Learning Solution

Advanced Docker Concepts & Container Orchestration

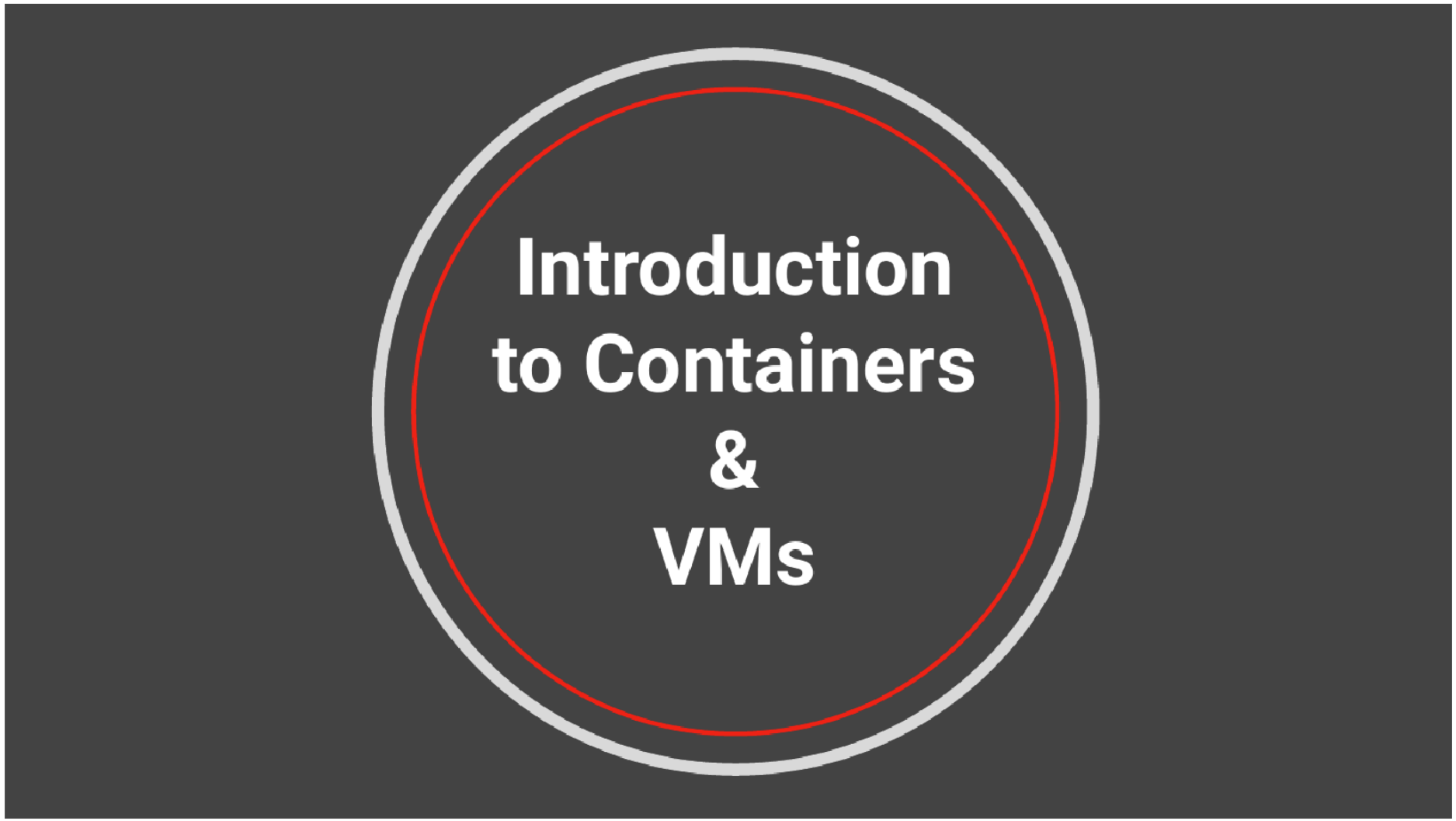
Gaurav

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


CONTAINERS, CONTAINERS
EVERYWHERE
memegenerator.net

Course Agenda

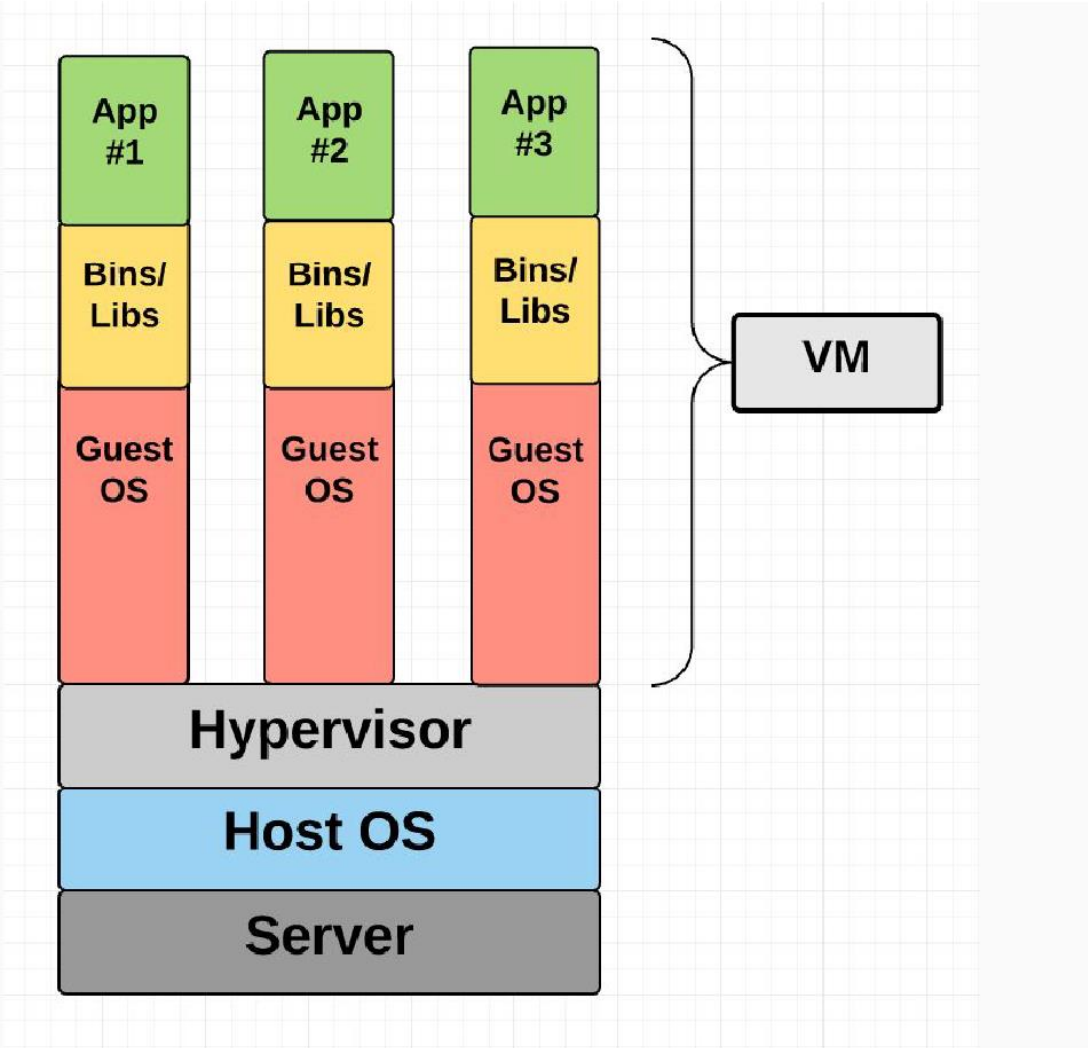
- Introducing Docker
- Installation of Docker
- Docker Client Operations
- Building Custom Images and Docker Registry
- Container Deep Dive
- Storage & Container Networking
- Docker Compose


Introduction
to Containers
&
VMs

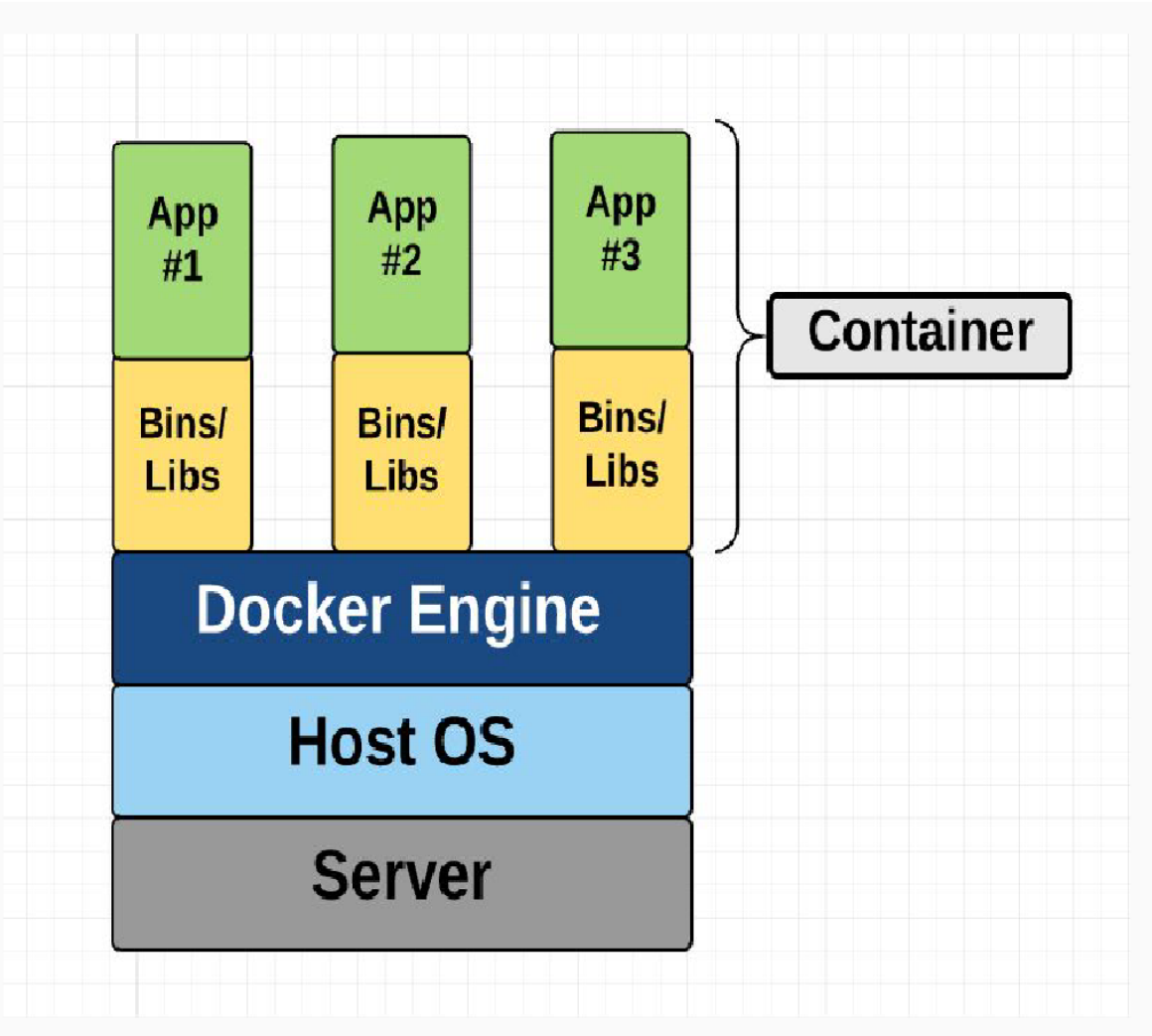
What are Containers & VMs ?

- Containers and VMs are similar in their goals: to isolate an application and its dependencies into a self-contained unit that can run anywhere.
- Moreover, containers and VMs remove the need for physical hardware, allowing for more efficient use of computing resources, both in terms of energy consumption and cost effectiveness.
- The main difference between containers and VMs is in their architectural approach.

- since the VM has a virtual operating system of its own, the hypervisor plays an essential role in providing the VMs with a platform to manage and execute this guest operating system.
- It allows for host computers to share their resources amongst the virtual machines that are running as guests on top of them.



- This diagram shows you that containers package up just the user space, and not the kernel or virtual hardware like a VM does.
- Each container gets its own isolated user space to allow multiple containers to run on a single host machine.
- We can see that all the operating system level architecture is being shared across containers.
- The only parts that are created from scratch are the bins and libs. This is what makes containers so lightweight.



Introduction to Docker

"Almost overnight, Docker has become the de facto standard that developers and system administrators use for packaging, deploying and running distributed and cloud native applications."

Learning Solution

It works on your machine, but it doesn't in production.

Problems we faced

How the services were traditionally configured and deployed ?

Dedicated Machines

- Let's say you had a single dedicated server that was running your PHP application.
- When you initially deployed the dedicated machine, all three of the applications, which make up your e-commerce website, worked with PHP 5.6, so there was no problem with compatibility.
- Your development team has been slowly working through the three PHP applications. You have deployed it on your host to make them work with PHP 7, as this will give them a good boost in performance.
- However, there is a single bug that they have not been able to resolve with App2, which means that it will not run under PHP 7 without crashing when a user adds an item to their shopping cart.
- If you have a single host running your three applications, you will not be able to upgrade from PHP 5.6 to PHP 7 until your development team has resolved the bug with App2.

Dedicated Machines

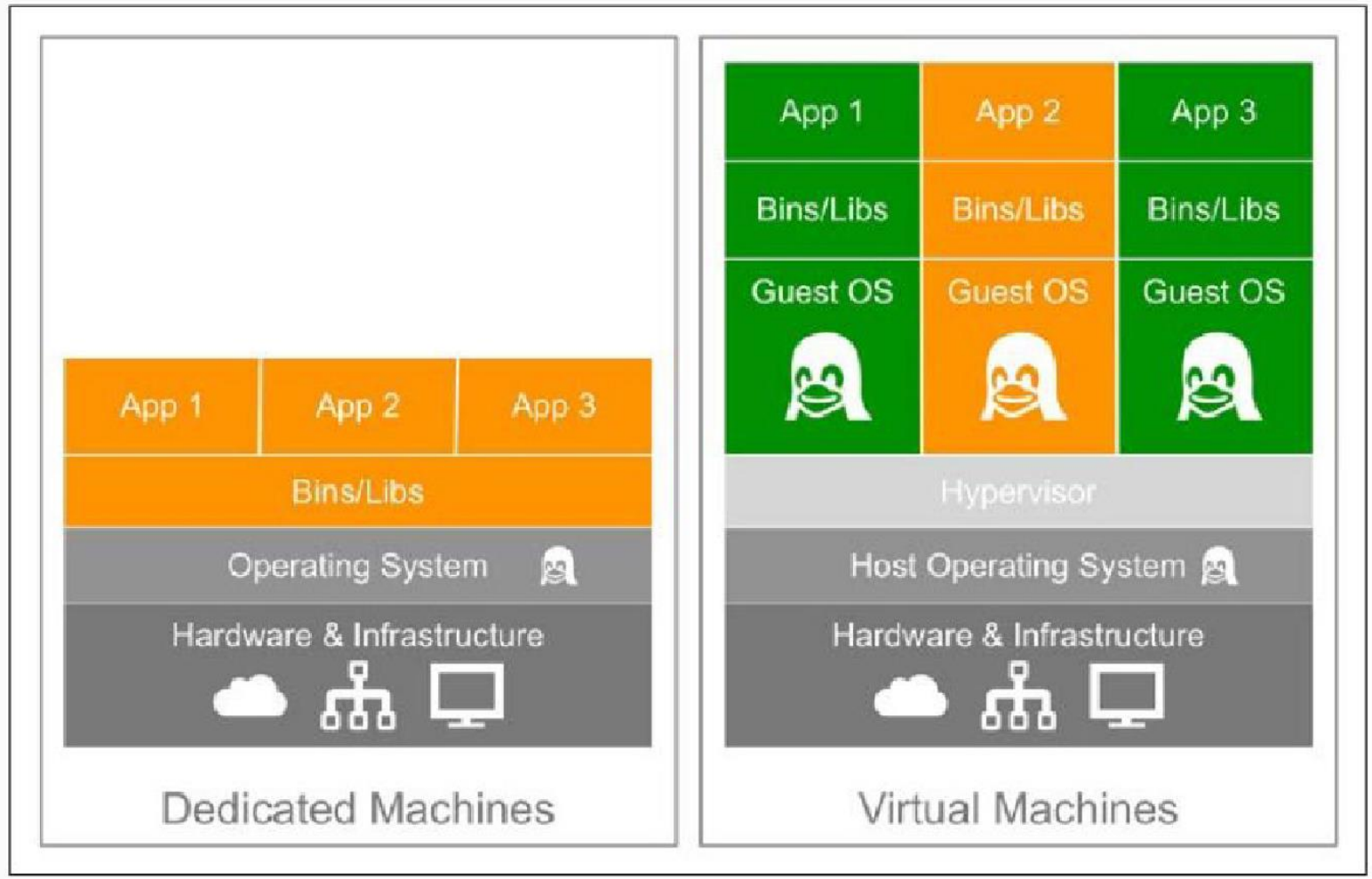
Unless you do one of the following:

- Deploy a new host running PHP 7 and migrate App1 and App3 to it; this could be both time consuming and expensive
- Deploy a new host running PHP 5.6 and migrate App2 to it; again this could be both time consuming and expensive
- Wait until the bug has been fixed; the performance improvements that the upgrade from PHP 5.6 to PHP 7 bring to the application could increase the sales and there is no ETA for the fix

Virtual Machines

One solution to the scenario detailed earlier would be to slice up your dedicated machine's resources and make them available to the application by installing a hypervisor.

- Once installed, you can then install your binaries and libraries on each of the different virtual hosts and also install your applications on each one.
- you will be able to upgrade to PHP 7 on the virtual machines with App1 and `App3` installed, while leaving App2 untouched and functional while the development work on the fix.
- **Great, So what's the catch**. From the developer's view, there is none as they have their applications running with the PHP versions, which work best for them; however, from an IT operations point of view:
    - **More CPU, RAM, and disk space:** Each of the virtual machines will require additional resources as the overhead of running three guest OS, as well as the three applications have to be taken into account
    - **More management:** IT operations now need to patch, monitor, and maintain four machines, the dedicated host machine along with three virtual machines, whereas before they only had a single dedicated host.


App 1
App 2
App 3
Bins/Libs
Operating System
Hardware & Infrastructure
Dedicated Machines
App 1
App 2
App 3
Bins/Libs
Bins/Libs
Bins/Libs
Guest OS
Guest OS
Guest OS
Hypervisor
Host Operating System
Hardware & Infrastructure
Virtual Machines

Containers

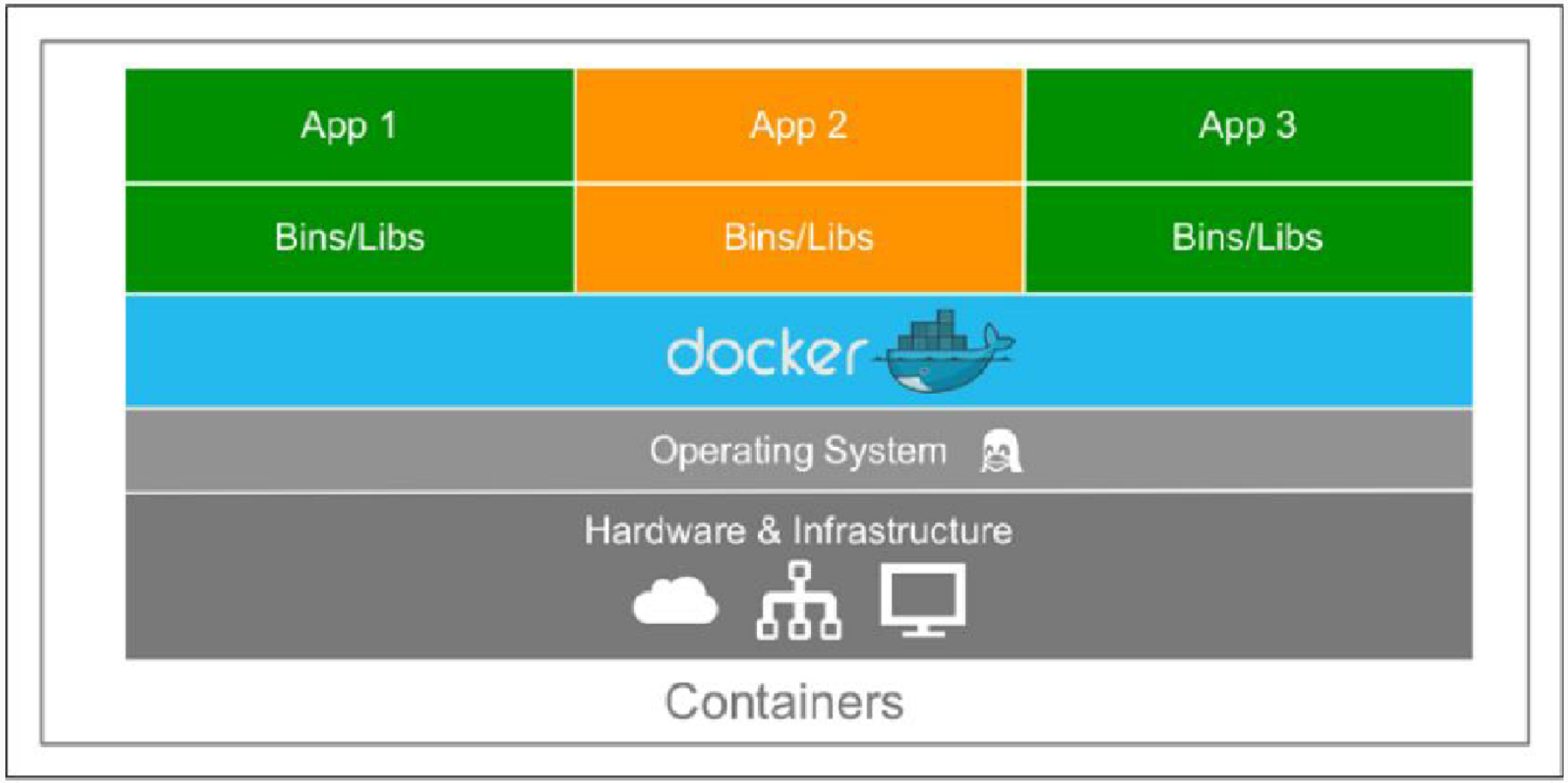
Let's look at what Docker adds to the mix :

- Installing Docker on the host and then deploying each of the applications as a container on this host gives you the benefits of the virtual machine.
- While vastly reducing the footprint, that is, removing the need for the hypervisor and guest operating system completely, and replacing them with a SlimLine interface directly into the host machines kernel.

Containers

The advantages this gives both the IT operations and development teams are as follows:

- **Low overhead:** As mentioned already, the resource and management for the IT operations team is lower.
- **Development provide the containers:** Rather than relying on the IT operations team to configure each of the three applications environments to machine the development environment, they can simply pass their containers to be put into production.


App 1
App 2
App 3
Bins/Libs
Bins/Libs
Bins/Libs
docker
Operating System
Hardware & Infrastructure
Containers

Containers are far from new.

Google has been using their own container technology for years.

Others Linux container technologies include Solaris Zones, BSD jails, and LXC, which have been around for many years.

Is Docker, our only solution ?

Learning Solution

Rkt, Mesos, LXC, OpenVZ, Containerd

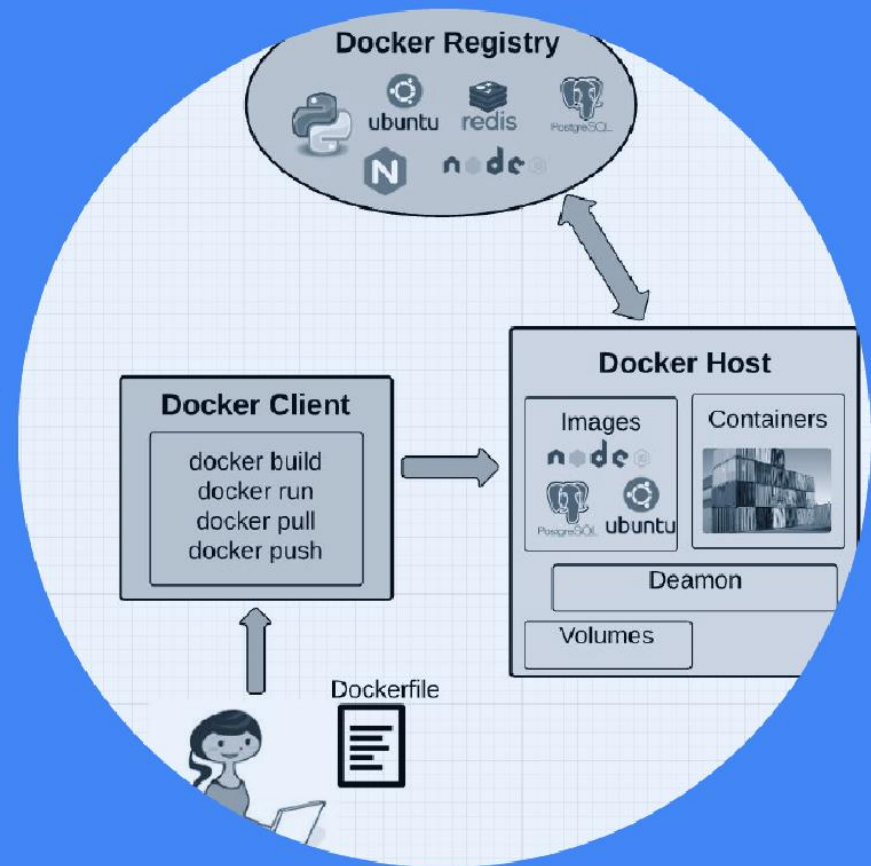
About

- 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.

About docker

- Ease of use
  - build once, run anywhere
- Speed
- Docker Hub
- Modularity and Scalability

Fundamental Docker Concepts



Docker Concepts

- Docker Engine
  - Docker Daemon
  - Docker Client
  - REST API
- Dockerfile
- Docker Image
- Docker Containers

Downloading and Installing Docker

Requirements

- Be running a 64-bit architecture
- Be running a Linux 3.8 or later kernel
- The kernel must support an appropriate storage driver.
  For example,
  - Device Mapper
  - UFS
  - VFS
  - BTRFS
  - Cgroups
  - Namespace

Checking for prerequisites:

- $ uname -a
- $ ls -l /sys/class/misc/device-mapper
- $ sudo grep device-mapper /proc/devices
- #wget https://download.docker.com/linux/centos/docker-ce.repo -O \
  /etc/yum.repos.d/docker-ce.repo
- # yum install docker-ce -y
- # systemctl start docker
- # systemctl enable docker
- # systemctl status docker

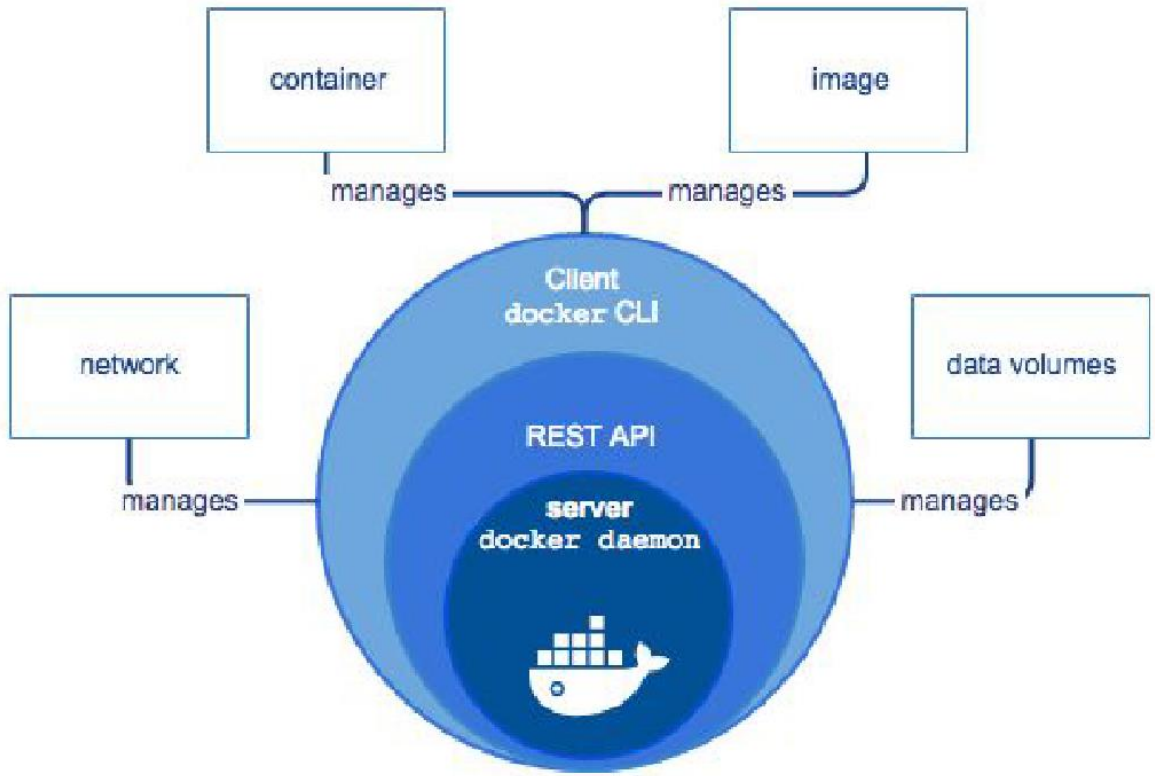
After Docker Installation

- # docker version
- # docker info
- # docker images
- # docker ps

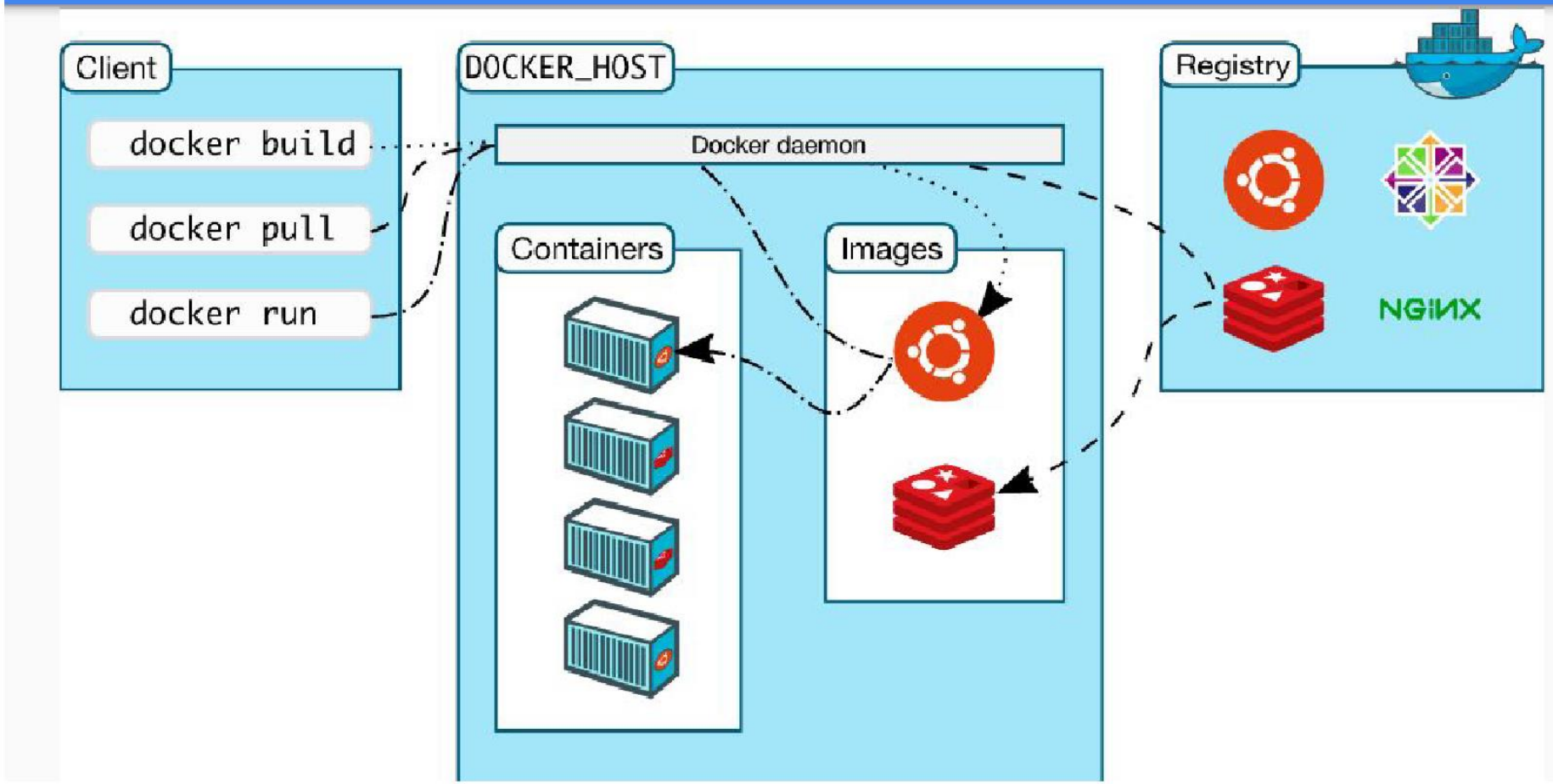
Docker Essential Commands

View all commands	$ docker
Search for images	$ docker search *<string>*
Download image	$ docker pull *<username>*/*<repository>*
Inspect a container	$ docker inspect *<container id>*
Access a running container	$ docker exec -it *<container-name/-id>* bash
View logs of a container	$ docker logs -f *<container-id/-name>*
View the host port that a container app's port maps:	$ docker port *<container_name>* *<container_app_port>*
Run a command from an image/container	$ docker run *<username>*/<repository> *<command args>*
Run the image on the same network stack of the host	$ docker run - -net=host *<username>*/*<image>*

Understanding Docker Engine



Docker Architecture



The Underlying Technology

- Docker is written in Go.
- Uses several features of Linux kernel to deliver its functionality.
- Docker uses a technology called ***namespaces*** to provide the isolated workspace called the ***container***.
- When you run a container, Docker creates a set of *namespaces* for that container.
- These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

The Underlying Technology

Docker Engine uses namespaces such as the following on Linux:

- **The pid namespace:** Process isolation (PID: Process ID).
- **The net namespace:** Managing network interfaces (NET: Networking).
- **The ipc namespace:** Managing access to IPC resources (IPC: InterProcess Communication).
- **The mnt namespace:** Managing filesystem mount points (MNT: Mount).
- **The uts namespace:** Isolating kernel and version identifiers. (UTS: Unix Timesharing System).

Running your first container

docker run -d -it -P --name c1 nginx /bin/bash

-d → detach from terminal (optional)

-i → interactive/user can interact (optional)

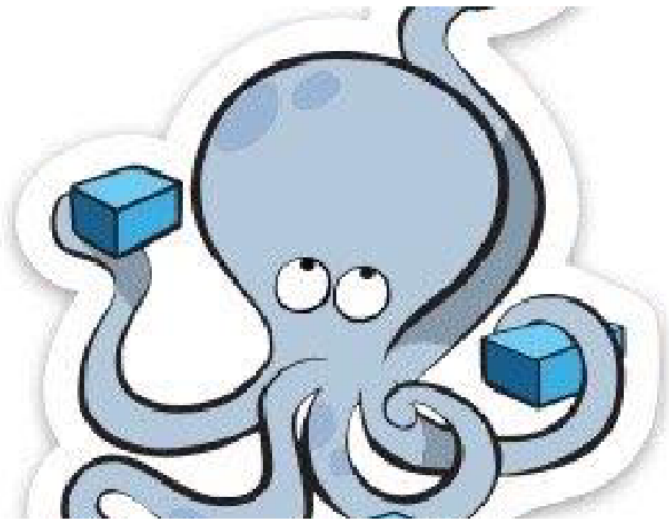
-t → assign terminal (tty) to the container (optional)

-P → Publish service running in container on a specific port to any random port on host

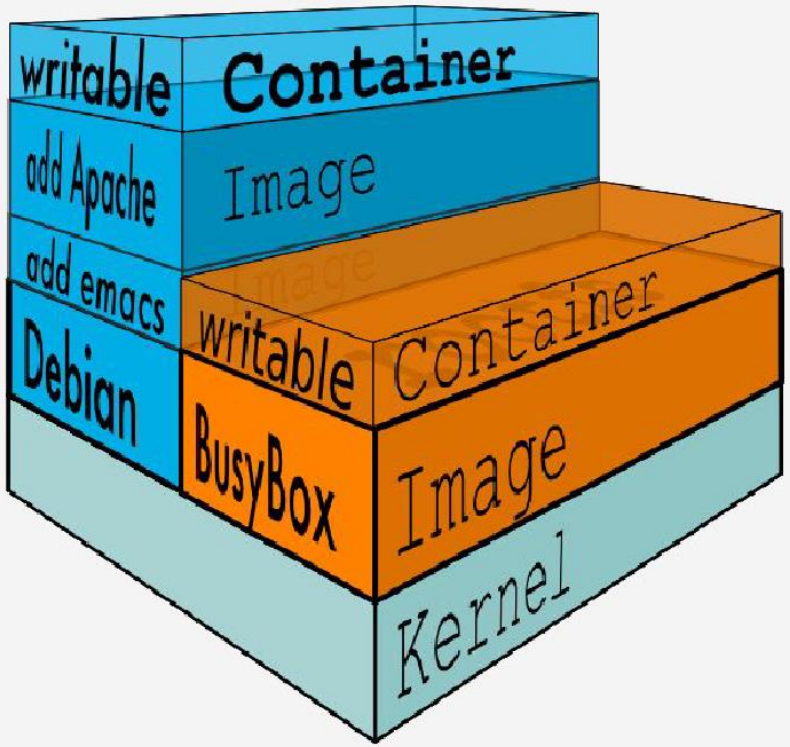
--name → name of the container (optional)

nginx → image name to be used to spin up the container with

/bin/bash → any command to run in container after creation (optional)



Deep dive into docker image


writable
Container
add Apache
Image
add emacs
writable
Container
Debian
BusyBox
Image
Kernel

Displaying Docker Images

To see the list of Docker images on the system, you can issue the following command.

docker images

Output :

```
demo@ubuntuserver:~$ sudo docker images
[sudo] password for demo:
REPOSITORY          TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
newcentos           latest              7a86f8ffcb25        9 days ago
196.5 MB
jenkins             latest              998d1854867e        2 weeks ago
714.1 MB
centos              latest              97cad5e16cb6        4 weeks ago
196.5 MB
demo@ubuntuserver:~$ _
```

The image command display following columns

- Repository
- Tag
- Image ID
- Created
- Size

Docker Images

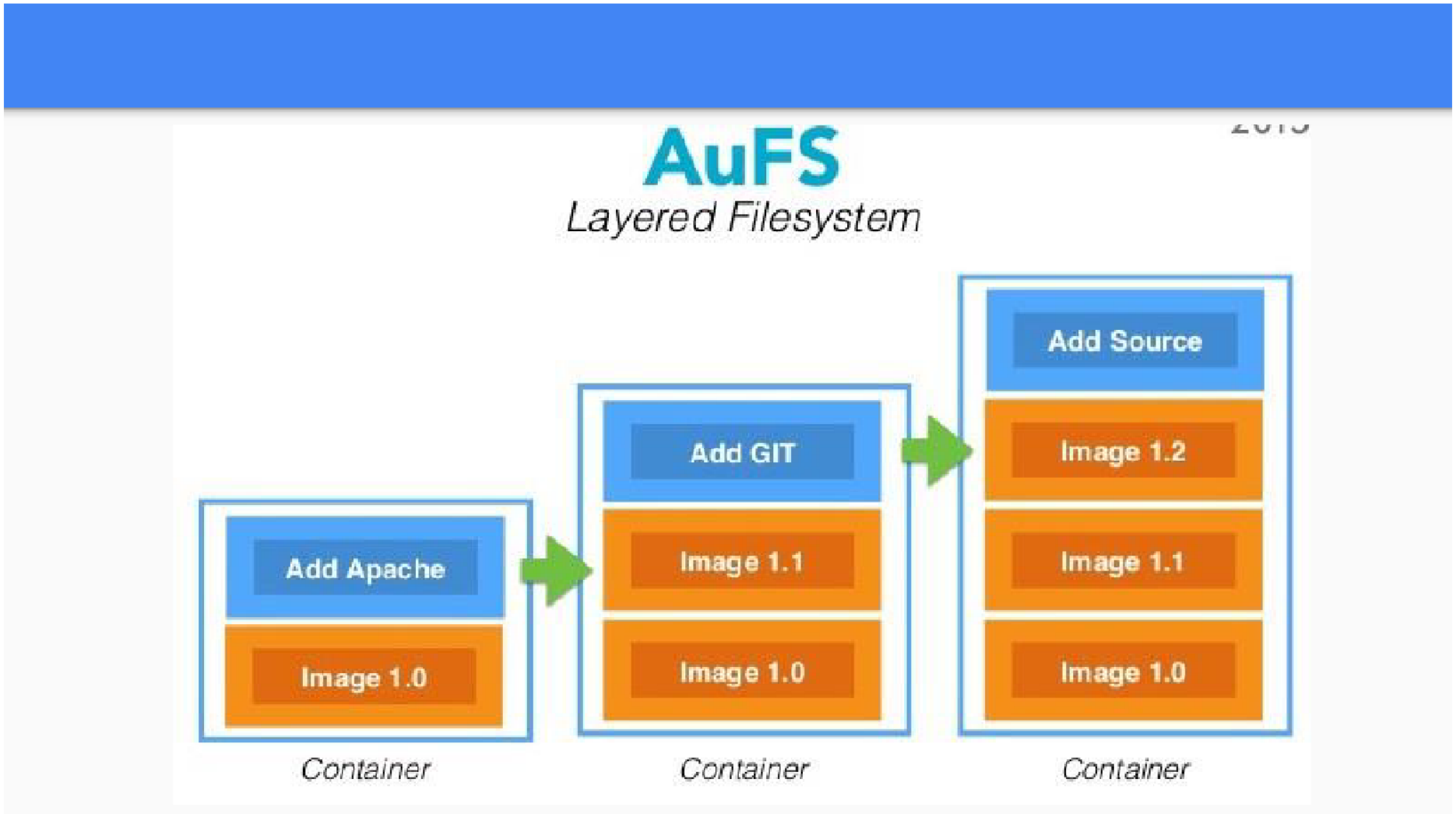
- Downloading docker images
  - docker pull *image*
- Remove docker images
  - docker rmi *imageID*
- To see imageID's
  - docker images -q
- Image detail
  - docker inspect *Repository*

Understanding Docker Image Layers

Union File Systems

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.


AuFS
Layered Filesystem
Add Apache
Image 1.0
Container
Add GIT
Image 1.1
Image 1.0
Container
Add Source
Image 1.2
Image 1.1
Image 1.0
Container

What are these layers keep coming Up ?

How are Images used to create Containers?

- When you spawn a container (*docker run <image-name>*), each gets its own thin writable container layer, and all changes are stored in this container layer, this means that multiple containers can share access to the same underlying image and yet have their own data state.
- When a container is deleted, all data stored is lost.
- For databases and data-centric apps, which require persistent storage, Docker allows mounting host's filesystem directly into the container.
- This ensures that the data is persisted even after the container is deleted, and the data can be shared across multiple containers.
- Docker also allows mounting data volumes from external storage arrays and storage services like AWS EBS via its Docker Volume Plug-ins.

Dockerfile

- Image Instructions
  - FROM
  - MAINTAINER
  - ADD
  - RUN
  - COPY
  - CMD
  - ENTRYPOINT
  - ENV
  - EXPOSE

What are these layers keep coming Up ?

Let's consider the following Dockerfile to build a simple Ubuntu image with an Apache installation

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y apache2
RUN touch /opt/a.txt
```

- When Docker builds the container from the above Dockerfile, each step corresponds to a command run in the Dockerfile.
- And each layer is made up of the file generated from running that command.
- Along with each step, the layer created is listed represented by its random generated ID.
- Once the image is built, you can view all the layers that make up the image with the ***docker history*** command
- An image becomes a container when the ***docker run*** command is executed.

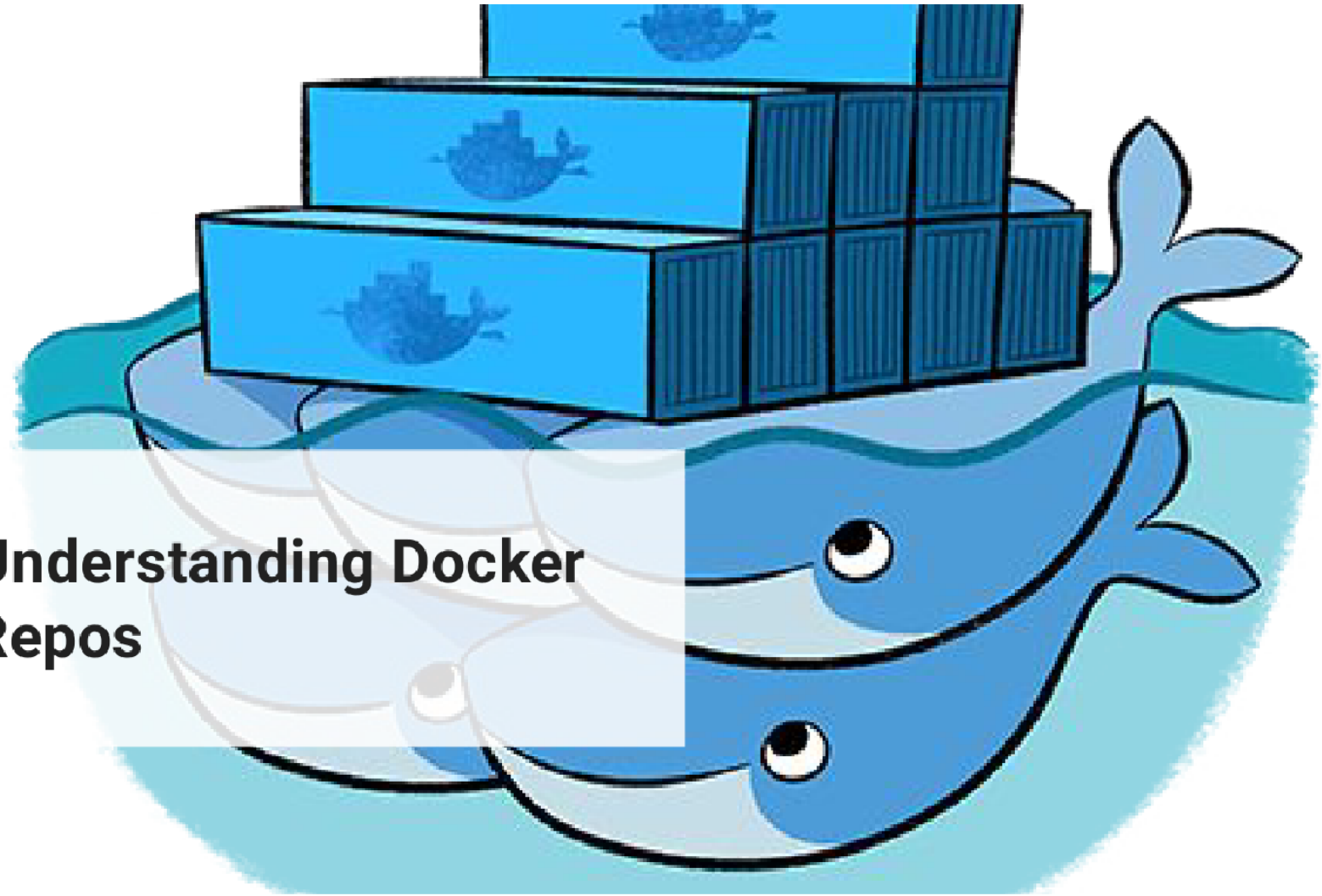
Now you might want to rollback the last layer of image

```
$ docker history test/b
IMAGE               CREATED             CREATED BY                                      SIZE
c0daf4be2ed4        42 seconds ago      /bin/sh -c touch /opt/b.txt                     8 B
9977b78fbad7        About a minute ago  /bin/sh -c apt-get install -y apache2           54.17 MB
e83b3bf07b42        3 minutes ago       /bin/sh -c apt-get update                       20.67 MB
9cd978db300e        3 months ago        /bin/sh -c #(nop) ADD precise.tar.xz in /       204.4 MB
6170bb7b0ad1        3 months ago        /bin/sh -c #(nop) MAINTAINER Tianon Gravi <ad   0 B
511136ea3c5a        10 months ago                                                       0 B
```

If we want to make a rollback and remove the last layer we can do so by tagging the layer 9977b78fbad7:

$ docker tag 9977b test/b

```
$ docker history test/b
IMAGE               CREATED             CREATED BY                                      SIZE
9977b78fbad7        3 hours ago         /bin/sh -c apt-get install -y apache2           54.17 MB
e83b3bf07b42        3 hours ago         /bin/sh -c apt-get update                       20.67 MB
9cd978db300e        3 months ago        /bin/sh -c #(nop) ADD precise.tar.xz in /       204.4 MB
6170bb7b0ad1        3 months ago        /bin/sh -c #(nop) MAINTAINER Tianon Gravi <ad   0 B
511136ea3c5a        10 months ago                                                       0 B
```



Understanding Docker Repos

Docker - Public Repositories

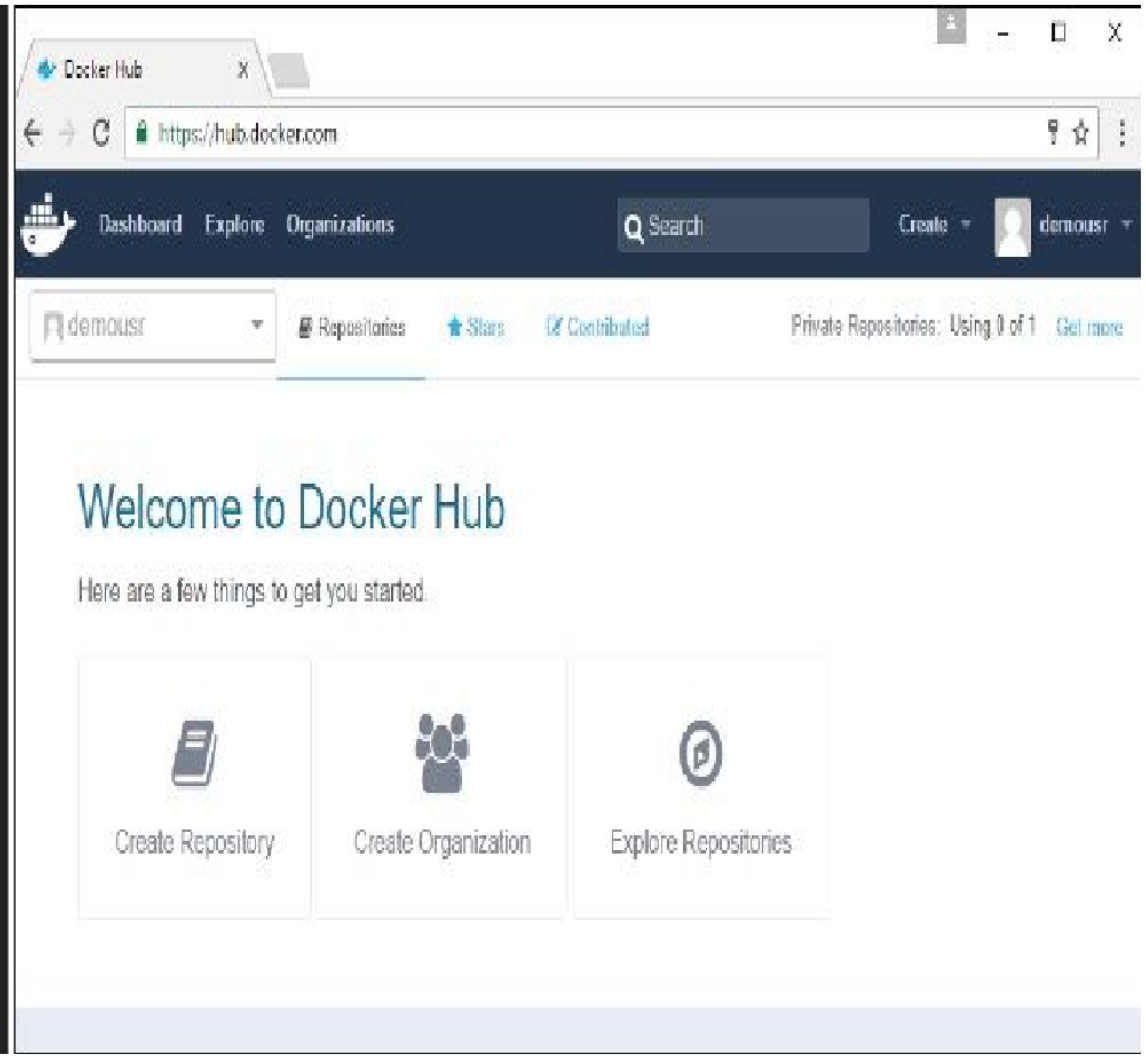
- Public repositories can be used to host Docker images which can be used by everyone else.
- Most of the images such as Centos, Ubuntu, and Jenkins are all publicly available for all.
- We can also make our images available by publishing it to the public repository on Docker Hub.

- Let's use this to upload to the Docker public repository.

The following steps explain how you can upload an image to public repository.

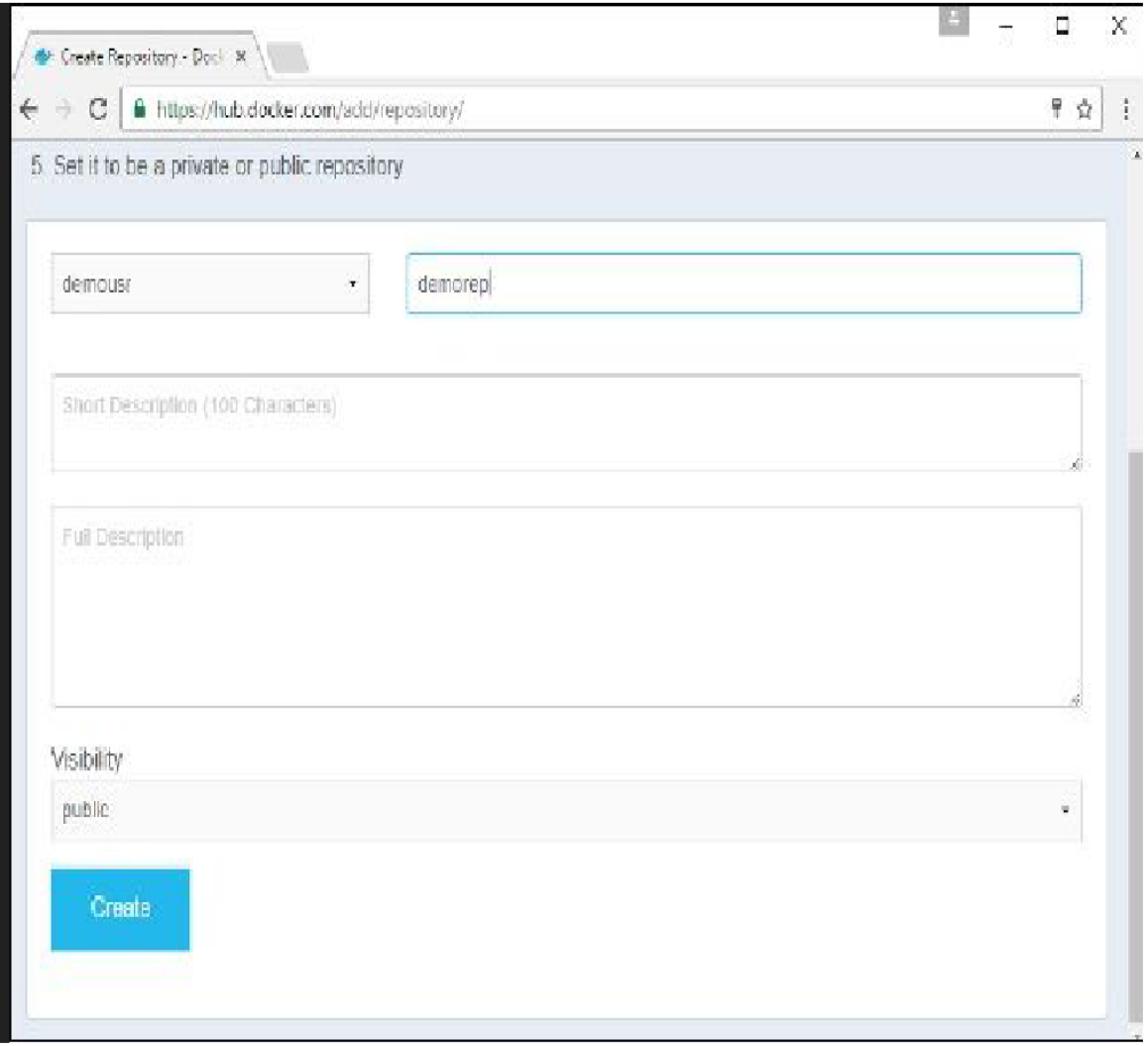
STEP 1

1. Log into Docker Hub and create your repository.
2. This is the repository where your image will be stored.
3. Go to https://hub.docker.com/ and log in with your credentials.



STEP 2

1. Click the button "Create Repository" on the above screen and create a repository with the name demorep.
2. Make sure that the visibility of the repository is public.

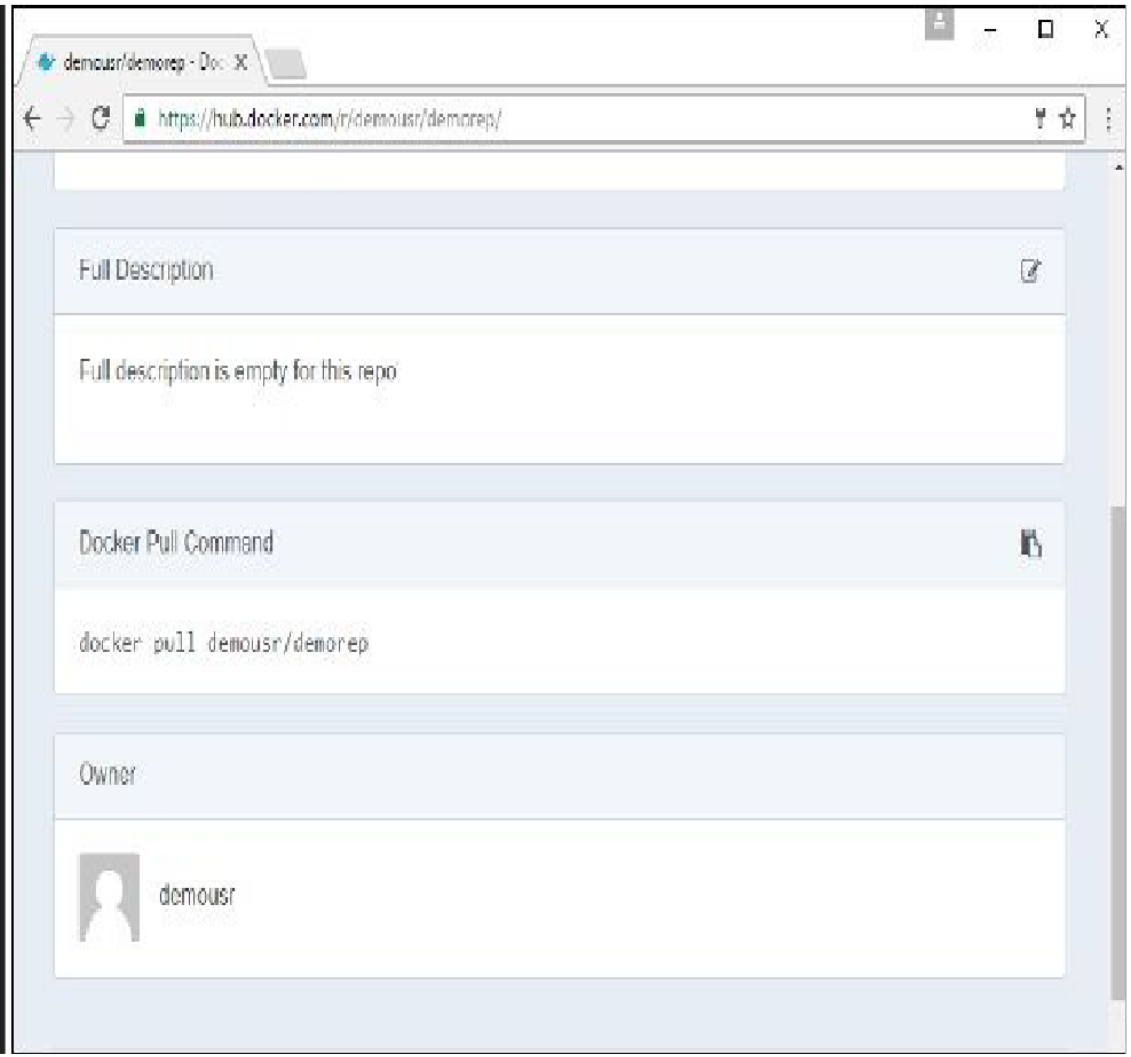


STEP 3

Once the repository is created, make a note of the pull command which is attached to the repository.

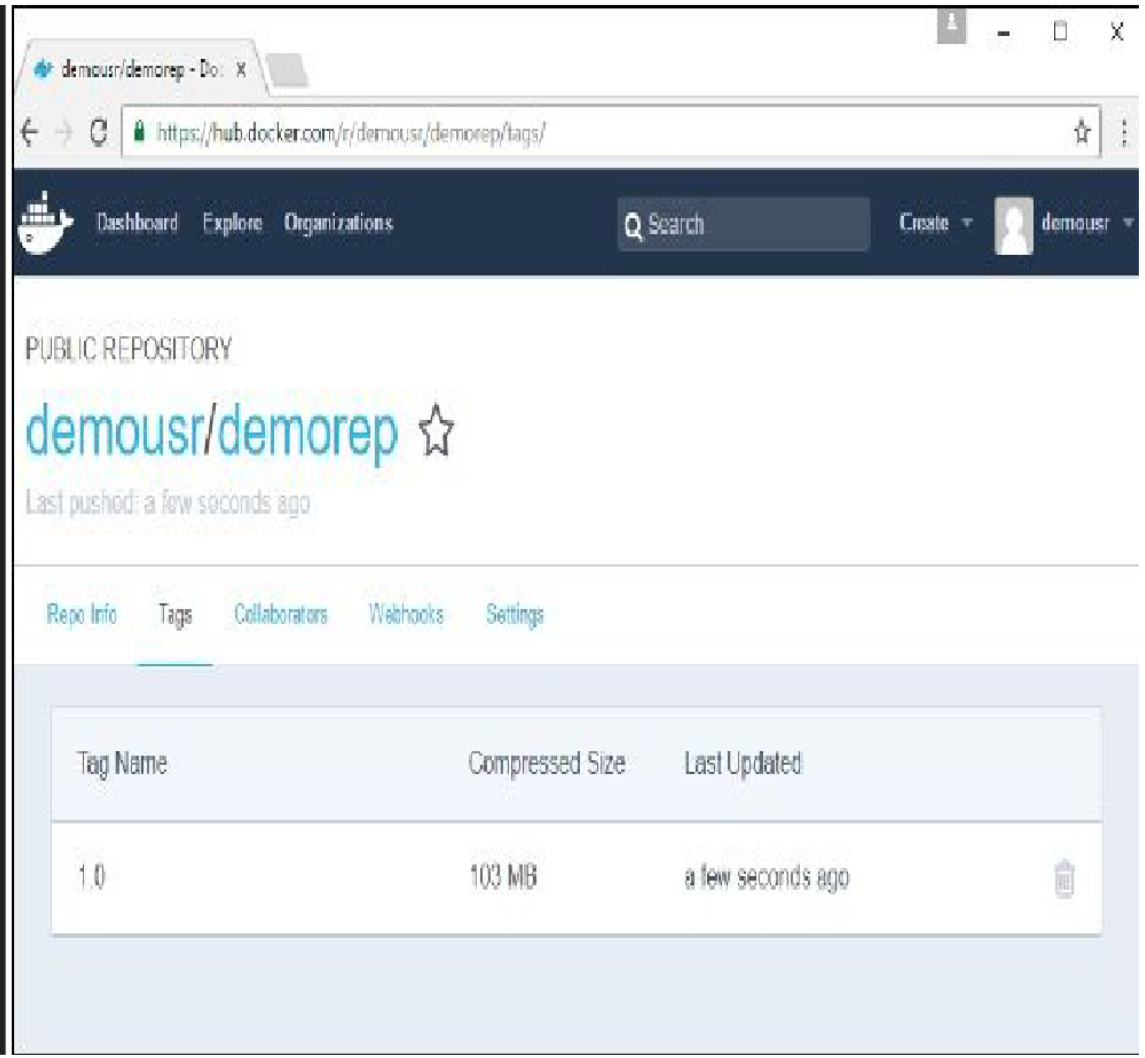
The pull command which will be used in our repository is as follows –

docker pull demousr/demorep



STEP 4

1. Now go back to the Docker Host. Here we need to tag our myimage to the new repository created in Docker Hub.
2. Issue the Docker login command to login into the Docker Hub repository from the command prompt.
3. The Docker login command will prompt you for the username and password to the Docker Hub repository.



DOCKER TAG

This method allows one to tag an image to the relevant repository.

docker tag imageID Repositoryname

imageID – This is the ImageID which needs to be tagged to the repository.

Repositoryname – This is the repository name to which the ImageID needs to be tagged to.

Example

sudo docker tag ab0c1d3744dddemousr/demorep:1.0

```
demo@ubuntudemo:~$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
myimage             0.1                 ab0c1d3744dd        6 minutes ago
225.3 MB
centos              latest              67591570dd29        2 days ago
191.8 MB
ubuntu              latest              104bec311bcd        2 days ago
129 MB
demo@ubuntudemo:~$ sudo docker tag ab0c1d3744dd demousr/demorep:1.0
demo@ubuntudemo:~$
```

DOCKER PUSH

This method allows one to push images to the Docker Hub.

docker push Repository name

Repository name – This is the repository name which needs to be pushed to the Docker Hub.

Example
sudo docker push demousr/demorep:1.0

```
demo@ubuntudemo:~$ sudo docker push demousr/demorep:1.0
The push refers to a repository [docker.io/demousr/demorep]
2fa3ddba4e69: Layer already exists
ef84b80e23cc: Layer already exists
5972ebe5b524: Layer already exists
3d515508d4eb: Layer already exists
bbe6cef52379: Layer already exists
87f743c24123: Pushed
32d75bc97c41: Layer already exists
1.0: digest: sha256:1bcdae3a9270a95798f02cd287b91956c5a6cf9fae08d82eb3d11f3a22d4
8d42 size: 1781
demo@ubuntudemo:~$
```

Private repositories on the Docker Hub

- The Docker Hub provides both a public and private repository. The public repository is free to users and private is a paid service.
- The plans with private repositories are available in different sizes, such as a micro, small, medium, or large subscription.
- Docker has published their public repository code to open source at https://github.com/docker/docker-registry.
- Normally, enterprises will not like to keep their Docker images either in a Docker public or private repository.
- They prefer to keep, maintain, and support their own repository. Hence, Docker also provides the option for enterprises to create and install their own repository.

Private repositories on the Docker Hub

Let's create a repository in the local machine using the registry image provided by Docker.
We will run the registry container on the local machine, using the registry image from Docker:

```
$ sudo docker run -p 5000:5000 -d registry
768fb5bcbe3a5a774f4996f0758151b1e9917dec21aedf386c5742d44beafa41
```

Let's tag the image ID 224affbf9a65 to our locally created registry image. This image registry may have multiple variants in the repository, so this tag will help you identify the particular image:

```
$ sudo docker tag 224affbf9a65 localhost:5000/vcodeburster/dockerfileimageforhub
```

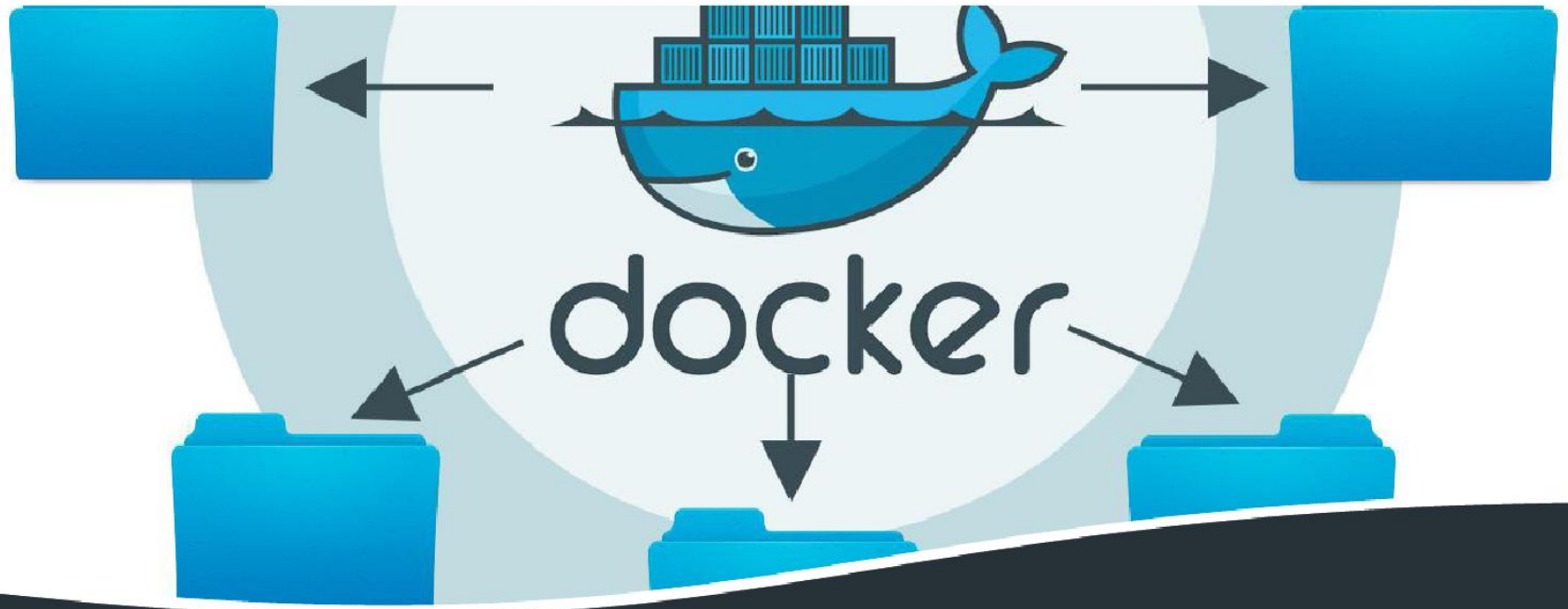
Once the tagging is done, push this image to a new registry using the docker push command:

```
$ sudo docker push localhost:5000/vinoddandy/dockerfile imageforhub
```

The push refers to a repository
[localhost:5000/codeburster/dockerfileimageforhub]

Docker container commands

- docker container logs <container-name>
- docker container export <container-name> → exports to a tar archive
- docker container kill <container-name>
- docker container pause <container-name>
- docker container prune <container-name> → remove all stopped container
- docker container rename <container-name>
- docker container stop <container-name>
- docker container rm <container-name>
- docker container top <container-name>
- docker container cp SRC_PATH <container-name>:DEST_PATH



Docker Storage

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

Image Storage

- The copy-on-write Mechanism
  - Docker engine does not make full copy of the already stored image.
- The Union File System
  - specialises in not storing duplicate data

Storage Plugins

Ideally, very little data is written to a container's writable layer, and you use Docker volumes to write data.

However, some workloads require you to be able to write to the container's writable layer.

This is where storage drivers come in.

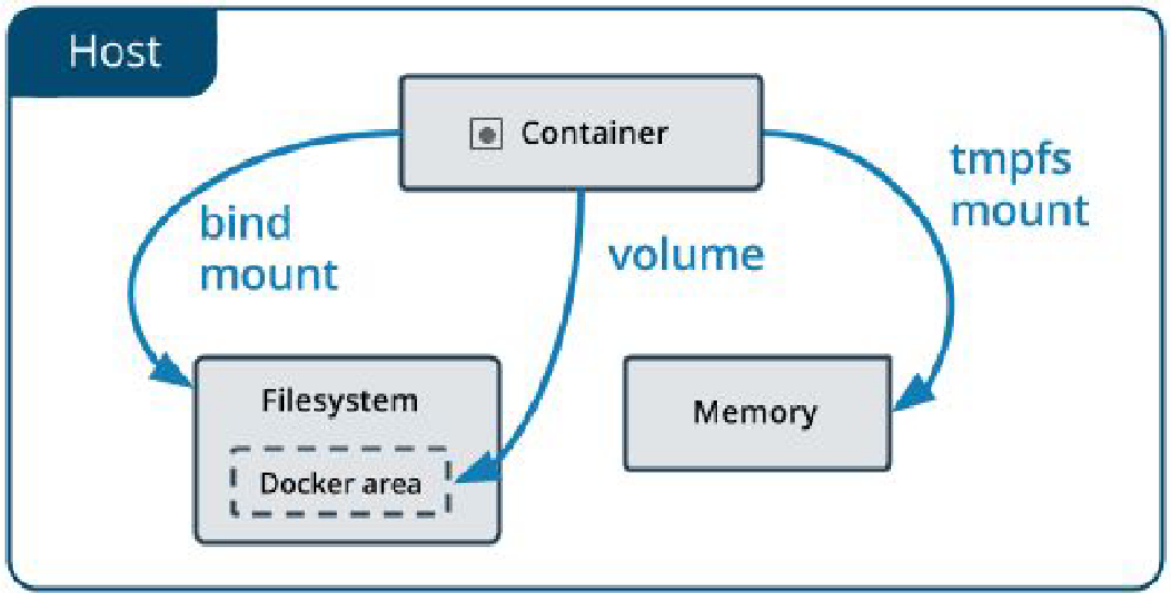
By introducing storage plugins in Docker, many options are available for the Copy-On-Write functionality, for example:

- OverlayFS (CoreOS)
- AUFS (Ubuntu)
- device mapper (RHEL)
- btrfs (next-gen RHEL)
- ZFS (next-gen Ubuntu releases)

Manage data in Docker

- Docker has two options for containers to store files in the host machine, so that the files are persisted even after the container stops
  - Volumes
  - Bind mounts
- If you're running Docker on Linux you can also use a *tmpfs* mount.

Choosing right type of mount



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***"

Use cases 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.

Commands for Volumes

- docker volume create my-vol
- docker volume ls
- docker volume inspect my-vol
- docker volume rm my-vol
- docker run -d --name devtest --mount source=myvol2,target=/app nginx:latest

Populate a volume using a container

- docker run -d --name nginxtest --mount source=nginx-vol,destination=/usr/share/nginx/html nginx:latest
- docker run -d --name nginxtest --mount \
  source=nginx-vol,destination=/usr/share/nginx/html,readonly nginx:latest

Bind mounts

- Bind mounts have limited functionality compared to volumes.
- When you use a bind mount, a file or directory on the host machine is mounted into a container.
- The file or directory does not need to exist on the Docker host already.
- Relies on the host machine's filesystem having a specific directory structure available.
- You can't use Docker CLI commands to directly manage bind mounts.

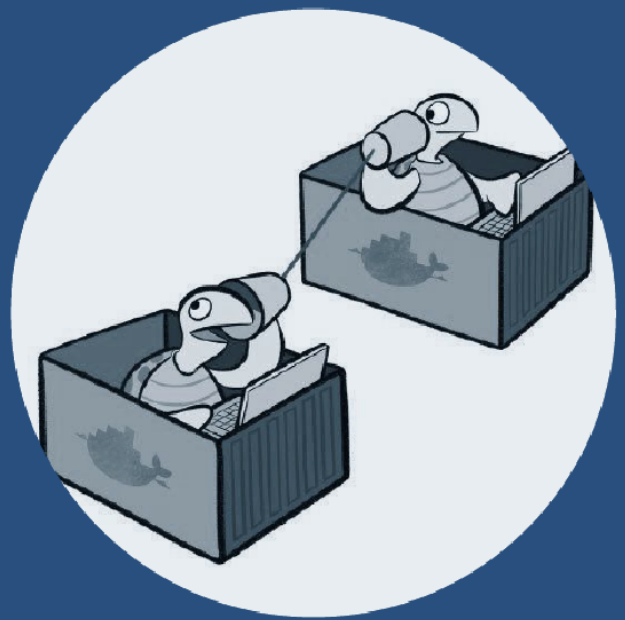
"One side effect of using bind mounts, for better or for worse, is that you can change the host filesystem via processes running in a container, including creating, modifying, or deleting important system files or directories."

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.

Learning Solution

Deep dive into Docker Networking



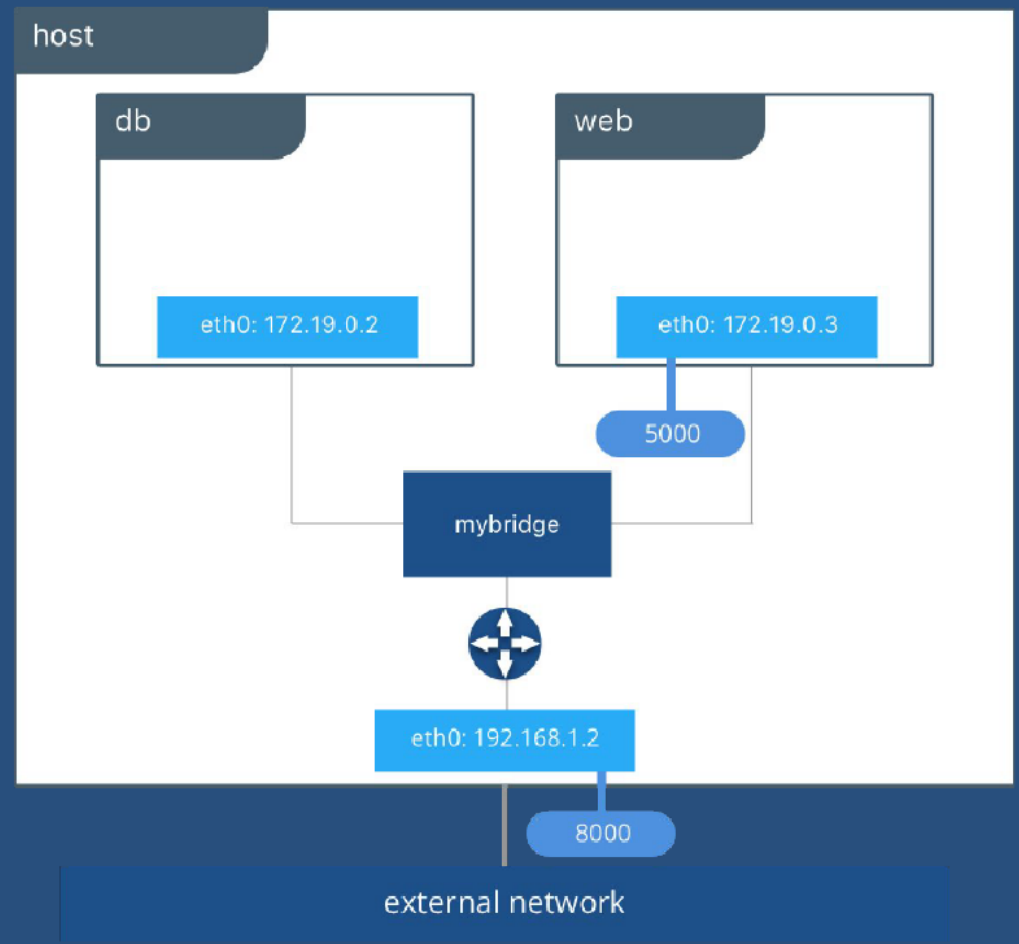
We have managed to make it so far without having to really think about networking.

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.

Let's see what I mean ...

- An application can be composed of a single process running in a Docker container.
- It could be made up of multiple processes running in their own containers and being replicated as the load increases.
- Each Docker container has its own network stack, because of Linux Kernel NET namespace.
- This NET namespace cannot be seen from outside the container or from other containers.



The docker0 bridge

“The docker0 bridge is the heart of default networking”

- Linux bridge is created on the host machine by default upon starting the service.
- The interfaces on containers talk to bridge.
- The bridge proxies to the external world.
- Multiple containers on the same host can talk to each other through Linux Bridge.
- docker0 can be configured via *--net* flag

Modes of --net flag

- --net default
- --net=none
- --net=container:$container2
- --net=host

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.

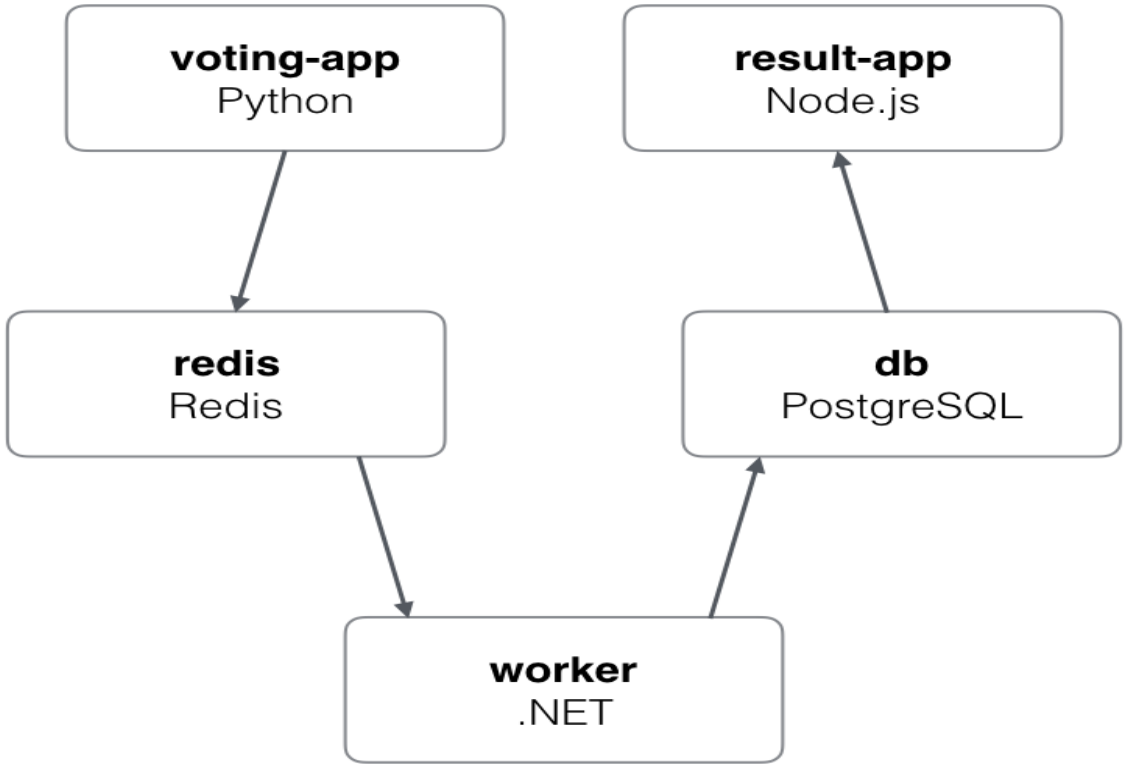
Docker allows only the following capabilities:

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

Linux capabilities

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

Deploying an Voting Application



Docker-Compose

- Compose is a tool for defining and running multi-container Docker applications.
- With Compose, you use a YAML file to configure your application's services.
- Then, with a single command, you create and start all the services from your configuration.

- Using Compose is basically 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 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.

Install Compose

- #curl -L "https://github.com/docker/compose/releases/download/1.26.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
- #chmod +x /usr/local/bin/docker-compose
- #docker-compose --version

Docker Compose - Versions

docker-compose.yml

```
redis:
    image: redis
db:
    image: postgres:9.4
vote:
    image: voting-app
    ports:
        - 5000:80
    links:
        - redis
```

version 1

docker-compose.yml

```
version: 2
services:
    redis:
        image: redis
    db:
        image: postgres:9.4
    vote:
        image: voting-app
        ports:
            - 5000:80
        depends_on:
            - redis
```

version 2

docker-compose.yml

```
version: 3
services:
    redis:
        image: redis
    db:
        image: postgres:9.4
    vote:
        image: voting-app
        ports:
            - 5000:80
        depends_on:
            - redis
```

version 3