Working with Docker

Mohanraj Shanmugam

Docker Command Line

- Docker Command is used to manage the docker
- Docker command can be used to manage the local host or remote host by “docker –H host
- By default, the Docker command line stores its configuration files in a directory called `.docker` within your `HOME` directory. However, you can specify a different location via the `DOCKER_CONFIG` environment variable or the`--config` command line option.
- If both are specified, then the `--config`option overrides the `DOCKER_CONFIG` environment variable.

Run a Docker Container

- Run a container using Command “docker run”

```
docker run ubuntu /bin/echo 'Hello world‘
Hello world
```

- The `docker run` combination *runs* containers.
- Next we specified an image: `ubuntu`.
- This is the source of the container we ran.
- Docker calls this an image.
- In this case we used the Ubuntu operating system image.
- When you specify an image, Docker looks first for the image on your Docker host.
- If it can’t find it then it downloads the image from the public image registry: Docker Hub.
- Next we told Docker what command to run inside our new container:

```
/bin/echo 'Hello world'
```

An interactive container

- You can get the console of the container by running a shell

```
docker run -t -i ubuntu /bin/bash
root@af8bae53bdd3:/#
```

`-t` flag assigns a pseudo-tty or terminal inside our new container and the `-i` flag allows us to make an interactive connection by grabbing the standard in (`STDIN`) of the container.

```
root@af8bae53bdd3:/#
root@af8bae53bdd3:/# pwd /
root@af8bae53bdd3:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr
var
root@af8bae53bdd3:/# exit
```

A daemonized Hello world

```
#docker run -d ubuntu /bin/sh -c "while true; do echo hello world; sleep 1; done“
```

- `-d` flag tells Docker to run the container and put it in the background, to daemonize it.
- The `docker ps` command queries the Docker daemon for information about all the containers it knows about.

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
1e5535038e28 ubuntu /bin/sh -c 'while tr 2 minutes ago Up 1 minute
insane_babbage
```

A daemonized Hello world

```
#docker run -d ubuntu /bin/sh -c "while true; do echo hello world; sleep 1; done“
```

- `-d` flag tells Docker to run the container and put it in the background, to daemonize it.
- The `docker ps` command queries the Docker daemon for information about all the containers it knows about.

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
1e5535038e28 ubuntu /bin/sh -c 'while tr 2 minutes ago Up 1 minute
insane_babbage
```

A daemonized Hello world

The `docker logs` command looks inside the container and returns its standard output: in this case the output of our command `hello world`.

- $ docker logs insane_babbage

```
hello world
hello world
hello world
. . .
```

A daemonized Hello world

- The `docker stop` command tells Docker to politely stop the running container. If it succeeds it will return the name of the container it has just stopped.

```
$ docker stop insane_babbage
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

A daemonized Hello world

- The `docker stop` command tells Docker to politely stop the running container. If it succeeds it will return the name of the container it has just stopped.

```
$ docker stop insane_babbage insane_babbage

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

Run a Web Server in a Container using Nginx

- Create a Website Directory to host

```
$mkdir -p ~/docker-nginx/html
$cd ~/docker-nginx/html
$echo “My Docker Website” > index.html
```

Run a Web Server in a Container using Nginx

- Start the web server container

```
sudo docker run --name docker-nginx -p 80:80 -d -v ~/docker-nginx/html:/usr/share/nginx/html nginx
```

- `-v` specifies that we're linking a volume
- -p denotes which port the host will listen and which port container will listen
- the part to the left of the `:` is the location of our file/directory on our virtual machine (`~/docker-nginx/html`)
- the part to the right of the `:` is the location that we are linking to in our container (`/usr/share/nginx/html`)

Run a Web Server in a Container using Nginx

- Verify by running a curl command

```
#curl http://localhost
```

Working with Images

- Docker images will list the images you have locally on our host.

```
$ docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
ubuntu 14.04 1d073211c498 3 days ago 187.9 MB
busybox latest 2c5ac3f849df 5 days ago 1.113 MB
training/webapp latest 54bb4e8718e8 5 months ago
```

- A repository potentially holds multiple variants of an image. In the case of our`ubuntu` image you can see multiple variants covering Ubuntu 10.04, 12.04, 12.10, 13.04, 13.10 and 14.04
- Tag is represt by name:tag (ubuntu:12:04)

```
$ docker run -t -i ubuntu:12.04 /bin/bash
```

Getting a new image

- Docker pull pulls the image from Repository
- By default it pulls from docker hub
- You can specify your own registry as well

```
$ docker pull centos
Pulling repository centos
b7de3133ff98: Pulling dependent layers
5cc9e91966f7: Pulling fs layer
511136ea3c5a: Download complete
ef52fb1fe610: Download complete . . .
Status: Downloaded newer image for centos
```

Running a local Registry

- AS the Proxy is not opened for docker hub we will use local registry
- Start a local Local Registry

```
$docker run -d -p 5000:5000 --restart=always --name registry registry
```

- Validate the local registry

```
$docker tag ubuntu localhost:5000/ubuntu
$docker push localhost:5000/ubuntu
$docker pull localhost:5000/ubuntu
```

Running a local Registry

- AS the Proxy is not opened for docker hub we will use local registry
- Start a local Local Registry

```
$docker run -d -p 5000:5000 --restart=always --name registry registry
```

- Validate the local registry

```
$docker tag ubuntu localhost:5000/ubuntu
$docker push localhost:5000/ubuntu
$docker pull localhost:5000/ubuntu
```

Searching images in registry

- Docker Search to search in the repository
- In Docker hub

```
$docker login
$docker search sinatra
NAME DESCRIPTION STARS OFFICIAL AUTOMATED
training/sinatra Sinatra training image 0 [OK]
marceldegraaf/sinatra Sinatra test app 0
```

- In local repository

```
$docker search localhost:5000/ubuntu
NAME            DESCRIPTION   STARS    OFFICIAL   AUTOMATED
library/ubuntu                0
```

Searching images in registry

- Docker Search to search in the repository
- In Docker hub

```
docker search sinatra
NAME DESCRIPTION STARS OFFICIAL AUTOMATED
training/sinatra Sinatra training image 0 [OK]
marceldegraaf/sinatra Sinatra test app 0
```

- In local repository

```
$docker search localhost:5000/ubuntu
NAME            DESCRIPTION   STARS    OFFICIAL   AUTOMATED
library/ubuntu                0
```

Pulling our image

- Docker pull to pull images
- In Docker hub

```
$docker pull training/sinatra
```

- In local repository

```
$docker pull localhost:5000/ubuntu
```

Creating our own images

```
#docker run -t -i centos /bin/bash
# vi /etc/yum.conf
# The proxy server - proxy server:port number proxy=http://proxy.cognizant.con:6080
# The account details for yum connections
proxy=http://proxy.cognizant.con:6080
proxy_username=<user name>
proxy_password=<password>
# yum install httpd
# vi /var/www/html/index.html
```

In another window

```
#docker ps
# docker commit -m "Web Server" -a Mohan a2b6064f23ad localhost:5000/centosweb:v2
# docker stop centos
#docker run –p 80:80 –d  localhost:5000/centosweb:v2 /bin/bash -c "exec /usr/sbin/apachectl start"
```

Building an image from a Dockerfile

- Dockerfile contains a set of instructions that tell Docker how to build our image.
- Docker build command is used to build the image from dokerfile
- The build is run by the Docker daemon.

```
$ docker build -f /path/to/a/Dockerfile .
```

Docker File Format

- *# Comment*
- INSTRUCTION arguments
- The instruction is not case-sensitive, however convention is for them to be UPPERCASE in order to distinguish them from arguments more easily.
- Docker runs the instructions in a dockerfile in order

Dockerfile Instructions

- FROM
- **FROM** <image>
- FROM <image>:<tag>
- **FROM** <image>@<digest>
- FROM instruction sets the *Base Image* for subsequent instructions.
- As such, a valid `Dockerfile` must have `FROM` as its first instruction.
- The image can be any valid image – it is especially easy to start by **pulling an image** from the *Public Repositories*.

Dockerfile Instructions

- FROM must be the first non-comment instruction in the Dockerfile.
- FROM can appear multiple times within a single Dockerfile in order to create multiple images.
- The tag or digest values are optional. If you omit either of them, the builder assumes a latest by default.

Dockerfile Instructions

- MAINTAINER
- **MAINTAINER** <name>
- The `MAINTAINER` instruction allows you to set the *Author* field of the generated images.

Dockerfile Instructions

- RUN
- *shell* form, the command is run in a shell - `/bin/sh -c`
    - RUN <command>
- Execution form (Any command)
    - RUN ["executable", "param1", "param2"]
- The `RUN` instruction will execute any commands in a new layer on top of the current image and commit the results.
- The resulting committed image will be used for the next step in the Dockerfile.
- Layering RUN instructions and generating commits conforms to the core concepts of Docker where commits are cheap and containers can be created from any point in an image’s history, much like source control.

Dockerfile Instructions

- CMD
- **The main purpose of a `CMD` is to provide defaults for an executing container.**
- These defaults can include an executable, or they can omit the executable, in which case you must specify an `ENTRYPOINT` instruction as well.
- The `CMD` instruction has three forms:
  - CMD ["executable","param1","param2"] (exec form)
  - CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
  - CMD command param1 param2 (shell form)
- There can only be one `CMD` instruction in a `Dockerfile`. If you list more than one `CMD` then only the last `CMD` will take effect.

Dockerfile Instructions

- CMD
- **FROM** ubuntu
- **CMD** ["/usr/bin/wc","--help"]

- **FROM** ubuntu
- **CMD** echo "This is a test." | wc -

Dockerfile Instructions

- LABEL
- The LABEL instruction adds metadata to an image. A LABEL is a key-value pair. To include spaces within a LABEL value, use quotes and backslashes as you would in command-line parsing.
- **LABEL** "com.example.vendor"="ACME Incorporated"
- **LABEL** com.example.label-with-value="foo"
- **LABEL** version="1.0"
- **LABEL** description="This text illustrates \
- that label-values can span multiple lines.“
- **LABEL** multi.label1="value1" multi.label2="value2" other="value3“
- Use Docker inspect command to see the Variable

Dockerfile Instructions

- EXPOSE
- **EXPOSE** <port> [<port>...]
- The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime.
- `EXPOSE` does not make the ports of the container accessible to the host.
- To do that, you must use either the `-p` flag to publish a range of ports or the `-P` flag to publish all of the exposed ports.
- You can expose one port number and publish it externally under another number.

Dockerfile Instructions

- ENV
- **ENV** <key> <value>
- **ENV** <key>=<value> ...
- The `ENV` instruction sets the environment variable `<key>` to the value`<value>`
- `The ENV instruction has two forms`
  - ENV <key> <value>
    - will set a single variable to a value. The entire string after the first space will be treated as the `<value>` - including characters such as spaces and quotes.
  - ENV <key>=<value> ...
    - allows for multiple variables to be set at one time. Like command line parsing, quotes and backslashes can be used to include spaces within values.

Dockerfile Instructions

- ENV
- **ENV** myName="John Doe" myDog=Rex\ The\ Dog \ myCat=fluffy
- **ENV** myName John Doe
- **ENV** myDog Rex The Dog
- **ENV** myCat fluffy

Dockerfile Instructions

- ADD
- ADD has two forms:
    - `ADD <src>... <dest>`
    - ADD ["<src>",”<src>”... "<dest>"]

•The `ADD` instruction copies new files, directories or remote file URLs from`<src>` and adds them to the filesystem of the container at the path `<dest>`.

• Multiple <src> resource may be specified

ADD hom* /mydir/

ADD hom?.txt /mydir/

ADD test relativeDir/

ADD test /absoluteDir/

Dockerfile Instructions

- `ADD`
- In the case where `<src>` is a remote file URL, the destination will have permissions of 600.
- The `<src>` path must be inside the *context* of the build; you cannot`ADD ../something /something`, because the first step of a`docker build` is to send the context directory (and subdirectories) to the docker daemon.
- `If <src> is a URL and <dest> does not end with a trailing slash, then a file is downloaded from the URL and copied to <dest>.`
- If `<src>` is a directory, the entire contents of the directory are copied, including filesystem metadata.
- If `<src>` is a *local* tar archive in a recognized compression format (identity, gzip, bzip2 or xz) then it is unpacked as a directory.
- If `<src>` is any other kind of file, it is copied individually along with its metadata.
- If multiple `<src>` resources are specified, either directly or due to the use of a wildcard, then `<dest>` must be a directory, and it must end with a slash `/`
- If `<dest>` does not end with a trailing slash, it will be considered a regular file and the contents of `<src>` will be written at `<dest>`.
- If `<dest>` doesn’t exist, it is created along with all missing directories in its path.

.

COPY

- COPY has two forms:
  - `COPY <src>... <dest>`
  - COPY ["<src>",... "<dest>"]
- The `COPY` instruction copies new files or directories from `<src>` and adds them to the filesystem of the container at the path `<dest>`.
- **COPY** hom* /mydir/
- **COPY** hom?.txt /mydir/
- **COPY** test relativeDir/
- **COPY** test /absoluteDir/

ENTRYPOINT

- ENTRYPOINT
  - An `ENTRYPOINT` allows you to configure a container that will run as an executable.
- ENTRYPOINT has two forms:
  - ENTRYPOINT ["executable", "param1", "param2"]
  - ENTRYPOINT command param1 param2
- docker **run** -i -t --rm -p 80:80 nginx```

ENTRYPOINT

- ENTRYPOINT
  - An `ENTRYPOINT` allows you to configure a container that will run as an executable.
- ENTRYPOINT has two forms:
  - ENTRYPOINT ["executable", "param1", "param2"]
  - ENTRYPOINT command param1 param2
- docker **run** -i -t --rm -p 80:80 nginx```

VOLUME

- The `VOLUME` instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.
- The value can be a JSON array, VOLUME ["/var/log/"], or a plain string with multiple arguments, such asVOLUME /var/log or VOLUME /var/log /var/db

FROM ubuntu

RUN mkdir /myvol

RUN echo "hello world" > /myvol/greeting

VOLUME /myvol

USER

- The `USER` instruction sets the user name or UID to use when running the image and for any `RUN`, `CMD` and`ENTRYPOINT` instructions that follow it in the `Dockerfile`

USER daemon

WORKDIR

- The `WORKDIR` instruction sets the working directory for any `RUN`, `CMD`, `ENTRYPOINT`, `COPY` and `ADD`instructions that follow it in the `Dockerfile`.
- **WORKDIR** /path/to/workdir

WORKDIR /a

WORKDIR b

WORKDIR c

RUN pwd

Building an image from a Dockerfile

- $ mkdir build
- $ cd build
- $ touch Dockerfile
- $vi Dockerfile

```
# This is a comment
FROM ubuntu:14.04
MAINTAINER Kate Smith <ksmith@example.com>
RUN touch /etc/test
```

```
$docker build -t localhost:5000/Ubuntu:v4 .
```

Dockerfile Best practices

Containers should be ephemeral

- The container produced by the image your `Dockerfile` defines should be as ephemeral as possible. By “ephemeral,” we mean that it can be stopped and destroyed and a new one built and put in place with an absolute minimum of set-up and configuration.

Use a .dockerignore file

- In most cases, it’s best to put each Dockerfile in an empty directory. Then, add to that directory only the files needed for building the Dockerfile. To increase the build’s performance, you can exclude files and directories by adding a .dockerignore file to that directory as well

Dockerfile Best practices

- Avoid installing unnecessary packages
  - In order to reduce complexity, dependencies, file sizes, and build times, you should avoid installing extra or unnecessary packages just because they might be “nice to have.” For example, you don’t need to include a text editor in a database image.
- Run only one process per container
  - In almost all cases, you should only run a single process in a single container. Decoupling applications into multiple containers makes it much easier to scale horizontally and reuse containers.

Dockerfile Best practices

- Minimize the number of layers
  - You need to find the balance between readability (and thus long-term maintainability) of the Dockerfile and minimizing the number of layers it uses. Be strategic and cautious about the number of layers you use.
- Sort multi-line arguments
  - Whenever possible, ease later changes by sorting multi-line arguments alphanumerically. This will help you avoid duplication of packages and make the list much easier to update.

```
RUN apt-get update && apt-get install -y \
bzr \
cvs \
git \
mercurial \
subversion
```

Setting tags on an image

- You can also add a tag to an existing image after you commit or build it.
- We can do this using the `docker tag` command.
- Now, add a new tag to your ouruser/sinatraimage.

```
$ docker tag 5db5f8471261 localhost:5000/myubuntu
```

Name a container

- You’ve already seen that each container you create has an automatically created name.
- You can also name containers yourself. This naming provides two useful functions:
  - You can name containers that do specific functions in a way that makes it easier for you to remember them
  - Names provide Docker with a reference point that allows it to refer to other containers. There are several commands that support this and you’ll use one in an exercise later.

Using names in the command

- $docker **run** -d -P --name web nginix
- $docker inspect web
- $docker stop web
- $docker rm web

Networking Container

- Launch a container on the default network
  - Docker includes support for networking containers through the use of **network drivers**.
  - By default, Docker provides two network drivers for you, the `bridge` and the `overlay` drivers.
  - You can also write a network driver plugin so that you can create your own drivers but that is an advanced task.

Networking Container

- Every installation of the Docker Engine automatically includes three default networks.

```
$ docker network ls
NETWORK ID NAME DRIVER
18a2866682b8 none null
c288470c46f6 host host
7b369448dccb bridge bridge
```

- The bridge network represents the docker0 network present in all Docker installations. Unless you specify otherwise with thedocker run --net=<NETWORK> option
- the Docker daemon connects containers to this network by default. You can see this bridge as part of a host's network stack by using the ifconfig command on the host

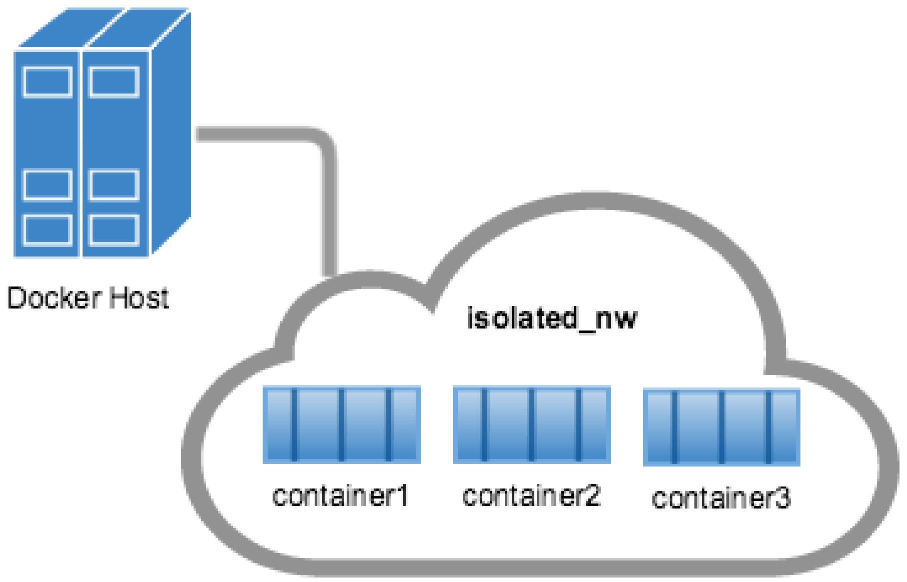
Networking Container

- If you don’t specify a network interface for a container then Container gets added to The `none` network.
- The `host` network adds a container on the hosts network stack. You’ll find the network configuration inside the container is identical to the host.
- You cannot remove these default network
- You can view the details using “docker network inspect”

Creating your own bridge network

- The easiest user-defined network to create is a `bridge` network. This network is similar to the historical, default `docker0` network.
- $ docker network **create** *--driver bridge isolated_nw*
- *$docker network inspect isolated_nw*
- $ docker run --net=isolated_nw -itd --name=container3 Ubuntu
- $ docker network inspect isolated_nw

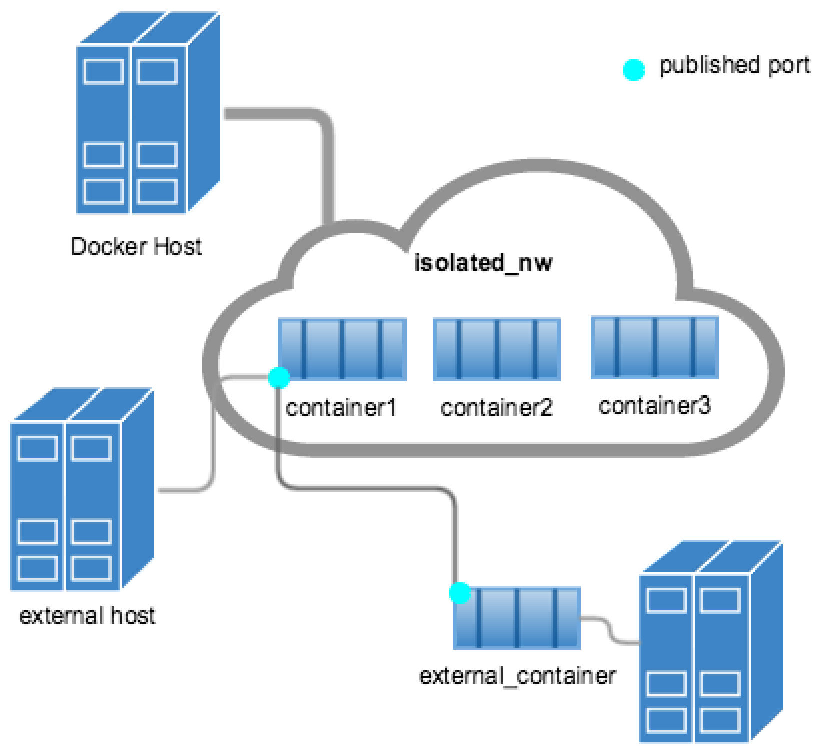
Creating your own bridge network



- The containers you launch into user defined network must reside on the same Docker host.

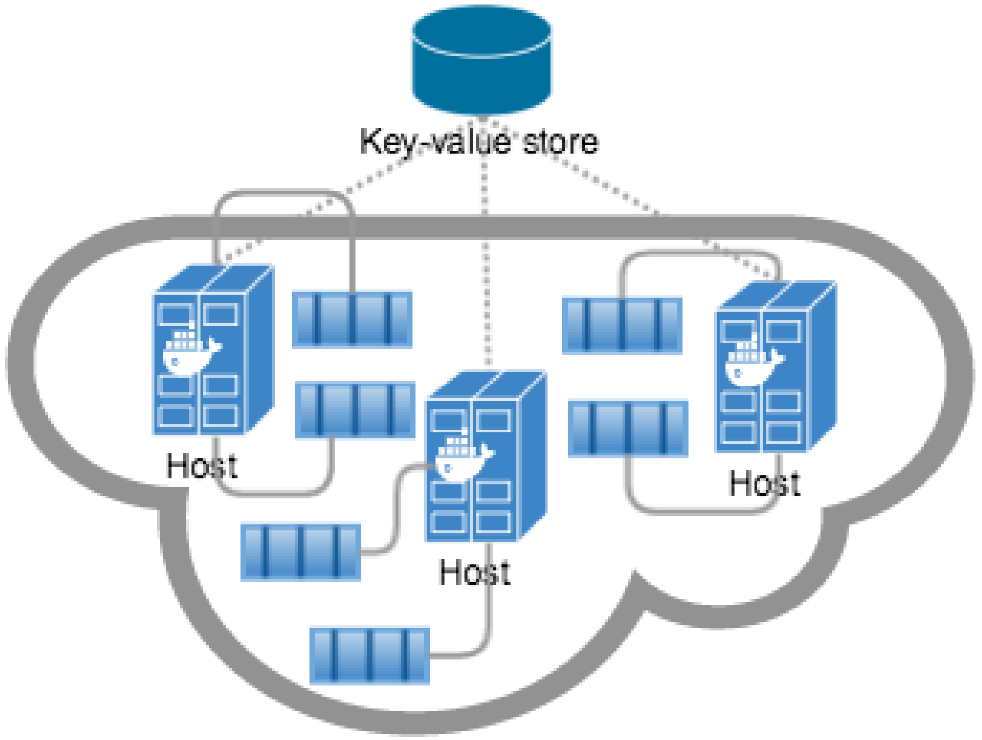
  Each container in the network can immediately communicate with other containers in the network. Though, the network itself isolates the containers from external networks.

Creating your own bridge network



- The containers you launch into this network must reside on the same Docker host. Each container in the network can immediately communicate with other containers in the network. Though, the network itself isolates the containers from external networks.
- A bridge network is useful in cases where you want to run a relatively small network on a single host.
- You can, however, create significantly larger networks by creating an `overlay` network.

An overlay network



- Docker’s overlay network driver supports multi-host networking natively out-of-the-box.
- This support is accomplished with the help of libnetwork, a built-in VXLAN-based overlay network driver, and Docker’s libkv library.
- The overlay network requires a valid key-value store service
- Currently, Docker’s libkv supports Consul, Etcd, and ZooKeeper (Distributed store).
- We will create a Overlay network using swarm later

An overlay network

You should open the following ports between each of your hosts.

Protocol	Port	Description
udp	4789	Data plane (VXLAN)
tcp/udp	7946	Control plane

Docker embedded DNS server

- Docker daemon runs an embedded DNS server to provide automatic service discovery for containers connected to user defined networks.
- Name resolution requests from the containers are handled first by the embedded DNS server.
- If the embedded DNS server is unable to resolve the request it will be forwarded to any external DNS servers configured for the container.
- To facilitate this when the container is created, only the embedded DNS server reachable at`127.0.0.11` will be listed in the container's `resolv.conf` file.

Docker embedded DNS server

- the docker daemon implements an embedded DNS server which provides built-in service discovery for any container created with a valid`name` or `net-alias` or aliased by `link`.
- --name=CONTAINER-NAME
  - Container name configured using`--name` is used to discover a container within an user-defined docker network. The embedded DNS server maintains the mapping between the container name and its IP address
- --net-alias=ALIAS
  - a container is discovered by one or more of its configured`--net-alias` (or `--alias` in`docker network connect`command) within the user-defined network.
  - The embedded DNS server maintains the mapping between all of the container aliases and its IP address on a specific user-defined network.
  - A container can have different aliases in different networks by using the `--alias` option in`docker network connect`command.

Docker embedded DNS server

- --link=CONTAINER_NAME:ALIAS
  - Using this option as you `run` a container gives the embedded DNS an extra entry named `ALIAS` that points to the IP address of the container identified by `CONTAINER_NAME`.
  - When using `--link` the embedded DNS will guarantee that localized lookup result only on that container where the`--link` is used.
  - This lets processes inside the new container connect to container without having to know its name or IP.
- --dns=[IP_ADDRESS...]
  - The IP addresses passed via the`--dns` option is used by the embedded DNS server to forward the DNS query if embedded DNS server is unable to resolve a name resolution request from the containers.
  - These`--dns` IP addresses are managed by the embedded DNS server and will not be updated in the container's`/etc/resolv.conf` file.

Docker embedded DNS server

- --dns-search=DOMAIN...
    - Sets the domain names that are searched when a bare unqualified hostname is used inside of the container.
    - These `--dns-search`options are managed by the embedded DNS server and will not be updated in the container's`/etc/resolv.conf` file.
    - When a container process attempts to access`host` and the search domain`example.com` is set, for instance, the DNS logic will not only look up `host`but also `host.example.com`.
- --dns-opt=OPTION...
    - Sets the options used by DNS resolvers. These options are managed by the embedded DNS server and will not be updated in the container's`/etc/resolv.conf` file.

Manage data in containers

- Two primary ways you can manage data in Docker.
  - Data volumes
  - Data volume containers.
- Data volumes are designed to persist data, independent of the container’s life cycle.
- Docker therefore *never* automatically deletes volumes when you remove a container, nor will it “garbage collect” volumes that are no longer referenced by a container.

Data volumes

- A *data volume* is a specially-designated directory within one or more containers that bypasses the *Union File System.*
- Data volumes provide several useful features for persistent or shared data:
  - Volumes are initialized when a container is created. If the container’s base image contains data at the specified mount point, that existing data is copied into the new volume upon volume initialization.
  - Data volumes can be shared and reused among containers.
  - Changes to a data volume are made directly.
  - Changes to a data volume will not be included when you update an image.
  - Data volumes persist even if the container itself is deleted.

Adding a data volume

- You can add a data volume to a container using the `-v` flag with the `docker create` and `docker run` command.
- You can use the -v multiple times to mount multiple data volumes.
- Let’s mount a single volume now in our web application container.
- $ docker run --name testc2 -t -i -v /webapp centos /bin/bash
- docker inspect testc2

Mount a host directory as a data volume

- $mkdir /test
- cd /test
- touch test1 test2 test3
- $ docker run --name testc2 -t -i -v /test:/webapp centos /bin/bash
- docker inspect testc2

Mount a host file as a data volume

- docker run --rm -it -v ~/.bash_history:/root/.bash_history ubuntu /bin/bash

Creating and mounting a data volume container

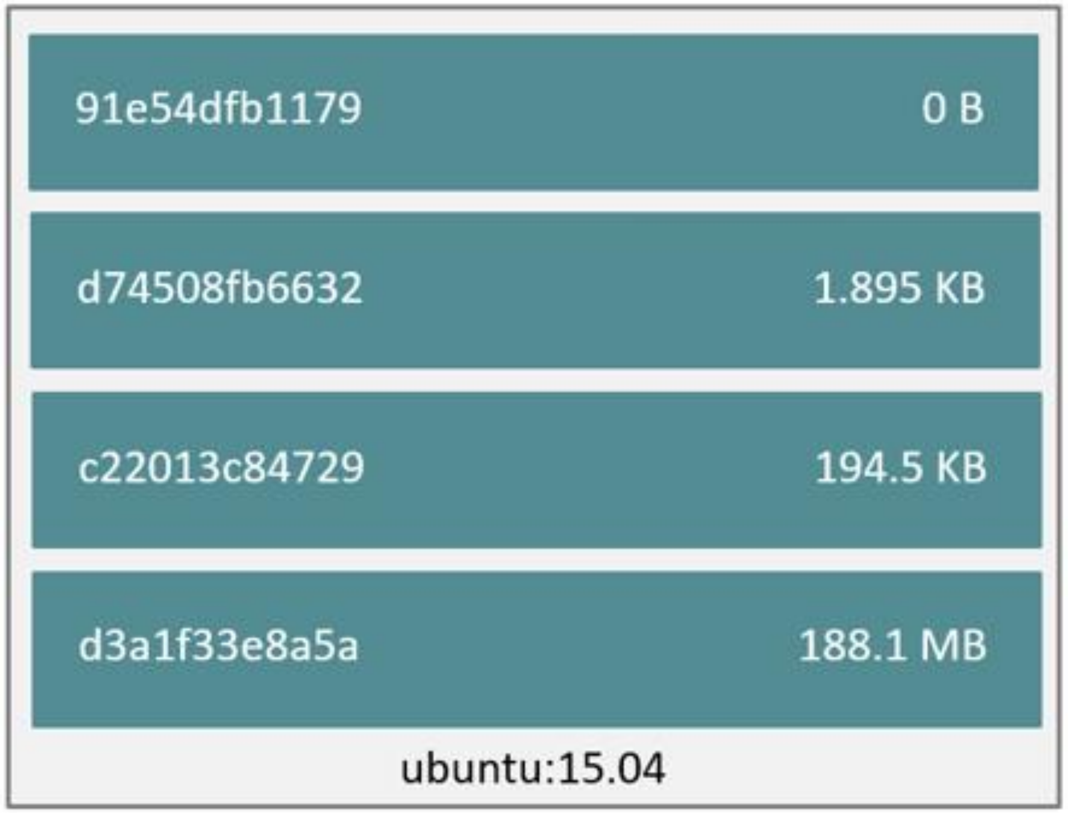
- If you have some persistent data that you want to share between containers, or want to use from non-persistent containers, it’s best to create a named Data Volume Container, and then to mount the data from it.
- $ docker **create** -v /dbdata *--name dbstore mysql /bin/true*
- *$ docker run --name mysql01 --volumes-from dbstore -e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql*
- $docker exec -it mysql01 bash
- $df –h
- docker run --name mysql02 --volumes-from dbstore -e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql

Backup, restore, or migrate data volumes

- Another useful function we can perform with volumes is use them for backups, restores or migrations. We do this by using the `--volumes-from` flag to create a new container that mounts that volume
- Backup
  - $ docker run --rm --volumes-from dbstore -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata
- Restore
  - $ docker **run** -v /dbdata --name dbstore2 ubuntu /bin/bash
  - $ docker run --rm --volumes-from dbstore2 -v $(pwd):/backup ubuntu bash -c "cd /dbdata && tar xvf /backup/backup.tar --strip 1"

Understand images, containers, and storage drivers

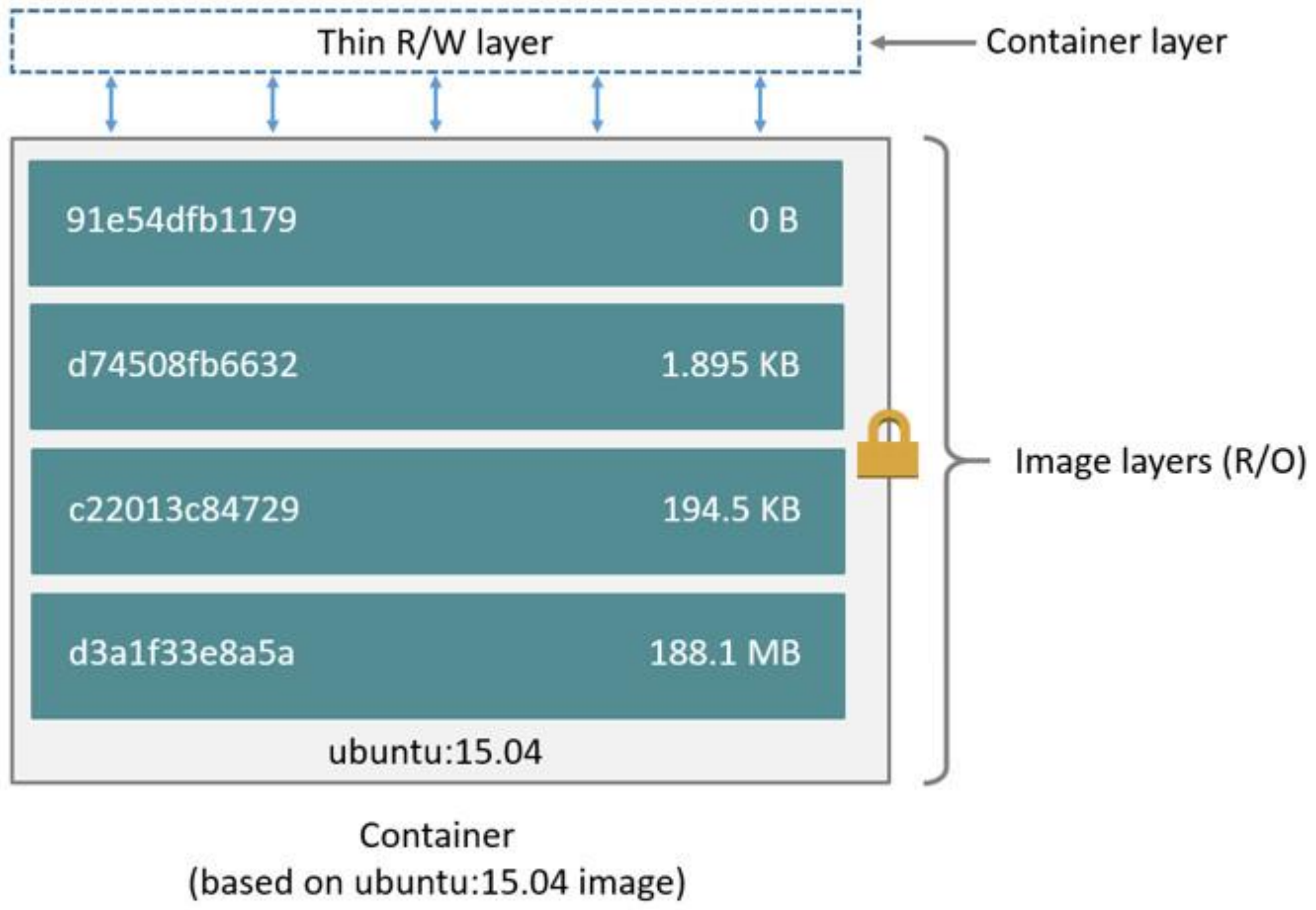
Images and layers



Image

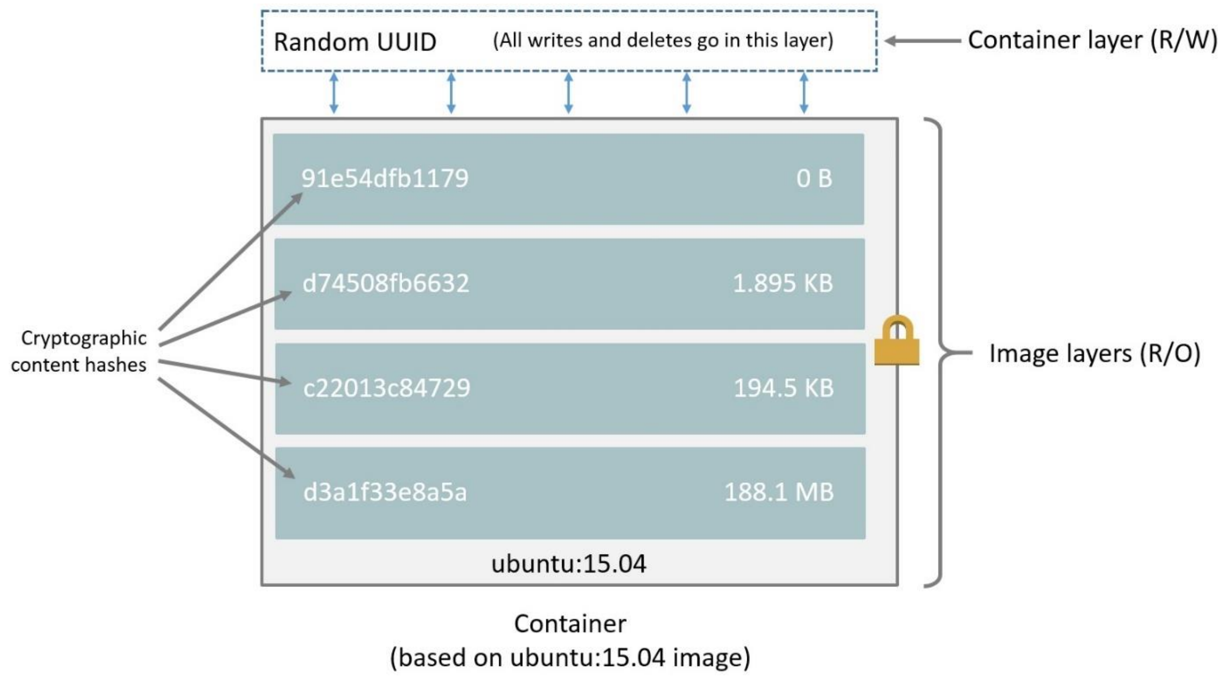
- Each Docker image references a list of read-only layers that represent filesystem differences. Layers are stacked on top of each other to form a base for a container's root filesystem.
- The Docker storage driver is responsible for stacking these layers and providing a single unified view.

Images and layers



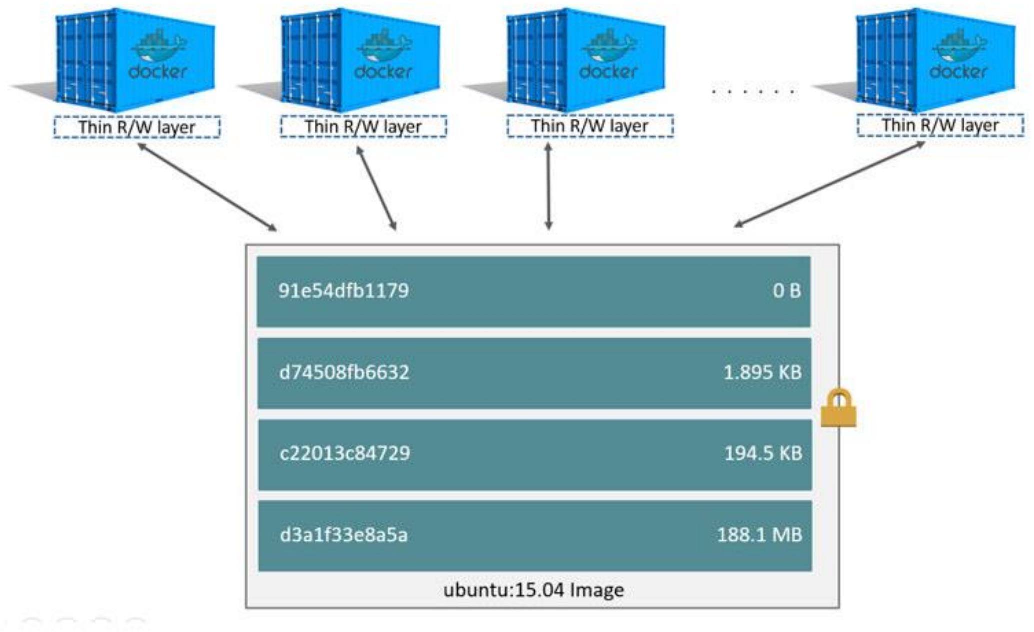
- When you create a new container, you add a new, thin, writable layer on top of the underlying stack. This layer is often called the “container layer”.
- All changes made to the running container - such as writing new files, modifying existing files, and deleting files - are written to this thin writable container layer

Content addressable storage



- model improves security, provides a built-in way to avoid ID collisions, and guarantees data integrity after pull, push, load, and save operations.
- It also enables better sharing of layers by allowing many images to freely share their layers even if they didn’t come from the same build.

Container and layers

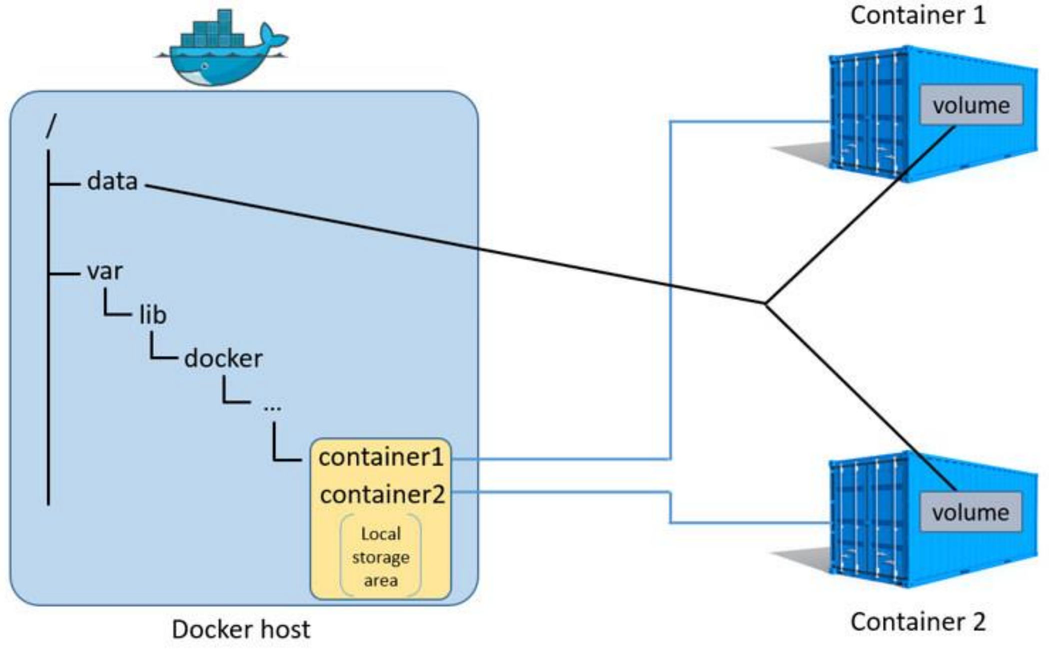


- The major difference between a container and an image is the top writable layer.
- All writes to the container that add new or modify existing data are stored in this writable layer.
- When the container is deleted the writable layer is also deleted.
- The underlying image remains unchanged.

copy-on-write strategy

- Copy-on-write is a similar strategy of sharing and copying. In this strategy, system processes that need the same data share the same instance of that data rather than having their own copy.
- At some point, if one process needs to modify or write to the data, only then does the operating system make a copy of the data for that process to use.
- Only the process that needs to write has access to the data copy. All the other processes continue to use the original data.
- Docker uses a copy-on-write technology with both images and containers.
- This CoW strategy optimizes both image disk space usage and the performance of container start times.

Data volumes and the storage driver



- When a container is deleted, any data written to the container that is not stored in a *data volume* is deleted along with the container.
- A data volume is a directory or file in the Docker host's filesystem that is mounted directly into a container.
- Data volumes are not controlled by the storage driver. Reads and writes to data volumes bypass the storage driver and operate at native host speeds.
- You can mount any number of data volumes into a container. Multiple containers can also share one or more data volumes.

storage driver

- Docker has a pluggable storage driver architecture.
- This gives you the flexibility to “plug in” the storage driver that is best for your environment and use-case.
- Each Docker storage driver is based on a Linux filesystem or volume manager.
- Further, each storage driver is free to implement the management of image layers and the container layer in its own unique way.
- This means some storage drivers perform better than others in different circumstances.

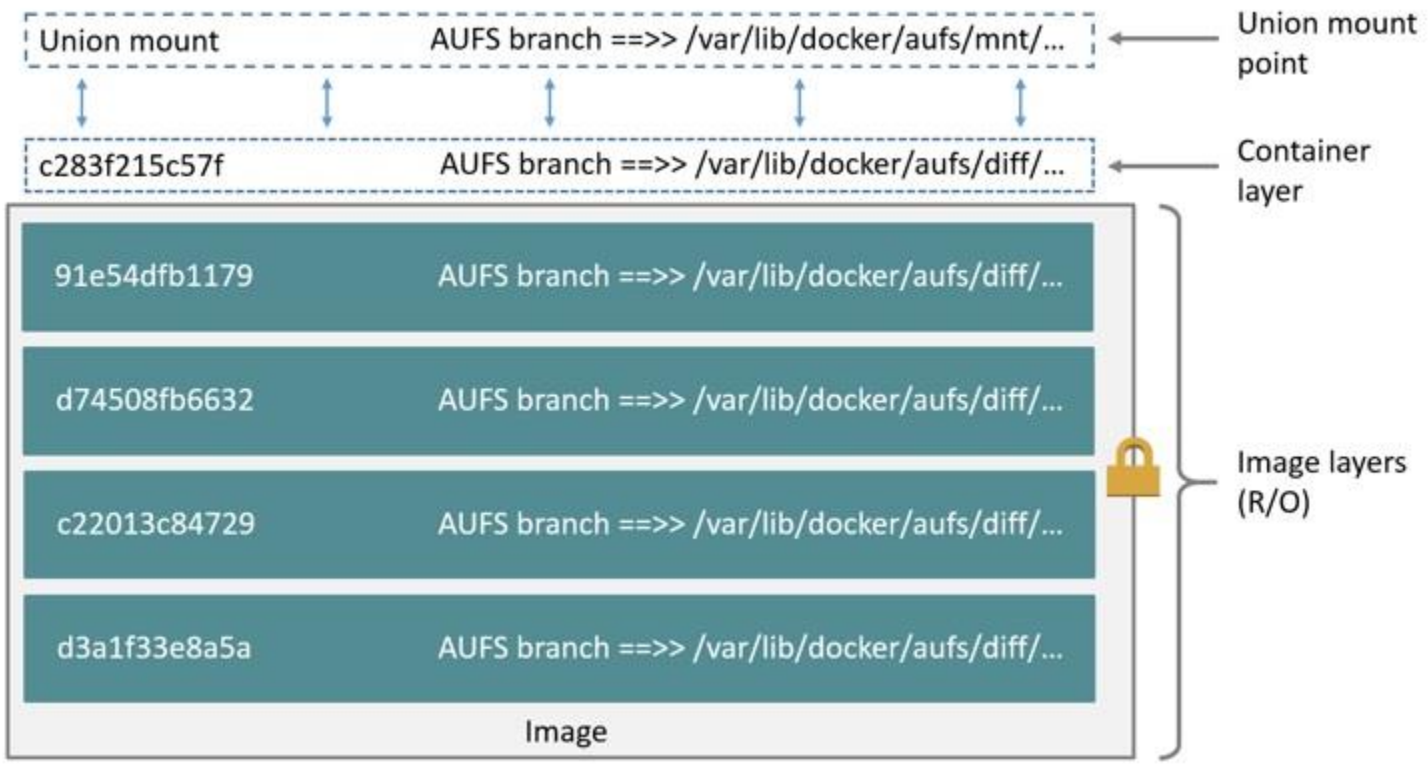
Types of Storage Driver

Technology	Storage driver name
OverlayFS	overlay
AUFS	aufs
Btrfs	btrfs
Device Mapper	devicemapper
VFS*	vfs
ZFS	zfs

AUFS

- AUFS was the first storage driver in use with Docker.
- These features enable:
  - Fast container startup times.
  - Efficient use of storage.
  - Efficient use of memory.
- some Linux distributions do not support AUFS. This is usually because AUFS is not included in the mainline (upstream) Linux kernel.

AUFS

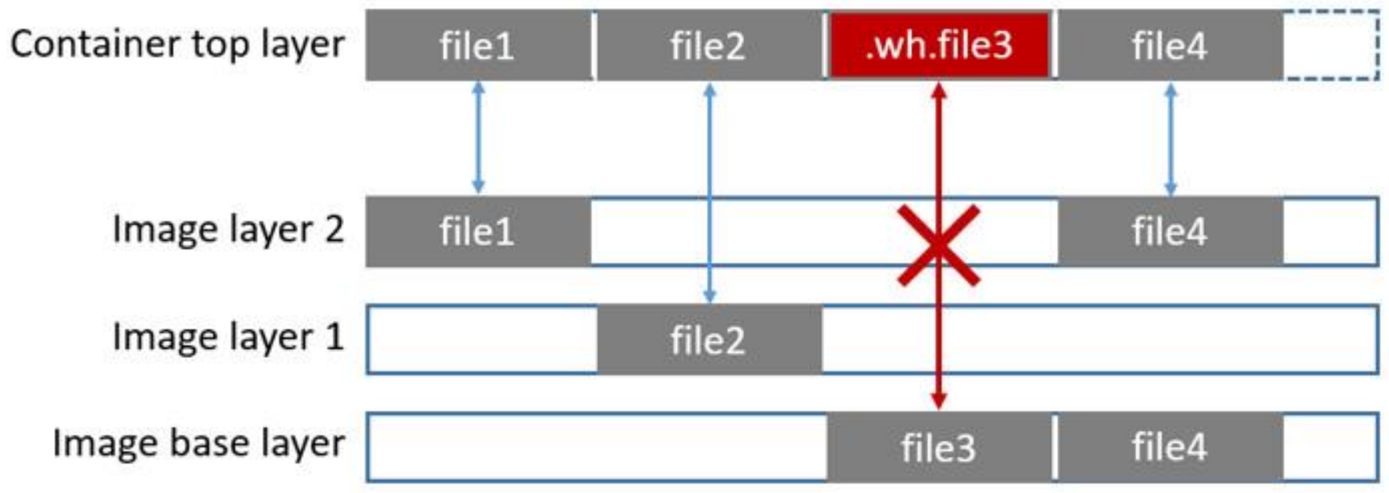

Union mount
AUFS branch ==>> /var/lib/docker/aufs/mnt/...
Union mount point
c283f215c57f
AUFS branch ==>> /var/lib/docker/aufs/diff/...
Container layer
91e54dfb1179
AUFS branch ==>> /var/lib/docker/aufs/diff/...
d74508fb6632
AUFS branch ==>> /var/lib/docker/aufs/diff/...
c22013c84729
AUFS branch ==>> /var/lib/docker/aufs/diff/...
d3a1f33e8a5a
AUFS branch ==>> /var/lib/docker/aufs/diff/...
Image
Image layers (R/O)

- The AUFS storage driver implements Docker image layers using this union mount system.
- AUFS branches correspond to Docker image layers.

Container reads and writes with AUFS

- Docker leverages AUFS CoW technology to enable image sharing and minimize the use of disk space
- AUFS works at the file level. This means that all AUFS CoW operations copy entire files - even if only a small part of the file is being modified.
- This behavior can have a noticeable impact on container performance, especially if the files being copied are large, below a lot of image layers, or the CoW operation must search a deep directory tree.

Deleting files with the AUFS storage driver



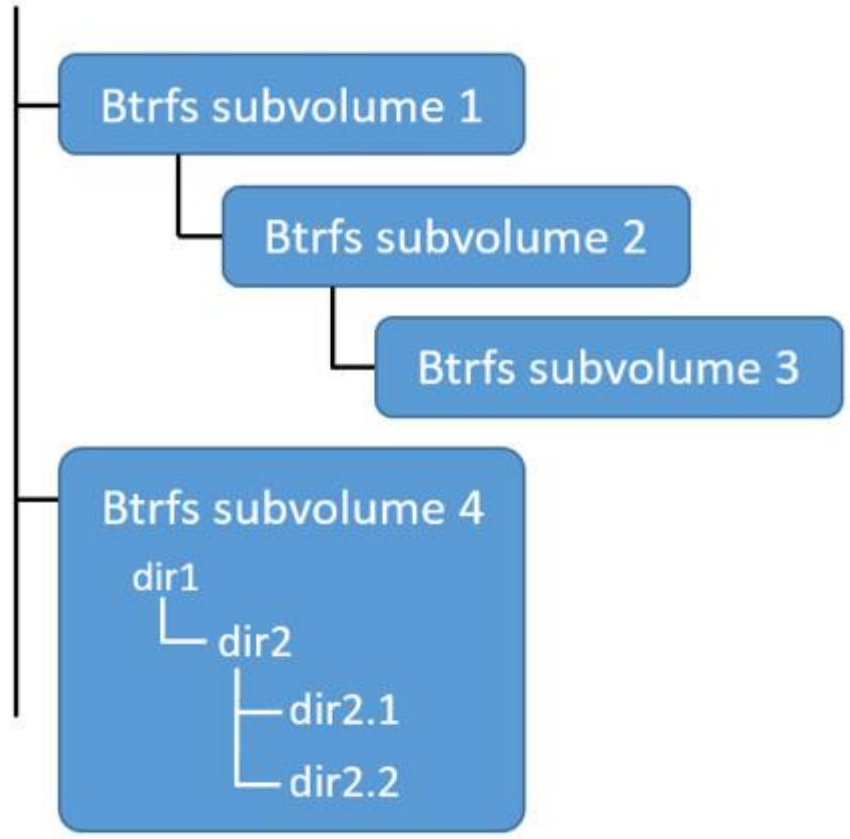
Docker container
(AUFS storage-driver demonstrating whiteout file)

- The AUFS storage driver deletes a file from a container by placing a *whiteout file* in the container's top layer.
- The whiteout file effectively obscures the existence of the file in the read-only image layers below.

BRTS Filesystem

- Btrfs is a next generation copy-on-write filesystem that supports many advanced storage technologies that make it a good fit for Docker.
- Btrfs is included in the mainline Linux kernel and its on-disk-format is now considered stable.
- You should only consider the `btrfs` driver for production deployments if you understand it well and have existing experience with Btrfs.

BRTS Filesystem

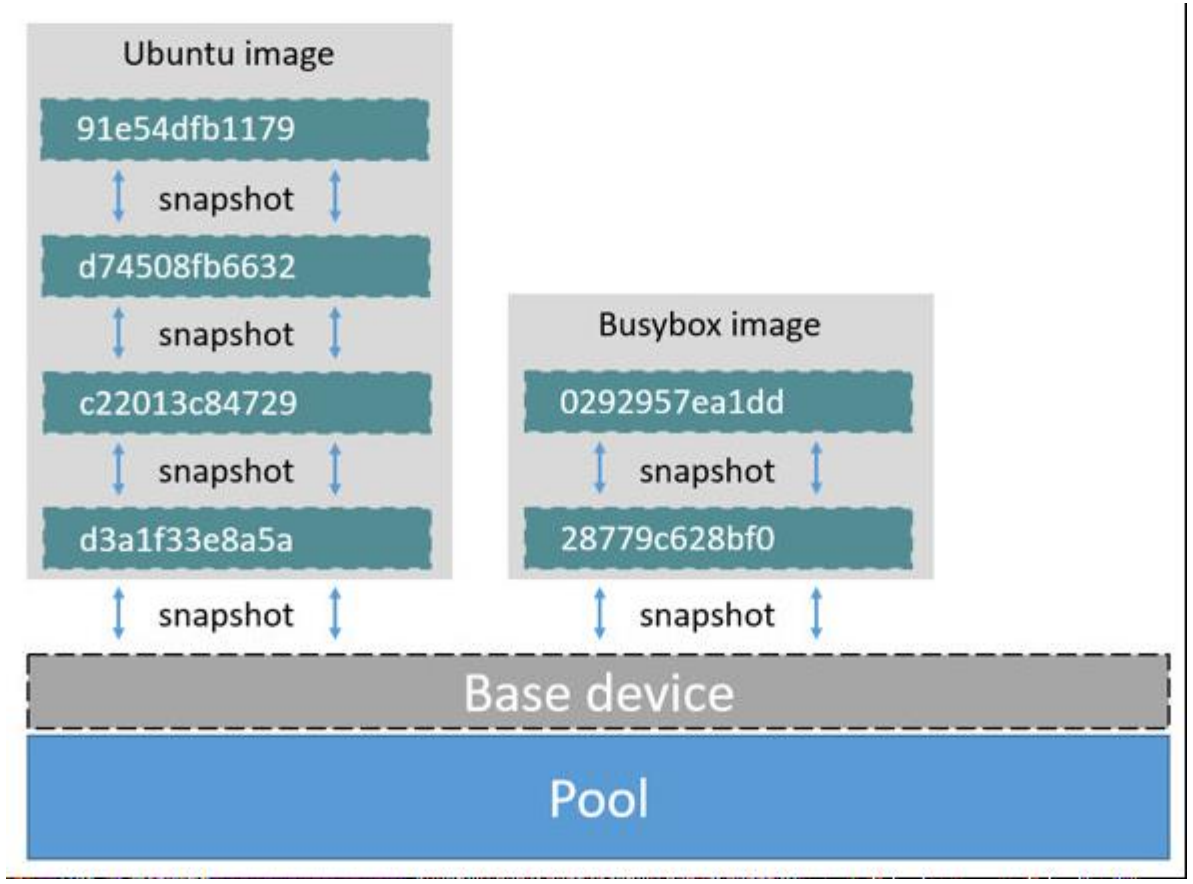


- Docker leverages Btrfs *subvolumes* and *snapshots* for managing the on-disk components of image and container layers.
- Btrfs subvolumes look and feel like a normal Unix filesystem. As such, they can have their own internal directory structure that hooks into the wider Unix filesystem.
- Subvolumes are natively copy-on-write and have space allocated to them on-demand from an underlying storage pool.
- They can also be nested and snapped. The diagram blow shows 4 subvolumes. 'Subvolume 2' and 'Subvolume 3' are nested, whereas 'Subvolume 4' shows its own internal directory tree.

Device Mapper Driver

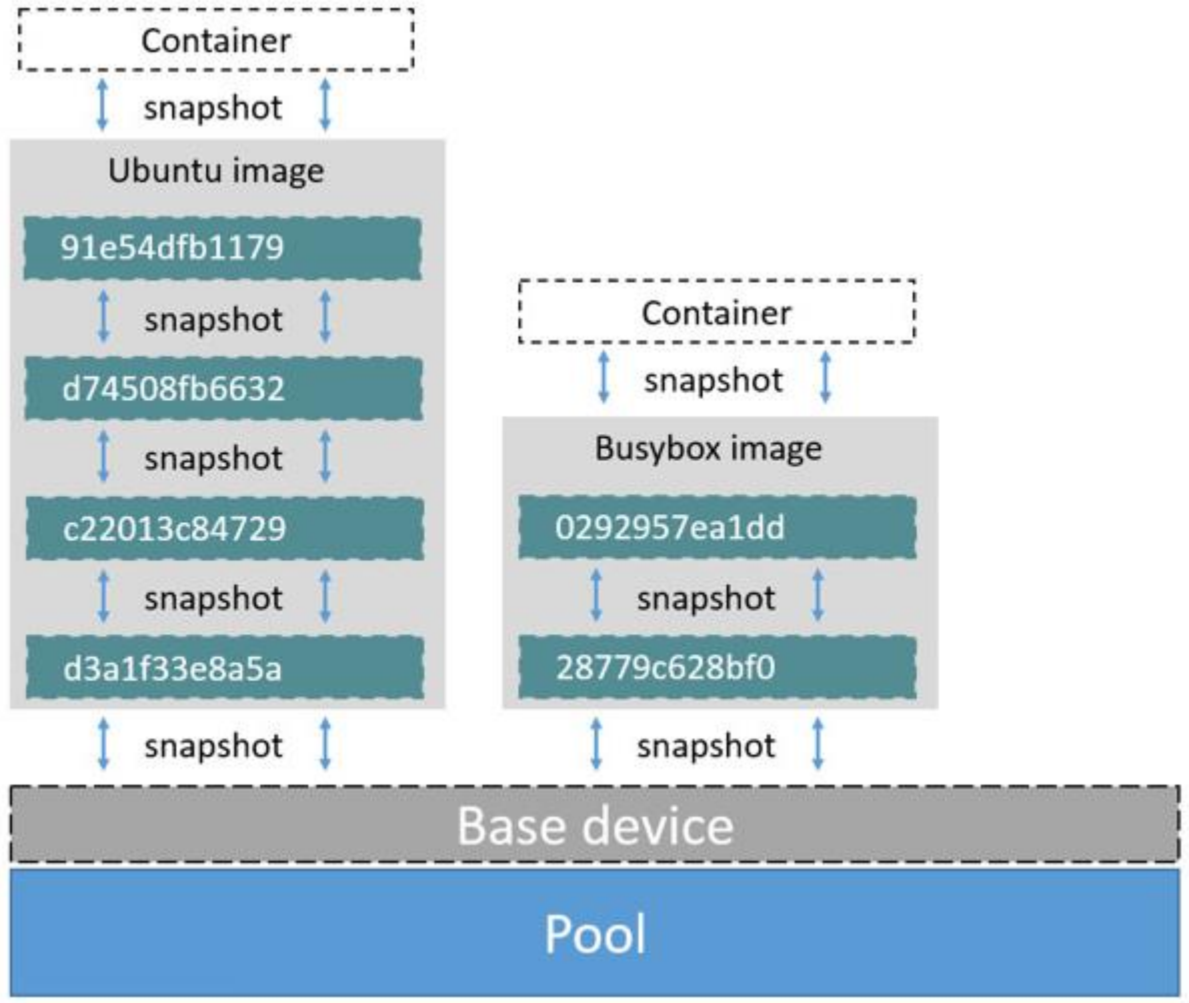
- The `devicemapper` driver stores every image and container on its own virtual device.
- These devices are thin-provisioned copy-on-write snapshot devices.
- Device Mapper technology works at the block level rather than the file level.
- This means that `devicemapper` storage driver’s thin provisioning and copy-on-write operations work with blocks rather than entire files.

Device Mapper Driver



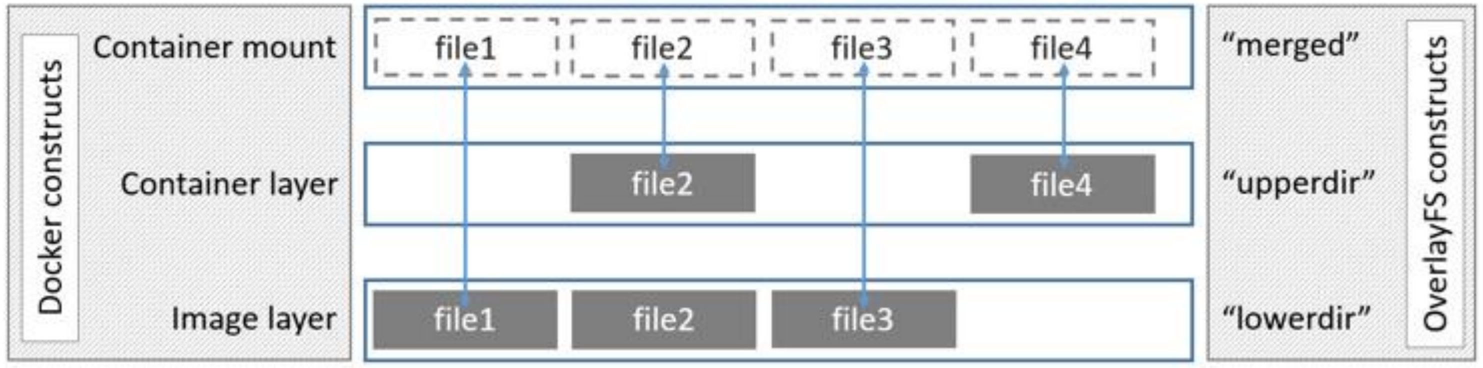
- The `devicemapper` storage driver creates a thin pool
- The pool is created from block devices or loop mounted sparse files.
- it creates a *base device*.
- A base device is a thin device with a filesystem. You can see which filesystem is in use by running the `docker info` command and checking the `Backing filesystem` value
- Each new image (and image layer) is a snapshot of this base device.
- These are thin provisioned copy-on-write snapshots. This means that they are initially empty and only consume space from the pool when data is written to them.

Device Mapper Driver



OverlayFS

- OverlayFS takes two directories on a single Linux host, layers one on top of the other, and provides a single unified view.
- These directories are often referred to as *layers* and the technology used to layer them is known as a *union mount*.
- The OverlayFS terminology is “lowerdir” for the bottom layer and “upperdir” for the top layer. The unified view is exposed through its own directory called “merged”.


Docker constructs
Container mount
Container layer
Image layer
file1
file2
file3
file4
file2
file4
file1
file2
file3
“merged”
“upperdir”
“lowerdir”
OverlayFS constructs