Docker Administration

Gaurav

RHCSA | RHCE | CKA | SCA | SCE | SCA+ HA | ANSIBLE | DOCKER | OPENSHIFT | OPEN SOURCE TECHNOLOGIES

Course Agenda

- Introducing Docker
- Installation of Docker
- Docker Client Operations
- Building Custom Images and Docker Registry
- Container Deep Dive
- Storage & Container Networking

KOENIG

VIRTUAL MACHINE VS CONTAINER

What's the difference?

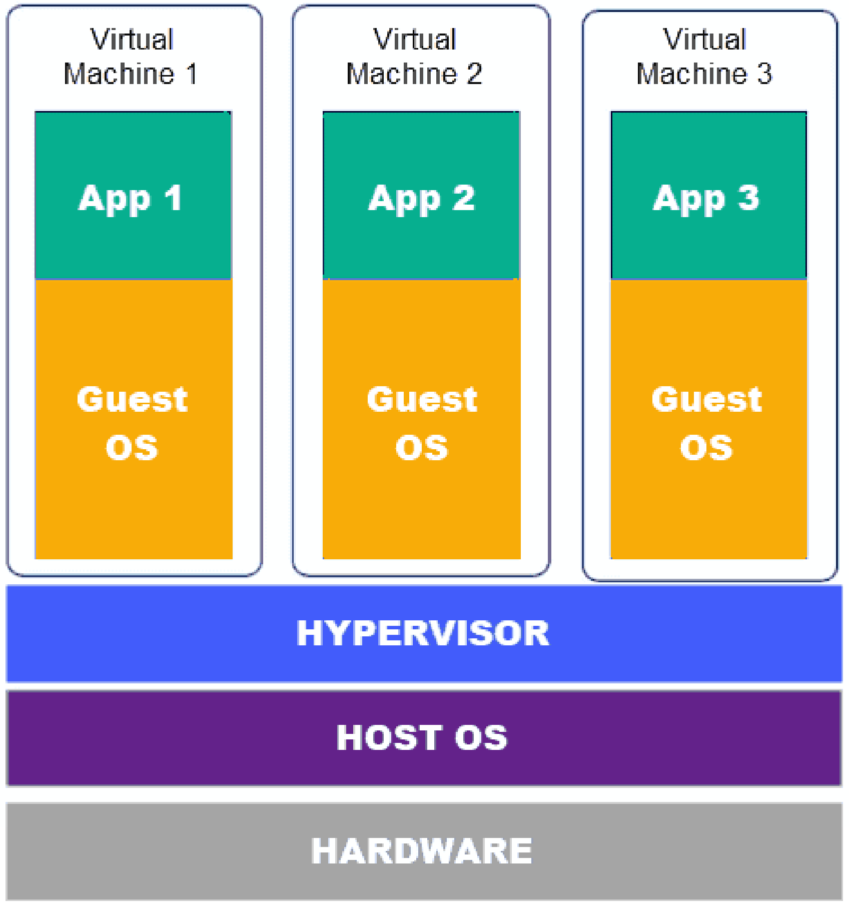
KOENIG

What are Virtual Machines?

A **virtual machine (VM)** is an operating system that shares the physical resources of one server. It is a guest on the host’s hardware, which is why it is also called a **guest machine**.

How a Virtual Machine Works?

Everything necessary to run an app is contained within the virtual machine – the virtualized hardware, an OS, and any required binaries and libraries. Therefore, virtual machines have their own infrastructure and are self-contained.



KOENIG

Virtual Machine: Advantage

VMs **reduce expenses**. Instead of running an application on a single server, a virtual machine enables utilizing one physical resource to do the job of many. Therefore, you do not have to buy, maintain and store enumerable stacks of servers.

Because there is one host machine, it allows you to **efficiently manage** all the virtual environments with the centralized power of the hypervisor. These systems are entirely separate from each other meaning you can install **multiple system environments**.

Most importantly, a virtual machine is isolated from the host OS and is a **safe** place for experimenting and developing applications.

Virtual Machine: Disadvantage

Virtual machines may take up a lot of system resources of the host machine, being many GBs in size. Running a single app on a virtual server means running a copy of an operating system as well as a virtual copy of all the hardware required for the system to run. This quickly adds up to a lot of RAM and CPU cycles.

The process of **relocating an app running on a virtual machine can also be complicated** as it is always attached to the operating system. Hence, you have to migrate the app as well as the OS with it. Also, when creating a virtual machine, the hypervisor allocates hardware resources dedicated to the VM.

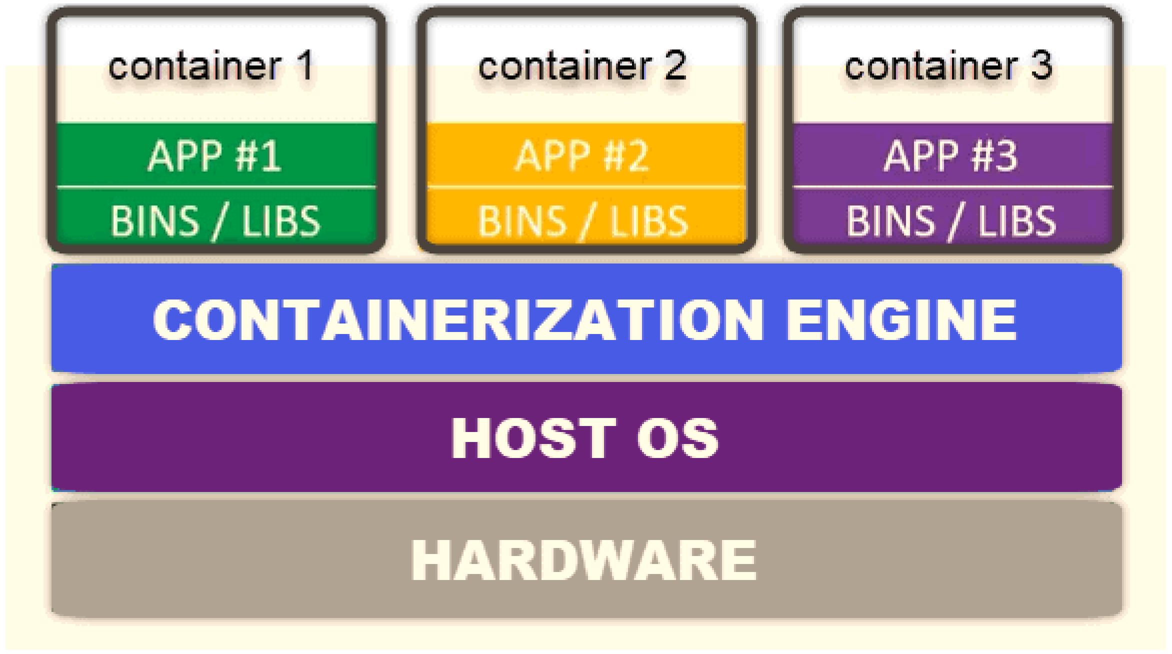
A virtual machine rarely uses all the resources available which can **make the planning and distribution difficult.** That's still economical compared to running separate actual computers.

What is a Container?

A container is an environment that runs an application that is not dependent on the operating system. It isolates the app from the host by virtualizing it. This allows users to created multiple workloads on a single OS instance.

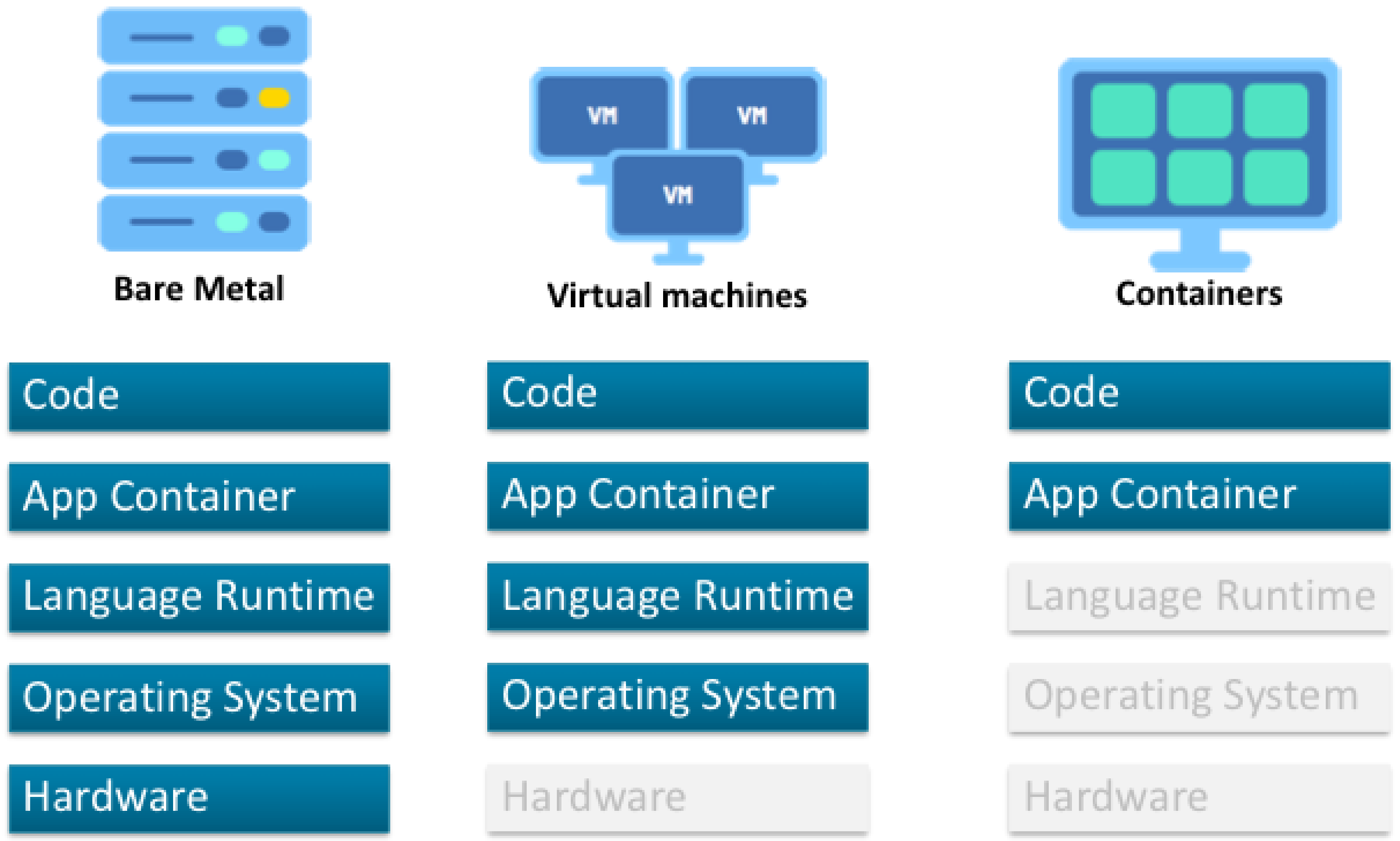
How do Containers Work?

Containers are a way for developers to easily package and deliver applications, and for operations to easily run them anywhere in seconds, with no installation or setup necessary.



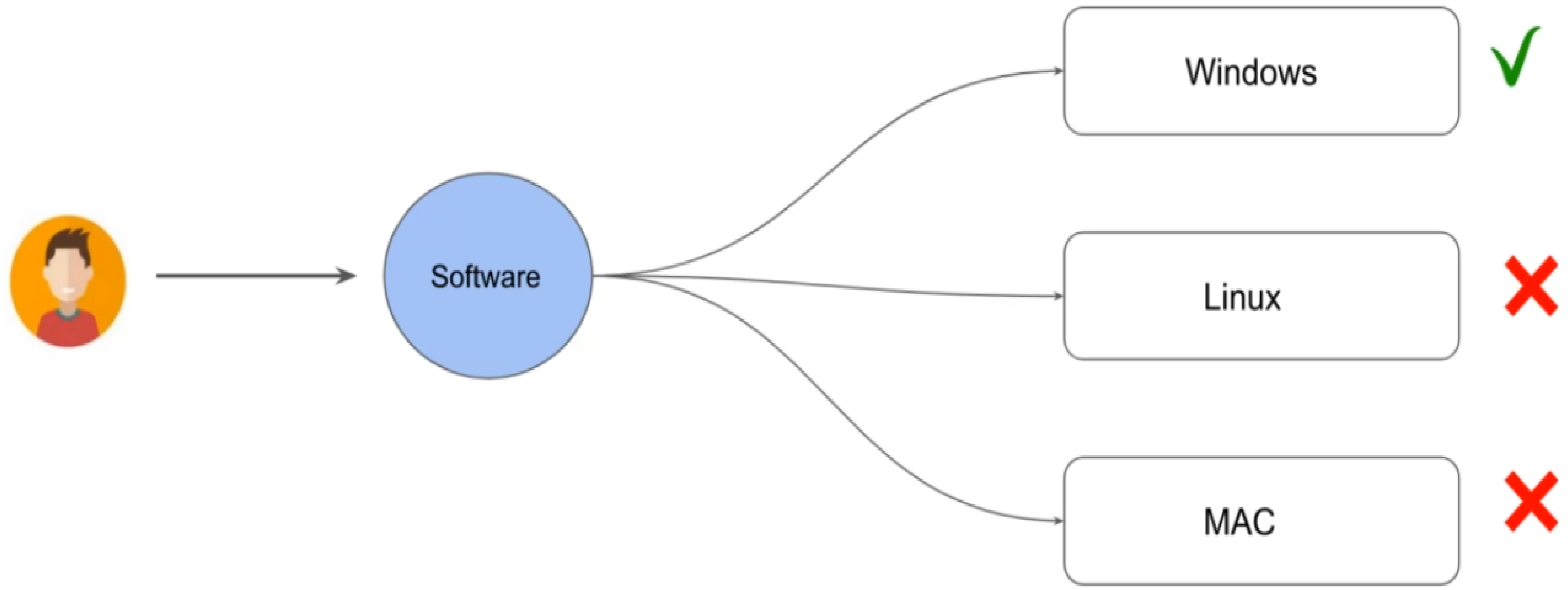
KOENIG

Bare Metal vs Container vs Virtual Machine



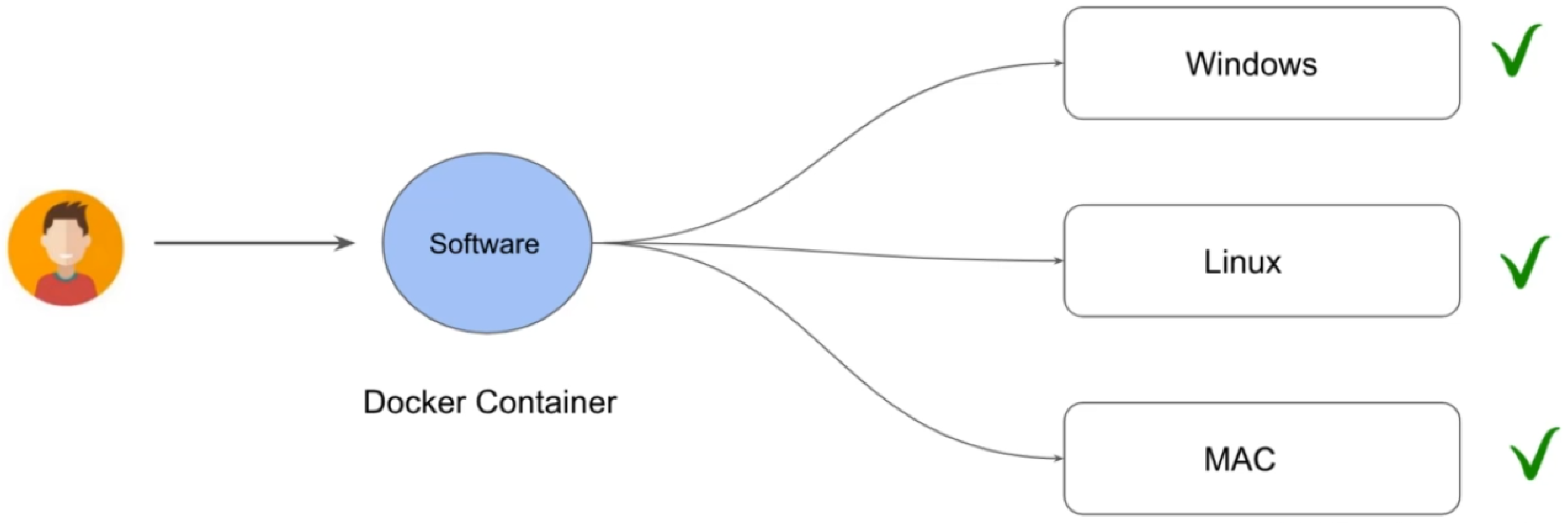
Bare Metal	Virtual machines	Containers
Code	Code	Code
App Container	App Container	App Container
Language Runtime	Language Runtime	Language Runtime
Operating System	Operating System	Operating System
Hardware	Hardware	Hardware

What is Docker Trying to Achieve?



KOENIG

What is Docker Trying to Achieve?



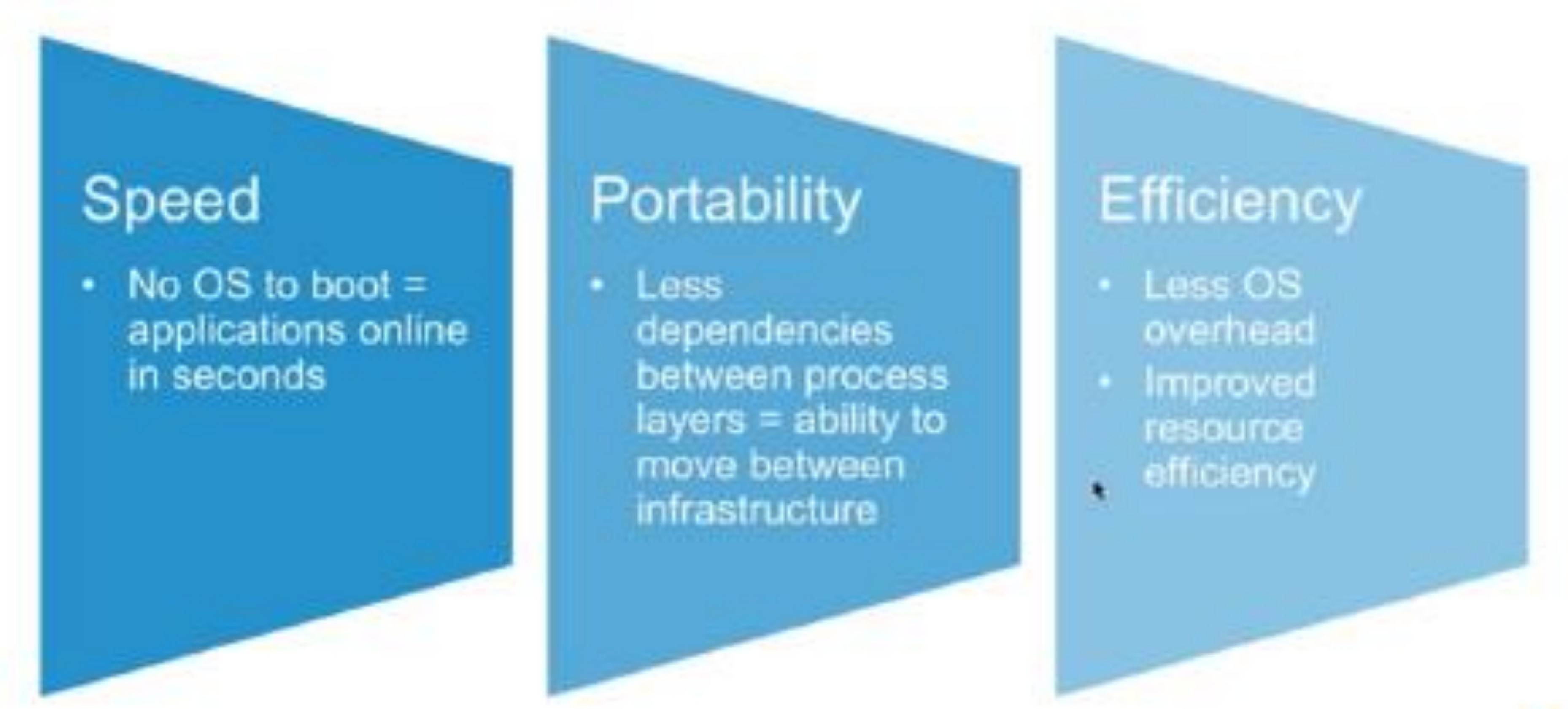
KOENIG

Introduction to Docker

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package.

KOENIG

Key Benefits of Docker



Speed

- No OS to boot = applications online in seconds

Portability

- Less dependencies between process layers = ability to move between infrastructure

Efficiency

- Less OS overhead
- Improved resource efficiency

Is Docker, our only solution ?

Rkt, Mesos, LXC, OpenVZ, Containerd

About

- Docker is a container management system that helps easily manage Linux Containers (LXC) in an easier and universal fashion.
- It provides tools for simplifying DevOps by enabling developers to create templates called ***images***.
- These images can be used to create lightweight virtual machines called ***containers.***
- Containers include their applications and all of their applications' dependencies.
- Docker makes it easier for organizations to automate infrastructure, isolate applications, maintain consistency, and improve resource utilizations.

KOENIG

Image

An image includes everything that is needed to run an application, including the application's executable code, any software on which the application depends, and any required configuration settings

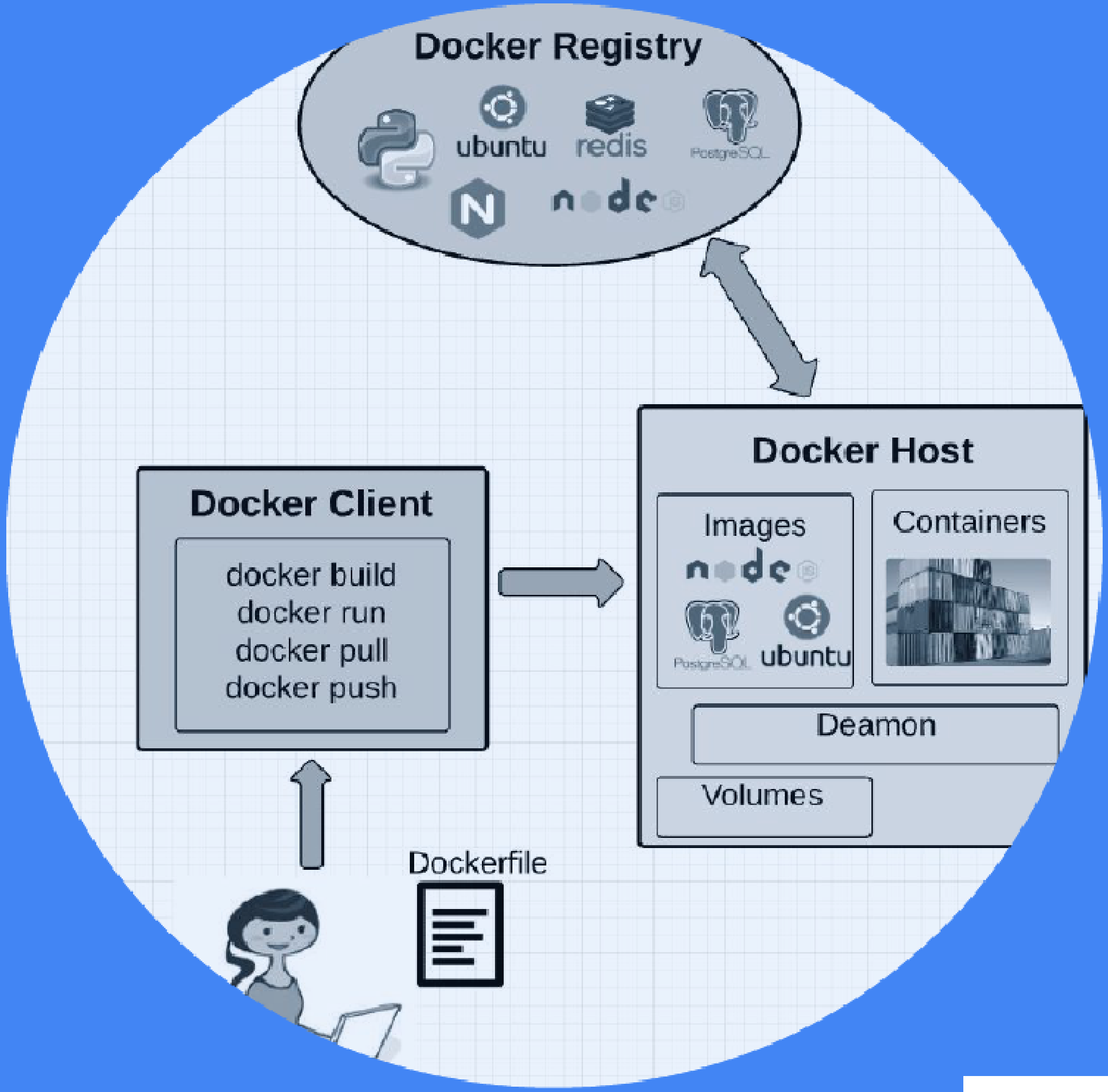
Images are, in a way, just **templates**, you cannot start or run them. What you can do is use that template as a base to build a container.

Container

A run-time instance of an image.

A **Docker container** is a virtualized run-time environment where users can isolate applications from the underlying system. These containers are compact, portable units in which you can start up an application quickly and easily.

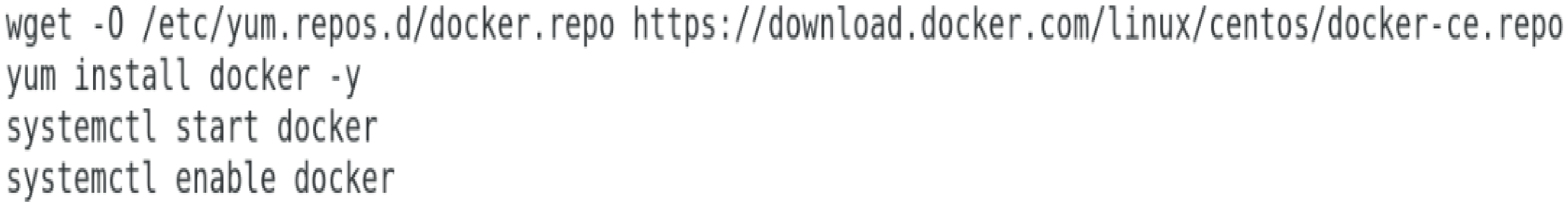
Fundamental Docker Concepts



KOENIG

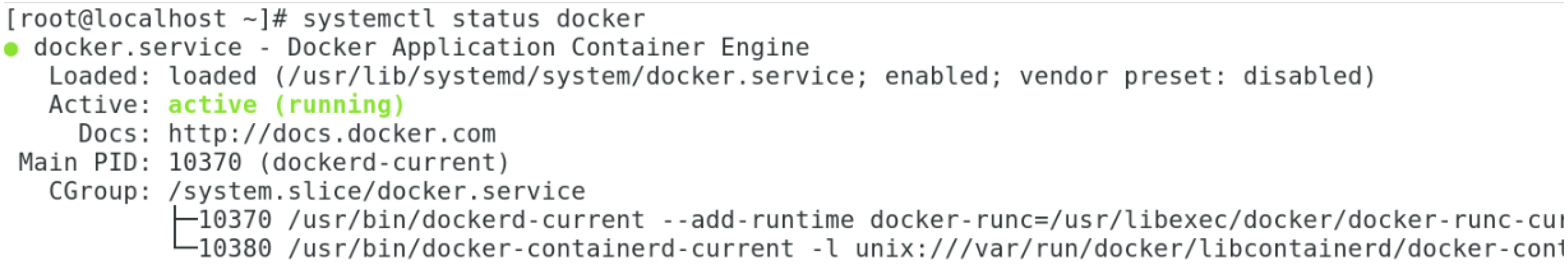
Downloading and Installing Docker

Installing Docker



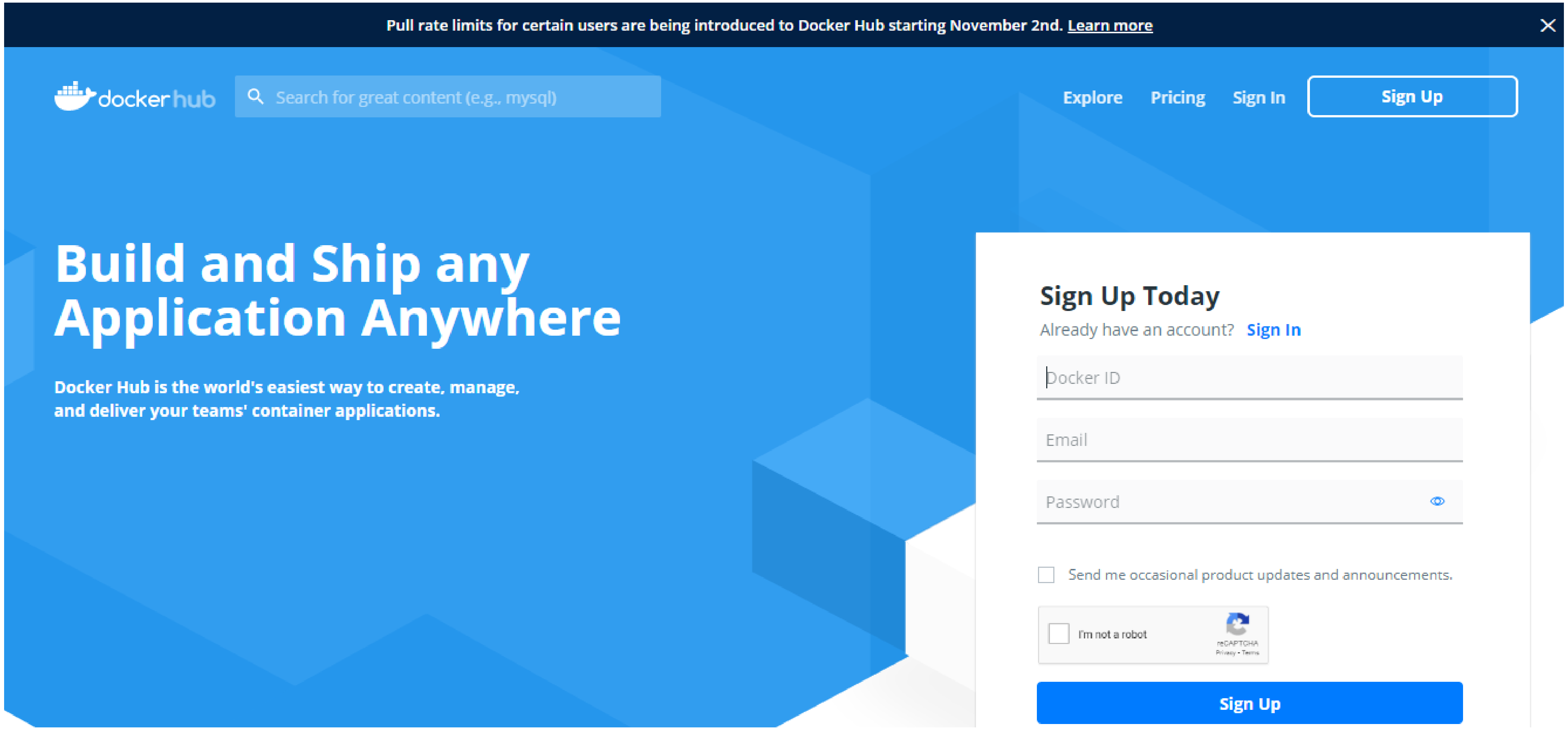
```
wget -O /etc/yum.repos.d/docker.repo https://download.docker.com/linux/centos/docker-ce.repo
yum install docker -y
systemctl start docker
systemctl enable docker
```

Verify Installation



```
[root@localhost ~]# systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; vendor preset: disabled)
   Active: active (running)
     Docs: http://docs.docker.com
 Main PID: 10370 (dockerd-current)
   CGroup: /system.slice/docker.service
           ├─10370 /usr/bin/dockerd-current --add-runtime docker-runc=/usr/libexec/docker/docker-runc-cu
           └─10380 /usr/bin/docker-containerd-current -l unix:///var/run/docker/libcontainerd/docker-con
```

Create Account on Docker Hub



Pull rate limits for certain users are being introduced to Docker Hub starting November 2nd. Learn more

Build and Ship any Application Anywhere

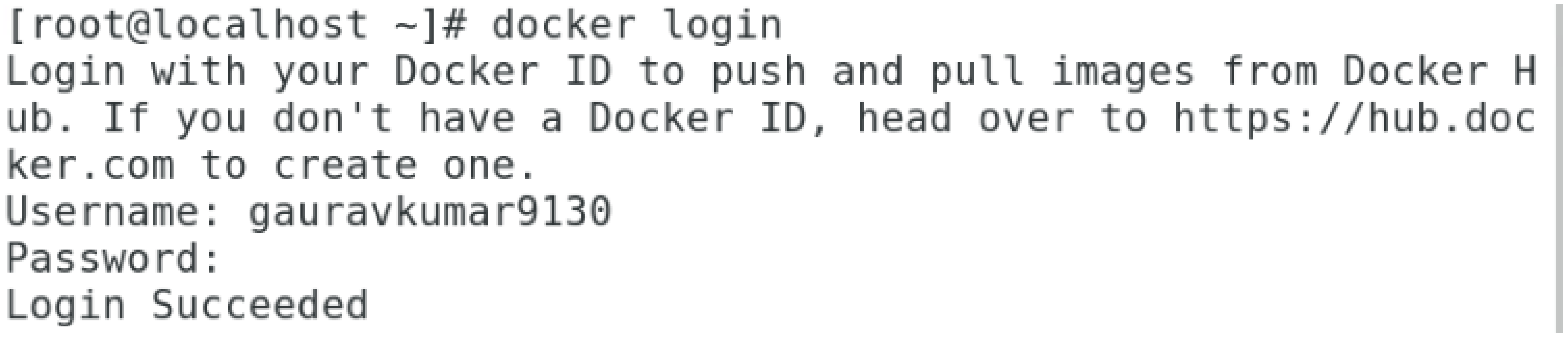
Docker Hub is the world's easiest way to create, manage, and deliver your teams' container applications.

Sign Up Today

Already have an account? Sign In

KOENIG

Docker Login



```
[root@localhost ~]# docker login
Login with your Docker ID to push and pull images from Docker H
ub. If you don't have a Docker ID, head over to https://hub.doc
ker.com to create one.
Username: gauravkumar9130
Password:
Login Succeeded
```

KOENIG

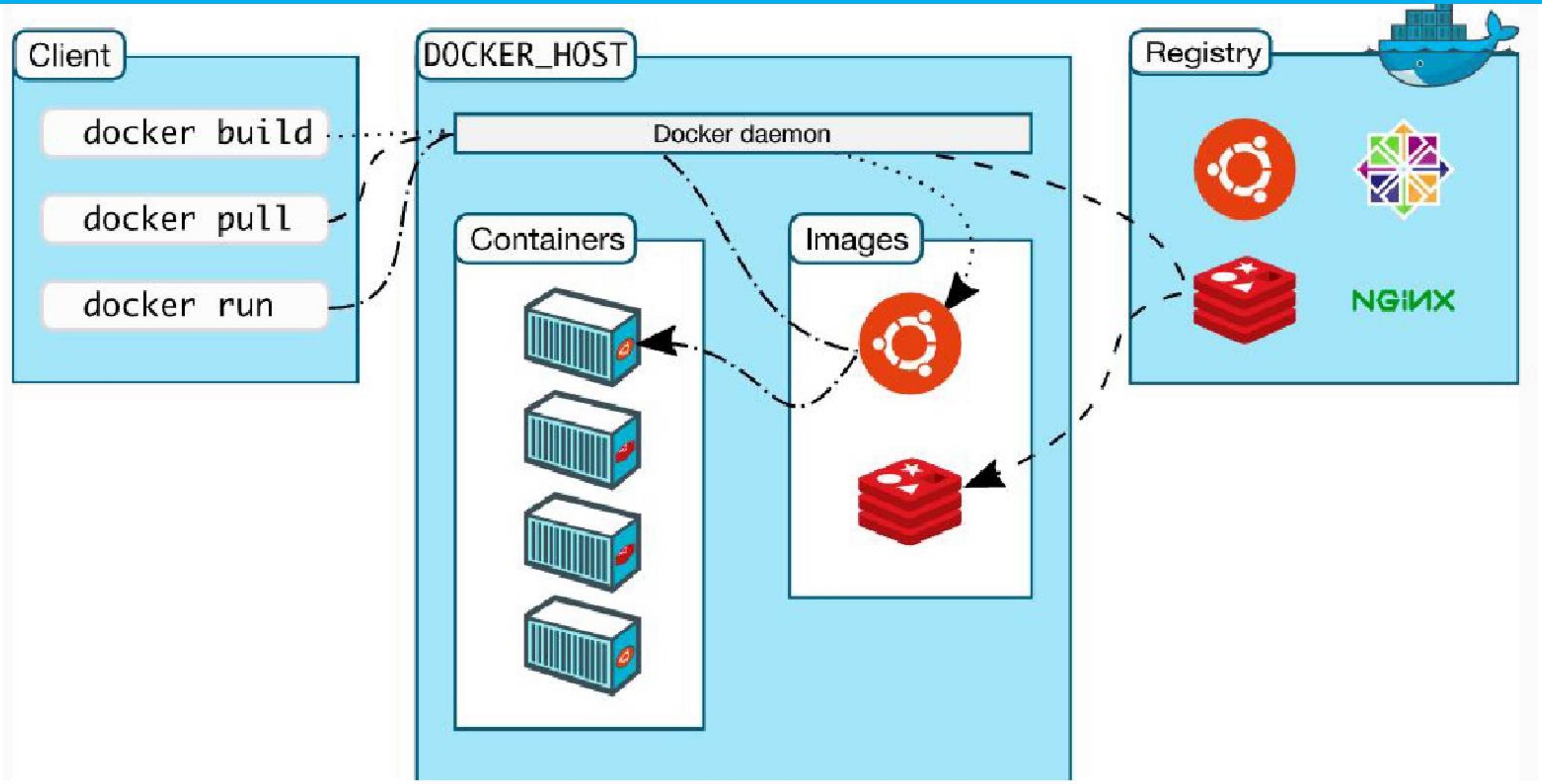
Docker Commands

To check Docker Version:	#docker version
To List Images:	#docker images
To Check running Containers:	#docker ps
To check all containers:	#docker ps -a

Getting Started With *Docker*

KOENIG

Docker Architecture



KOENIG

Create Container

```
#docker run -it -d --name container01 nginx
```

-d	-> Run in Background
-i	-> interactive/user can interact
-t	-> assign terminal to the container
--name	-> to give container name
nginx	-> image name

Docker Essential Commands

View all commands:	$docker
Search for images:	$docker search <string>
Download Image:	$docker pull <username>/<repository>
Start Container:	$docker start <containername>
Stop Container:	$docker stop <containername>
Inspect Container:	$docker inspect <containername>
Access Container:	$docker exec -it <containername> bash

Container Registries

Docker Hub

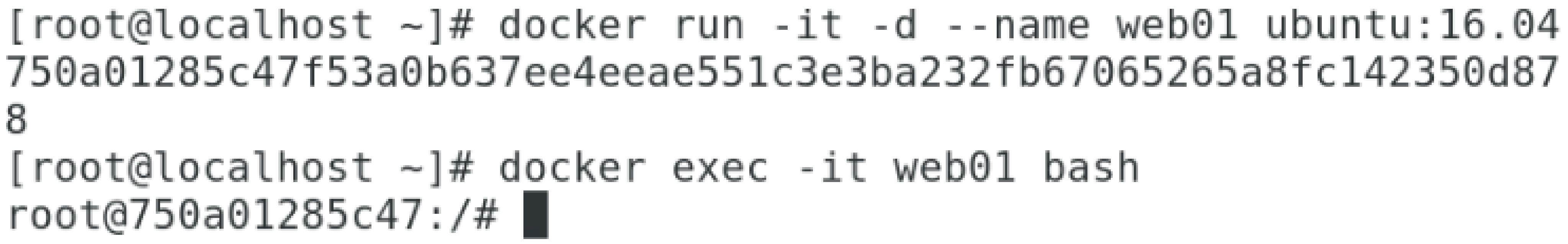
docker

Google Container Registry

RED HAT® Quay.io

KOENIG

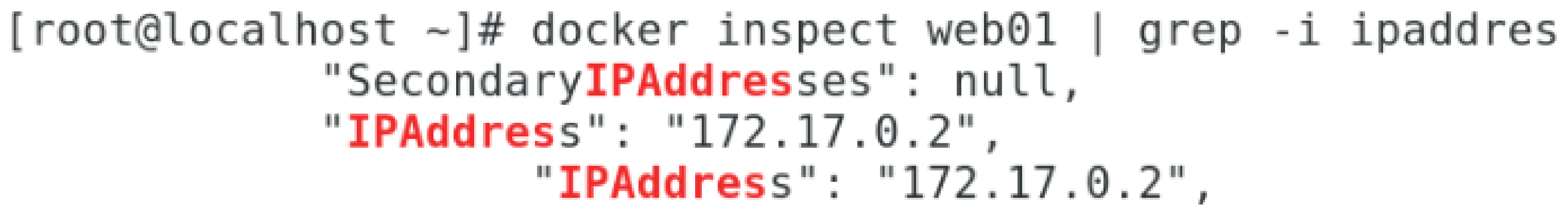
Let’s Deploy a Web Application



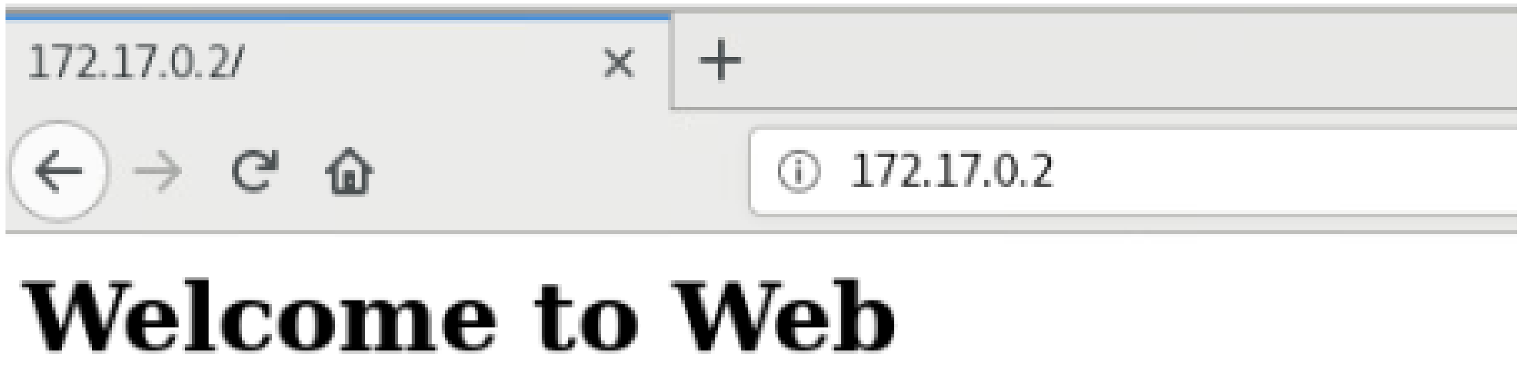
```
[root@localhost ~]# docker run -it -d --name web01 ubuntu:16.04
750a01285c47f53a0b637ee4eeae551c3e3ba232fb67065265a8fc142350d87
8
[root@localhost ~]# docker exec -it web01 bash
root@750a01285c47:/#
```

```
apt-get update
apt-get install apache2 -y
echo "<h1> Welcome to Web </h1>" > /var/www/html/index.html
service apache2 restart
```

Access Web Application



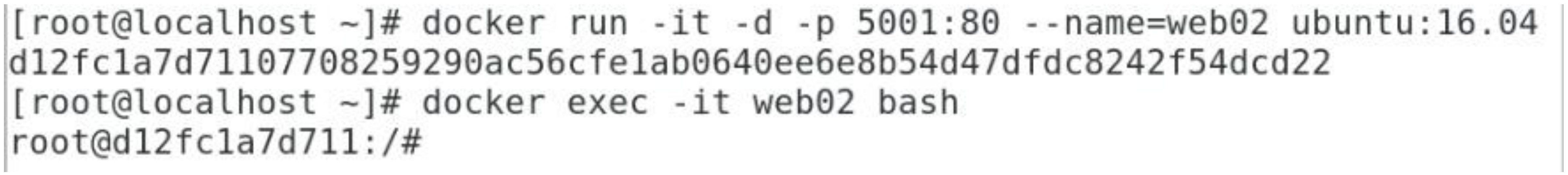
```
[root@localhost ~]# docker inspect web01 | grep -i ipaddres
            "SecondaryIPAddresses": null,
            "IPAddress": "172.17.0.2",
                    "IPAddress": "172.17.0.2",
```



Welcome to Web

KOENIG

Web Application with Port Mapping

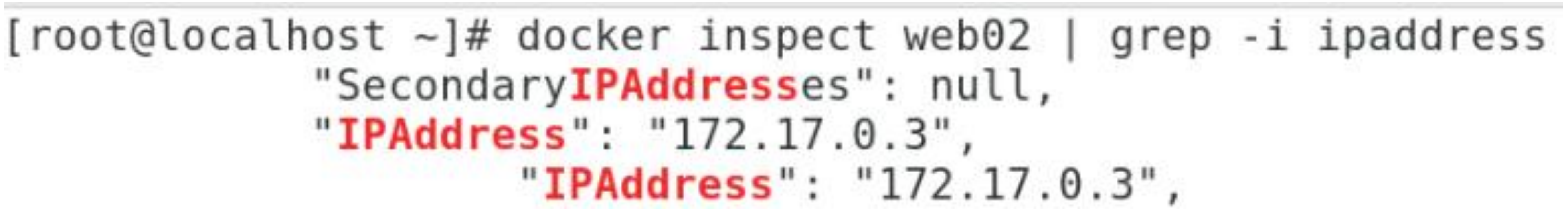


```
[root@localhost ~]# docker run -it -d -p 5001:80 --name=web02 ubuntu:16.04
d12fc1a7d71107708259290ac56cfe1ab0640ee6e8b54d47dfdc8242f54dcd22
[root@localhost ~]# docker exec -it web02 bash
root@d12fc1a7d711:/#
```

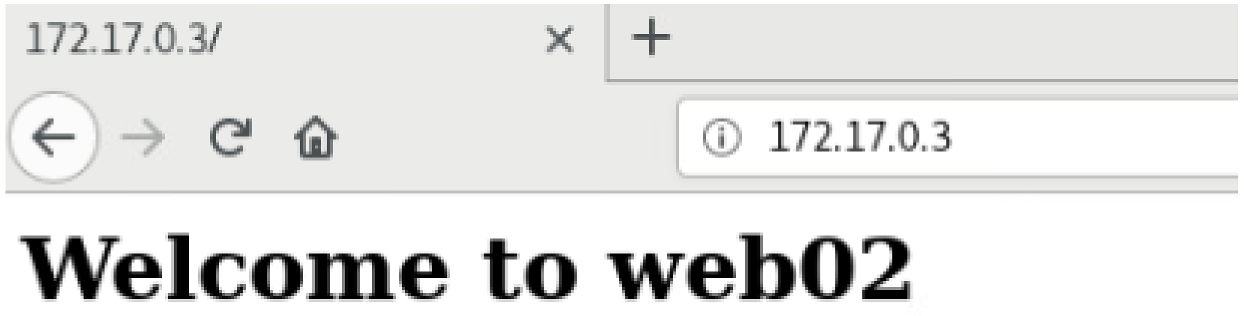
```
apt-get update
apt-get install apache2 -y
echo "<h1> Welcome to web02 </h1>" > /var/www/html/index.html
service apache2 restart
```

KOENIG

Access Web Application



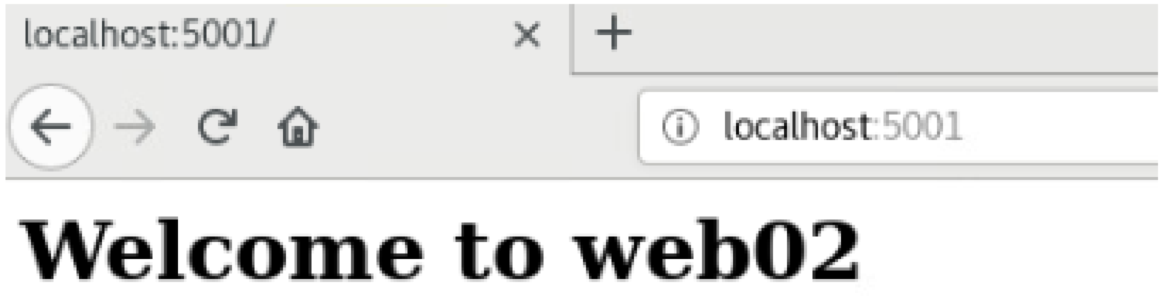
```
[root@localhost ~]# docker inspect web02 | grep -i ipaddress
            "SecondaryIPAddresses": null,
            "IPAddress": "172.17.0.3",
                    "IPAddress": "172.17.0.3",
```



172.17.0.3/

172.17.0.3

Welcome to web02



localhost:5001/

localhost:5001

Welcome to web02

KOENIG

Docker Image

KOENIG

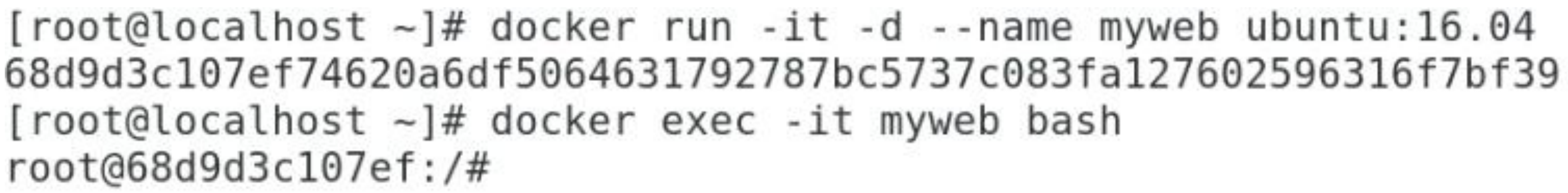
Image Creation Methods

There are two methods to create image:

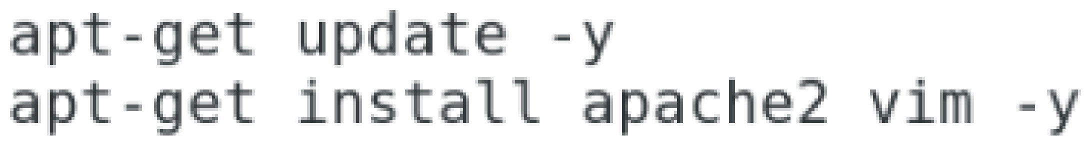
1) docker commit
2) Dockerfile

KOENIG

Method 1: Docker Commit

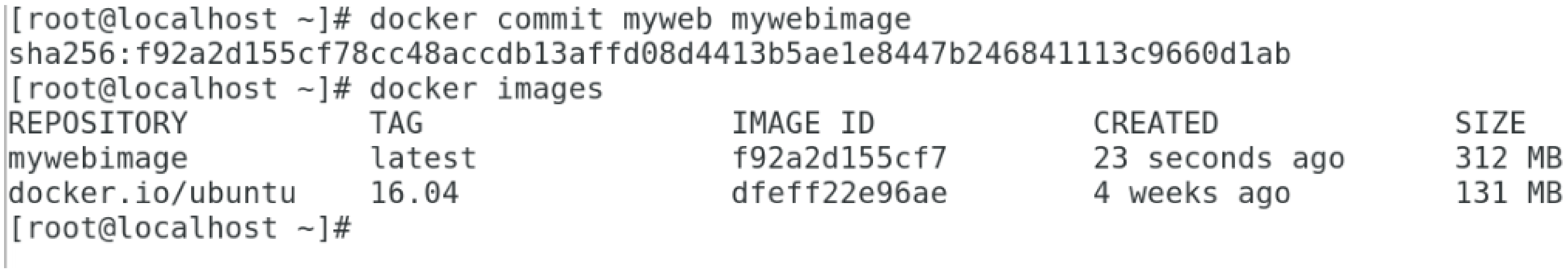


```
[root@localhost ~]# docker run -it -d --name myweb ubuntu:16.04
68d9d3c107ef74620a6df5064631792787bc5737c083fa127602596316f7bf39
[root@localhost ~]# docker exec -it myweb bash
root@68d9d3c107ef:/#
```



```
apt-get update -y
apt-get install apache2 vim -y
```

Method 1: Docker Commit



```
[root@localhost ~]# docker commit myweb mywebimage
sha256:f92a2d155cf78cc48accdb13affd08d4413b5ae1e8447b246841113c9660d1ab
[root@localhost ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
mywebimage          latest              f92a2d155cf7        23 seconds ago      312 MB
docker.io/ubuntu    16.04               dfeff22e96ae        4 weeks ago         131 MB
[root@localhost ~]#
```

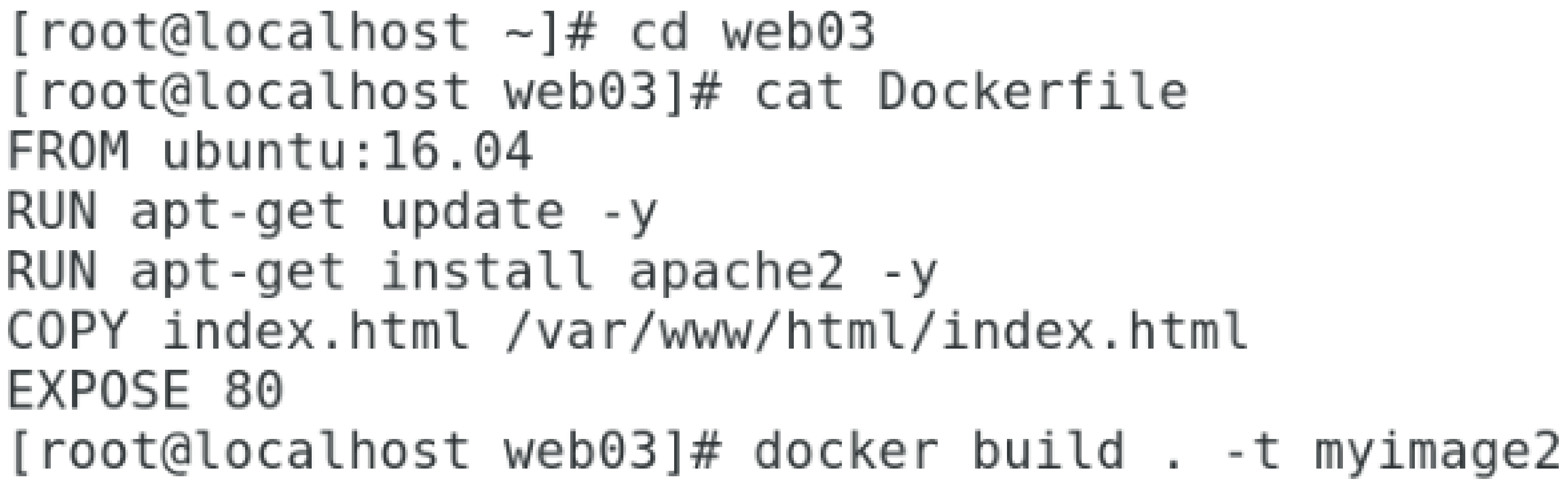
KOENIG

Method 2: Dockerfile

Dockerfile Instructions

FROM	-> Image you want to use
RUN	-> Run the commands
COPY	-> Copy file from host machine to container
ADD	-> Copy file from host machine or remote location to container
MAINTAINER	-> Author Information
EXPOSE	-> Define Port Number
ENTRYPOINT	-> Set image's main command
CMD	-> Define Command for run
ENV	-> Environment Variables

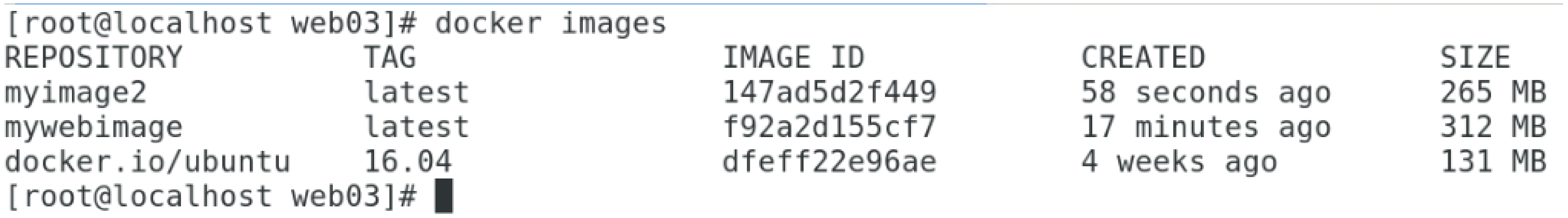
Method 2: Dockerfile



```
[root@localhost ~]# cd web03
[root@localhost web03]# cat Dockerfile
FROM ubuntu:16.04
RUN apt-get update -y
RUN apt-get install apache2 -y
COPY index.html /var/www/html/index.html
EXPOSE 80
[root@localhost web03]# docker build . -t myimage2
```

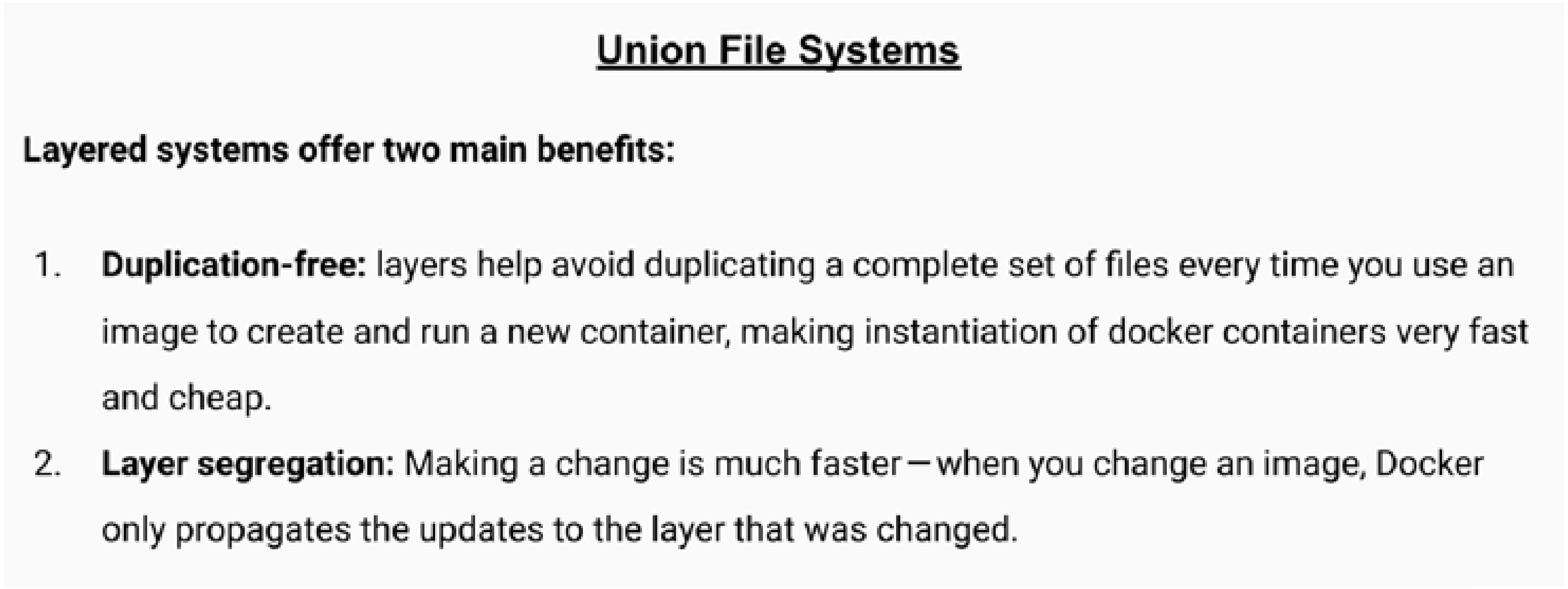
KOENIG

Method 2: Dockerfile



```
[root@localhost web03]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
myimage2            latest              147ad5d2f449        58 seconds ago      265 MB
mywebimage          latest              f92a2d155cf7        17 minutes ago      312 MB
docker.io/ubuntu    16.04               dfeff22e96ae        4 weeks ago         131 MB
[root@localhost web03]# 
```

Understanding Docker Image Layers



<u>Union File Systems</u>

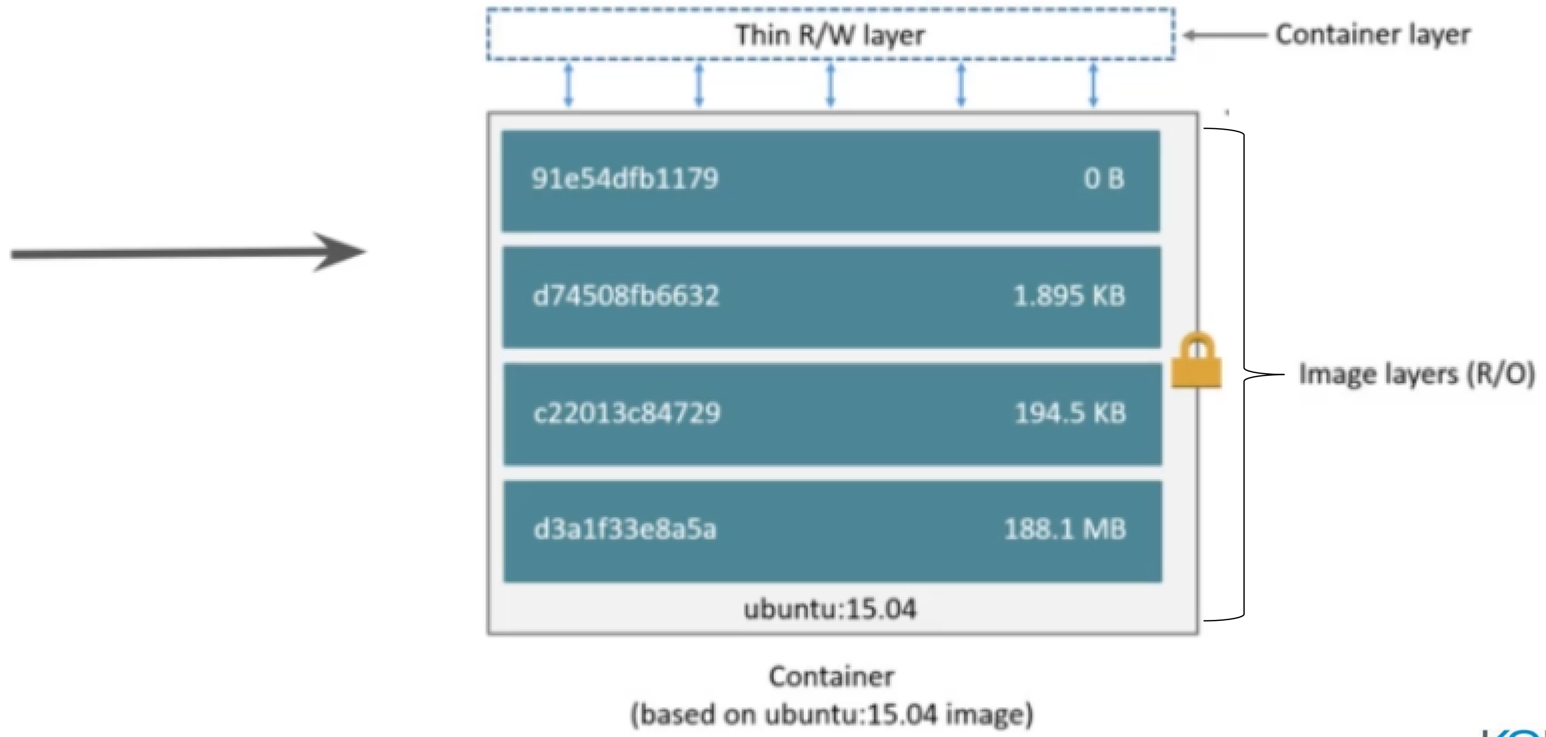
Layered systems offer two main benefits:

1. **Duplication-free:** layers help avoid duplicating a complete set of files every time you use an image to create and run a new container, making instantiation of docker containers very fast and cheap.
2. **Layer segregation:** Making a change is much faster – when you change an image, Docker only propagates the updates to the layer that was changed.

Layers of Docker Images

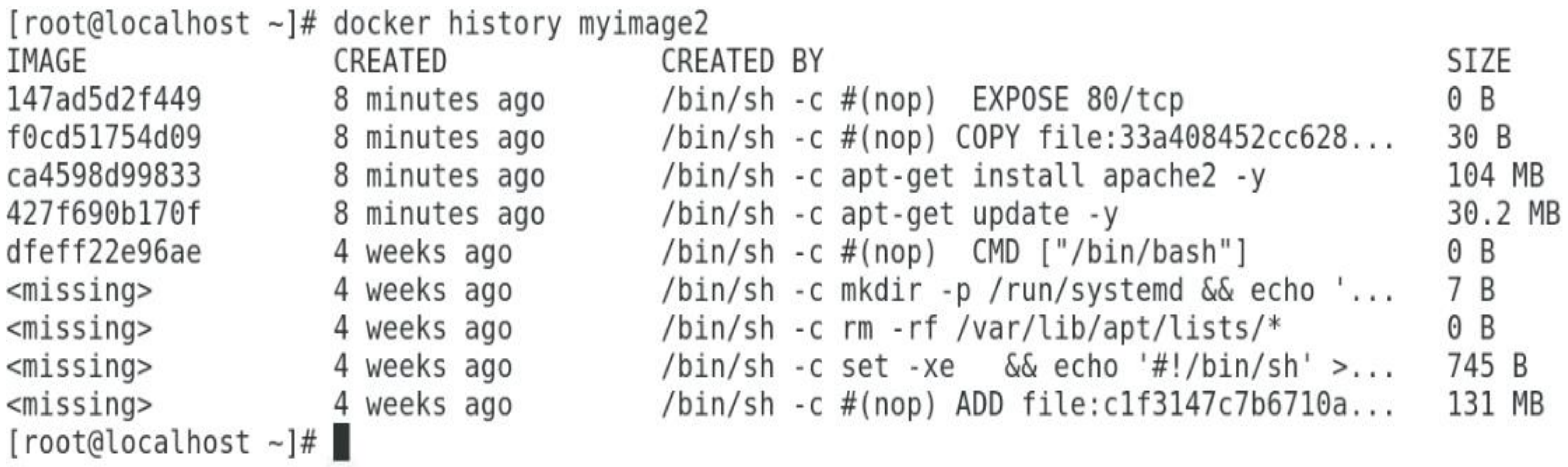
- A docker image is built up from a series of layers.
- Each layer represents an instruction in the image's Dockerfile.

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```



KOENIG

History of Docker Image Layers


```
[root@localhost ~]# docker history myimage2
IMAGE               CREATED             CREATED BY                                      SIZE
147ad5d2f449        8 minutes ago       /bin/sh -c #(nop)  EXPOSE 80/tcp                0 B
f0cd51754d09        8 minutes ago       /bin/sh -c #(nop) COPY file:33a408452cc628...   30 B
ca4598d99833        8 minutes ago       /bin/sh -c apt-get install apache2 -y           104 MB
427f690b170f        8 minutes ago       /bin/sh -c apt-get update -y                    30.2 MB
dfeff22e96ae        4 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]            0 B
<missing>           4 weeks ago         /bin/sh -c mkdir -p /run/systemd && echo '...   7 B
<missing>           4 weeks ago         /bin/sh -c rm -rf /var/lib/apt/lists/*          0 B
<missing>           4 weeks ago         /bin/sh -c set -xe   && echo '#!/bin/sh' >...   745 B
<missing>           4 weeks ago         /bin/sh -c #(nop) ADD file:c1f3147c7b6710a...   131 MB
[root@localhost ~]#
```

Docker Registry

Two types of Registry:

1) Public
2) Private

Registry:- Place where you are pushing your images(like docker hub,quay.io)

Repository:- Collection of images(like we are create repository and we are pushing image)

Working with Public Registry

Steps:

1) Create a account on hub.docker.com
2) Create a Repository
3) Tag your Image
4) Docker Login
5) Push Image

Working with Public Registry

Steps:

1) **Create a account on hub.docker.com**
2) **Create a Repository**
3) **Tag your Image**

```
#docker tag <imageid> <username>/<repositoryname>:<tag>
```

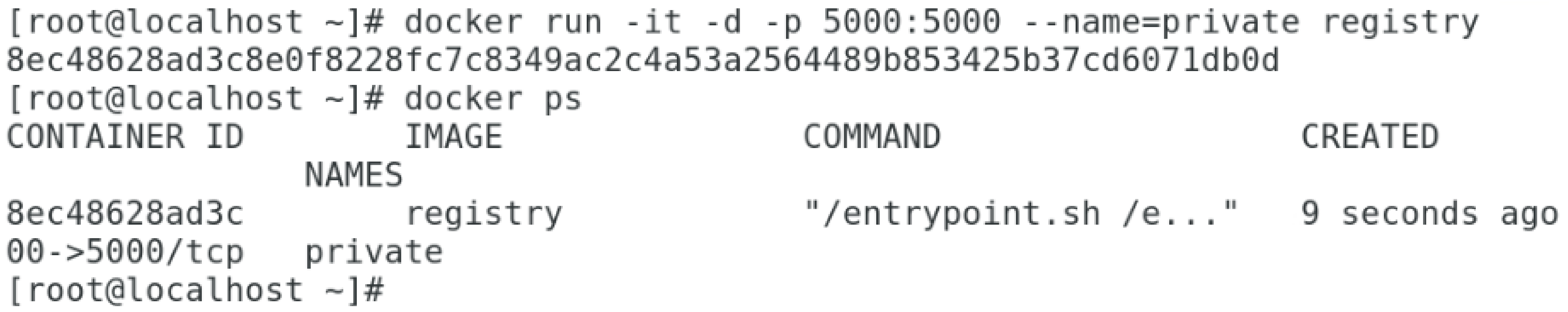
4) **Docker Login**

```
#docker login
```

5) **Push Image**

```
#docker push <username>/<repositoryname>
```

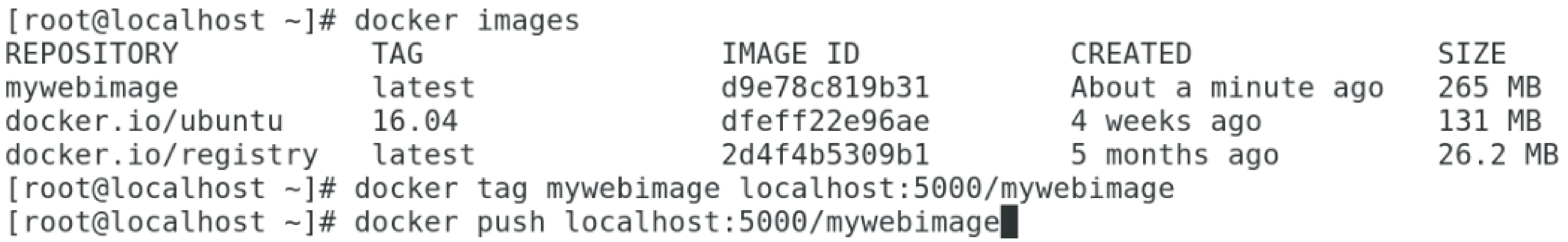
Deploying Private Registry



```
[root@localhost ~]# docker run -it -d -p 5000:5000 --name=private registry
8ec48628ad3c8e0f8228fc7c8349ac2c4a53a2564489b853425b37cd6071db0d
[root@localhost ~]# docker ps
CONTAINER ID        IMAGE                    COMMAND                  CREATED
               NAMES
8ec48628ad3c        registry                 "/entrypoint.sh /e..."   9 seconds ago
00->5000/tcp   private
[root@localhost ~]#
```

KOENIG

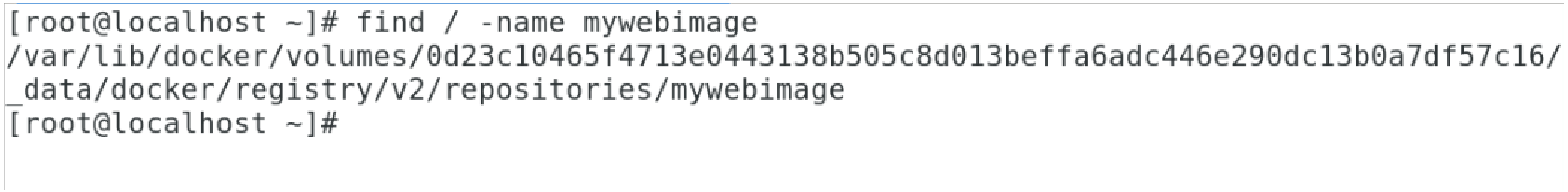
Working with Private Registry



```
[root@localhost ~]# docker images
REPOSITORY           TAG                 IMAGE ID            CREATED              SIZE
mywebimage           latest              d9e78c819b31        About a minute ago   265 MB
docker.io/ubuntu     16.04               dfeff22e96ae        4 weeks ago          131 MB
docker.io/registry   latest              2d4f4b5309b1        5 months ago         26.2 MB
[root@localhost ~]# docker tag mywebimage localhost:5000/mywebimage
[root@localhost ~]# docker push localhost:5000/mywebimage
```

KOENIG

Private Registry Location



```
[root@localhost ~]# find / -name mywebimage
/var/lib/docker/volumes/0d23c10465f4713e0443138b505c8d013beffa6adc446e290dc13b0a7df57c16/
_data/docker/registry/v2/repositories/mywebimage
[root@localhost ~]#
```

Managing Resources for Container

To show resource consumption:

```
#docker stats
```

To restrict memory:

```
#docker run -it -d -m 300M --name=<containername> <imagename>
```

To restrict cpu and memory:

```
#docker run -it -d -m 100M --cpuset-cpus=1 --name=<containername>
<imagename>
```

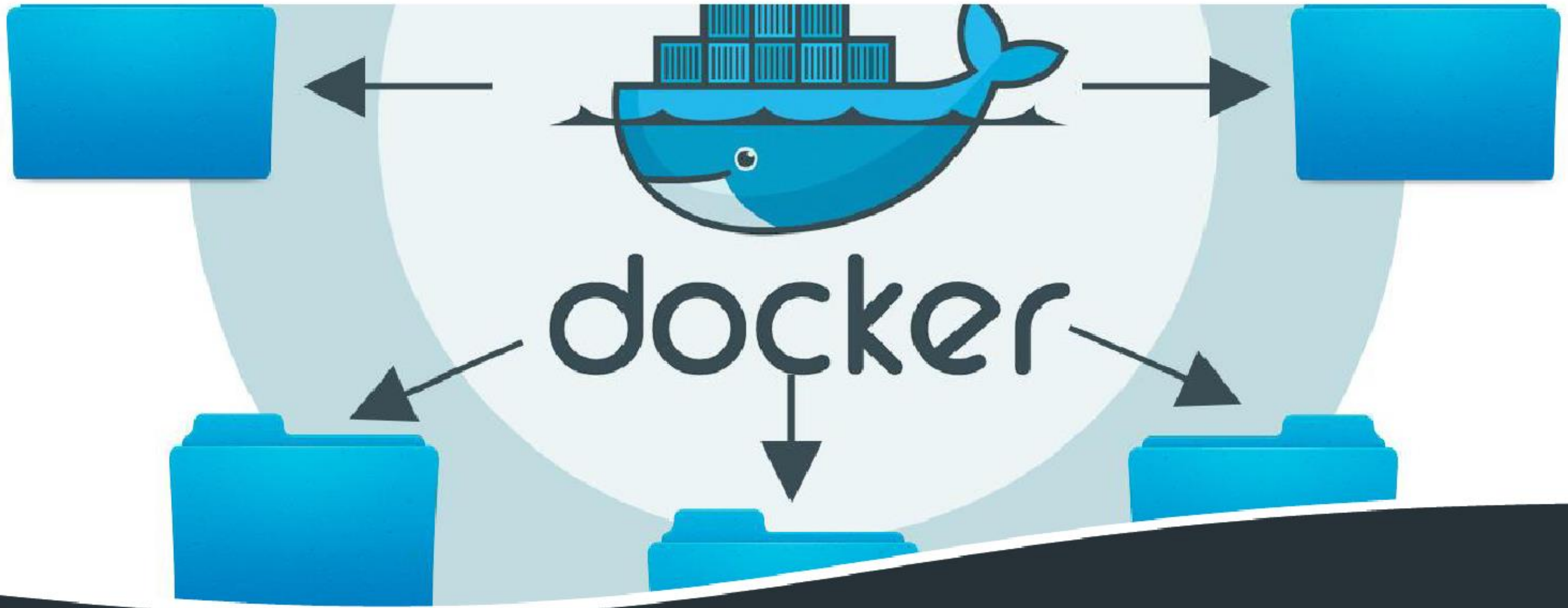
Disk Usage Metrics for Docker Components

To check disk usage:

```
#docker system df
```

To check disk usage in details:

```
#docker system df -v
```



Docker Storage

There are lots of places inside Docker (both at the engine level and container level) that use or work with storage.

KOENIG

Image Storage

The copy-on-write mechanism using Union File System:

- Docker engine doesn’t make full copy of the already stored image.

Data Storage Methods

Three ways to store data:

1) tmpfs -> temporary

2) Volumes -> persist data

3) bind mounts -> persist data (share direct file from host machine to container)

Volumes

- Created and managed by Docker.
- You can create a volume explicitly using the ***docker volume create*** command
- Docker can create a volume during container or service creation.
- Stored within a directory on the Docker host. → ***/var/lib/docker/volumes***
- This is similar to the way that bind mounts work.
- Volumes are managed by Docker and are isolated from the core functionality of the host machine.
- A volume can be mounted into multiple containers simultaneously.
- The volume will be available to Docker even if no running container is using it.
- Volumes are not automatically deleted.
- Volumes also support the use of volume drivers, which allow you to store your data on remote hosts or cloud providers, among other possibilities.

"You can remove unused volumes using ***docker volume prune***"

Docker Volume Commands

#docker volume ls -> to list volume

#docker volume create <volname> -> to create volume

#docker volume inspect <volumename> -> inspect volume

#docker volume rm <volumename> -> to delete volume

To use Volume in container:

#docker run -it -d -v <vol name>:<container path> --name c1 ubuntu

KOENIG

Use case for Volumes

- Sharing data among multiple running containers.
- Multiple containers can mount the same volume simultaneously, either read-write or read-only.
- When the Docker host is not guaranteed to have a given directory or file structure. Volumes help you decouple the configuration of the Docker host from the container runtime.
- When you want to store your container's data on a remote host or a cloud provider, rather than locally.
- When you need to backup, restore, or migrate data from one Docker host to another, volumes are a better choice.

Bind Mounts

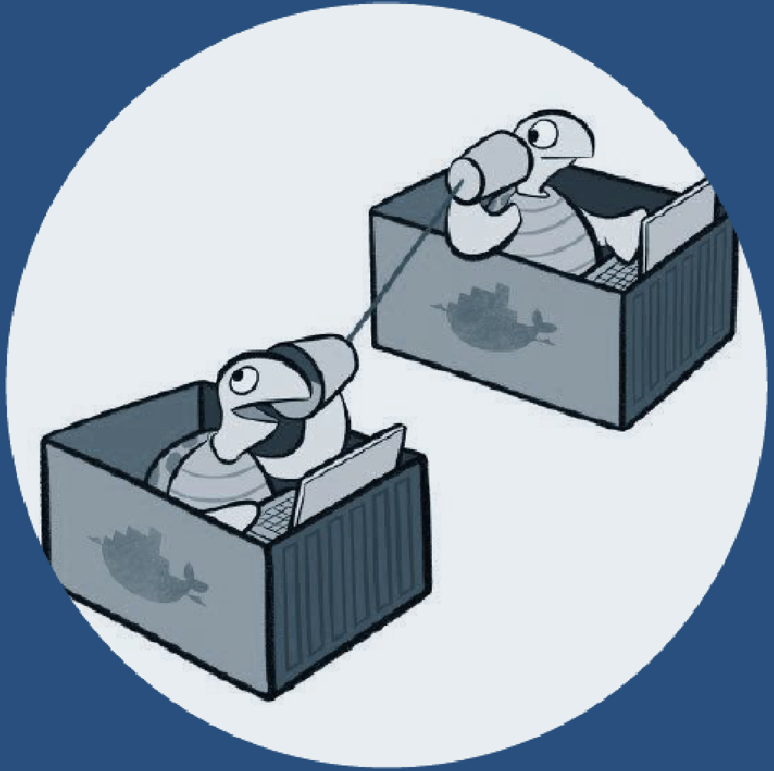
To use Bind Mounts in container:

```
#docker run -it -d -v <hostfile path>:container path> --name c1 ubuntu
```

Use cases for Bind Mounts

- Sharing configuration files from the host machine to containers.
- By default, docker provides DNS resolution to containers by mounting /etc/resolv.conf from the host machine into each container.
- Sharing source code or build artifacts between a development environment on the Docker host and a container.
- If you use Docker for development this way, your production Dockerfile would copy the production-ready artifacts directly into the image, rather than relying on a bind mount.
- When the file or directory structure of the Docker host is guaranteed to be consistent with the bind mounts the containers require.

Deep dive into Docker Networking

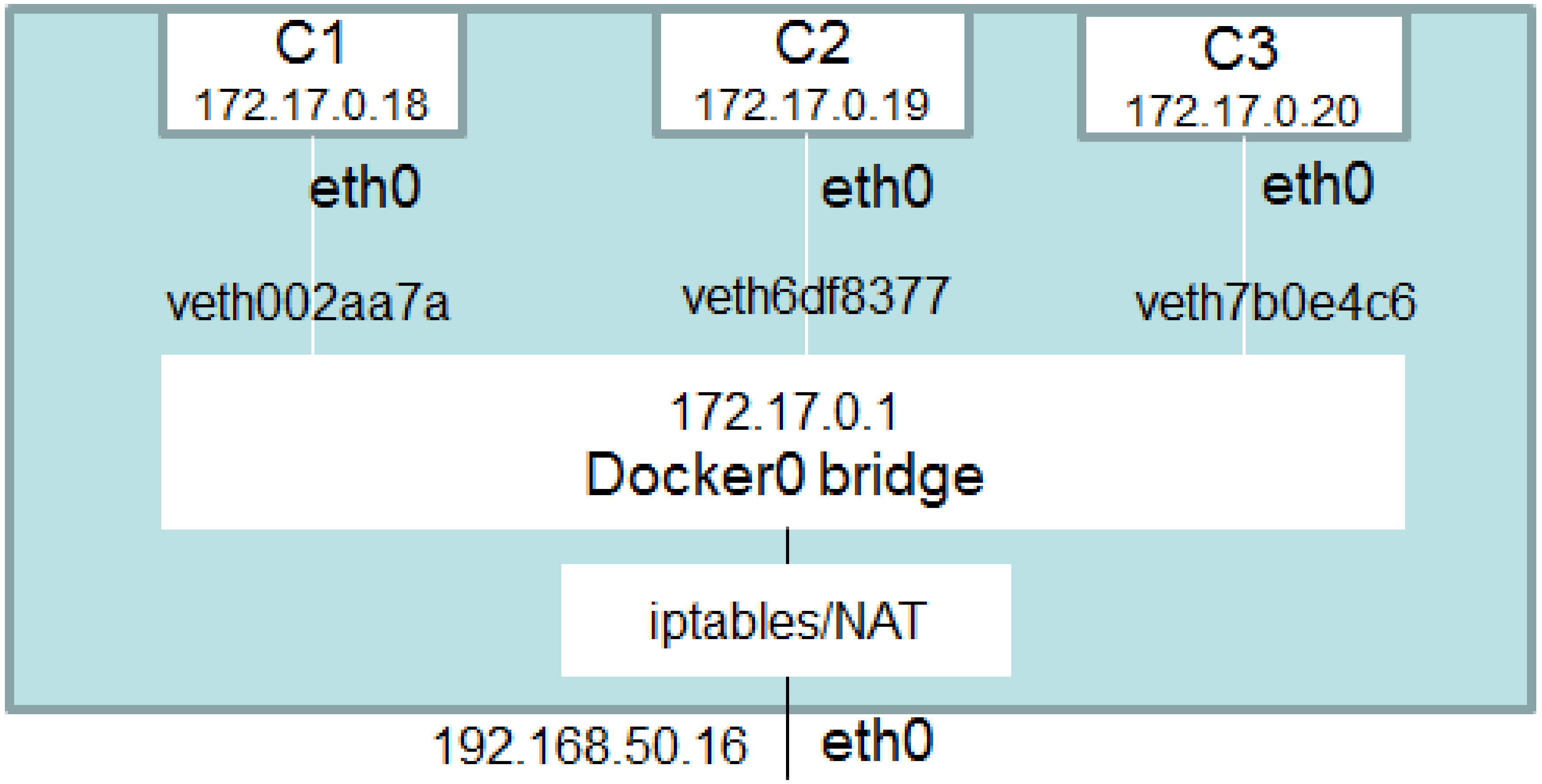


We have managed to make it so far without having to really think about networking.

Because by default, Docker creates a network bridge between the containers and your host machine's network interface.

This is Docker networking at its most basic form.

Default Bridge



KOENIG

Network Drivers

There are several drivers available by default, and provides core networking functionality.

- Bridge
- Host
- None

Docker Network Commands

To List Network:

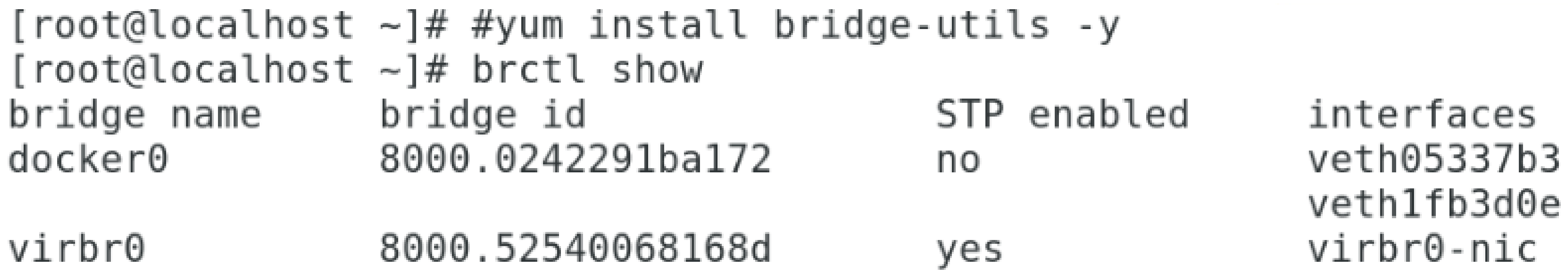
```
#docker network ls
```

To inspect network drivers:

```
#docker network inspect bridge
```

Understanding Bridge Network Driver

A bridge network uses a software bridge which allows containers connected to the same bridge network to communicate, while providing isolation from containers which are not connected to that bridge network.



```
[root@localhost ~]# #yum install bridge-utils -y
[root@localhost ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.0242291ba172       no              veth05337b3
                                                        veth1fb3d0e
virbr0          8000.52540068168d       yes             virbr0-nic
```

User Defined Bridge Network

To create network:

#docker network create --driver=bridge <network name>

To create container using user defined bridge:

#docker run -it -d --name <container name> --network <net name> <image name>

Install Network Utilities in Container:

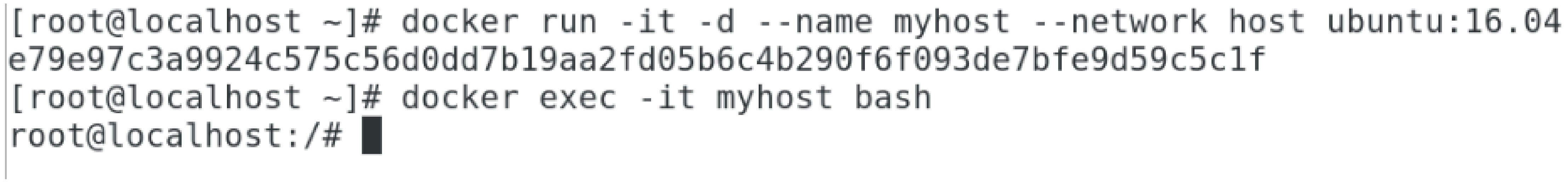
#apt-get update && apt-get install net-tools && apt-get install iputils-ping

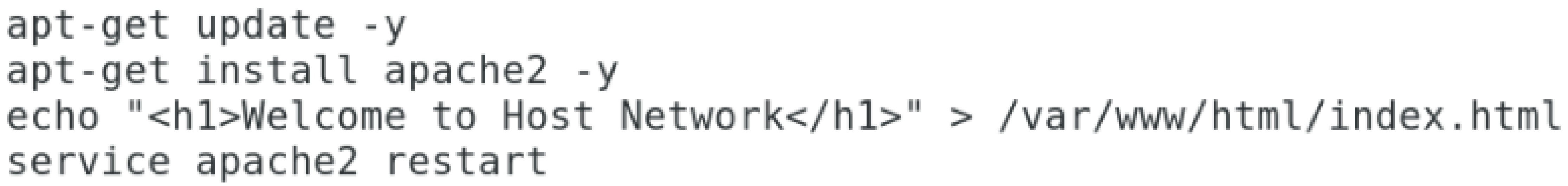
Understanding Host Network Driver

This driver removes the network isolation between the docker host and the docker containers to use the host's networking directly.

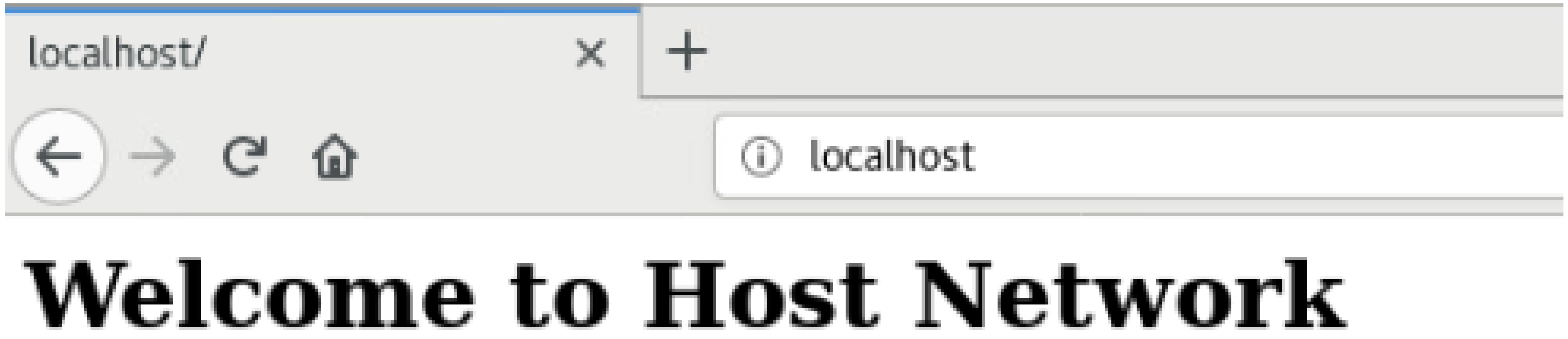
For instance,if you run a container which binds to port 80 and you use host networking, the container's application will be avilable on port 80 on the host's IP address.

Demo of Host Network Driver


```
[root@localhost ~]# docker run -it -d --name myhost --network host ubuntu:16.04
e79e97c3a9924c575c56d0dd7b19aa2fd05b6c4b290f6f093de7bfe9d59c5c1f
[root@localhost ~]# docker exec -it myhost bash
root@localhost:/#
```


```
apt-get update -y
apt-get install apache2 -y
echo "<h1>Welcome to Host Network</h1>" > /var/www/html/index.html
service apache2 restart
```

Access Application



Welcome to Host Network

KOENIG

Understanding None Network Driver

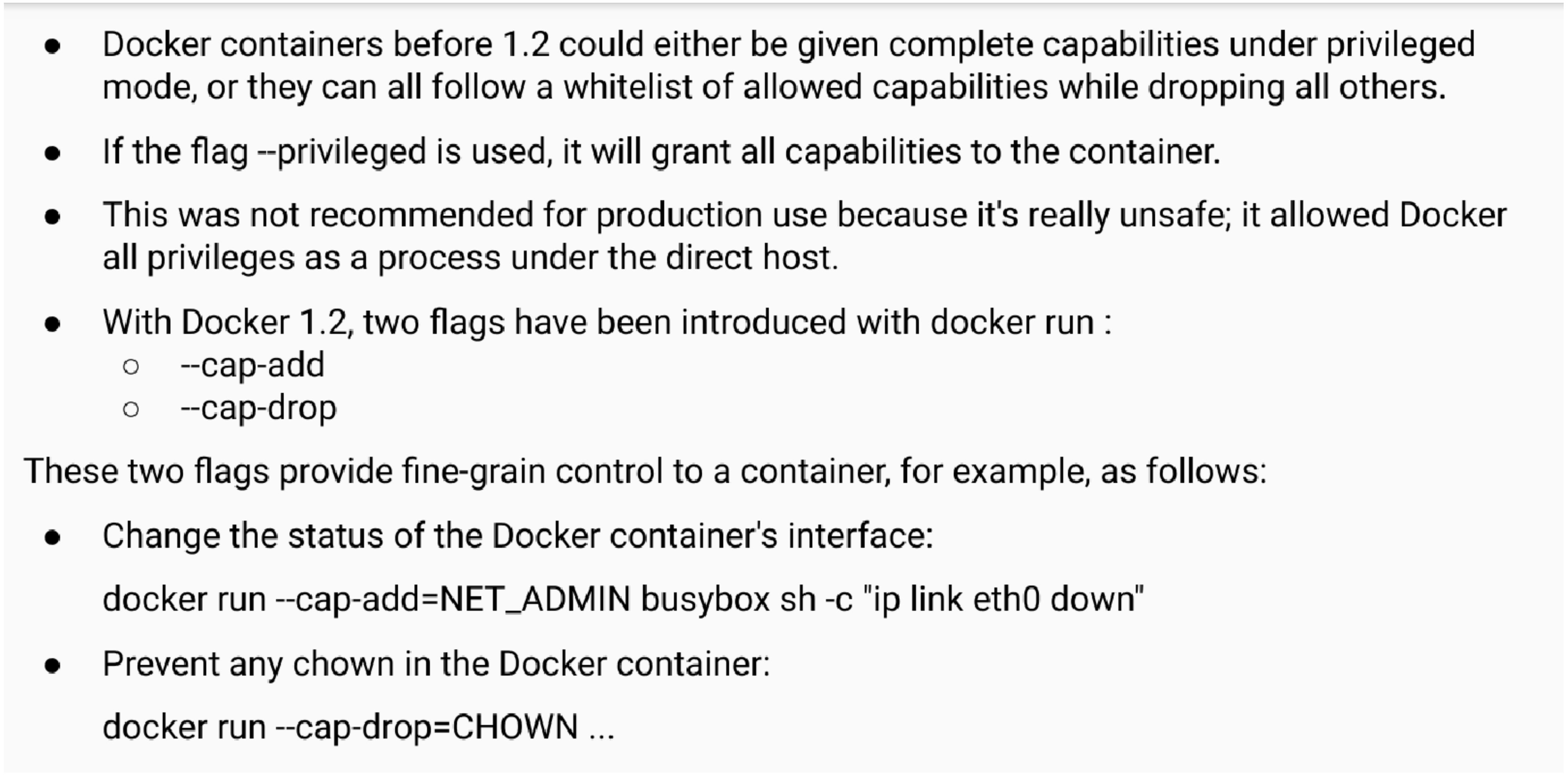
If you want to completely disable the networking stack on a container, you can use the none network.

This mode will not configure any IP for the container and doesn't have any access to the external network as well as for other containers.

```
#docker run -it -d --name mynone --network none alpine
```

KOENIG

Linux Capabilities



- Docker containers before 1.2 could either be given complete capabilities under privileged mode, or they can all follow a whitelist of allowed capabilities while dropping all others.
- If the flag --privileged is used, it will grant all capabilities to the container.
- This was not recommended for production use because it's really unsafe; it allowed Docker all privileges as a process under the direct host.
- With Docker 1.2, two flags have been introduced with docker run :
  - --cap-add
  - --cap-drop

These two flags provide fine-grain control to a container, for example, as follows:

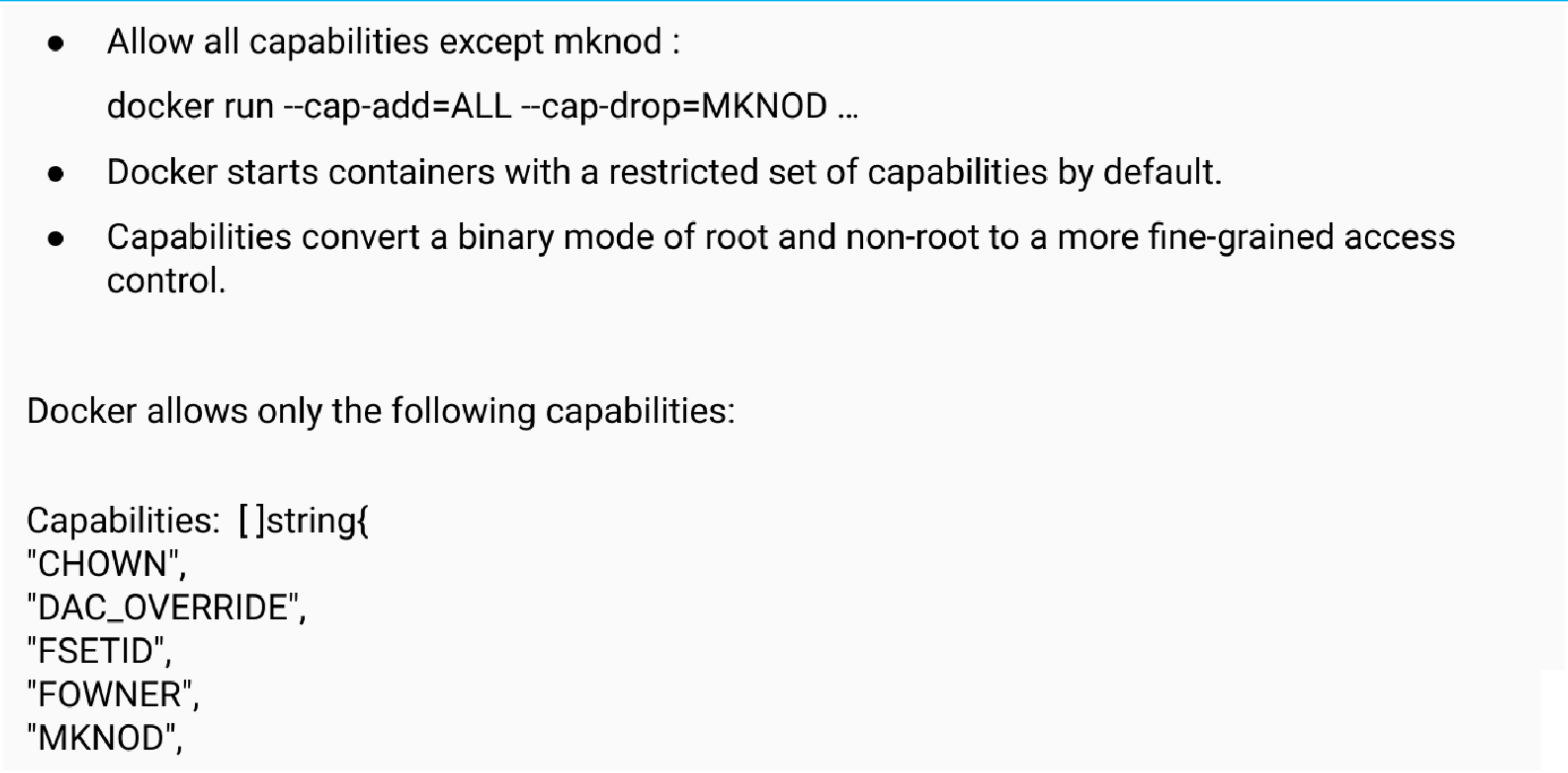
- Change the status of the Docker container's interface:

  ```
  docker run --cap-add=NET_ADMIN busybox sh -c "ip link eth0 down"
  ```

- Prevent any chown in the Docker container:

  ```
  docker run --cap-drop=CHOWN ...
  ```

Linux Capabilities



- Allow all capabilities except mknod :

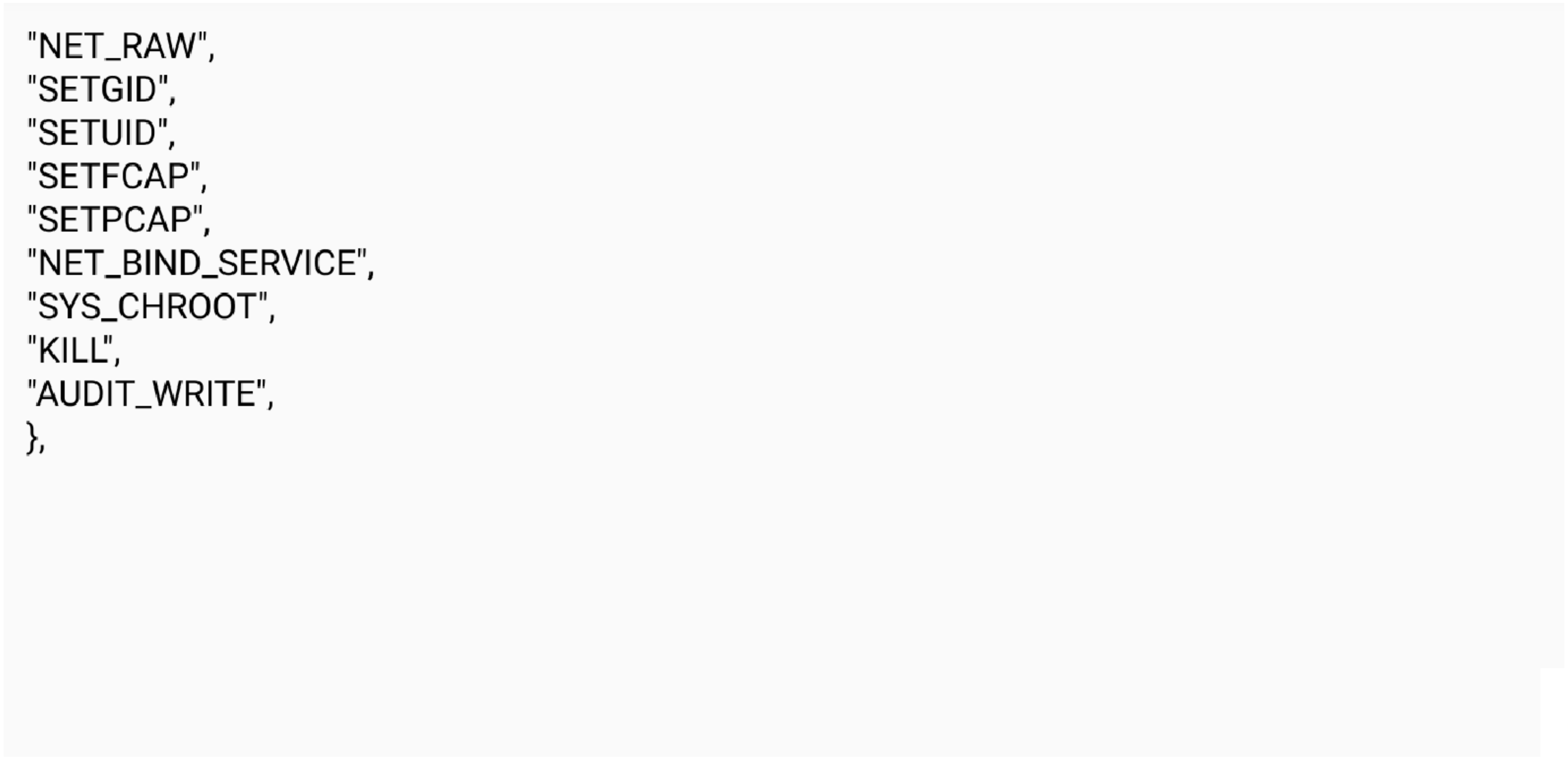
  docker run --cap-add=ALL --cap-drop=MKNOD ...

- Docker starts containers with a restricted set of capabilities by default.
- Capabilities convert a binary mode of root and non-root to a more fine-grained access control.

Docker allows only the following capabilities:

```
Capabilities: [ ]string{
"CHOWN",
"DAC_OVERRIDE",
"FSETID",
"FOWNER",
"MKNOD",
```

Linux Capabilities



```
"NET_RAW",
"SETGID",
"SETUID",
"SETFCAP",
"SETPCAP",
"NET_BIND_SERVICE",
"SYS_CHROOT",
"KILL",
"AUDIT_WRITE",
},
```

Docker Download Rate Limit

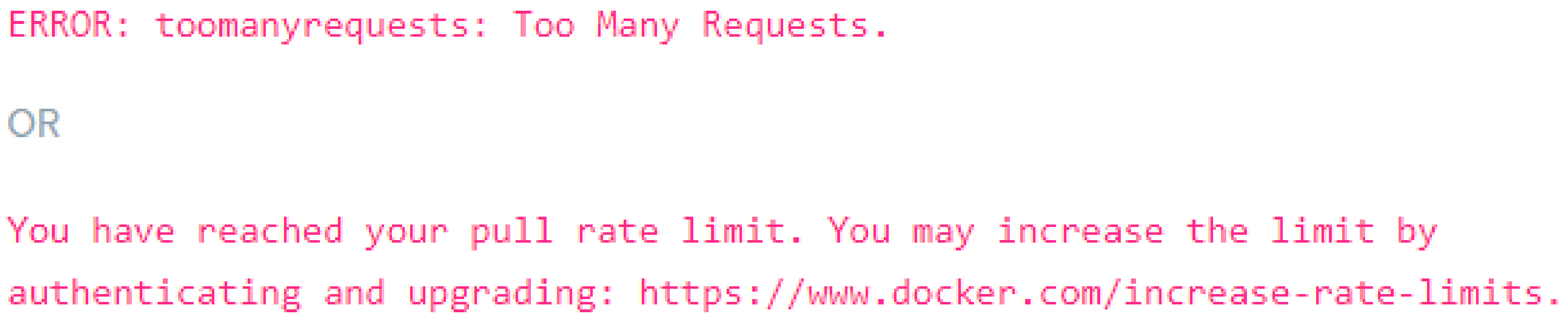
Docker Pull Rate Limit

Docker has enabled download rate limits for pull requests on Docker Hub. Limits are determined based on the account type.

Anonymous free users will be limited to **100 pulls per six hours**, and authenticated free users will be limited to **200 pulls per six hours**. **Docker Pro and Team subscribers** can pull container images from Docker Hub without restriction as long as the quantities are not excessive or abusive.

https://www.docker.com/pricing/resource-consumption-updates

Pull Rate Limit Error

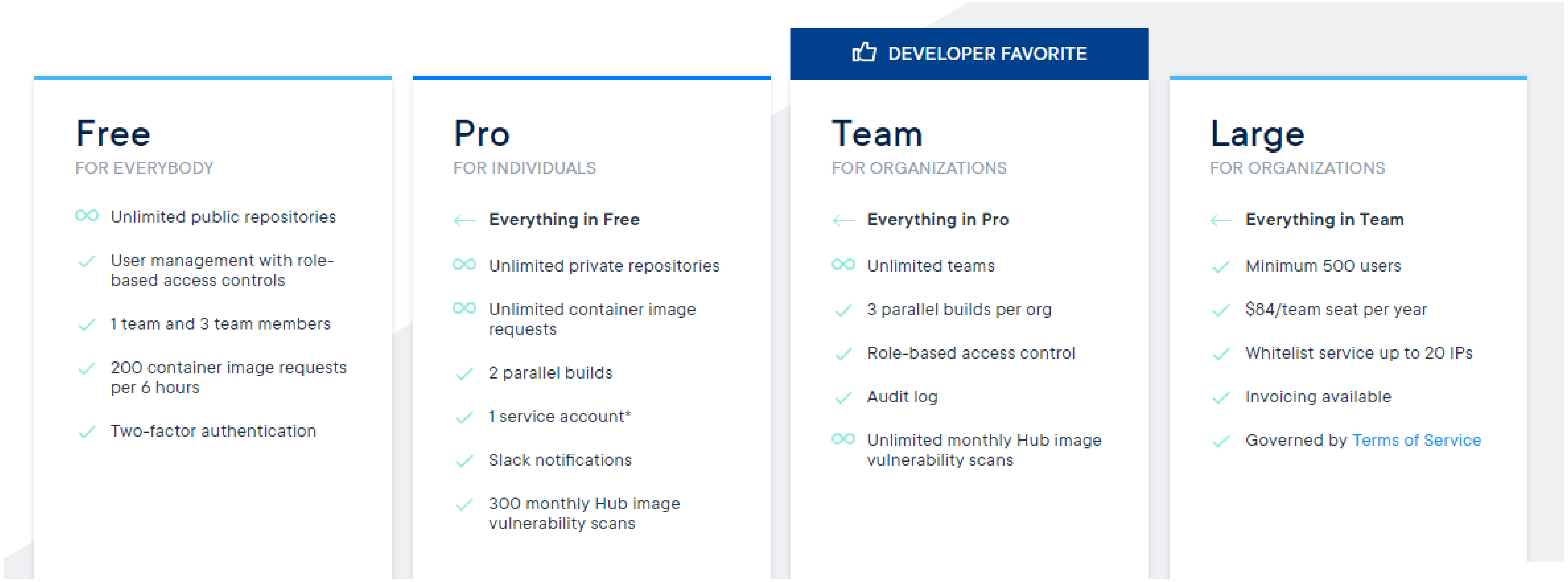


```
ERROR: toomanyrequests: Too Many Requests.
```

OR

```
You have reached your pull rate limit. You may increase the limit by
authenticating and upgrading: https://www.docker.com/increase-rate-limits.
```

Docker Subscription



DEVELOPER FAVORITE

Free
FOR EVERYBODY

- Unlimited public repositories
- User management with role-based access controls
- 1 team and 3 team members
- 200 container image requests per 6 hours
- Two-factor authentication

Pro
FOR INDIVIDUALS

- **Everything in Free**
- Unlimited private repositories
- Unlimited container image requests
- 2 parallel builds
- 1 service account*
- Slack notifications
- 300 monthly Hub image vulnerability scans

Team
FOR ORGANIZATIONS

- **Everything in Pro**
- Unlimited teams
- 3 parallel builds per org
- Role-based access control
- Audit log
- Unlimited monthly Hub image vulnerability scans

Large
FOR ORGANIZATIONS

- **Everything in Team**
- Minimum 500 users
- $84/team seat per year
- Whitelist service up to 20 IPs
- Invoicing available
- Governed by Terms of Service

KOENIG

Determining Current Limit Rate

For Anonymous User:

#yum install epel-release -y

#yum install jq -y

#TOKEN=$(curl "https://auth.docker.io/token?service=registry.docker.io&scope=repository:ratelimitpreview/test:pull" | jq -r .token)

#curl --head -H "Authorization: Bearer $TOKEN" https://registry-1.docker.io/v2/ratelimitpreview/test/manifests/latest 2>&1 | grep RateLimit

https://www.docker.com/blog/checking-your-current-docker-pull-rate-limits-and-status/

Determining Current Limit Rate

For Authenticated User:

#yum install epel-release -y

#yum install jq -y

#TOKEN=$(curl --user 'username:password' "https://auth.docker.io/token?service=registry.docker.io&scope=repository:ratelimitpreview/test:pull" | jq -r .token)

#curl --head -H "Authorization: Bearer $TOKEN" https://registry-1.docker.io/v2/ratelimitpreview/test/manifests/latest 2>&1 | grep RateLimit

https://www.docker.com/blog/checking-your-current-docker-pull-rate-limits-and-status/

KOENIG

Any
Question

“Thank You”