Docker Training

Pragra

ONWARD & UPWARD

Welcome

[www.docker.com]

Agenda

- Understanding Docker
- Docker Containers
- Docker Images
- Docker Networking
- Docker volume
- Docker Compose

Pragra
ONWARD & UPWARD

What is Docker

Docker is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating system–level virtualization on Linux

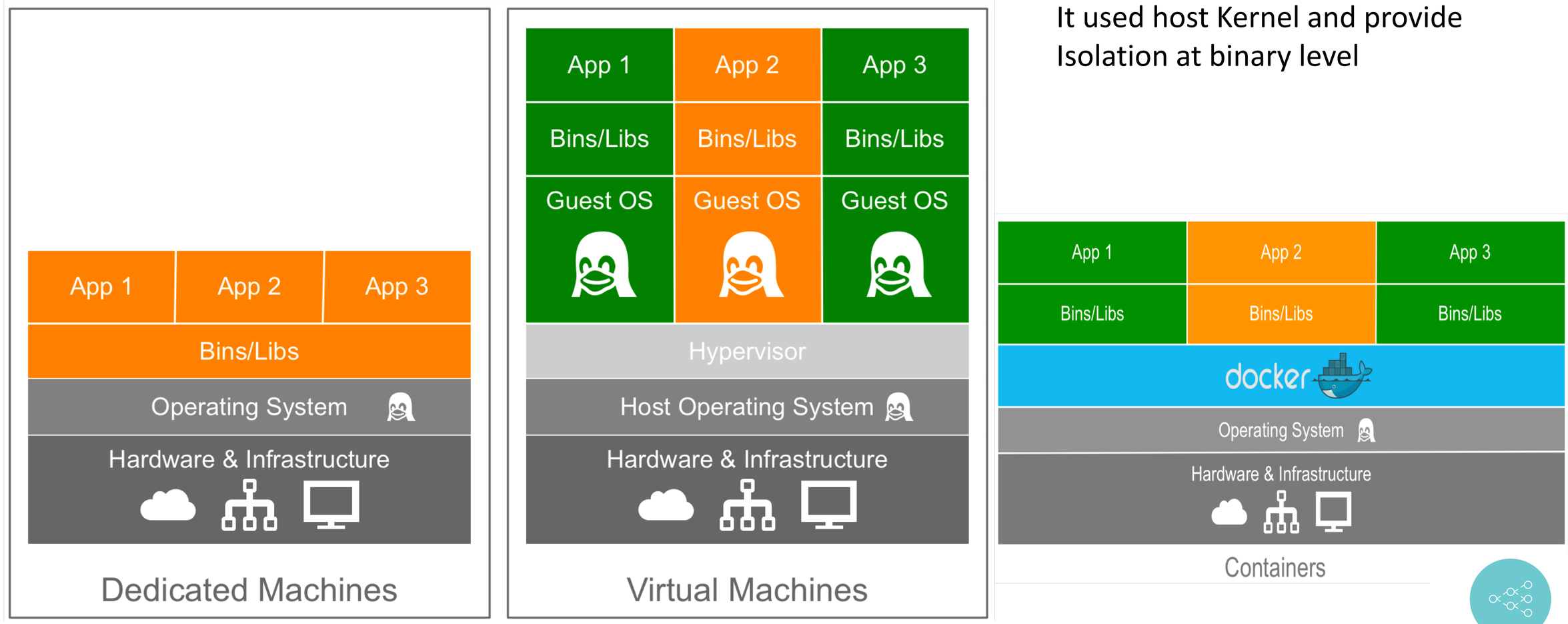
Provide a uniformed wrapper around a software package: *«Build, Ship and Run Any App, Anywhere»* [www.docker.com]

Similar to shipping containers: The container is always the same, regardless of the contents and thus fits on all trucks, cranes, ships, ...

Difference Between Host/VM and Docker

- Docker is a container management system that helps us easily manage **Linux Containers (LXC)** in an easier and universal fashion.

Unlike VM docker doesn't install guest Operating system with own kernel. I It used host Kernel and provide Isolation at binary level



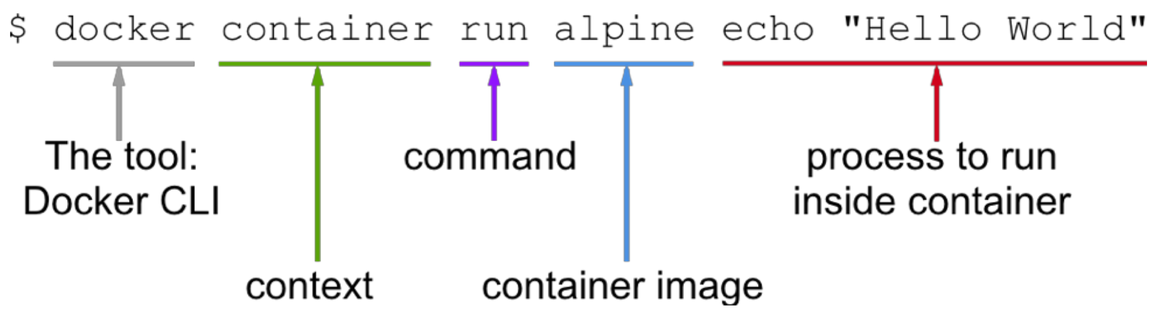
Pragra
ONWARD & UPWARD

Docker CLI

- Docker comes with sophisticated CLI - docker cli provide various management command.

Check version of your docker installation

```
➜  ~ docker –v
Docker version 18.09.2, build 6247962
```



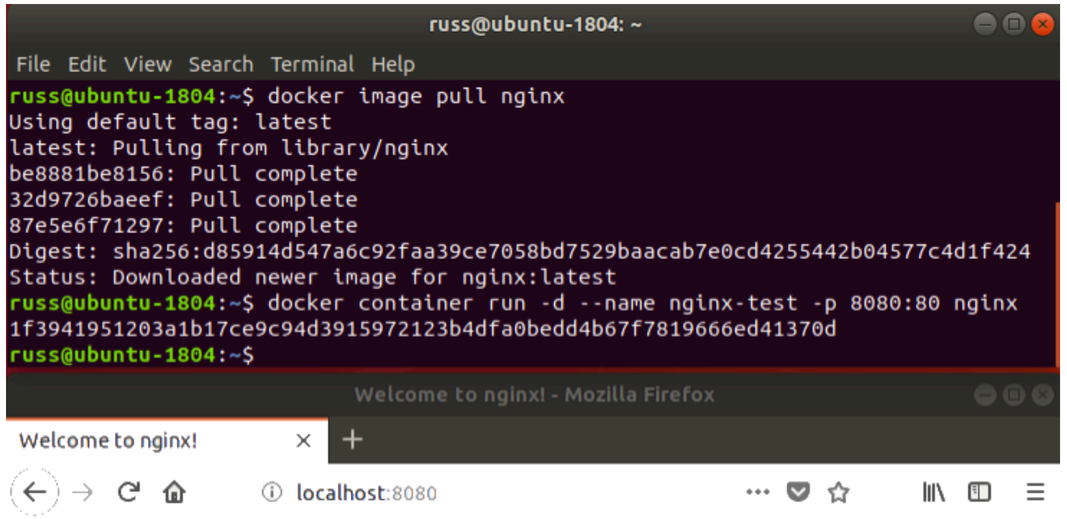

```
Management Commands:
  builder     Manage builds
  config      Manage Docker configs
  container   Manage containers
  image       Manage images
  network     Manage networks
  node        Manage Swarm nodes
  plugin      Manage plugins
  secret      Manage Docker secrets
  service     Manage services
  stack       Manage Docker stacks
  swarm       Manage Swarm
  system      Manage Docker
  volume      Manage volumes
```

Pragra
ONWARD & UPWARD

Run your first Container


```
~ docker container run –d ––name nginx–test –p 8080:80 nginx
```

- Command will run the a container name ngnix-test and will forward port 80 from container to 8080 of localhost



Pragra
ONWARD & UPWARD

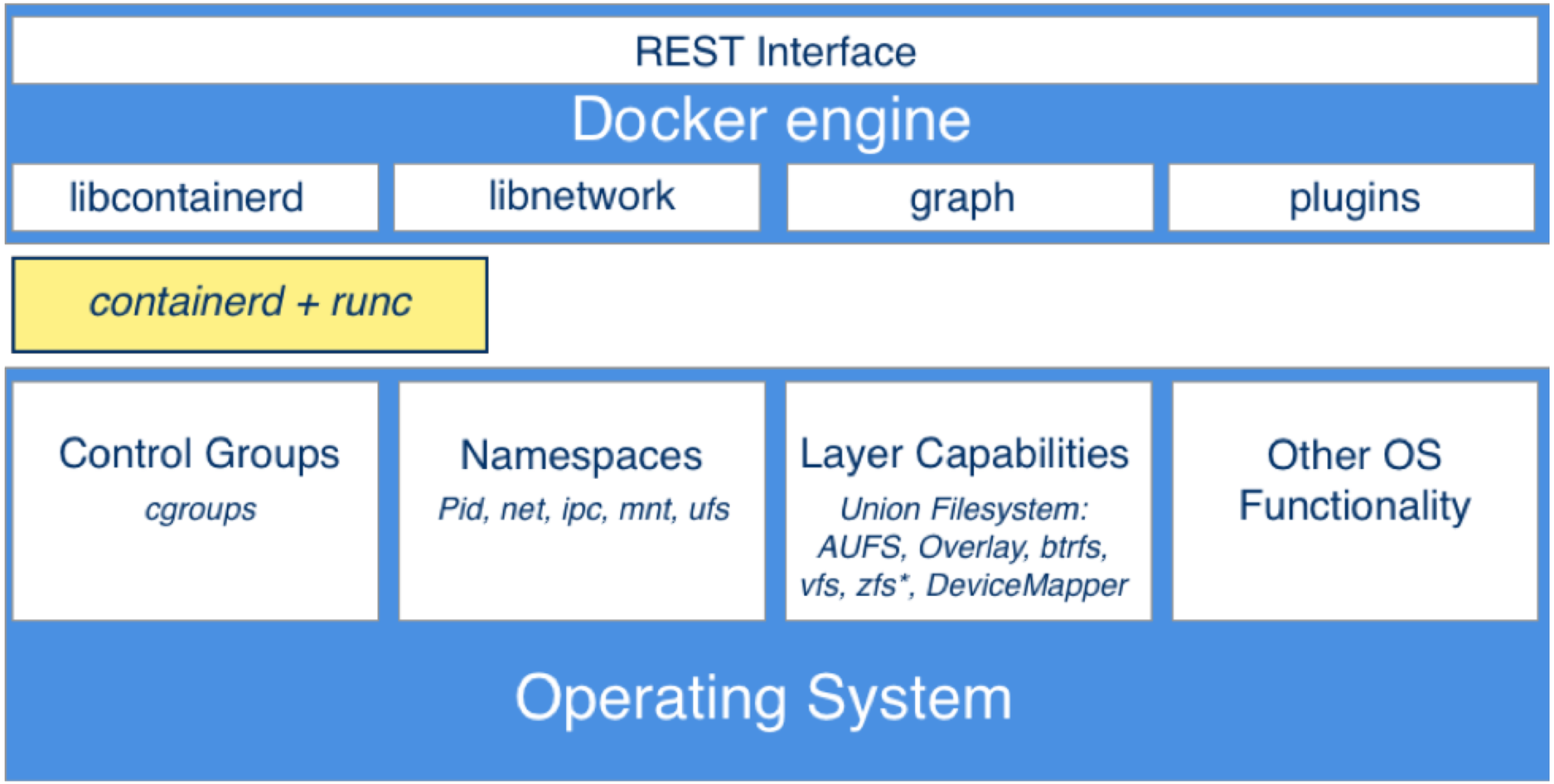
Questions ?

- What command did we use to run a container?
- Can you run a docker in a VM
- Can we have docker and VM on same machine ?
- Why Docker is better than VM ?

•Docker CLI restructure blog post: https://blog.docker.com/2017/01/whats-new-in-docker-1-13/

•Docker Extended Support Announcement: https://blog.docker.com/2018/07/extending-support-cycle-docker-community-edition/

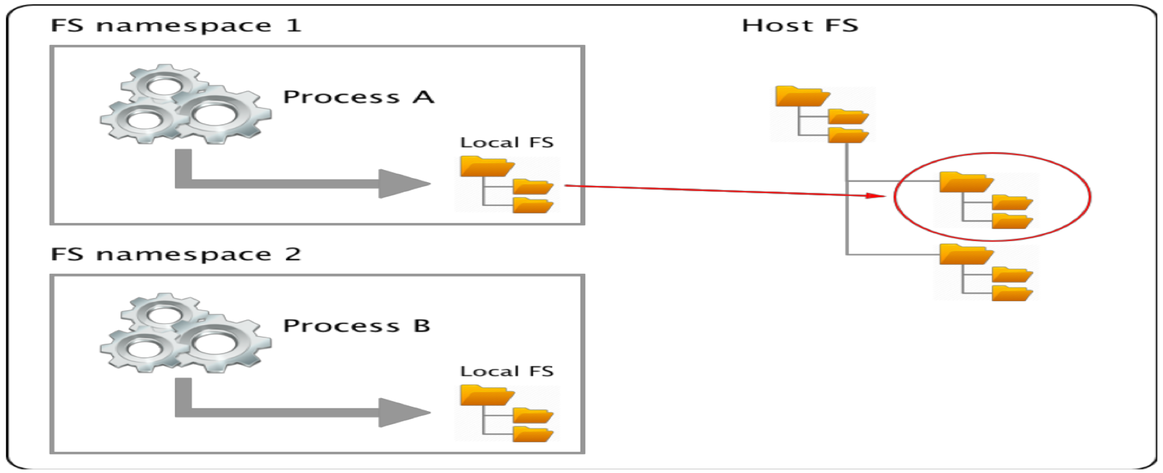
Docker Architecture

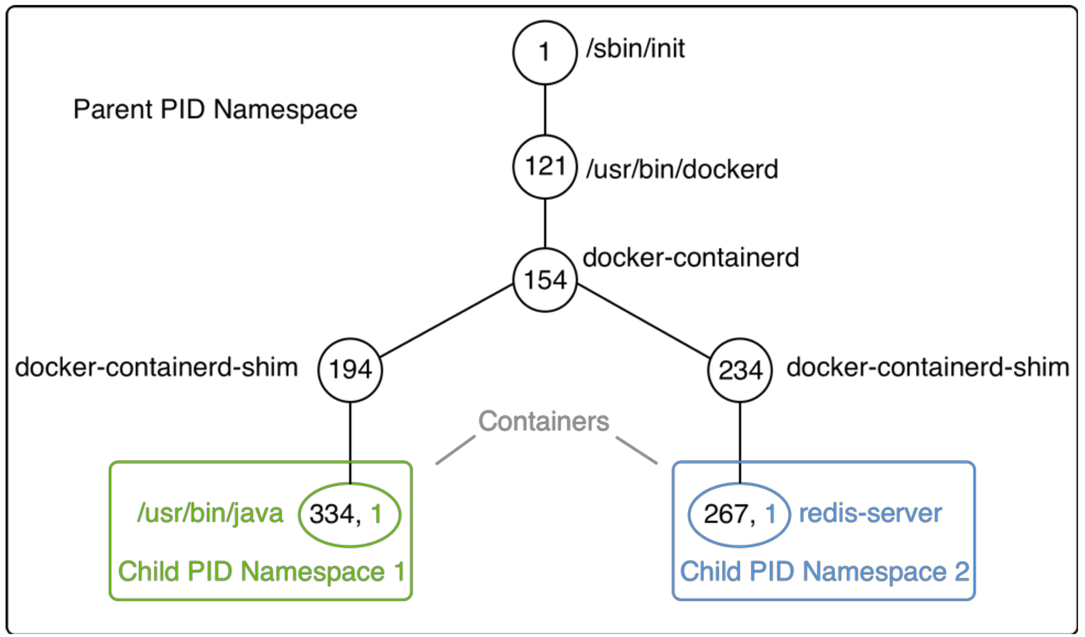


Linux Namespaces

A namespace is an abstraction of global resources such as filesystems, network access, process tree (also named PID namespace) or the system group IDs, and user IDs. A Linux system is initialized with a single instance of each namespace type. After initialization, additional namespaces can be created or joined.

Docker creates child namespace for each container.

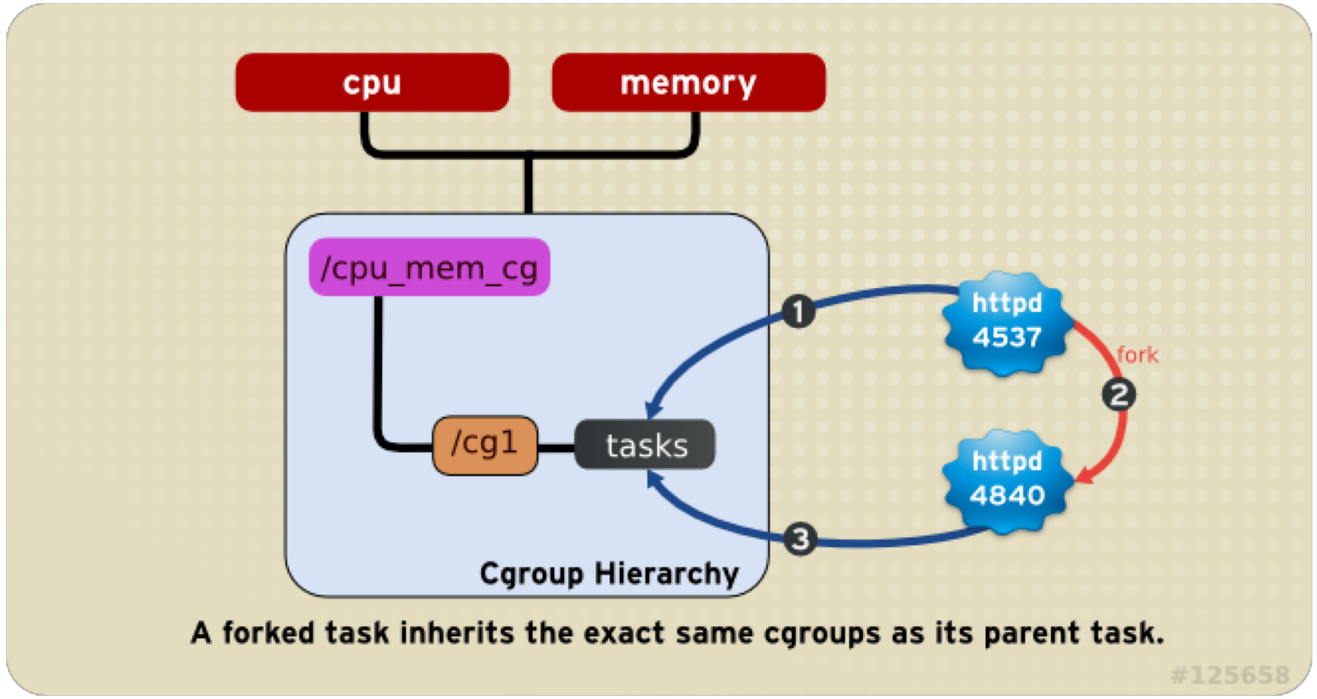




Pragra
ONWARD & UPWARD

Control Groups (CGROUP)

Linux cgroups are used to limit, manage, and isolate resource usage of collections of processes running on a system. Resources are CPU time, system memory, network bandwidth, or combinations of these resources, and so on.

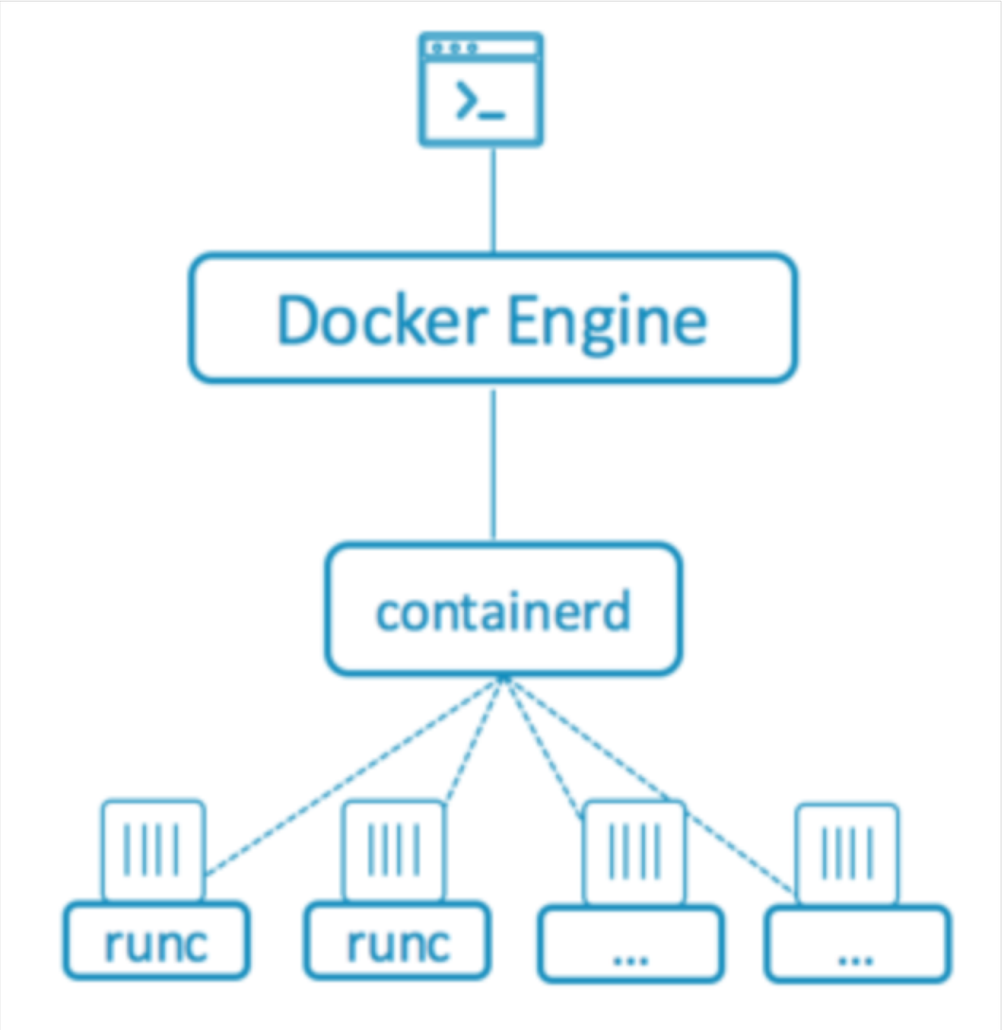


Pragra
ONWARD & UPWARD

Runc

Runc is a lightweight, portable container runtime. It provides full support for Linux namespaces as well as native support for all security features available on Linux, such as SELinux, AppArmor, seccomp, and cgroups.

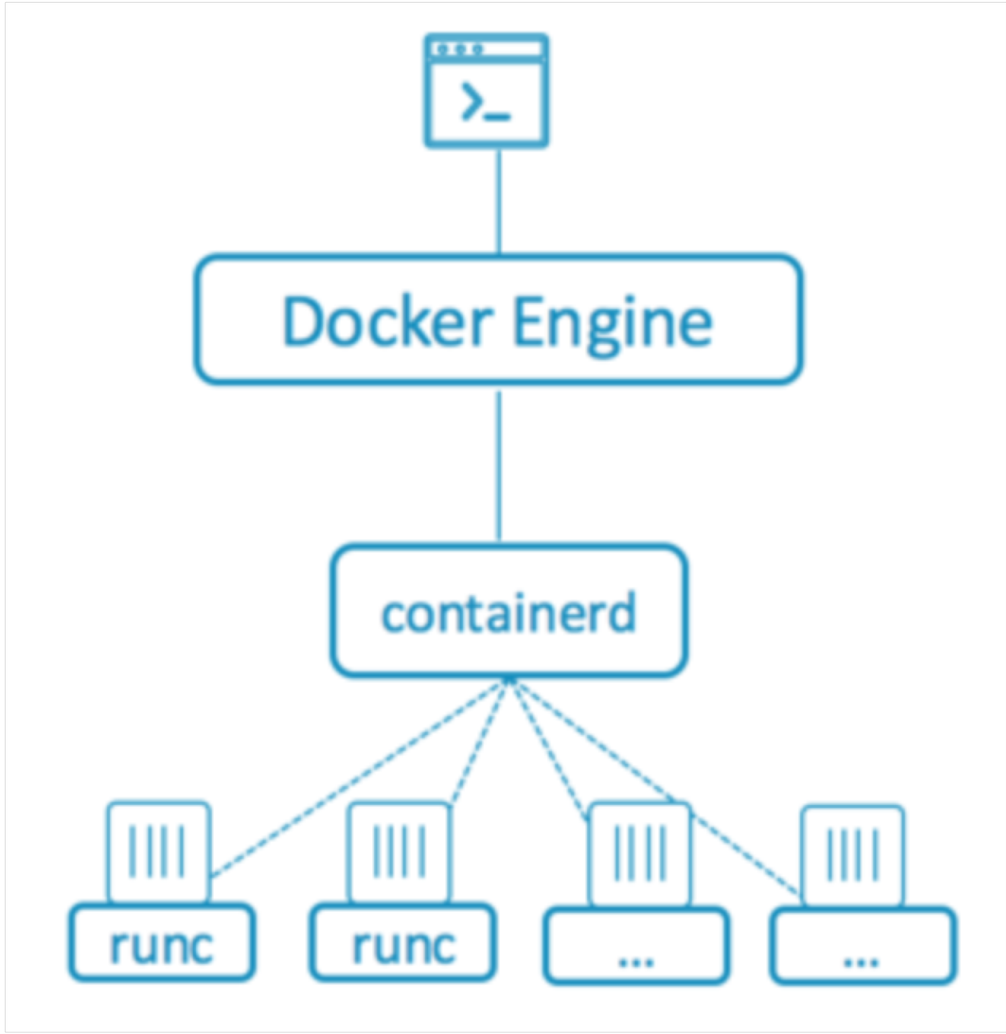
Runc is a tool for spawning and running containers according to the **Open Container Initiative**(**OCI**) specification. It is a formally specified configuration format, governed by the **Open Container Project** (**OCP**) under the auspices of the Linux Foundation.



Pragra
ONWARD & UPWARD

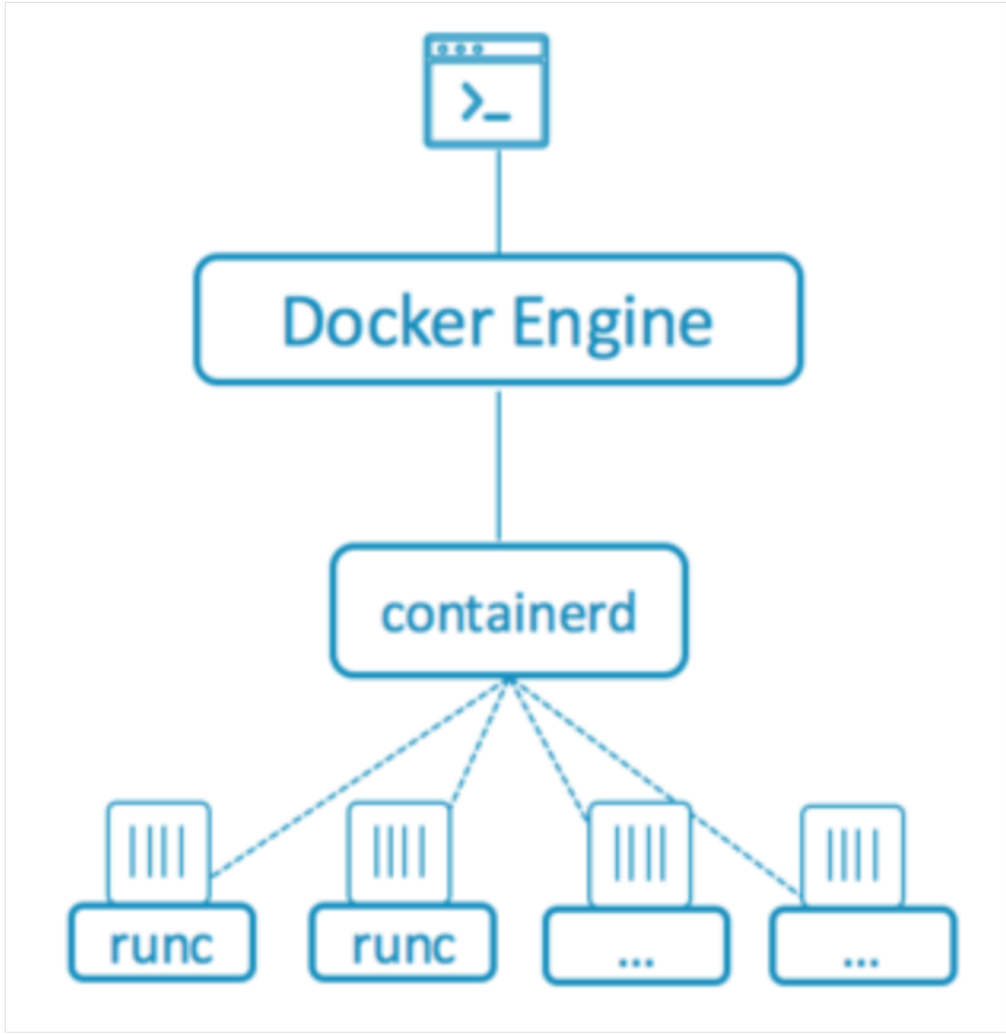
Container.d

containerd builds on top of runc (container runtime), and adds higher-level features, such as image transfer and storage, container execution, and supervision, as well as network and storage attachments. With this, it manages the complete life cycle of containers. Containerd is the reference implementation of the OCI specifications and is by far the most popular and widely-used container runtime.



Container.d

containerd builds on top of runc (container runtime), and adds higher-level features, such as image transfer and storage, container execution, and supervision, as well as network and storage attachments. With this, it manages the complete life cycle of containers. Containerd is the reference implementation of the OCI specifications and is by far the most popular and widely-used container runtime.



Container – Commands -RUN

Usage: docker container run [OPTIONS] IMAGE [COMMAND] [ARG...]

```
docker container run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest:
sha256:6540fc08ee6e6b7b63468dc3317e3303aae178cb8a45ed3123180328
bcc1d20f
Status: Downloaded newer image for hello-world:latest
```

Docker unsuccessfully searched the local cache and then downloaded the hello-world:latest container image from the repository. Each container image description is made up of three parts:

- Docker registry host name
- Slash-separated name
- Tag name

Example hello-world image is stored this way – docker.io/hello-world:latest
Where docker.io is registry , hellow-world is name and latest is tag

Container – Commands -RUN

Usage: docker container run [OPTIONS] IMAGE [COMMAND] [ARG...]

To run container as detached

```
➜  ~ docker container run -d hello-world
```

To run a container with name

```
➜  ~ docker container run --name helloworld  -d hello-world
```

Run a container with port mapping and forwarding : Port forwarding uses host: guest format

```
➜  ~ docker container run --name helloworld  -d -p 8080:80 hello-world
```

Running a command in the container during launch of container

```
➜  ~ docker container run --name helloworld  -d  hello-world echo 'enjoy'
```

Running a container with interactive attached terminal

```
➜  ~ docker container run --name helloworld  -idt hello-world
```

Exercise

- *1. Launch a ubuntu container and execute ls command in container*
- *2. Launch in centos container and ssh to this container*
- *3. Install git to this container*

Container – Commands - LS

 # Usage: docker container ls [OPTIONS]

Docker container ls will list all the container currently active

```
➜  ~ docker container ls
CONTAINER
ID        IMAGE               COMMAND             CREATE
D             STATUS              PORTS               NA
MES
```

Lists all container even the killed and exited ones

```
➜  ~ docker container ls -a
```

List last 2 container

```
➜  ~ docker container ls -n 2
```

Just list only id of container

```
~ docker container ls -n 2 -q
```

```
Aliases:
  ls, ps, list


Options:
  -a, --all             Show all containers (default shows just
running)
  -f, --filter filter   Filter output based on conditions
provided
      --format string   Pretty-print containers using a Go
template
  -n, --last int        Show n last created containers (includes
all states) (default -1)
  -l, --latest          Show the latest created container
(includes all states)
      --no-trunc        Don't truncate output
  -q, --quiet           Only display numeric IDs
  -s, --size            Display total file sizes
```

Pragra

ONWARD & UPWARD

Exercise

- *1. List all container created*
- *2.Just list the container which are running*
- *3. List containers with name helloword and mysql*

Container – Commands - RM

 # Usage: docker container rm [OPTIONS]

Docker container rm, will remove the container from system

```
➜  ~ docker container rm <container_ids>
```

Removing the running container without stopping them

```
➜  ~ docker container rm –f 98378377a
```

Usage: docker container rm [OPTIONS] CONTAINER [CONTAINER...]

Remove one or more containers

Options:
-f, --force Force the removal of a running container (uses SIGKILL)
-l, --link Remove the specified link
-v, --volumes Remove the volumes associated with the container

Exercise

- *1. Remove all container running non –running*

Container – Commands – Create/Stop/Start

 # Usage: docker container create [OPTIONS] IMAGE

```
➜  ~ docker container create --name ngnix-test -p 8080:80 nginx
```

Container create command create a container but it will not run . To run a container you need to use start command

```
➜  ~ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
78433ad748e2        nginx               "nginx -g 'daemon of…"   6 seconds ago       Created
```

To run a container

```
➜  ~ docker container start 78433ad748e2
```

To stop a container

```
➜  ~ docker container stop 78433ad748e2
```

To kill a container

```
➜  ~ docker container kill 78433ad748e2
```

Pragra
ONWARD & UPWARD

Exercise

- *1. Run mongoDB, create a collection in MongoDB. Stop the container and start it again, verify collection is still there.*
- *Create mysql container and run and kill it.*

Container – Commands – LOGS

 # Usage: docker container logs [OPTIONS] CONTAINER

```
➜  ~ docker container logs --follow --timestamps 78433ad748e2
```

```
➜  ~ docker container logs --follow --timestamps 78433ad748e2
2019-07-24T18:00:42.225431266Z 172.17.0.1 - - [24/Jul/2019:18:00:42 +0000] "GET / HTTP/1.1" 200
612 "-" "curl/7.54.0" "-"
```

the short form of the command

docker container logs -f -t web-server

the short form of the command

```
➜  ~ docker container logs --follow --tail 5 78433ad748e2
```

The --details, --follow, --timestamps, and --tail parameters are all optional,

Pragra

ONWARD & UPWARD

Exercise

- *1. Run mongoDB, get the logs for the container*

Container – Commands – TOP

using the new syntax
Usage: docker container top CONTAINER [ps OPTIONS]
docker container top web-server

using the old syntax
docker top web-server

List processes are running inside a container

```
→  ~ docker container top 78433ad748e2
PID                 USER                TIME                COMMAND
2673                root                0:00                nginx: master process nginx -g daemon off;
2708                101                 0:00                nginx: worker process
```

Inspecting a container

using the new syntax
Usage: docker container inspect [OPTIONS] CONTAINER [CONTAINER...]

```
➜  ~ docker container inspect 78433ad748e2
```

Inspect will dump a json with all details.

Get details of particular json node and format

```
➜  ~ docker container inspect --format '{{json .State}}'  78433ad748e2 |jq
{
  "Status": "running",
  "Running": true,
  "Paused": false,
  "Restarting": false,
  "OOMKilled": false,
  "Dead": false,
  "Pid": 2673,
  "ExitCode": 0,
  "Error": "",
  "StartedAt": "2019-07-24T17:59:03.780192719Z",
  "FinishedAt": "0001-01-01T00:00:00Z"
}
```

Get particular value

```
➜  ~ docker container inspect --format '{{json .State}}'  78433ad748e2 |jq '.StartedAt'
"2019-07-24T17:59:03.780192719Z"
```

Pragra
ONWARD & UPWARD

Inspecting a container

Get IP address of container using inspect

```
➜  ~ docker container inspect -f '{{json .NetworkSettings}}'
78433ad748e2  |jq .IPAddress
"172.17.0.2"
➜  ~
```

Get ports of container

```
➜  ~ docker container inspect --format '{{json .NetworkSettings}}'
78433ad748e2 | jq '.Ports'
{
  "80/tcp": [
    {
      "HostIp": "0.0.0.0",
      "HostPort": "8080"
    }
  ]
}
```

Getting a container stats

Provides live, continually-updated usage statistics for one or more running containers

```
➜  ~ docker container stats 78433ad748e2
```

```
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT     MEM %               NET
I/O                 BLOCK I/O           PIDS
78433ad748e2        ngnix-test          0.00%               1.895MiB / 1.952GiB   0.09%               2.02kB /
1.27kB     0B / 0B             2
```

Exercise

- *1. Run mongo container.*
- *2. Run a mysql container*
- *3. list ip and port for the containers*
- *4. get CPU and % Memory used by them.*

Attaching a container

run a container detached

```
➜  ~ docker container run -idt  --name ngnix-test3 -p 8081:80 nginx
50908027af2d234d6b5c374e8a746044f866e3fbe6f7b6097a99909ce60eae6b
```

Attach container to the terminal

```
➜  ~ docker container attach 50908027
```

issue a *Ctrl* + *PQ* keystroke to detach

Pragra

ONWARD & UPWARD

Docker container exec

Sometimes, when you have a container running detached, you might want to get access to it, but don't want to attach to the executing command. You can accomplish this by using the container exec command

```
➜  ~ docker container exec 78433ad748e2 ls
```

Run exec with interactive mode, example running shell

```
➜  ~ docker container exec -it 78433ad748e2 /bin/bash
root@78433ad748e2:/#
```

Docker Images

Pragra

ONWARD & UPWARD

Images

In Linux, everything is a file. The whole operating system is basically a filesystem with files and folders stored on the local disk. This is an important fact to remember when looking at what container images are. As we will see, an image is basically a big tarball containing a filesystem. More specifically, it contains a layered filesystem

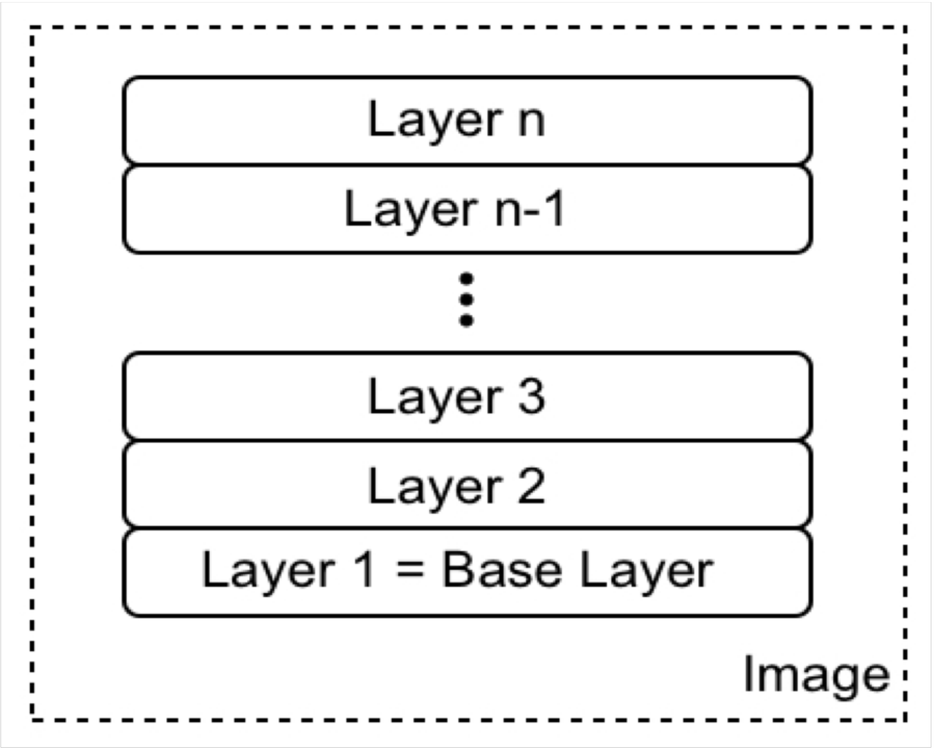
Layered FileSystem

Container images are templates from which containers are created. These images are not just one monolithic block, but are composed of many layers. The first layer in the image is also called the base layer:

Each individual layer contains files and folders.

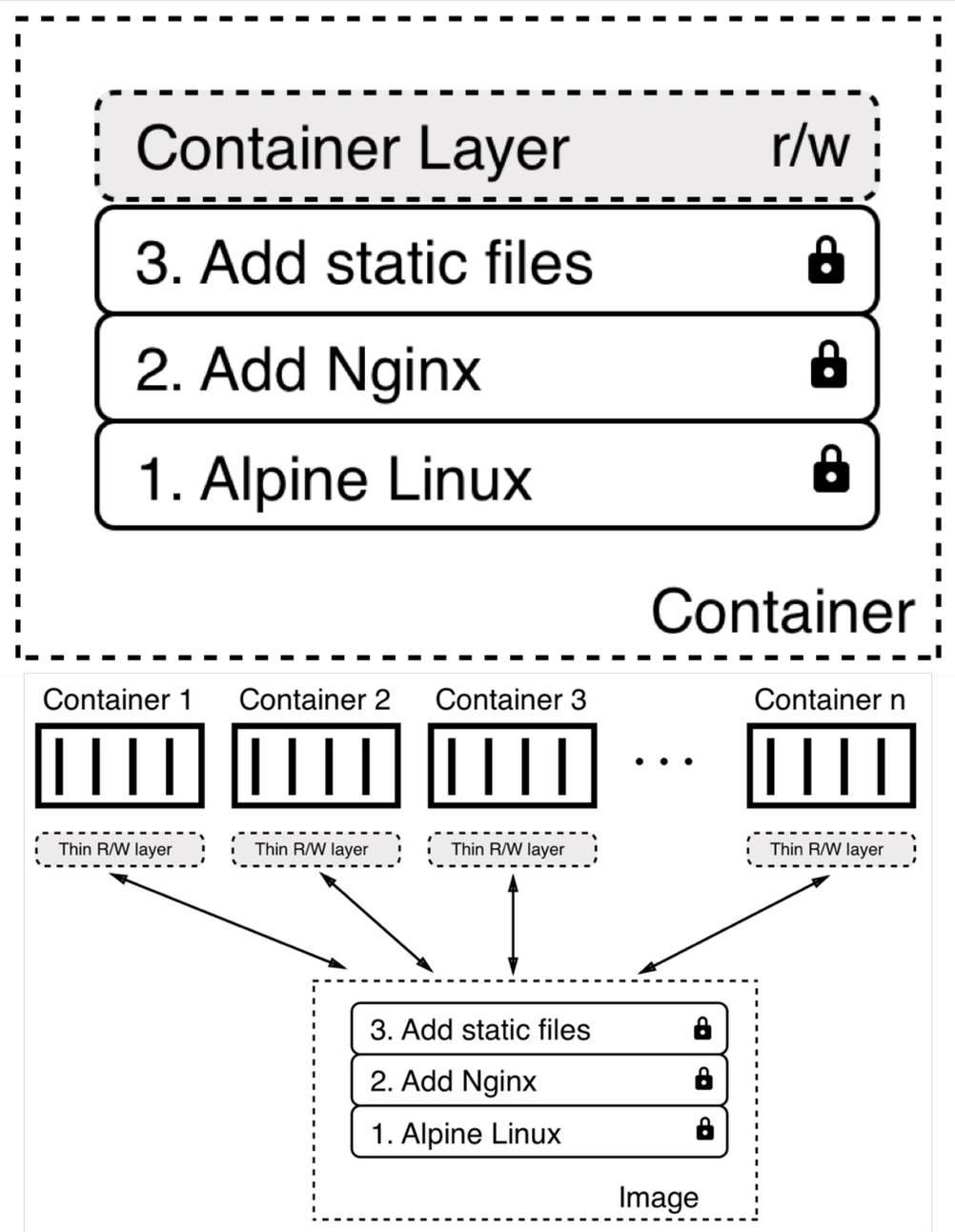
Each layer only contains the changes to the filesystem with respect to the underlying layers

In the following image, we can see what a custom image for a web application using Nginx as a web server could look like:



Writetable Container Layer

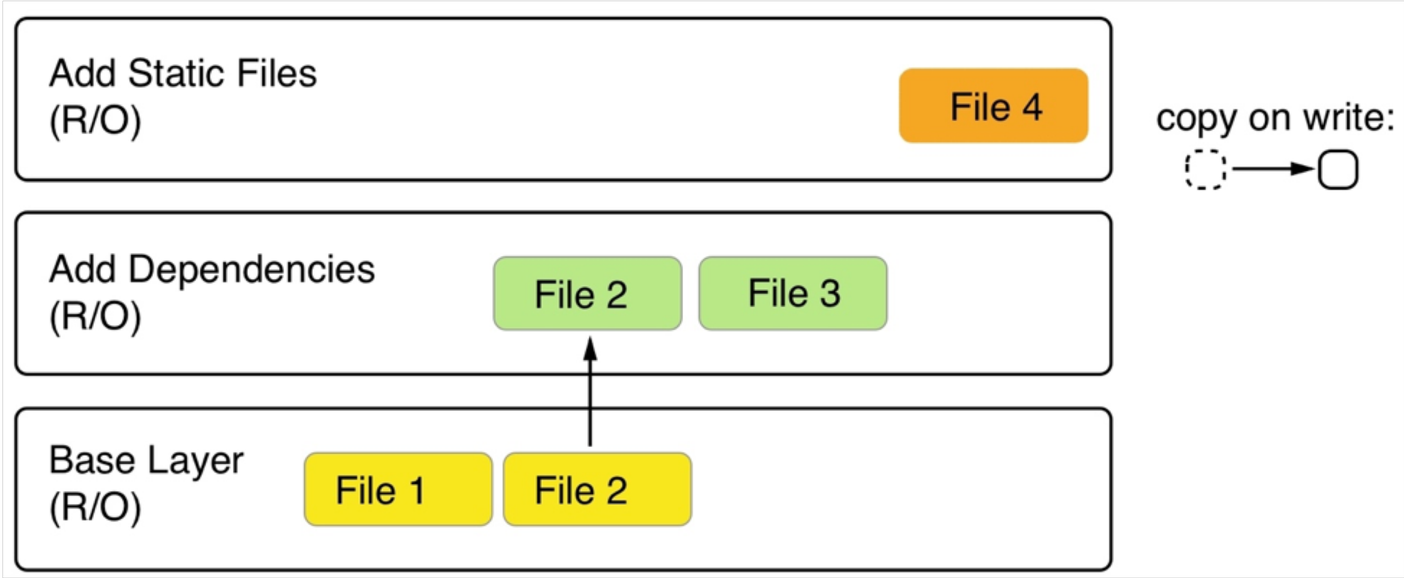
container image is made of a stack of immutable or read-only layers. When the Docker engine creates a container from such an image, it adds a writable container layer on top of this stack of immutable layers. Our stack now looks as follows



Pragra
ONWARD & UPWARD

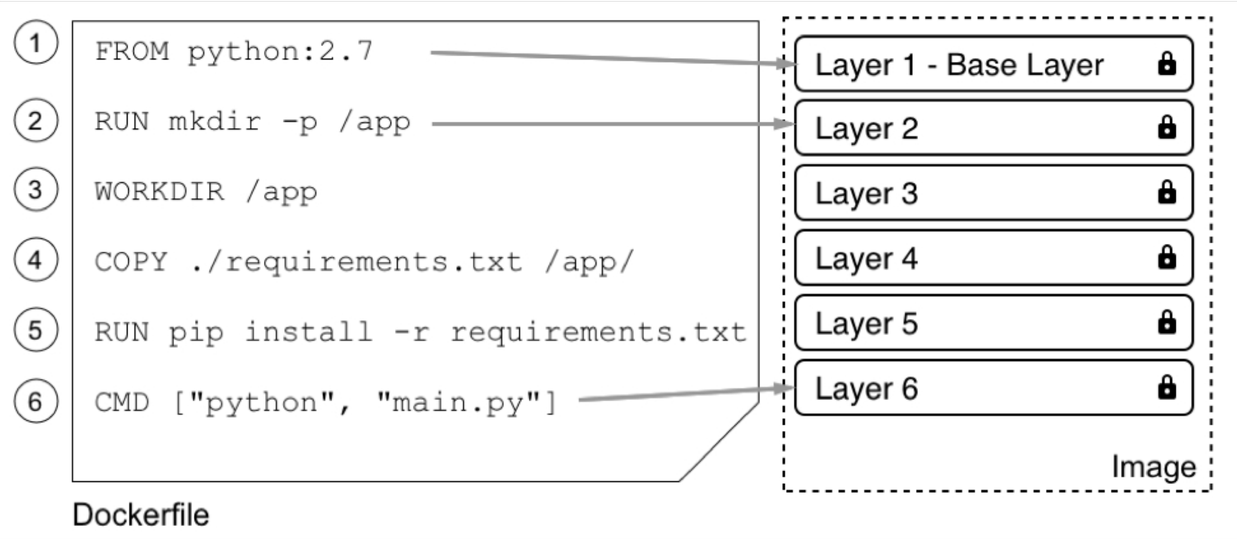
Copy on Write

Docker uses the copy-on-write technique when dealing with images. Copy-on-write is a strategy of sharing and copying files for maximum efficiency. If a layer uses a file or folder that is available in one of the low-lying layers, then it just uses it. If, on the other hand, a layer wants to modify, say, a file from a low-lying layer, then it first copies this file up to the target layer and then modifies it. In the following figure, we can see a glimpse of what this means:



Dockerfile

A Dockerfile is a text file that contains instructions used by the Docker daemon to create a Docker image. The instructions are defined using a type of value pair syntax. Each one has an instruction word followed by the parameters for that instruction. Every command gets its own line in the Dockerfile. Although the Dockerfile instructions are not case-sensitive, there is a well-used convention that the instruction word is always uppercase.

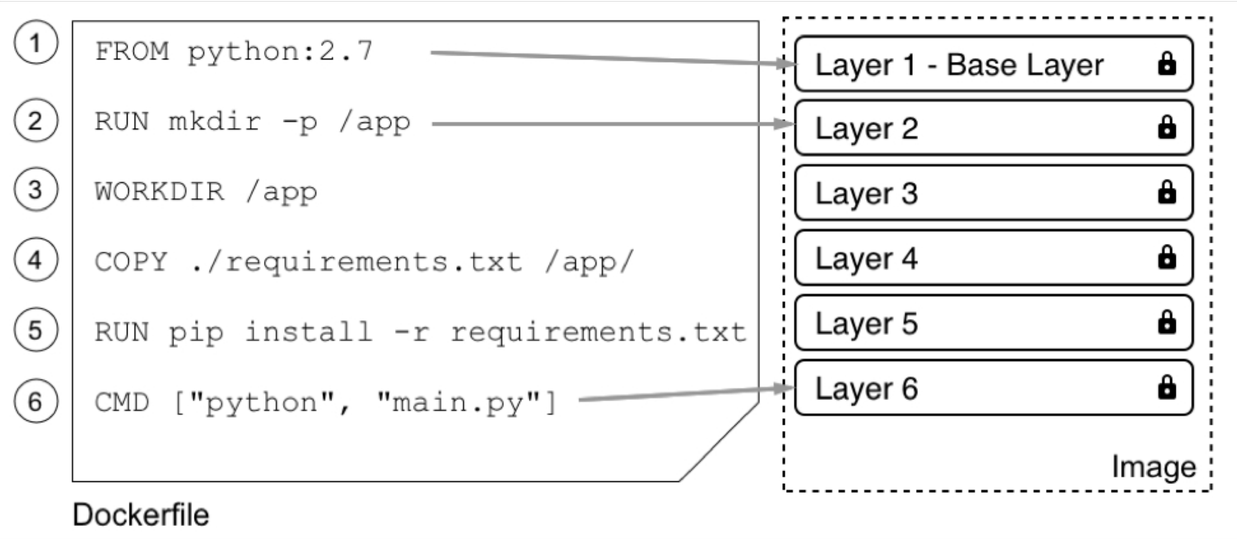


Pragra
ONWARD & UPWARD

FROM Instruction

Every Dockerfile must have a FROM instruction, and it must be the first instruction in the file.

The FROM instruction sets the base for the image being created and instructs the Docker daemon that the base of the new image should be the existing Docker image specified as the parameter



Pragra
ONWARD & UPWARD

The LABEL instruction

The LABEL instruction is a way to add metadata to your Docker image. This instruction adds embedded key-value pairs to the image. The LABEL instruction adds a zero-byte-sized layer to the image when it is created. An image can have more than one LABEL, and each LABEL instruction can provide one or more LABELs. The most common use for the LABEL instruction is to provide information about the image maintainer.

```
LABEL maintainer="Earl Waud
<earlwaud@mycompany.com>"
LABEL "description"="My development Ubuntu
image"
LABEL version="1.0"
LABEL label1="value1" \
label2="value2" \
lable3="value3"
LABEL my-multi-line-label="Labels can span \
more than one line in a Dockerfile."
LABEL support-email="support@mycompany.com"
support-phone="(123) 456-7890"
```

The COPY instruction

The COPY instruction is used to copy files and folders into the Docker image being built

Without the --chown parameter, the owner ID and group ID will both be set to 0.

The <src> or source is a filename or folder path and is interpreted to be relative to the context of the build. We will talk more about the build context later in this chapter, but for now, think of it as where the build command is run. The source may include wildcards

The <dest> or destination is a filename or path inside of the image being created

```
# COPY instruction syntax
COPY [--chown=<user>:<group>] <src>... <dest>
# Use double quotes for paths containing whitespace)
COPY [--chown=<user>:<group>] ["<src>",... "<dest>"]
```

```
# COPY instruction Dockerfile for Docker Quick Start
FROM alpine:latest
LABEL maintainer="Earl Waud <earlwaud@mycompany.com>"
LABEL version=1.0
# copy multiple files, creating the path
"/theqsg/files" in the process
COPY file* theqsg/files/
# copy all of the contents of folder "folder1" to
"/theqsg/"
# (but not the folder "folder1" itself)
COPY folder1 theqsg/
# change the current working directory in the image to
"/theqsg"
WORKDIR theqsg
# copy the file special1 into "/theqsg/special-files/"
COPY --chown=35:35 special1 special-files/
# return the current working directory to "/"
WORKDIR /
CMD ["sh"]
```

The ADD instruction

The ADD instruction is used to copy files and folders into the Docker image being built

ADD instruction can actually do more than the COPY instruction. The more is dependent upon the values used for the source input. With the COPY instruction, the source can be files or folders. However, with the ADD instruction, the source can be files, folders, a local .tar file, or a URL.

```
# ADD instruction syntax
ADD [--chown=<user>:<group>] <src>... <dest>
# Use double quotes for paths containing whitespace)
ADD [--chown=<user>:<group>] ["<src>",... "<dest>"]
```

```
# ADD instruction Dockerfile for Docker Quick Start
FROM alpine
LABEL maintainer= “Atin<atin@pragra.co>"
LABEL version=3.0
ADD https://github.com/docker-library/hello-
world/raw/master/amd64/hello-world/hello /
RUN chmod +x /hello
CMD ["/hello"]
```

Pragra

ONWARD & UPWARD

The ENV instruction

ENV instruction is used to define environment variables that will be set in the running containers created from the image being built. The variables are defined using typical key-value pairs. A Dockerfile can have one or more ENV instructions. Here is the ENV instruction syntax:

```
# ENV instruction syntax
ENV <key> <value>
ENV <key>=<value> ..
```

```
# ENV instruction Dockerfile for Docker Quick Start
FROM alpine
LABEL maintainer="Atin <atin@pragra.co>"
ENV appDescription This app is a sample of using ENV instructions
ENV appName=env-demo
ENV note1="The First Note First" note2=The\ Second\ Note\ Second \
note3="The Third Note Third"
ENV changeMe="Old Value"
CMD ["sh"]
```

The ENV instruction

ENV instruction is used to define environment variables that will be set in the running containers created from the image being built. The variables are defined using typical key-value pairs. A Dockerfile can have one or more ENV instructions. Here is the ENV instruction syntax:

```
# ENV instruction syntax
ENV <key> <value>
ENV <key>=<value> ..
```

```
# ENV instruction Dockerfile for Docker Quick Start
FROM alpine
LABEL maintainer="Atin <atin@pragra.co>"
ENV appDescription This app is a sample of using ENV instructions
ENV appName=env-demo
ENV note1="The First Note First" note2=The\ Second\ Note\ Second \
note3="The Third Note Third"
ENV changeMe="Old Value"
CMD ["sh"]
```

The ARG instruction

When building Docker images, you may need to use variable data to customize the build. The ARG instruction is the tool to handle that situation. To use it, you add ARG instructions to your Dockerfile, and then when you execute the build command, you pass in the variable data with a --build-arg parameter.

```
➜  ~ docker image build --rm \
 --build-arg key1="buildTimeValue" \
 --build-arg key2="good till env
instruction" \
 --tag arg-demo:2.0 .
```

The ARG instruction syntax

ARG <varname>[=<default value>]

The build-arg parameter syntax

docker image build --build-arg <varname>[=<value>]

...

```
# ENV instruction Dockerfile for Docker Quick Start
FROM alpine
LABEL maintainer="Atin <atin@pragra.co>"
ENV key1="ENV is stronger than an ARG"
RUN echo ${key1}
ARG key1="not going to matter"
RUN echo ${key1}

RUN echo ${key2}
ARG key2="defaultValue"
RUN echo ${key2}
ENV key2="ENV value takes over"
RUN echo ${key2}
CMD ["sh"]
```

Pragra

ONWARD & UPWARD

The USER instruction

The USER instruction allows you to set the current user (and group) for all of the instructions that follow in the Dockerfile, and for the containers that are run from the built image. The syntax for the USERinstruction is as follows

User instruction syntax

USER <user>[:<group>] or

USER <UID>[:<GID>]

```
# ENV instruction Dockerfile for Docker
Quick Start
FROM alpine
LABEL maintainer="Atin <atin@pragra.co>"
RUN id
USER games:games
run id
CMD ["sh"]
```

Pragra

ONWARD & UPWARD

The WORKDIR instruction

The WORKDIR instruction will change the current working directory in the image to the value provided in the instruction. If any segment of the path in the parameter of the WORKDIR instruction does not yet exist, it will be created as part of the execution of the instruction.

WORKDIR instruction syntax

WORKDIR instruction syntax

WORKDIR /path/to/workdir

```
# WORKDIR instruction Dockerfile for Docker Quick Start
FROM alpine
# Absolute path...
WORKDIR /
# relative path, relative to previous WORKDIR instruction
# creates new folder
WORKDIR sub-folder-level-1
RUN touch file1.txt
# relative path, relative to previous WORKDIR instruction
# creates new folder
WORKDIR sub-folder-level-2
RUN touch file2.txt
# relative path, relative to previous WORKDIR instruction
# creates new folder
WORKDIR sub-folder-level-3
RUN touch file3.txt
# Absolute path, creates three sub folders...
WORKDIR /l1/l2/l3
CMD ["sh"]
```

Pragra

ONWARD & UPWARD

The VOLUME instruction

The VOLUME instruction will create a storage location that is outside of the United File System, and by so doing, allow storage to persist beyond the life of your container

```
# VOLUME instruction syntax

VOLUME ["/data"]

# or for creating multiple volumes with a single instruction

VOLUME /var/log /var/db /moreData
```

```
# VOLUME instruction Dockerfile for Docker Quick Start
FROM alpine
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
CMD ["sh"]
```

Pragra

ONWARD & UPWARD

The EXPOSE instruction

The EXPOSE instruction is a way to document what network ports the image expects to be opened when a container is run from the image built using the Dockerfile.

It is important to understand that including the EXPOSE instruction in the Dockerfile does not actually open network ports in containers. When containers are run from the images with the EXPOSE instruction in their Dockerfile, it is still necessary to include the -p or -P parameters to actually open the network ports to the container.

```
# EXPOSE instruction syntax

EXPOSE <port> [<port>/<protocol>...]
```

```
# VOLUME instruction Dockerfile for Docker
Quick Start
FROM alpine
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
CMD ["sh"]
```

Pragra
ONWARD & UPWARD

The RUN instruction

The RUN instruction is the real workhorse of the Dockerfile. It is the tool by which you affect the most change in the resulting docker image. Basically, it allows you to execute any command in the image. There are two forms of the RUN instruction Every RUN instruction creates a new layer in the image, and the layers for each instruction that follow will be built on the results of the RUN instruction's layer

RUN <command>

Exec form

RUN ["executable", "param1", "param2"]

```
# RUN instruction Dockerfile for Docker
Quick Start
FROM ubuntu
RUN useradd --create-home -m -s /bin/bash
dev
RUN mkdir /myvol
RUN echo "hello DQS Guide" > /myvol/greeting
RUN ["chmod", "664", "/myvol/greeting"]
RUN ["chown", "dev:dev", "/myvol/greeting"]
VOLUME /myvol
USER dev
CMD ["/bin/bash"]
```

Exec form of RUN instruction using bash

RUN ["/bin/bash", "-c", "echo hello world > /myvol/greeting"]

Pragra

ONWARD & UPWARD

The CMD instruction

The CMD instruction is used to define the default action taken when containers are run from images built with their Dockerfile. While it is possible to include more than one CMD instruction in a Dockerfile, only the last one will be significant. Essentially, the final CMD instruction provides the default action for the image. This allows you to either override or use the CMD in the image used in the FROM instruction of your Dockerfile

CMD instruction syntax

CMD command param1 param2 (shell form)

CMD ["executable","param1","param2"] (exec form)

CMD ["param1","param2"] (as default parameters to ENTRYPOINT)

```
# RUN instruction Dockerfile for Docker
Quick Start
FROM ubuntu
RUN useradd --create-home -m -s /bin/bash
dev
RUN mkdir /myvol
RUN echo "hello DQS Guide" > /myvol/greeting
RUN ["chmod", "664", "/myvol/greeting"]
RUN ["chown", "dev:dev", "/myvol/greeting"]
VOLUME /myvol
USER dev
CMD ["/bin/bash"]
```

Pragra

ONWARD & UPWARD

The ENTRYPOINT instruction

The ENTRYPOINT instruction is used to configure a docker image to run like an application or a command. For example, we can use the ENTRYPOINT instruction to make an image that displays help for the curl command.

```
FROM alpine
RUN apk add curl
ENTRYPOINT ["curl"]
CMD ["--help"]
```

We can run the container image with no overriding CMD parameter and it will show help for the curl command.
However, when we run the container with a CMD override parameter, in this case, a URL, the response will be to curl the URL

```
# CMD instruction syntax
CMD command param1 param2 (shell form)
CMD ["executable","param1","param2"] (exec form)
CMD ["param1","param2"] (as default parameters to
ENTRYPOINT)
```

Pragra
ONWARD & UPWARD

The HEALTHCHECK instruction

The HEALTHCHECK instruction, which is a fairly new addition to the Dockerfile, is used to define the command to run inside a container to test the container's application health. When a container has a HEALTHCHECK, it gets a special status variable. Initially, that variable will be set to starting. Any time a HEALTHCHECK is performed successfully, the status will be set to healthy. When a HEALTHCHECK is performed and fails, the fail count value will be incremented and then checked against a retries value. If the fail count equals or exceeds the retries value, the status is set to unhealthy.

HEALTHCHECK instruction syntax

HEALTHCHECK [OPTIONS] CMD command

HEALTHCHECK NONE (disable any HEALTHCHECK inherited from the base image)

HEALTHCHECK CMD options

--interval=DURATION (default: 30s)
--timeout=DURATION (default: 30s)
--start-period=DURATION (default: 0s)
--retries=N (default: 3)

```
# HEALTHCHECK instruction Dockerfile for Docker Quick Start
FROM alpine
RUN apk add curl
EXPOSE 80/tcp
HEALTHCHECK --interval=30s --timeout=3s \
CMD curl -f http://localhost/ || exit 1
CMD while true; do echo 'DQS Expose Demo' | nc -l -p 80; done
```

Pragra
ONWARD & UPWARD

The ONBUILD instruction

The ONBUILD instruction is a tool used when creating images that will become the parameter to the FROM instructions in another Dockerfile. The ONBUILD instruction just adds metadata to your image, specifically a trigger that is stored in the image and not otherwise used. However, that metadata trigger does get used when your image is supplied as the parameter in the FROM command of another Dockerfile

ONBUILD instruction syntax

ONBUILD [INSTRUCTION]

```
# my-base Dockerfile
FROM alpine
LABEL maintainer="Atin <atin@pragra.co>"
ONBUILD LABEL version="1.0"
CMD ["sh"]
```

Pragra
ONWARD & UPWARD

The STOPSIGNAL instruction

The STOPSIGNAL instruction is used to set the system call signal that will be sent to the container to tell it to exit. The parameter used in the instruction can be an unsigned number, which equals a position in the kernel's syscall table, or it can be an actual signal name in uppercase. Here is the syntax for the instruction:

```
# STOPSIGNAL instruction syntax

STOPSIGNAL signal
```

```
# my-base Dockerfile
FROM alpine
LABEL maintainer="Atin <atin@pragra.co>"
ONBUILD LABEL version="1.0"
CMD ["sh"]
```

Docker image build command

Docker image build command syntax
Usage: docker image build [OPTIONS] PATH | URL | -

Common options used with the image build command
--rm Remove intermediate containers after a successful build
--build-arg Set build-time variables
--tag Name and optionally a tag in the 'name:tag' format
--file Name of the Dockerfile (Default is 'PATH/Dockerfile')

```
➜  ~ docker image build --rm --build-arg username=35 --tag arg-demo:2.0 .
```

```
➜  ~ docker image build --rm -t arg-demo:2.0 .
```

.dockerignore file

The .dockerignore file is used to exclude files that you do not want to be included with the build context during a docker image build. Using it helps to prevent sensitive and other unwanted files from being included in the build context, and potentially in the docker image. It is an excellent tool to help keep your Docker images small.

```
# Example of a .dockerignore file
# Exclude unwanted files
**/*~
**/*.log
**/.DS_Store
```

Pragra

ONWARD & UPWARD

Docker image command

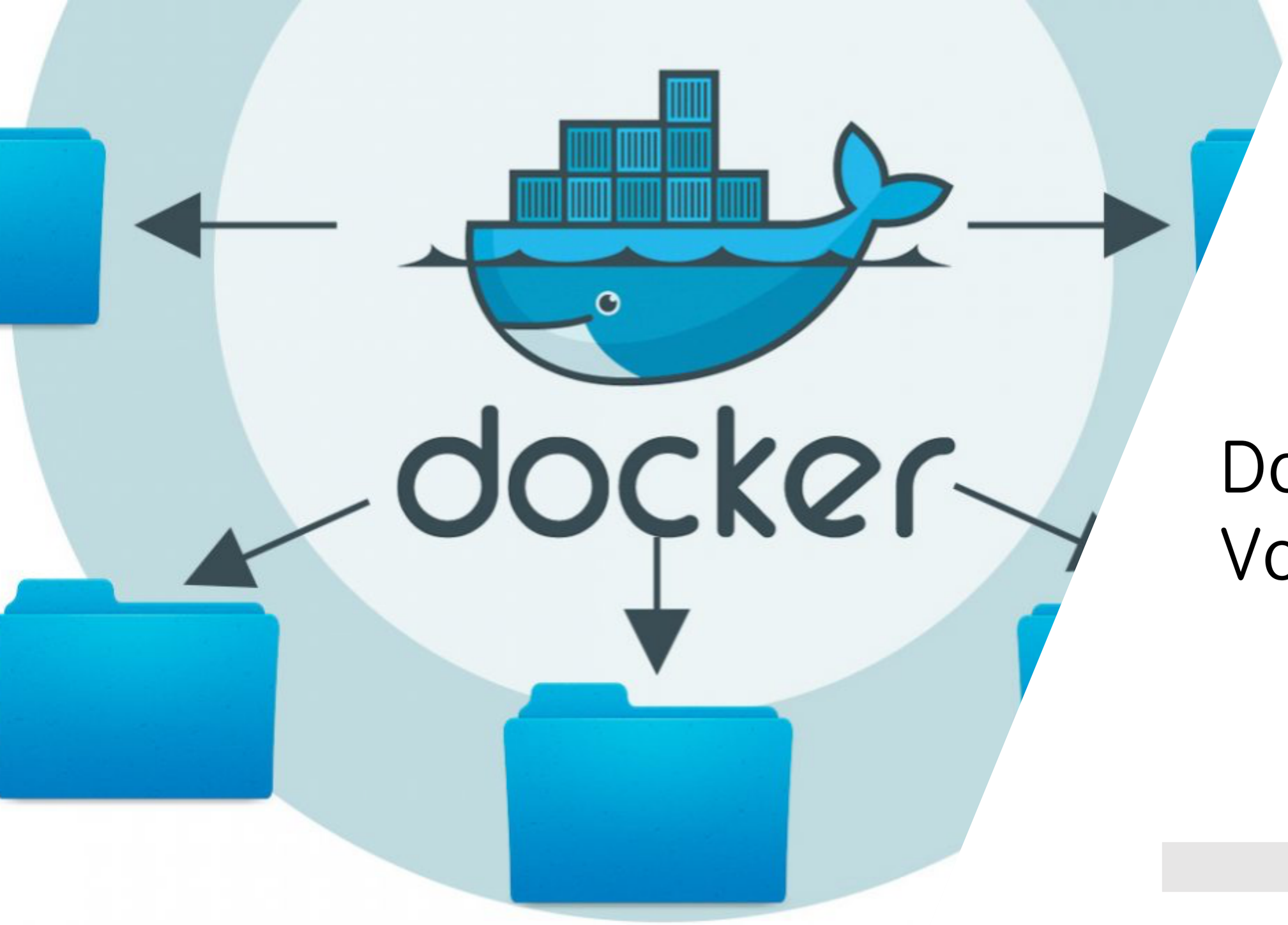

```
Usage: docker image COMMAND


Manage images


Commands:
  build       Build an image from a Dockerfile
  history     Show the history of an image
  import      Import the contents from a tarball to create a
filesystem image
  inspect     Display detailed information on one or more images
  load        Load an image from a tar archive or STDIN
  ls          List images
  prune       Remove unused images
  pull        Pull an image or a repository from a registry
  push        Push an image or a repository to a registry
  rm          Remove one or more images
  save        Save one or more images to a tar archive (streamed
to STDOUT by default)
  tag         Create a tag TARGET_IMAGE that refers to
SOURCE_IMAGE
```

Pragra

ONWARD & UPWARD

Exercise

- *Build a docker image from git repo https://github.com/atinsingh/mongoapi*
- *Build docker image from alpine linux which has curl*
- *Remove all docker image from your localenv*
- *Create new version of image and tag it with version2.0*
- *Run multiple container from same image and make change to one of the container and save change in image layer.*



Docker Volumes

Docker Volumes

```
# Docker volume management command
docker volume
```


When you run a container from a Docker image, the Docker daemon creates a new read-write layer that holds all of the live data that represents your container. When your container makes changes to its filesystem, those changes go into that read-write layer. As such, when your container goes away, taking the read-write layer goes with it

The Docker volume is a storage location that is completely outside of the Union File System. As such, it is not bound by the same rules that are placed on the read-only layers of an image or the read-write layer of a container. A Docker volume is a storage location that, by default, is on the host that is running the container that uses the volume.

```
# Docker volume management subcommands
docker volume create # Create a volume
docker volume inspect # Display information on one or more volumes
docker volume ls # List volumes
docker volume rm # Remove one or more volumes
docker volume prune # Remove all unused local volumes
```

CREATE Volumes

```
# Syntax for the volume create command
Usage: docker volume create [OPTIONS] [VOLUME]
```


When you create a volume default driver is local

```
# The options available to the volume create command:
-d, --driver string # Specify volume driver name (default "local")
--label list # Set metadata for a volume
-o, --opt map # Set driver specific options (default map[])
```

To create volume without specifying the driver and name

```
➜  ~ docker volume create
54f20f4a18ca858fc91c9b309fc713254a1c4b0c33ade516bb737dd6b0285c57

➜  ~ docker volume ls
DRIVER              VOLUME NAME
local               54f20f4a18ca858fc91c9b309fc713254a1c4b0c33ade516
bb737dd6b0285c57
```

CREATE Volumes

```
# Syntax for the volume create command
Usage: docker volume create [OPTIONS] [VOLUME]
```


To create volume by specifying name

```
➜  ~ docker volume create pragra-volume
pragra-volume
```

```
➜  ~ docker volume ls
DRIVER              VOLUME NAME
local               54f20f4a18ca858fc91c9b36bb737dd6b0285c57
local               pragra-volume
```

Lets inspect a volume

```
➜  ~ docker volume inspect pragra-volume
[
    {
        "CreatedAt": "2019-07-24T22:07:17Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/pragra-volume/_data",
        "Name": "pragra-volume",
        "Options": {},
        "Scope": "local"
    }
]
```

MOUNT Volume

Syntax for the volume create command
Usage: docker volume create [OPTIONS] [VOLUME]

Lets bake a image and mount the volume.

```
➜  ~ docker container run --rm -d \
--mount source=pragra-volume,target=/myvol \
--name vol-demo2 \
pragra-vol-demo:1.0 tail -f /dev/null
```

Alternative # Using --mount with source and destination

```
➜  ~ docker container run --rm -d \
   --mount source=my-
volume,destination=/myvol,readonly \
   --name vol-demo2 \
   volume-demo:latest tail -f /dev/null
```

#Using -v

```
➜  ~ docker container run --rm -d \
   -v my-volume:/myvol:ro \
   --name vol-demo3 \
   volume-demo:latest tail -f /dev/null
```

Dockerfile

```
FROM alpine
RUN mkdir /myvol
RUN echo "Data from image" > /myvol/both-places.txt
CMD ["sh"]
```

Pragra
ONWARD & UPWARD

Remove Volume

Remove specified volume from system

```
→  ~ docker volume rm pragra-volume
```

```
# Remove volumes command syntax
Usage: docker volume rm [OPTIONS] VOLUME [VOLUME...]
# Prune volumes command syntax
Usage: docker volume prune [OPTIONS]
```

Remove all volume from system

```
→  ~ docker volume prune
```

Find all dangling unused volume

```
→  ~ docker volume ls --filter
dangling=true
```

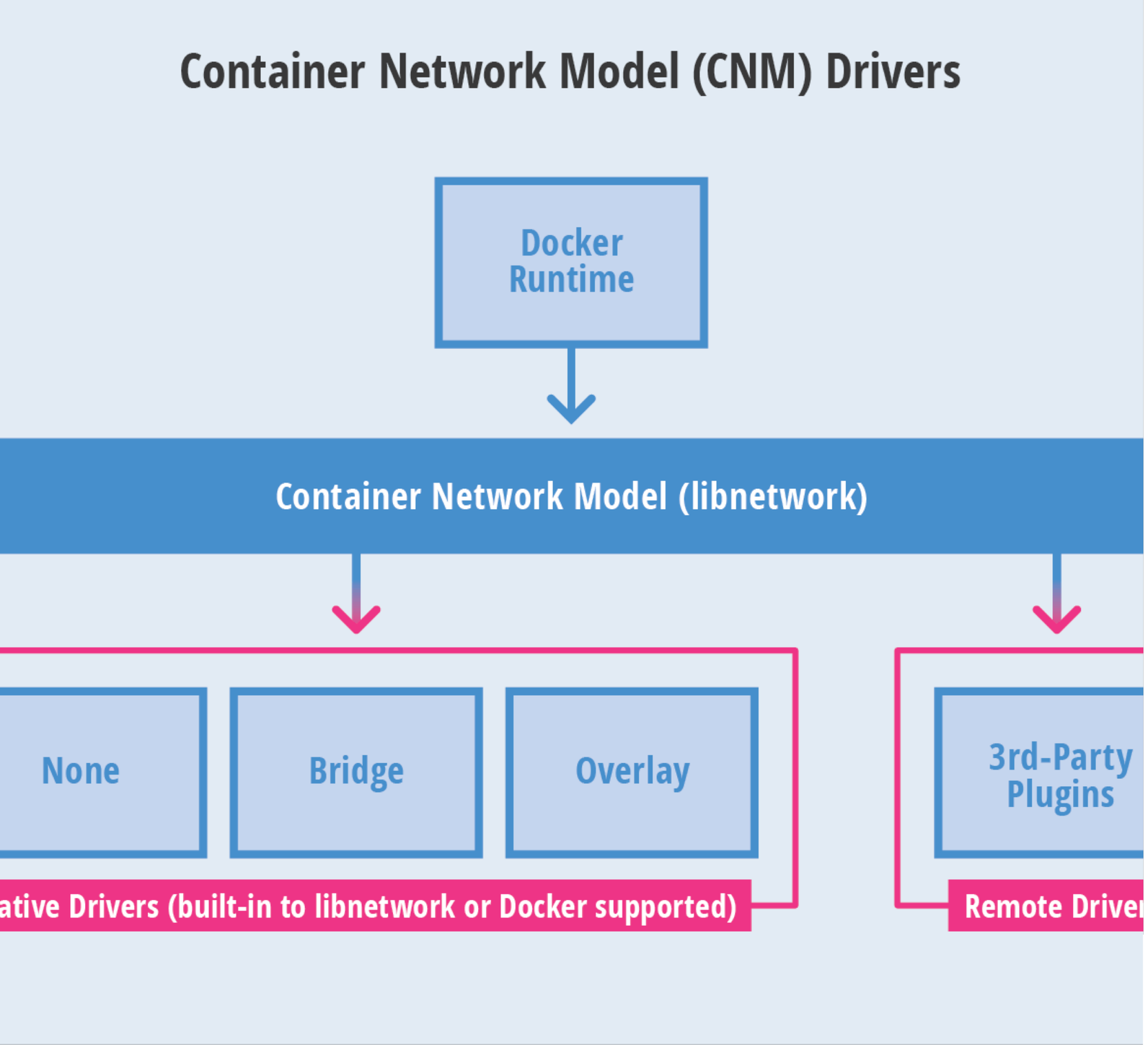
```
FROM alpine
RUN mkdir /myvol
RUN echo "Data from image" > /myvol/both-places.txt
CMD ["sh"]
```

Removed all unused and dangling

```
~ docker volume rm $(docker volume ls --filter dangling=true -q)
```

Pragra

ONWARD & UPWARD



Docker NETWORK

Pragra
ONWARD & UPWARD

Q&A

Pragra

ONWARD & UPWARD