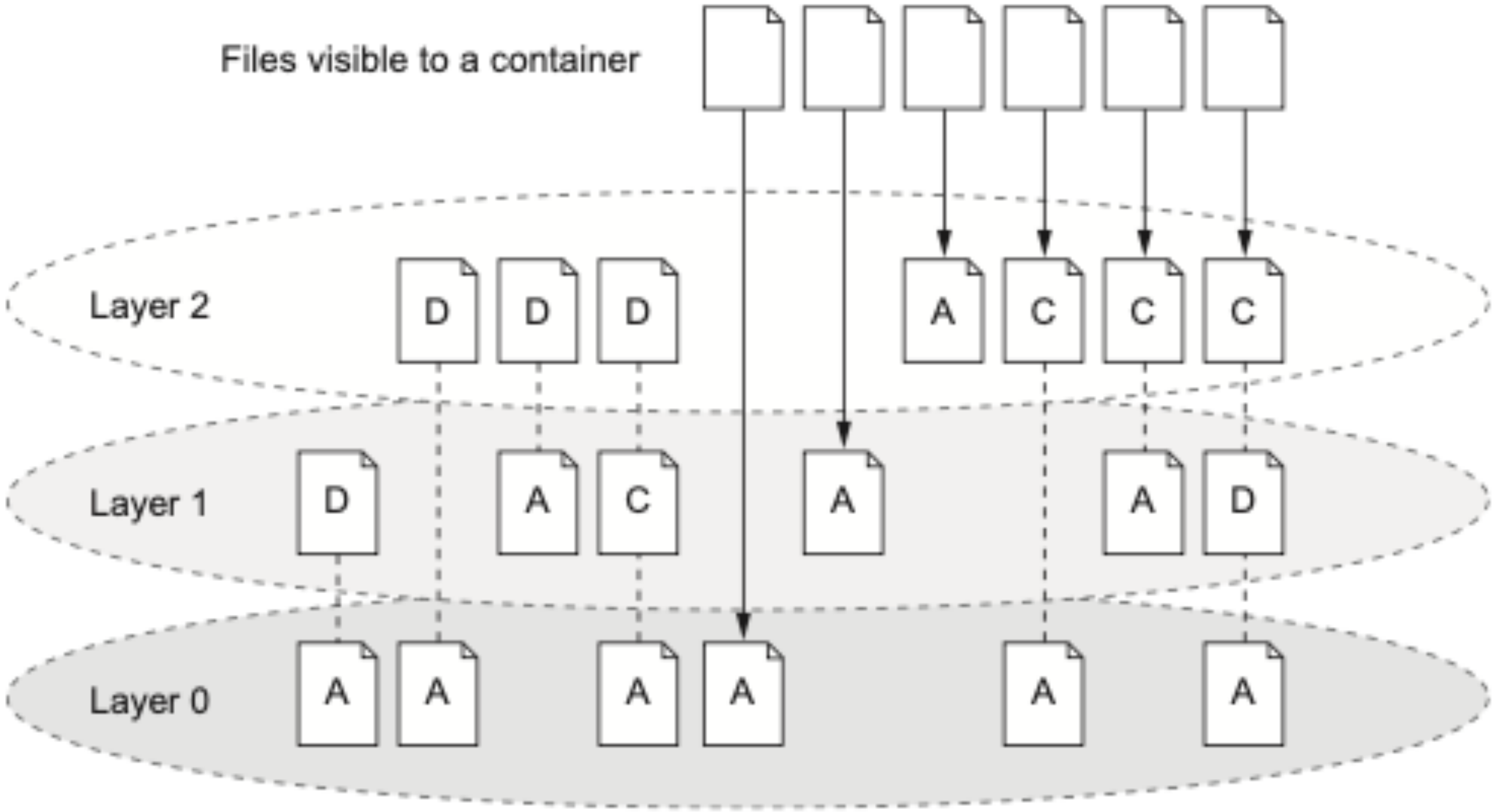

Container created with:
docker run --name mod_ubuntu ubuntu:latest touch /mychange
/mychange is written to the union file system mount created from ubuntu:latest
Files are read by the container from its union file system mount
Union file system mount: Perspective from the container
/mychange
other files
Container created with:
docker run --name mod_ubuntu ubuntu:latest touch /mychange
/mychange is written to a new layer that depends on ubuntu:latest
All reads from unchanged files are read from the layers that make up the original image
Union file system mount: Layered perspective
mod_ubuntu write layer
ubuntu:latest

File Changes and Deletions

- Like additions, both file changes and deletions work by modifying the top layer of the UFS.
- When a file is **deleted**, a delete record is written to the top layer:
  - This record overshadows any versions of that file on lower layers.
- When a file is **changed**, that change is written to the top layer, which again shadows any versions of that file on lower layers.
- Changes to filesystem attributes (file ownership, permissions) are also recorded in the same way, as changes to files.

Copy-on-write

- Most union file systems use the copy-on-write mechanism to implement changes.
- When a file in a read-only layer is modified:
  - The whole file is copied from the read-only layer into the writable layer
  - The change is made on the file
- When you commit the layer, a new ID is generated for it, and copies of all the file changes are saved.
- This approach results in a negative impact on runtime performance and image size.



Various file addition, change, and deletion combinations over a three-layered image

File-change mechanics are the most important thing to understand about union file systems.

Immutable Layers

- All layers below the writable layer created for a container are immutable - they can **never be modified.**
- Consequently:
  - It is possible to share access to images instead of creating independent copies for every container.
  - Individual layers are highly reusable.
  - Anytime you make changes to an image, you need to add a new layer, and **old layers are never removed**.

Image Layer Metadata

- The metadata for an image layer include:
  - The generated ID of the new layer
  - The identifier of the layer below it (parent)
  - The execution context of the container that the layer was created from.
- To review all the layers of an image, you can use the: `docker image history` command. It will give you:
  - Abbreviated layer ID
  - Age of the layer
  - Initial command of the creating container
  - Total file size of that layer

Packaging Software in Images

Revisiting Images

What is an Image?

- An image is a stack of layers constructed by traversing a layer dependency graph from some starting layer - the top of the stack.
- The layer dependency graph comprises:
    - Layers as nodes
    - Links connecting each layer to its parent layer, as represented in its metadata, which contained the parent-layer's ID.
- The layer's ID is also the ID of the image that is derived from it.

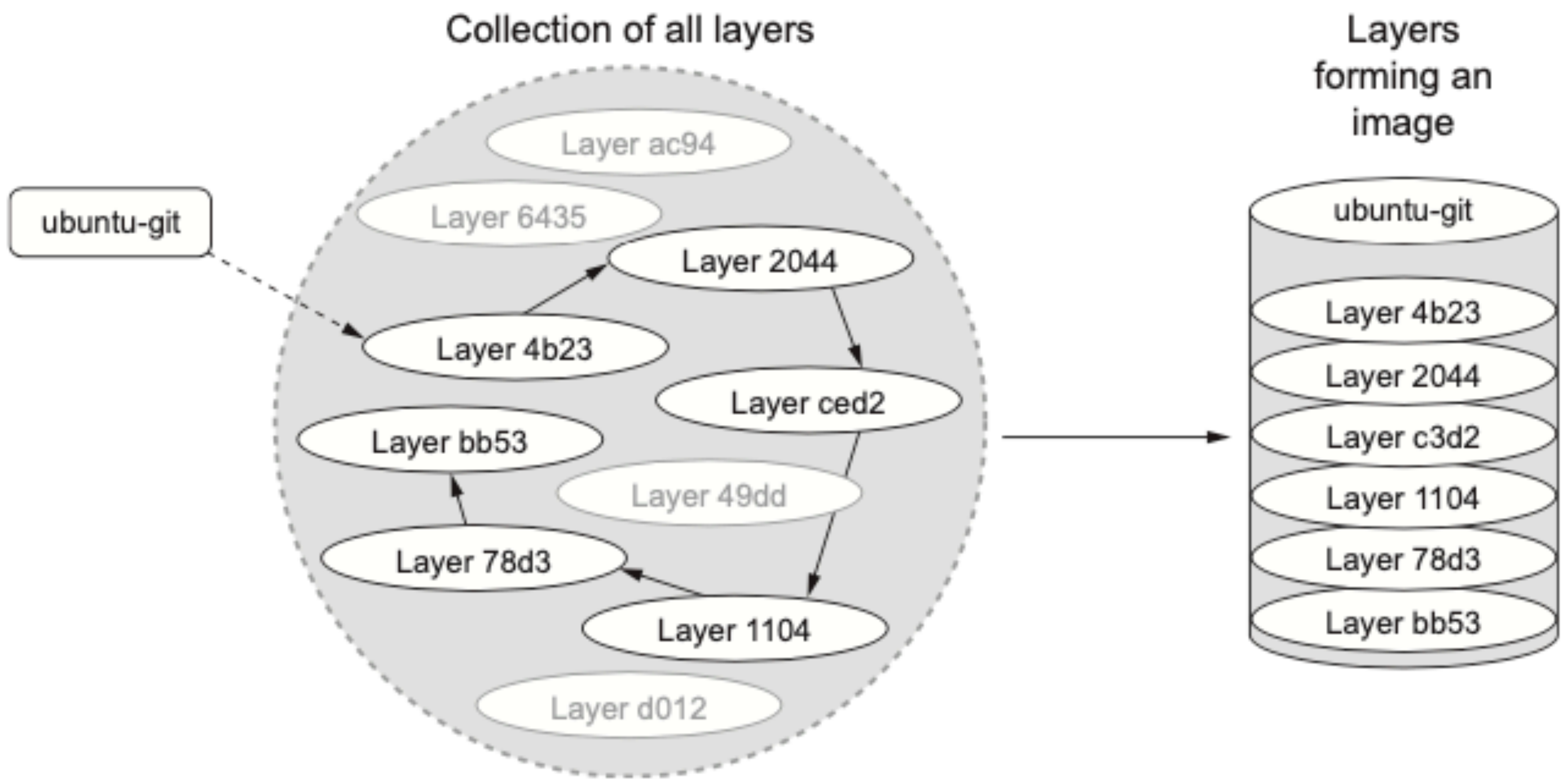
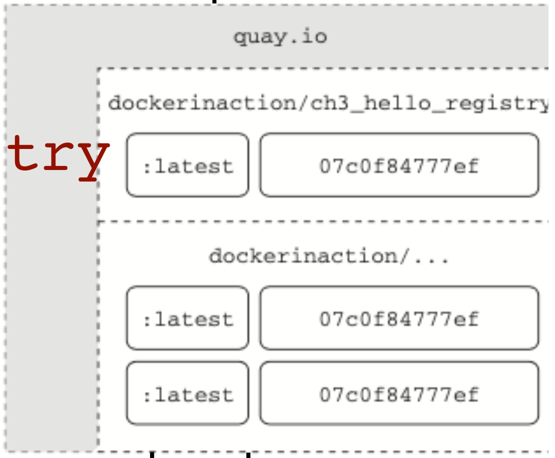


Figure 7.5 An image is the collection of layers produced by traversing the parent graph from a top layer.

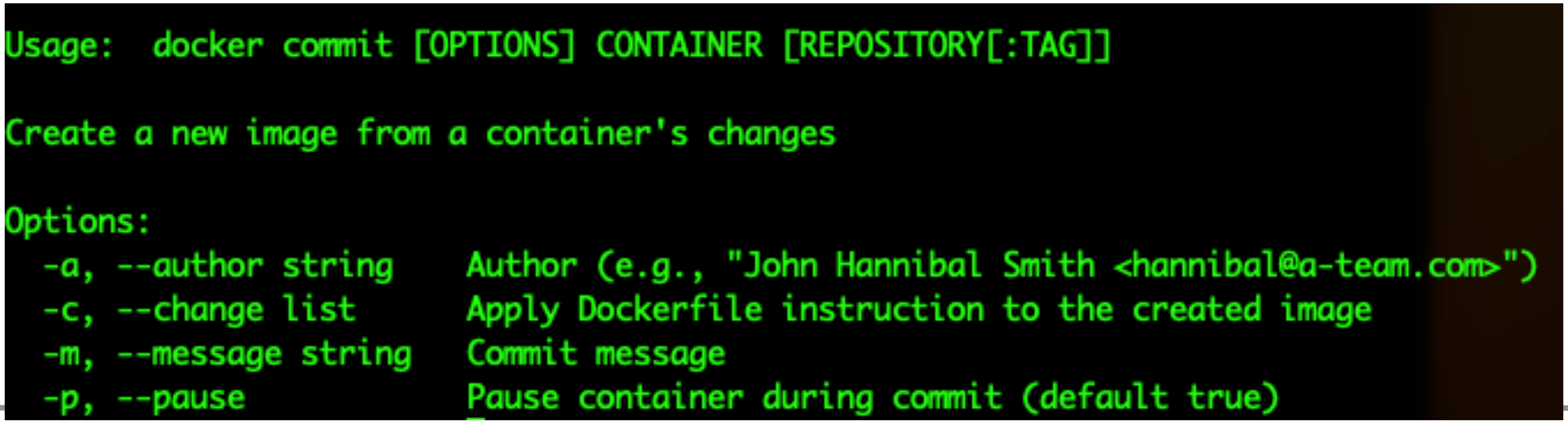
Images & Repositories

- Layer and image IDs are large hexadecimal numbers.
- Docker provides repositories to help users organize their images with **mnemonic names** and **tags**.
- A **repository** is a location/name pair that points to a set of specific layer identifiers.
- E.g.: `quay.io/dockerinaction/ch3_hello_registry`
- This repository is:
  - located in the registry hosted at `quay.io`.
  - named for the user (`dockerinaction`) and a unique short name (`ch3_hello_registry`).
- Pulling this repository would pull all the images defined for each tag in the repository.



Repositories

- Each repository contains:
  - **At least one tag** that points to a specific layer identifier and thus the image definition.
  - A "`latest`" tag by default, if definition of specific tag is omitted at creation time.
- Repositories and tags are created with the `docker tag`, `docker commit`, or `docker build` commands.



```
Usage:  docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

Create a new image from a container's changes

Options:
  -a, --author string    Author (e.g., "John Hannibal Smith <hannibal@a-team.com>")
  -c, --change list      Apply Dockerfile instruction to the created image
  -m, --message string   Commit message
  -p, --pause            Pause container during commit (default true)
```

- Launch a container, add a file mychange in it, check images installed on your computer and verify the change done in the container:

```
% docker run --name mod_ubuntu ubuntu:latest touch /mychange
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
6e3729cf69e0: Pull complete
Digest: sha256:27cb6e6ccef575a4698b66f5de06c7ecd61589132d5a91d098f7f3f9285415a9
Status: Downloaded newer image for ubuntu:latest
% docker images
REPOSITORY    TAG       IMAGE ID       CREATED       SIZE
ubuntu        latest    6b7dfa7e8fdb   10 days ago   77.8MB
% docker diff mod_ubuntu
A /mychange
```

- Commit the altered container, creating an un-named image:

```
% docker commit mod_ubuntu
sha256:36cb491a770772292b7f486abafc39bcbf34a9b0a8427b6397c9bd4453fdc49c
% docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
<none>        <none>    36cb491a7707   4 seconds ago   77.8MB
ubuntu        latest    6b7dfa7e8fdb   10 days ago     77.8MB
```

- Commit again the altered container, but under a given name and tag:

```
docker commit mod_ubuntu myuser/myfirstrepo:mytag
sha256:5265e062504be610bad61a18b699a162dc7c92b8d8a201c8c370148736449d1f
```

- You now have another repository with a new image ID:

```
% docker images
REPOSITORY           TAG       IMAGE ID       CREATED          SIZE
myuser/myfirstrepo   mytag     5265e062504b   3 seconds ago    77.8MB
<none>               <none>    36cb491a7707   34 seconds ago   77.8MB
ubuntu               latest    6b7dfa7e8fdb   10 days ago      77.8MB
```

- Use docker tag to create a repository from `myuser/myfirstrepo:mytag` with the name `myuser/mod_ubuntu`. Since a tag is not specified, it takes by default the tag `latest`:

```
docker tag myuser/myfirstrepo:mytag myuser/mod_ubuntu
(base) mdd@princeton ~ % docker images
REPOSITORY           TAG       IMAGE ID       CREATED              SIZE
myuser/mod_ubuntu    latest    5265e062504b   About a minute ago   77.8MB
myuser/myfirstrepo   mytag     5265e062504b   About a minute ago   77.8MB
<none>               <none>    36cb491a7707   2 minutes ago        77.8MB
ubuntu               latest    6b7dfa7e8fdb   10 days ago          77.8MB
```

Title Text

Container Management Frameworks

Application-oriented Infrastructure

Container technology enables the development of management APIs around containers instead of machines.

- App developers and operations teams relieved from having to worry about specific details of machines & OS.
- Infrastructure teams have the flexibility to roll out new hardware and upgrade OS with minimal impact on running apps & their developers.
- Ties telemetry collected by the management system to applications rather than machines:
  - No need to demultiplex signals from multiple apps running inside a physical or virtual machine.
  - Dramatically improves application monitoring and introspection.
- Management system can communicate information into the container:
  - Resource limits, container metadata for propagation to logging & monitoring, termination warnings

Applications as Containers

- In reality, an application does not "consume" only one container.
- Applications use nested-containers that are co-scheduled on the same machine:
  - Outer-most container is called a resource allocation: ***alloc*** in Borg, ***pod*** in Kubernetes
  - Major part of the application sits in one of the child containers
  - Other child containers run supporting functions
- Advantages: robustness, composability, fine-grained resource isolation

Container Management Services

- Basic services provide resource orchestration and allocation, application configuration and control, monitoring, load balancing.
- Additional services arising:
  - Naming and service discovery (the Borg Name Service, or BNS).
  - Master election.
  - Application-aware load balancing.
  - Horizontal (number of instances) and vertical (size of an instance) autoscaling.
  - Rollout tools that manage the careful deployment of new binaries and configuration data.
  - Workflow tools (e.g., to allow running multijob analysis pipelines with interdependencies between the stages).
  - Monitoring tools to gather information about containers, aggregate it, present it on dashboards, and use it to trigger alerts

Overview

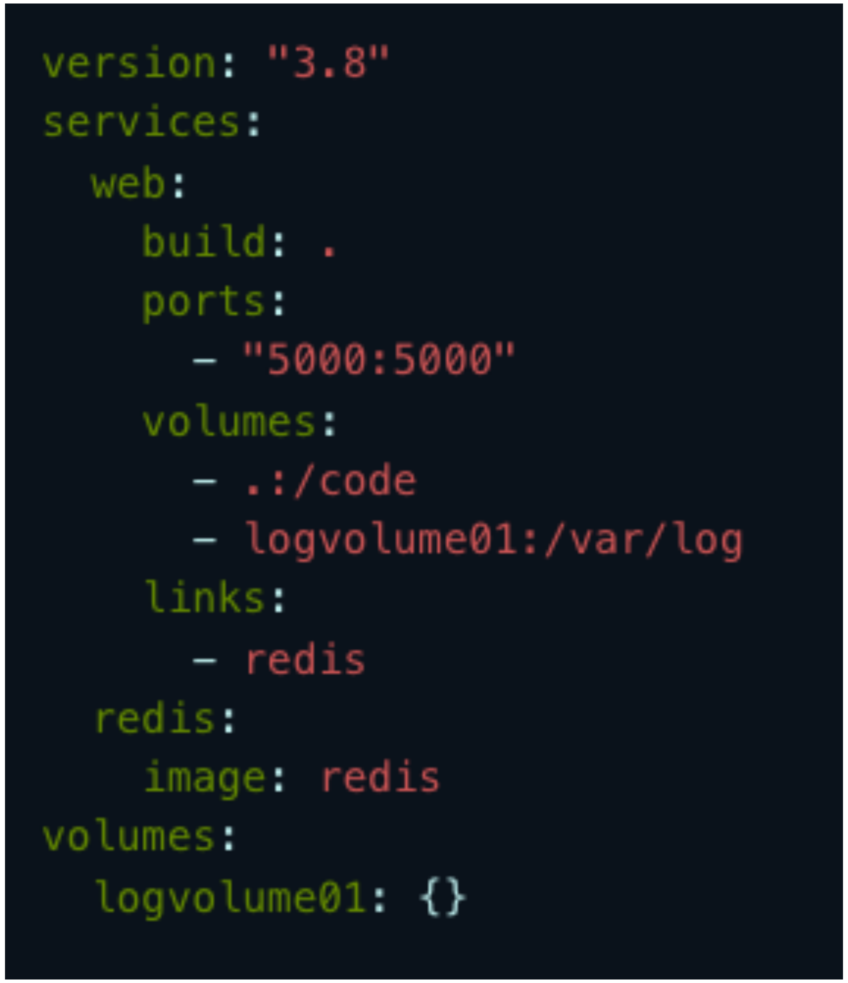
- **Docker compose**
- **Docker swarm** is a container orchestration tool, meaning that it allows the user to manage multiple containers deployed across multiple host machines. One of the key benefits associated with the operation of a docker swarm is the high level of availability offered for applications.
- **Borg**
- **Omega**
- **Kubernetes**

Docker Compose

- A tool for defining and running multi-container Docker applications.
- Using Compose is a three-step process:
  - Define your app's environment with a `Dockerfile` so it can be reproduced anywhere.
  - Define the services that make up your app in a YAML file `docker-compose.yml` so they can be run together in an isolated environment.
  - Run `docker-compose up` and Compose starts and runs your entire app.
- Docker Compose features that make it effective are:
  - Support for multiple isolated environments on a single host
  - Preserve all volumes used by your services when containers are created
  - Only recreate containers that have changed
  - Support for variables in the Compose file, and moving a composition between environments

Docker Compose and YAML

- Docker Composes uses YAML ("YAML Ain't Markup Language") to describe is a human-readable data-serialization language.
- Commonly used for configuration files and in applications where data is being stored or transmitted.



```yaml
version: "3.8"
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Docker Swarm

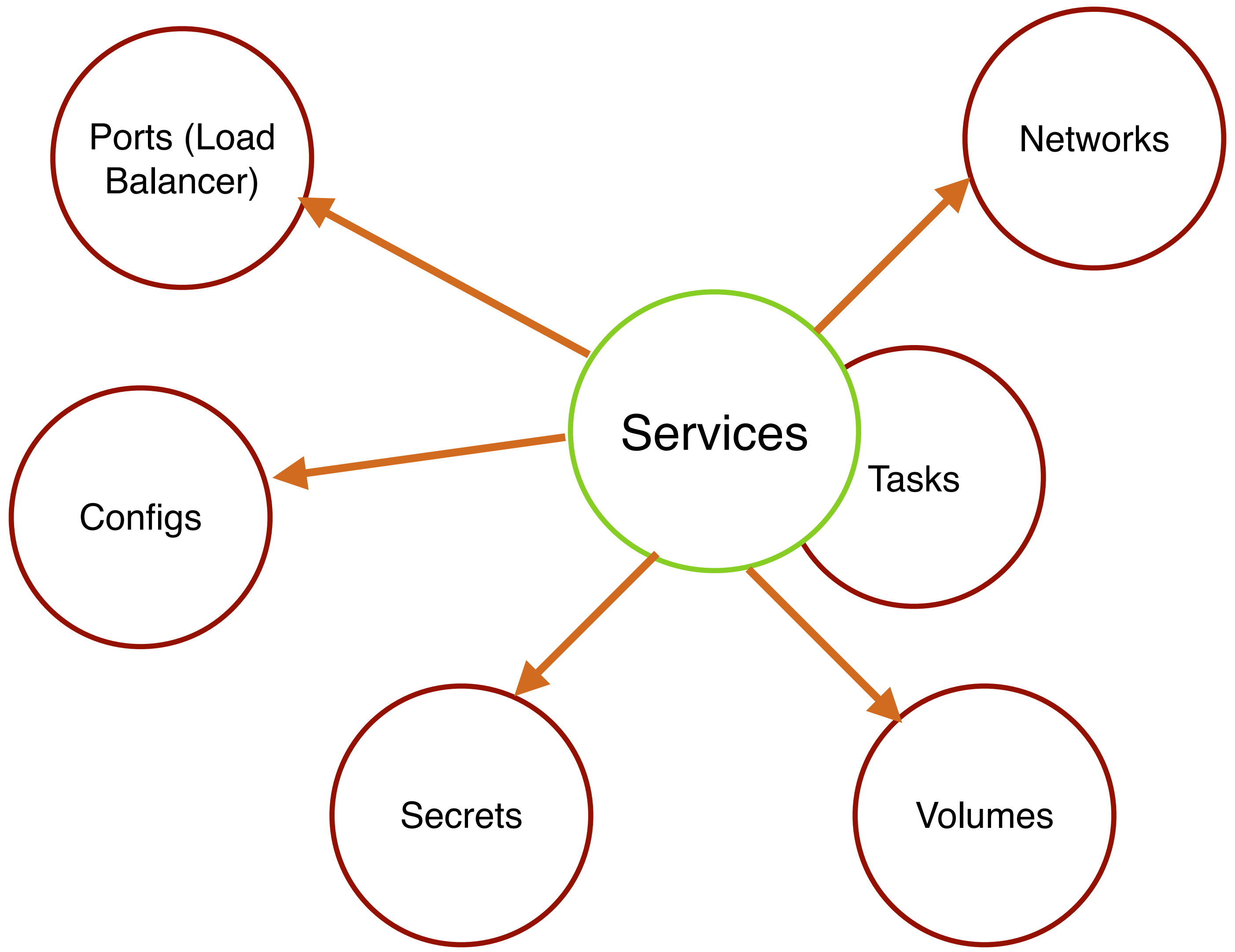
- A container orchestration tool that allows the user to manage multiple containers deployed across multiple host machines.
  - Included in Docker Engine & command-line tool.
- Provides a platform for deploying and operating a containerised application across a set of Docker hosts.
- Automates the process of deploying:
  - A new Docker service to the cluster
  - Changes to an existing service
- Supervises deployed applications to detect and repair possible problems.
- Schedules tasks according to the application's resource requirements and machine capabilities.
- Routes user requests to service containers.

Docker Swarm Structure

Machines joining a Swarm cluster can be Managers or Workers.

- Managers:
  - Listen for instructions to create, change, remove definitions for Docker services, configuration and secrets.
  - Instruct worker nodes to create containers and volumes that implement Docker service instances.
  - At least one manager per cluster - production clusters have 3-5.
- Workers: clusters can scale reliably to hundreds of worker nodes.
- Client requests may be sent to any node of the cluster on the port published for that service.
  - Swarm's network mesh routes the request from whichever node received the request to a healthy service container that can handle it.
- Swarm deploys and manages a load-balancer and network traffic components to receive and transport network traffic for each published port.

Docker Swarm Resource Types



Docker Swarm Resource Types

- Services—A Docker service defines the application processes that run on the Swarm cluster's nodes. Swarm managers interpret the service definition and create tasks that are executed on the cluster's manager and worker nodes.
- Tasks—Tasks define a containerized process that Swarm will schedule and run once until completion.
  - Restart policies
  - Dependencies
- Networks—Applications can use Docker overlay networks for traffic between services.
  - Docker networks have low overhead, so you can create network topologies that suit your desired security model.
- Volumes—Volumes provide persistent storage to service tasks.
  - Volumes are bound to a single node.
- Configs and secrets— provide environment—specific configurations to services deployed on the cluster.

Borg

- **Borg**: unified container-management system, built at Google to manage both:
  - long-running services
  - batch jobs.
- Expanded with mechanisms for:
  - configuring and updating jobs;
  - predicting resource requirements;
  - dynamically pushing configuration files to running jobs;
  - service discovery and load balancing;
  - auto-scaling;
  - machine- lifecycle management;
  - quota management etc

Omega

- Built from the ground up to have a more consistent, principled architecture than Borg.
- Stored the state of the cluster in a centralized Paxos-based transaction-oriented store that was accessed by the different parts of the cluster control plane (such as schedulers).
- Decoupling, allowed the Borgmaster's functionality to be broken into separate components that acted as peers.
- Omega's innovations folded into Borg.

Paxos

- Family of protocols for solving consensus in a network of unreliable or fallible processors.
- Consensus is the process of agreeing on one result among a group of participants.
- This problem becomes difficult when the participants or their communications may experience failures.

Kubernetes

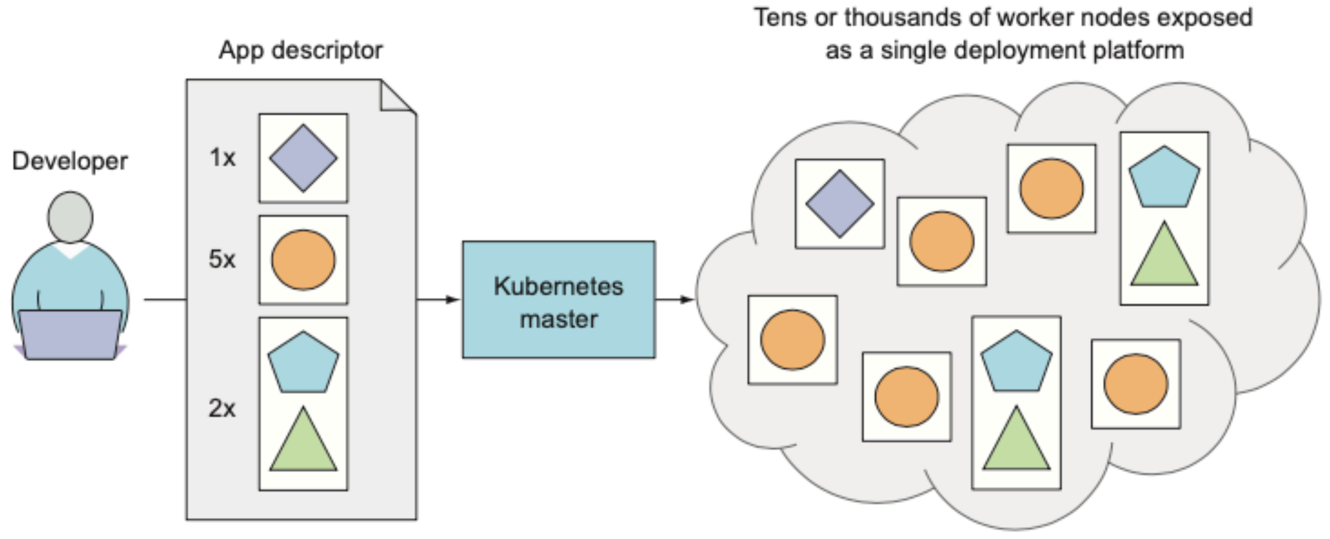
- Open source container management system: emerged from experiences with Borg and Omega.
- Allows you to easily deploy and manage containerized applications on top of it.
- Relies on the features of Linux containers to:
  - Run heterogeneous applications without having to know their internal details.
  - Deploy automatically these applications on each host.
- At its core a shared persistent store, with components watching for changes to relevant objects.
- State in Kubernetes is accessed exclusively through a domain-specific REST API.

Kubernetes abstraction

- Kubernetes enables you to run your software applications on thousands of computer nodes as if all those nodes were a single, enormous computer.
- It abstracts away the underlying infrastructure and simplifies development, deployment, and management for both development and the operations teams.
- Deploying applications through Kubernetes is always the same, whether your cluster contains only a couple of nodes or thousands of them.

Kubernetes basic view

- The system is composed of a master node and any number of worker nodes.
- When the developer submits a list of apps to the master, Kubernetes deploys them to the cluster of worker nodes.


App descriptor
Tens or thousands of worker nodes exposed as a single deployment platform
Developer
1x
5x
2x
Kubernetes master

on doesn't
to the
ministrator.

Figure 1.8 Kubernetes exposes the whole datacenter as a single deployment platform.

Kubernetes functionality

- Kubernetes can be thought of as an operating system for the cluster.
  - It relieves application developers from having to implement certain infrastructure-related services into their apps; instead they rely on Kubernetes to provide these services.
  - Application developers can therefore focus on implementing the actual features of the applications and not waste time figuring out how to integrate them with the infrastructure.
- Kubernetes **services**:
  - service **discovery**
  - **scaling**
  - load-**balancing**
  - self-**healing**, and even
  - leader **election**.

Architecture of a Kubernetes cluster

- The Kubernetes master node hosts the Kubernetes Control Plane that controls and manages the whole Kubernetes system.
- The Kubernetes worker nodes run the actual applications you deploy.

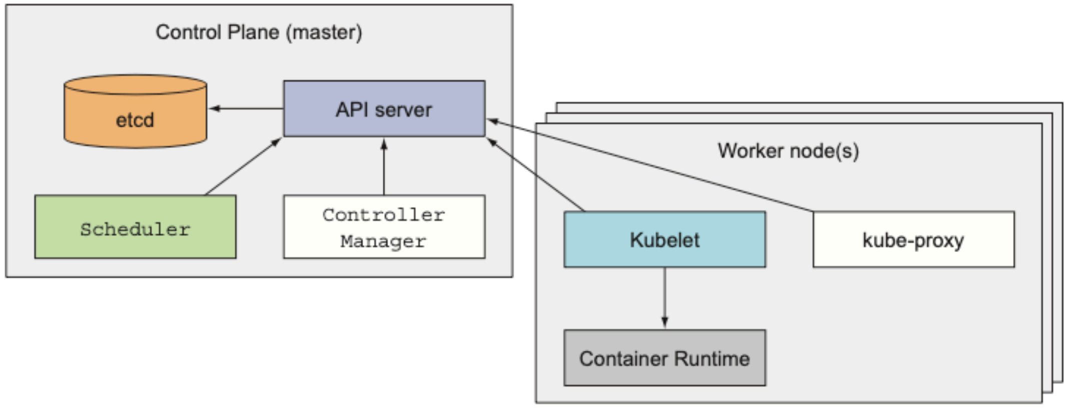


Figure 1.9 The components that make up a Kubernetes cluster

Kubernetes Control Plane

- Controls the cluster and makes it function.
- It consists of multiple components that can run on a single master node or be split across multiple nodes and replicated to ensure high availability:
- The Kubernetes API Server, which you and the other Control Plane components communicate with.
- The Scheduler, which schedules your apps (assigns a worker node to each deployable component of your application)
- The Controller Manager, which performs cluster-level functions, such as replicating components, keeping track of worker nodes, handling node failures, and so on
- etcd, a reliable distributed data store that persistently stores the cluster configuration.

Kubernetes Worker Nodes

- The machines that run your containerized applications.
- The task of running, monitoring, and providing services to your applications is done by the following components:
  - Docker, rkt, or another container runtime, which runs your containers
  - The Kubelet, which talks to the API server and manages containers on its node
  - The Kubernetes Service Proxy (kube-proxy), which load-balances network traffic between application components

Running Applications on Kubernetes

1. Package you application up into one or more container images.
2. Push those images to an image registry
3. Post a description of your app to the Kubernetes API server.

- The description includes information such as:
  - The container image or images that contain your application components.
  - How those components are related to each other
  - Which components need to be run co-located and which don't.
  - For each component, you can also specify how many copies (or replicas) you want to run.
  - Which of those components provide a service to either internal or external clients and should be exposed through a single IP address and made discoverable to the other components.

From Description to Running Containers

- When the API server processes an app description, the Scheduler schedules the specified groups of containers onto the available Worker Nodes.
  - The Scheduler takes into account the computational resources required by each group and the unallocated resources on each node at that moment.
- The Kubelets on those nodes then instruct the Container Runtime (Docker, for example) to pull the required container images and run the containers.
- Once the application is running, Kubernetes continuously makes sure that the deployed state of the application always matches the description you provided.
  - For example, if you specify that you always want five instances of a web server running, Kubernetes will always keep exactly five instances running.
  - If one of those instances stops working properly, like when its **process crashes** or when it **stops responding**, Kubernetes will restart it automatically.
  - Similarly, if a whole worker node dies or becomes inaccessible, Kubernetes will select new nodes for all the containers that were running on the node and run them on the newly selected nodes.

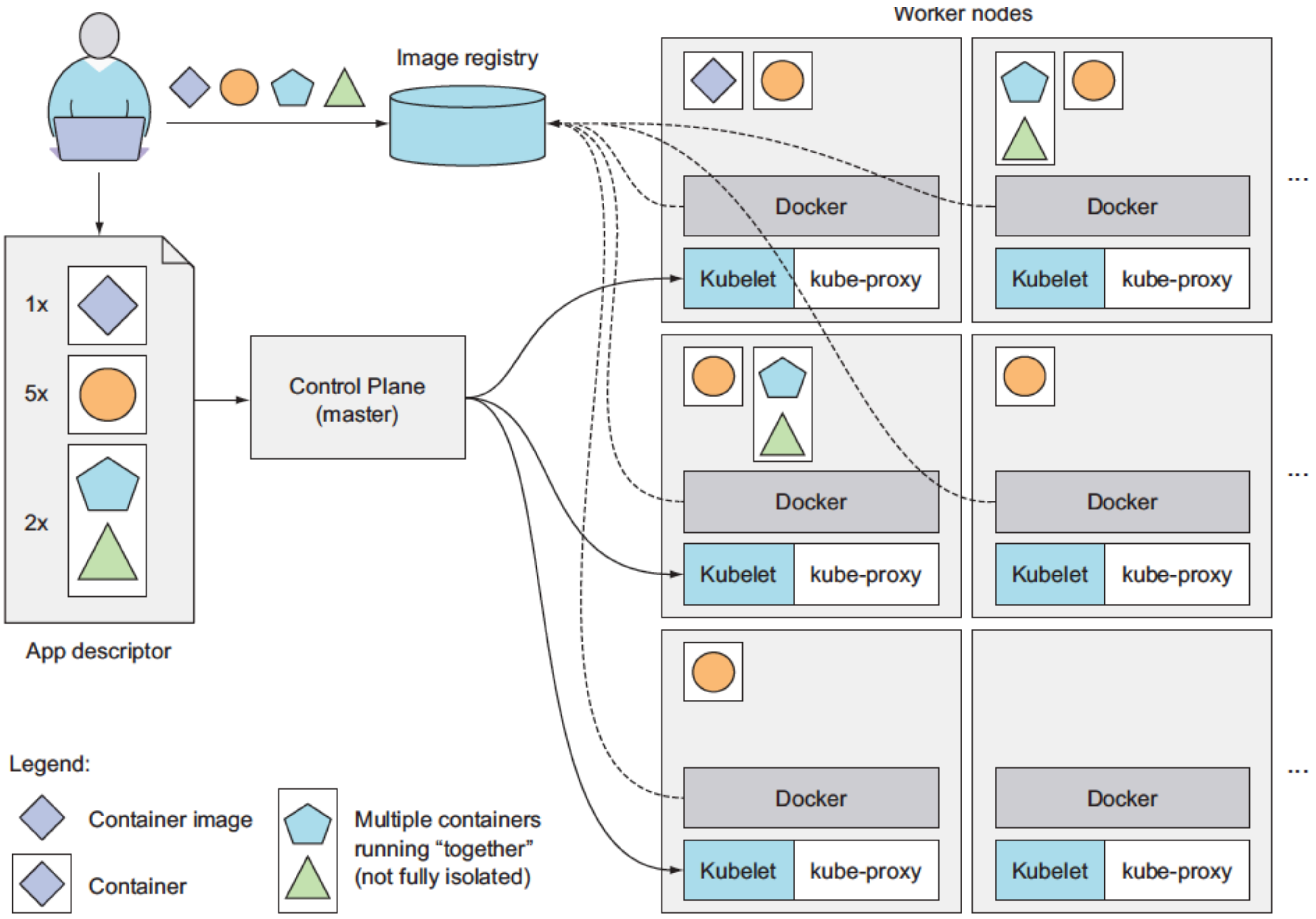


Figure 1.10 A basic overview of the Kubernetes architecture and an application running on top of it

Scaling and Moving Around

- While the application is running, you can decide you want to increase or decrease the number of copies of running containers:
  - Kubernetes will spin up additional ones or stop the excess ones, respectively.
- Alternatively, Kubernetes can decide itself the optimal number of copies:
  - It can automatically keep adjusting the number, based on real-time metrics, such as CPU load, memory consumption, queries per second, or any other metric your app exposes.
  - It can move containers around the cluster, when a node they were running on has failed or when they are evicted from a node to make room for other containers.

Service interface

- If the container is providing a service to external clients or other containers running in the cluster, how can they use the container properly if it's constantly moving around the cluster?
- And how can clients connect to containers providing a service when those containers are replicated and spread across the whole cluster?
- To allow clients to easily find containers that provide a specific service, you can tell Kubernetes which containers provide the same service and Kubernetes will expose all of them at a single static IP address and expose that address to all applications running in the cluster.
    - This is done through environment variables, but clients can also look up the service IP through good old DNS.
    - The kube-proxy will make sure connections to the service are load balanced across all the containers that provide the service.
    - The IP address of the service stays constant, so clients can always connect to its containers, even when they're moved around the cluster.