Introduction to Linux Containers & Docker

Tejas Parikh (t.parikh@northeastern.edu)

CSYE 7125 - Advanced Cloud Computing

Fall 2023

Northeastern University

Containers & Global Trade

- Uniform metal containers were invented by Malcom McLean, an American trucking magnate, in 1956.
    - Theft of transported goods was rampant: as an old joke put it, dock workers used to earn “$20 a day and all the Scotch you could carry home.”
- Containers changed this in several ways.
    - The price of everything fell, starting with the cost of loading and unloading.
    - Because containers were packed and sealed at the factory, losses to theft plummeted, which in turn drastically reduced insurance costs.
    - Ships could now get bigger and more efficient while still spending less time in port.
    - Containers made inland distribution by train and lorry easier
- Looking at 22 industrialized countries, study found that containerization is associated with a 320% increase in bilateral trade over the first five years and 790% over 20 years.
    - A bilateral free-trade agreement, by contrast, boosts trade by 45% over 20 years, and membership of GATT raises it by 285%.
    - In other words, containers have boosted globalisation more than all trade agreements in the past 50 years put together.
    - Not bad for a simple box.

Source: https://www.economist.com/the-economist-explains/2013/05/21/why-have-containers-boosted-trade-so-much

Motivation for Containerization

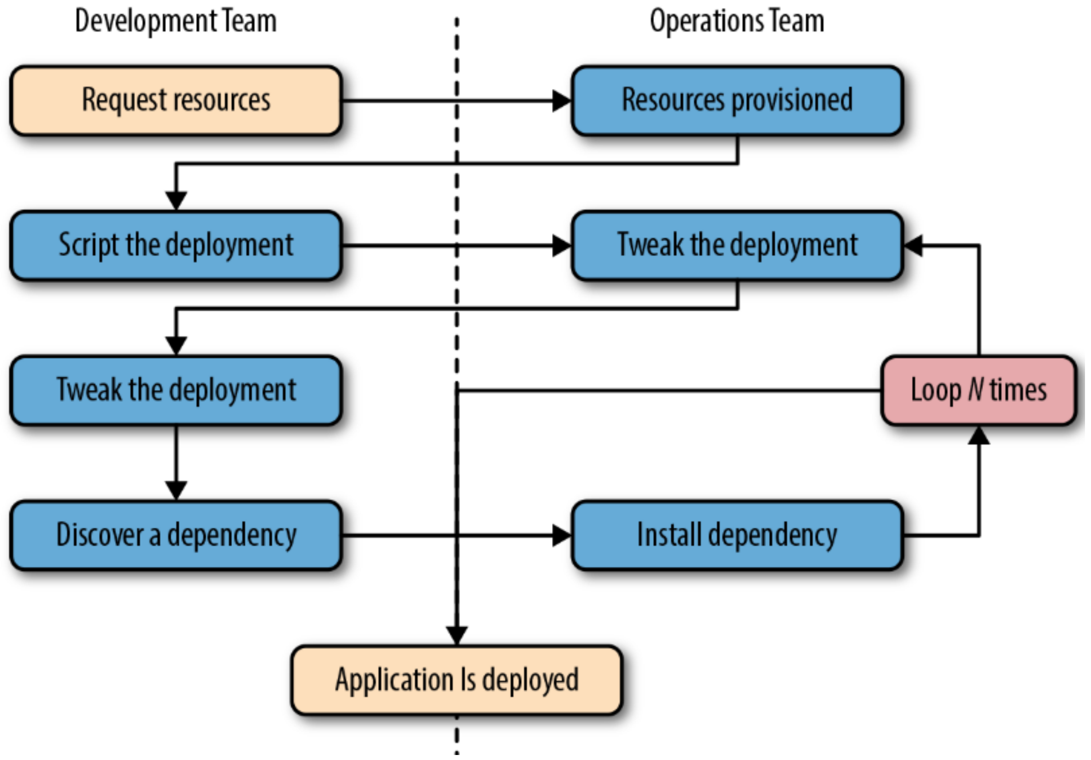
- Ease of Deployment - Write once run anywhere.
- Isolation – Reduce interference between applications.
- Efficiency – Better resource utilization.
- Better packaging – Bundling application software and required OS filesystems together in a single standardized image format.
- Using packaged artifacts to test and deliver the exact same artifact to all systems in all environments.
- Abstract software applications from the hardware without sacrificing resources.

Toward an Immutable Infrastructure

- By deploying most of your applications within containers, you can start simplifying your configuration management story by moving toward an immutable infrastructure, where components are replaced entirely rather than being changed in place. The idea of an immutable infrastructure has gained popularity in response to how difficult it is, in reality, to maintain a truly idempotent configuration management codebase.
- With Docker it is possible to deploy a very lightweight Docker server that needs almost no configuration management, or in many cases, none at all. You handle all of your application management simply by deploying and redeploying containers to the server. When the server needs an important update to something like the Docker daemon or the Linux kernel, you can simply bring up a new server with the changes, deploy your containers there, and then decommission or reinstall the old server.

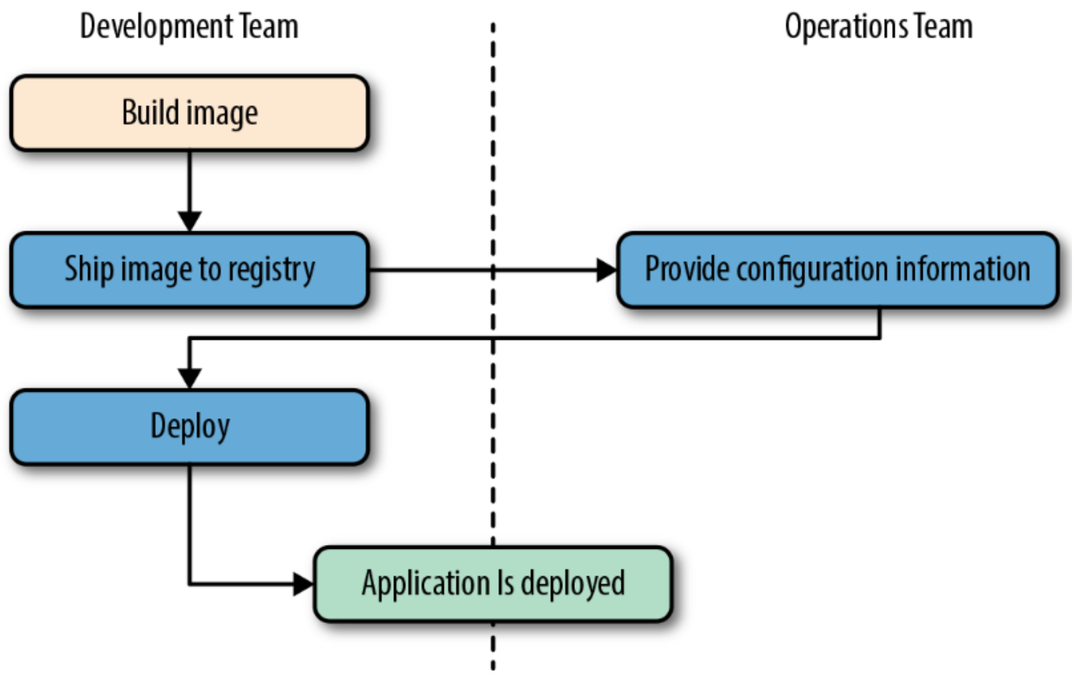
Traditional Deployment Workflow

- Application developers request resources from operations engineers.
- Resources are provisioned and handed over to developers.
- Developers script and tool their deployment.
- Operations engineers and developers tweak the deployment repeatedly.
- Additional application dependencies are discovered by developers.
- Operations engineers work to install the additional requirements.
- Loop over steps 5 and 6 N more times.
- The application is deployed.



Container Based Deployment Workflow

- Developers build the Docker image and ship it to the registry.
- Operations engineers provide configuration details to the container and provision resources.
- Developers trigger deployment.



Containers Are Not Virtual Machines

- Containers are not virtual machines.
- Containers are ephemeral (temporary): they may come and go much more readily than a traditional virtual machine.
- Virtual machines are stand-in for real hardware. Because a real server is what they’re abstracting, virtual machines are often long-lived in nature.

Containers Are Lightweight

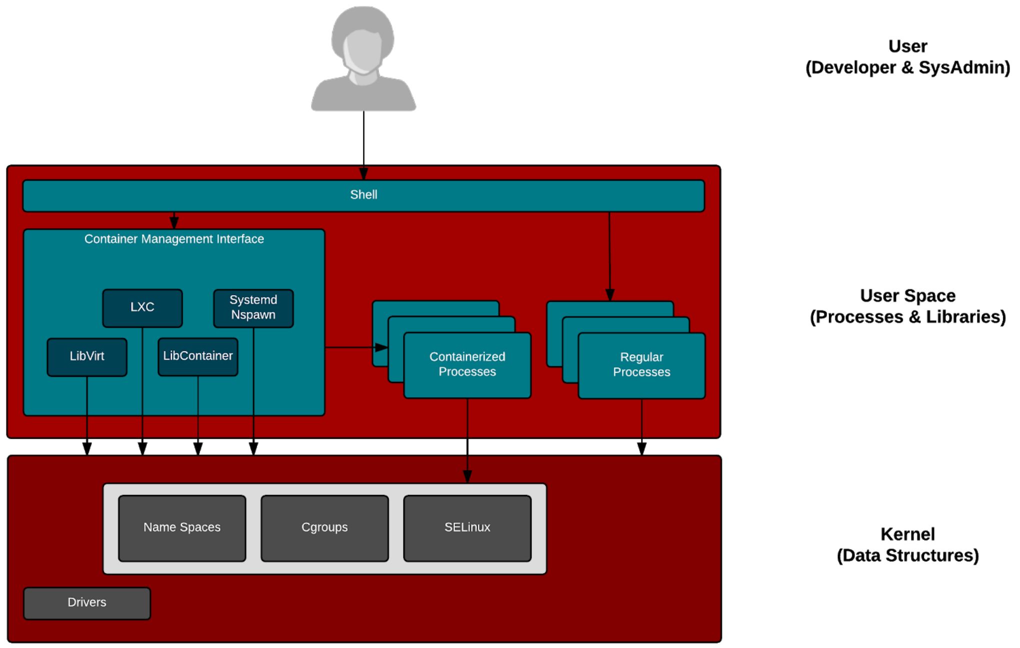
- Container are small because it is just a reference to a layered filesystem image and some metadata about the configuration.
- There is no copy of the data allocated to the container.
- Containers are just processes on the existing system, so there may not be a need to copy any data for the exclusive use of the container.

Containers are Linux

- Containers do not run ON docker.
- Containers are processes - they run on the Linux kernel.
- Containers are Linux.
- The docker daemon is one of the many user space tools/libraries that talks to the kernel to set up containers.
- Containers have been a part of the Linux operating system for more than a decade and go back even further in UNIX.
- A Linux container is nothing more than a process that runs on Linux. It shares a host kernel with other containerized processes.
- So what actually makes this process a “container”?

What makes a process a Container?

- Each containerized process is isolated from other processes running on the same Linux host, using kernel namespaces. Kernel namespaces provide a virtualized world for the container processes to run in.
- The resources consumed by each container process (memory, cpu, I/O, etc.) are confined to specified limits, using Linux control groups (or "cgroups"). This helps eliminate noisy neighbor issues, by keeping one container from over-consuming Linux host resources and starving other containers.
- This ability to both isolate containerized processes and confine the resources they consume is what enables multiple application containers to run more securely on a shared Linux host. The combination of isolation and resource confinement is what makes a Linux process a Linux container.



Control Groups (cgroups)

- Resource metering and limiting
  - memory
  - CPU
  - block I/O
  - network*
  - Device node (/dev/*) access control
- Each subsystem (memory, CPU...) has a hierarchy (tree)
- Each process belongs to exactly 1 node in each hierarchy
- Each hierarchy starts with 1 node (the root)
- Each node = group of processes (sharing the same resources)
- When a process is created, it is placed in the same groups as its parent

Namespaces

- Provide processes with their own view of the system
- cgroups limits how much you can use
- Namespaces limits what you can see (and therefore use)
- Multiple namespaces: pid, net, mnt, uts, ipc, user
- Each process is in one namespace of each type
- Processes within a PID namespace only see processes in the same PID namespace
- Each PID namespace has its own numbering (starting at 1)
- When PID 1 goes away, the whole namespace is killed

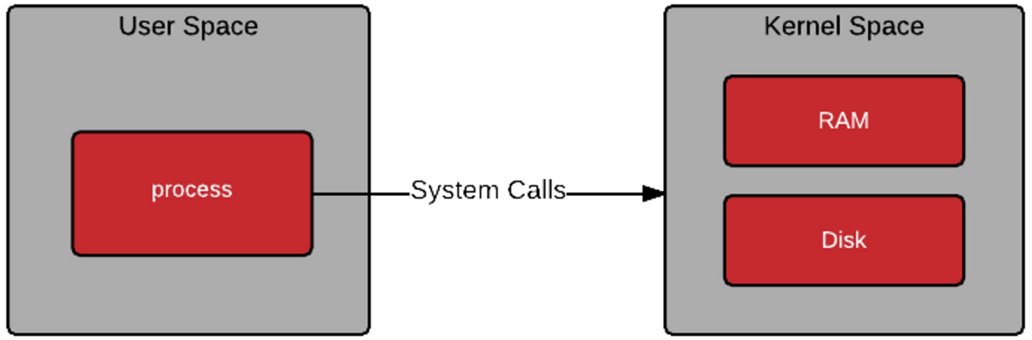
What is Docker?

- The docker container engine manages the configuration of Linux kernel namespaces, additional security features, and cgroups.
- Docker introduced a layered packaging format for content that runs inside containers.
- This made it easy for developers to run containers on their local machines and create immutable images that would run consistently across other machines and in different environments.
- **The runtime for those containers isn't docker, it's Linux.**

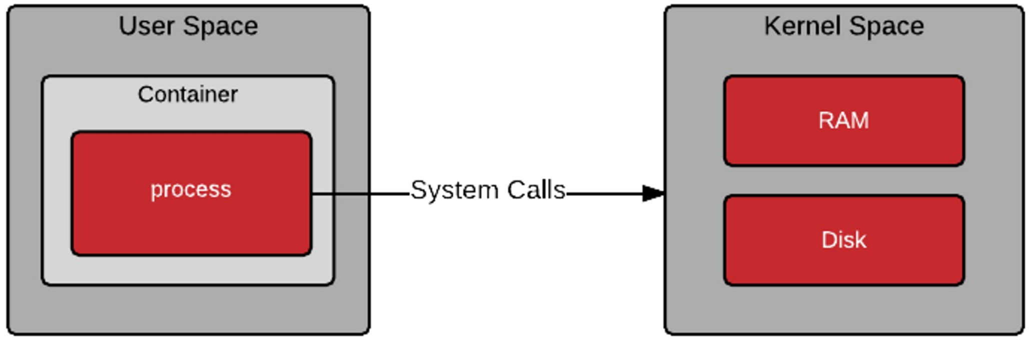
Linux Containers

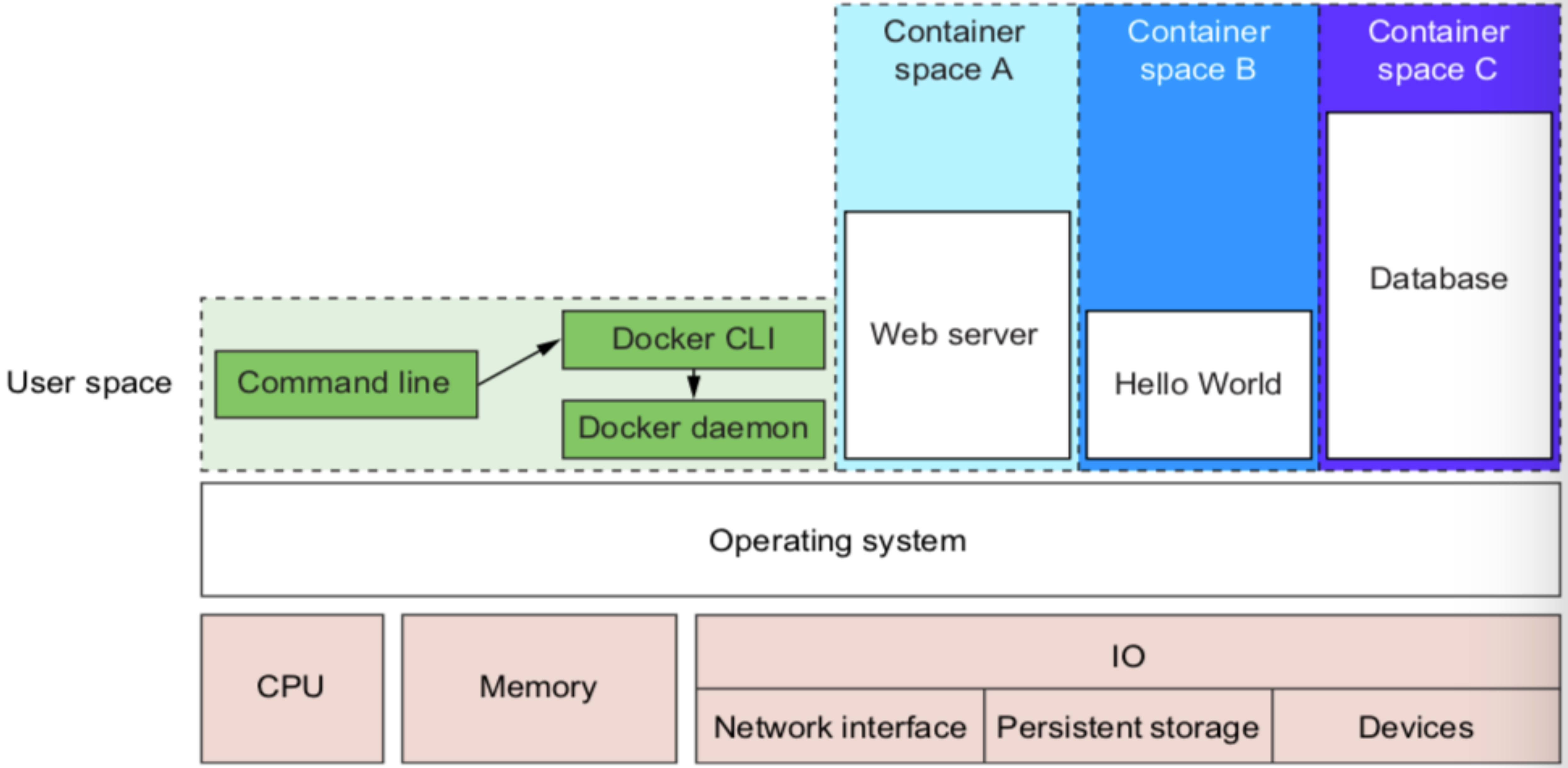
Fancy Processes

Regular Linux Process

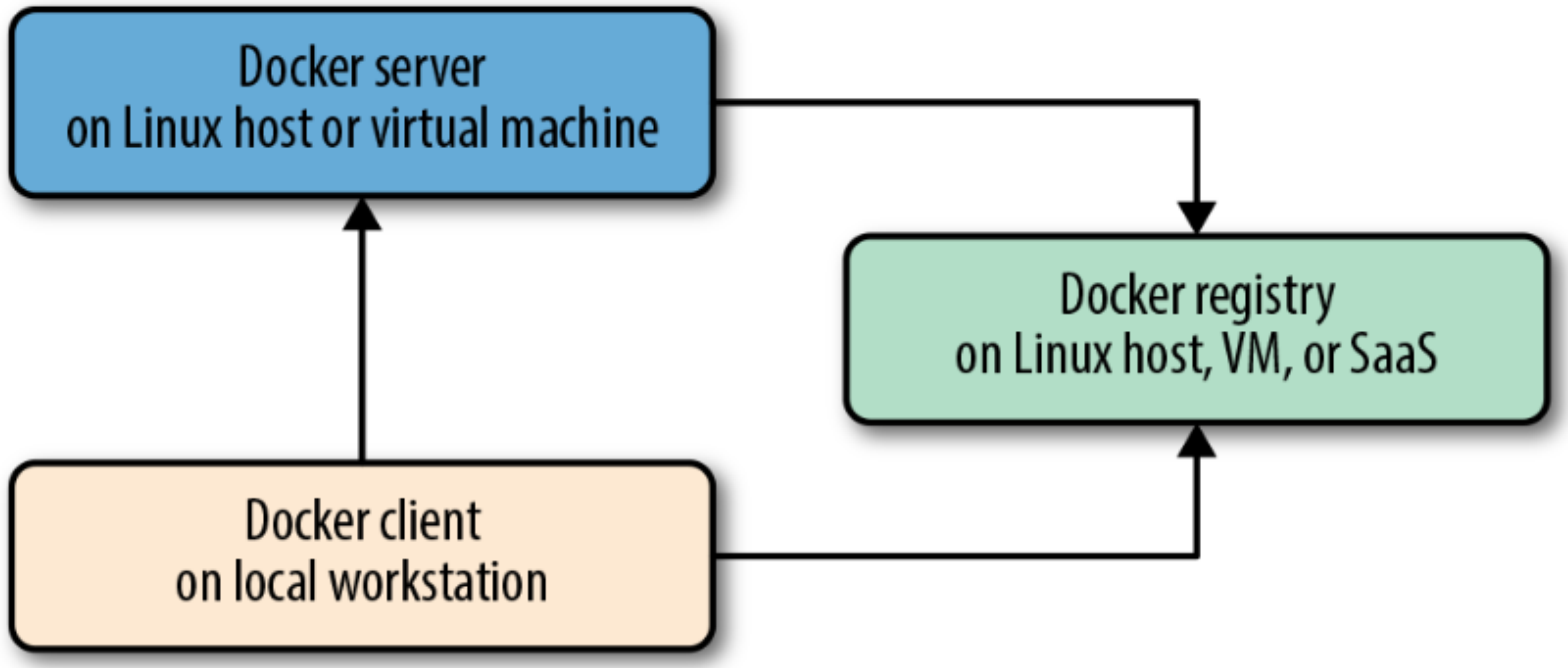


Containerized Process

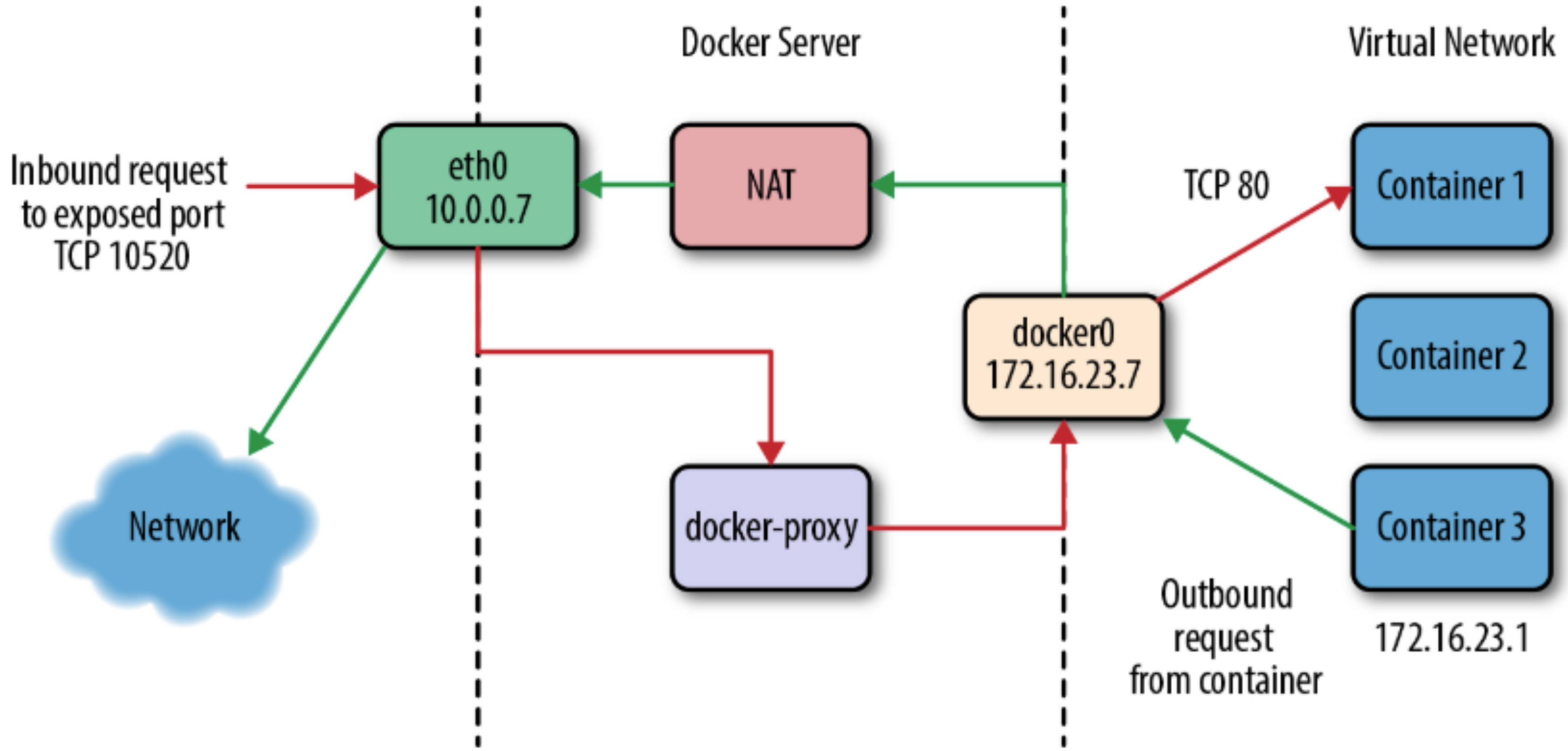



Container space A
Container space B
Container space C
Web server
Hello World
Database
User space
Command line
Docker CLI
Docker daemon
Operating system
CPU
Memory
IO
Network interface
Persistent storage
Devices

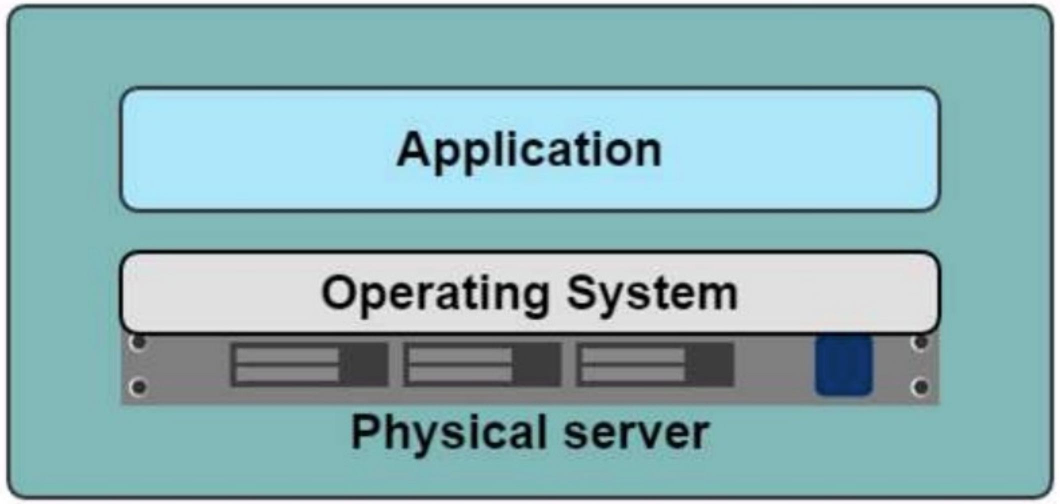
Docker Client/Server Model



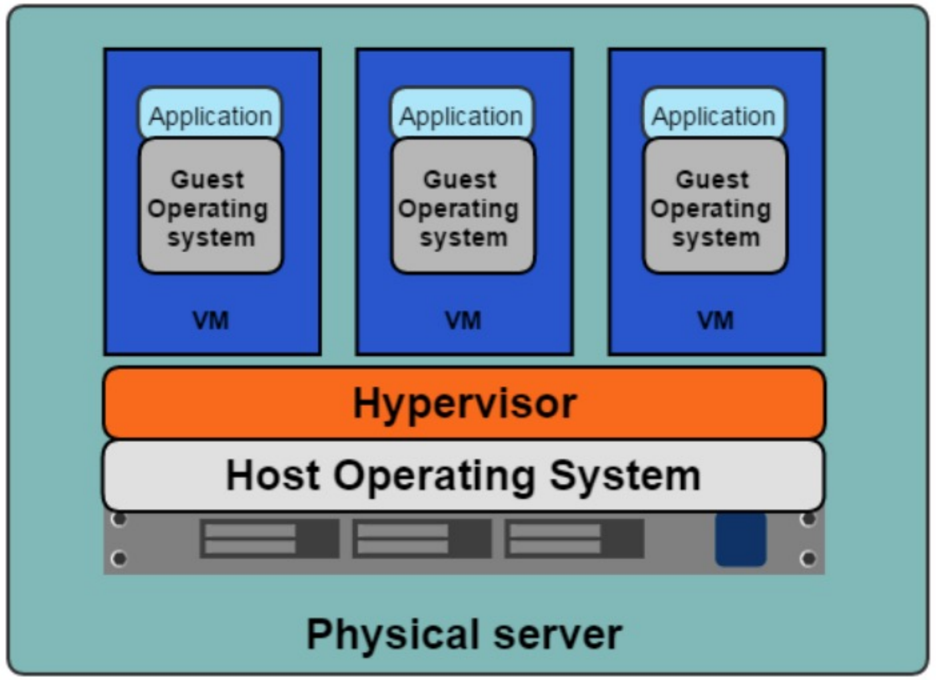
Docker & Networking



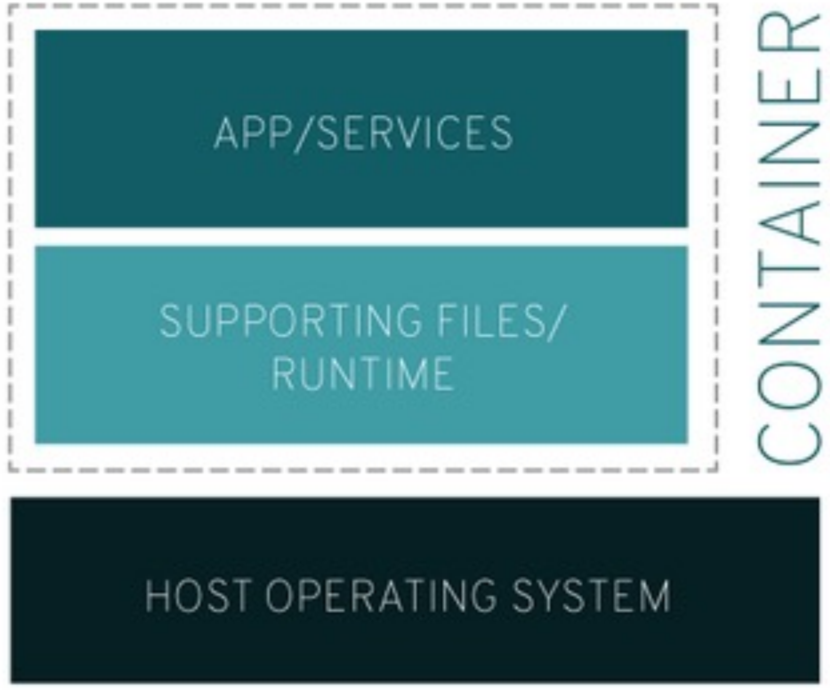
One Application on One Physical Server



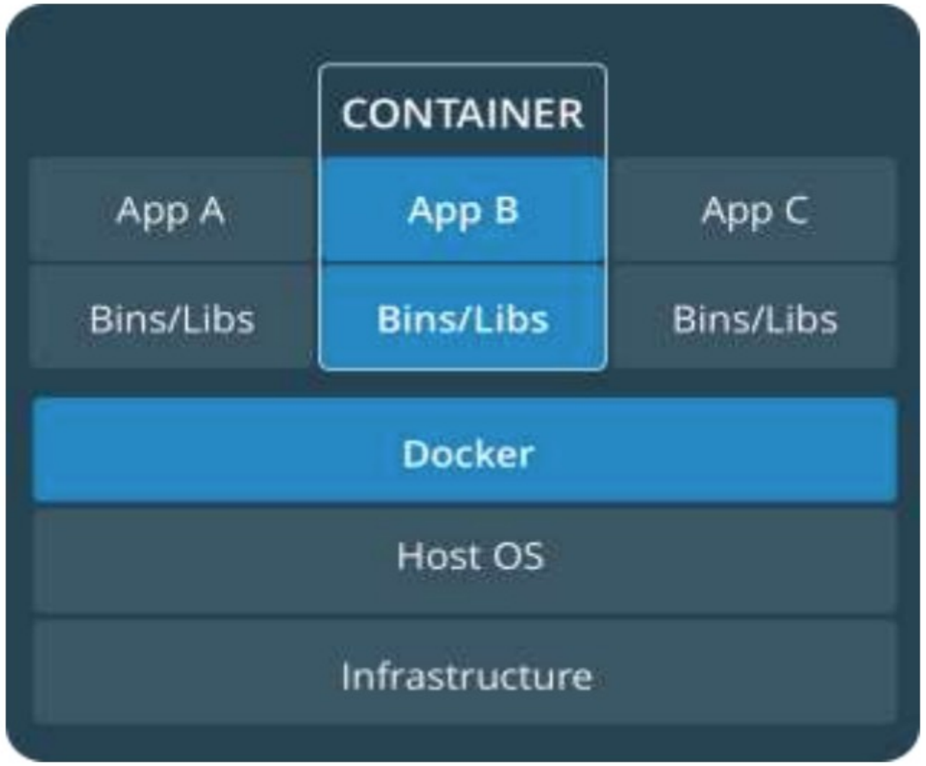
Virtual Machines

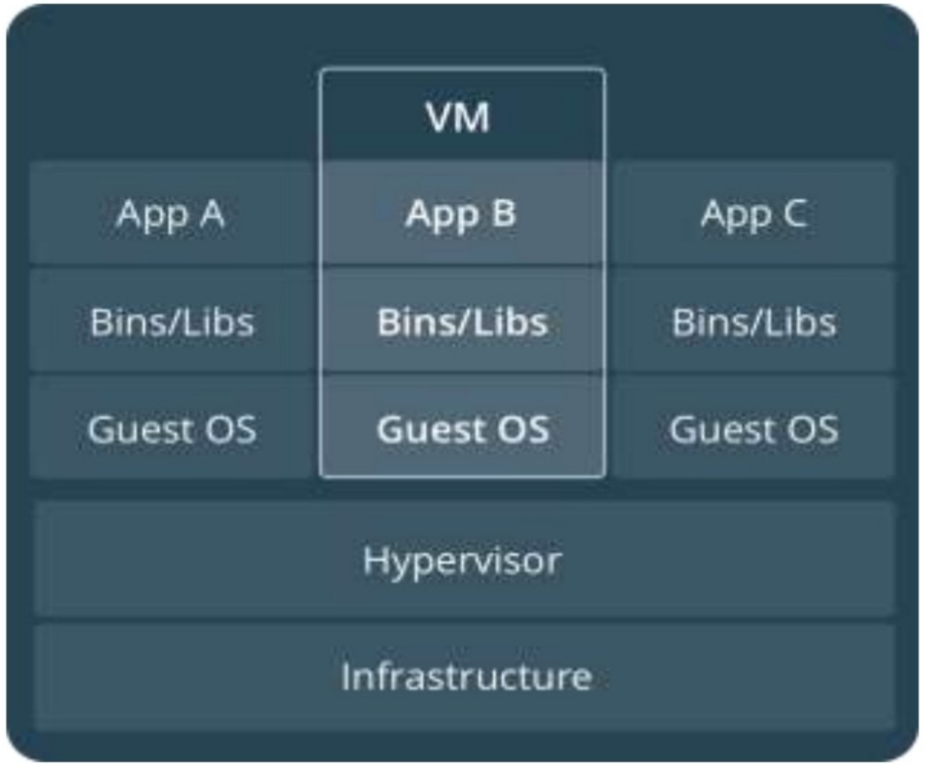


Containers


APP/SERVICES
SUPPORTING FILES/ RUNTIME
CONTAINER
HOST OPERATING SYSTEM

Comparing Containers and VMs

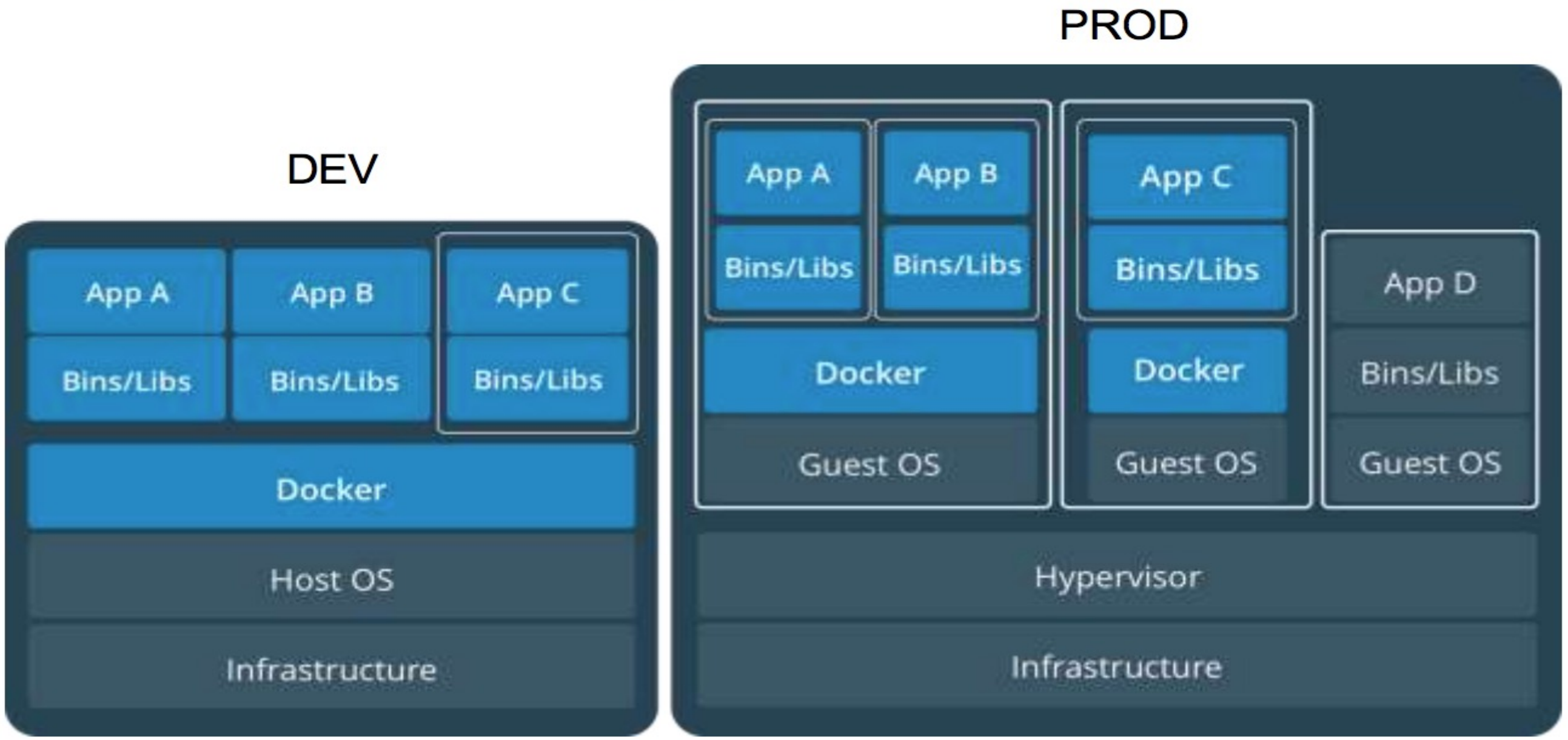




Containers are an app level construct

VMs are an infrastructure level construct to turn one machine into many servers

Containers and VMs together



Containers and VMs together provide a tremendous amount of flexibility for IT to optimally deploy and manage apps.

Docker Architecture

- Docker is a powerful technology, and that often means something that comes with a high level of complexity.
- Fundamental architecture of Docker is a simple client/server model, with only one executable that acts as both components, depending on how you invoke the *docker* command.
- There is a third component to Docker called registry which stores Docker images and metadata about those images.
- The server does the work of running and managing the containers.
- The client is used to tell the server what to do.

Docker Vocabulary

- **Docker Client** – The docker command used to control most of the Docker workflow and to talk to remote Docker servers.
- **Docker Server** – The docker command run in the daemon mode.
- **Docker Images** – Docker images consist of one or more filesystem layers and some important metadata that represent all the files required to run a Dockerized application. A single docker image can be copied to numerous hosts. A container will typically have both a name and a tag. The tag is used to identify a particular release of an image.
- **Docker Container** - A Docker container is a Linux container that has be instantiated from a Docker image. A specific container can only exist once; however, you can easily create multiple containers from the same image.

Docker/Container Image

- Every container is based on an image.
- Images are the underlying definition of a running container.
- Images provide the basis for everything that you will deploy and run with.
- Docker introduced the concept of a container image, which is a standard TAR file that combines:
  - Rootfs (container root filesystem): A directory on the system that looks like the standard root (/) of the operating system. For example, a directory with /usr, /var, /home, etc.
  - JSON file (container configuration): Specifies how to run the rootfs; for example, what command or entrypoint to run in the rootfs when the container starts; environment variables to set for the container; the container's working directory; and a few other settings.
- Docker "tar's up" the rootfs and the JSON file to create the base image. This enables you to install additional content on the rootfs, create a new JSON file, and tar the difference between the original image and the new image with the updated JSON file. This creates a layered imag
- The definition of a container image was eventually standardized by the Open Container Initiative (OCI) standards body as the OCI Image Specification.

Dockerfile & Layers

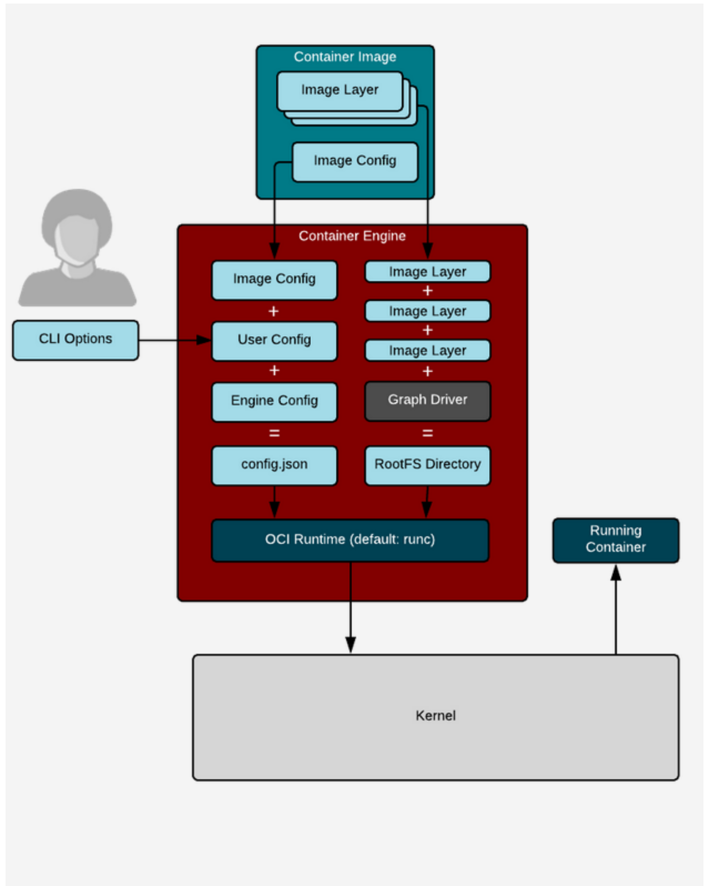
- Each line in a Dockerfile creates a new image layer that is stored by Docker.
- Layer contains all the changes that are a result of that command being issued.
- This means that when you build new images, Docker will only need to build layers that deviate from previous builds: you can reuse all the layers that haven’t changed.

Container Registry

- Docker took these **container images** (tarballs) and moved them to a web service from which they could be pulled, developed a protocol to pull them, and called the web service a **container registry**.

Container Engines

- Container engines are programs that can pull container images from container registries and reassemble them onto container storage.
- Container engines also launch container runtimes.



Linux container internals. Illustration by Scott McCarty. CC BY-SA 4.0

Container Storage

- Container storage is usually a copy-on-write (COW) layered filesystem. When you pull down a container image from a container registry, you first need to untar the rootfs and place it on disk.
- If you have multiple layers that make up your image, each layer is downloaded and stored on a different layer on the COW filesystem.
- The COW filesystem allows each layer to be stored separately, which maximizes sharing for layered images.

Dockerfile

- **Dockerfile** describes all the steps that are required to create an image and would usually be contained within the root directory of the source code repository for your application.

```
FROM node:0.10

LABEL "maintainer"="anna@example.com"
LABEL "rating"="Five Stars" "class"="First Class"

USER root

ENV AP /data/app
ENV SCPATH /etc/supervisor/conf.d

RUN apt-get -y update

# The daemons
RUN apt-get -y install supervisor
RUN mkdir -p /var/log/supervisor

# Supervisor Configuration
ADD ./supervisord/conf.d/* $SCPATH/

# Application Code
ADD *.js* $AP/

WORKDIR $AP

RUN npm install

CMD ["supervisord", "-n"]
```

Anatomy of Dockerfile

Order of Commands in Dockerfile

- **The order of commands in a Dockerfile can have a very significant impact on ongoing build times**.
- You should try to order commands so that things that change between every single build are closer to the bottom.
- This means that adding your code and similar steps should be held off until the end.
- When you rebuild an image, every single layer after the first introduced change will need to be rebuilt.

FROM

- The **FROM** instruction initializes a new build stage and sets the Base Image for subsequent instructions.
- A valid Dockerfile must start with a FROM instruction.

LABEL

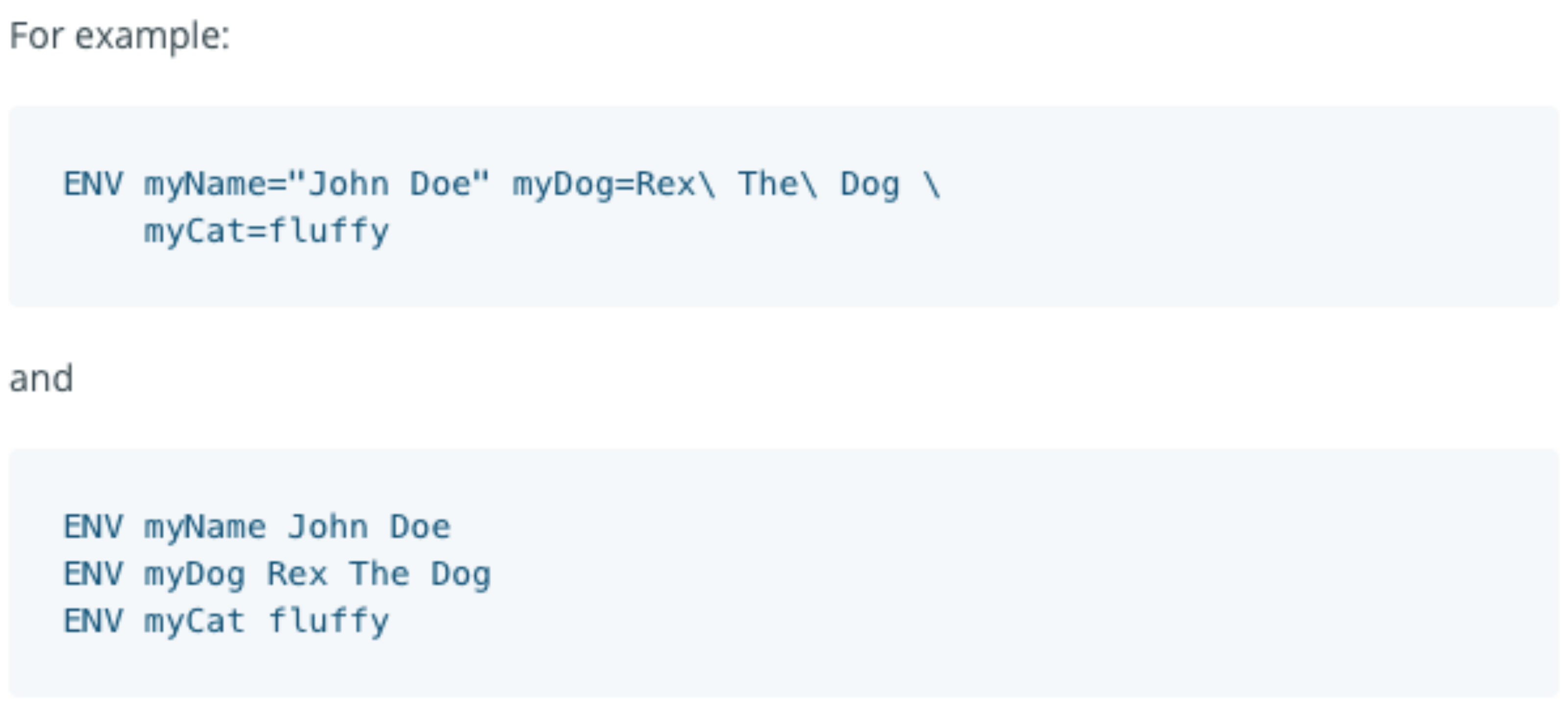
- The **LABEL** instruction adds metadata to an image.
- A LABEL is a key-value pair.
- To include spaces within a LABEL value, use quotes and backslashes as you would in command-line parsing.

```
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."
```

ENV

- The **ENV** instruction sets the environment variable <key> to the value <value>. This value will be in the environment for all subsequent instructions in the build stage and can be replaced inline in many as well.
- The ENV instruction has two forms.
  - The first form, **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 whitespace characters. The value will be interpreted for other environment variables, so quote characters will be removed if they are not escaped.
  - The second form, **ENV <key>=<value> ...,** allows for multiple variables to be set at one time. Notice that the second form uses the equals sign (=) in the syntax, while the first form does not. Like command line parsing, quotes and backslashes can be used to include spaces within values.

ENV (example)


For example:

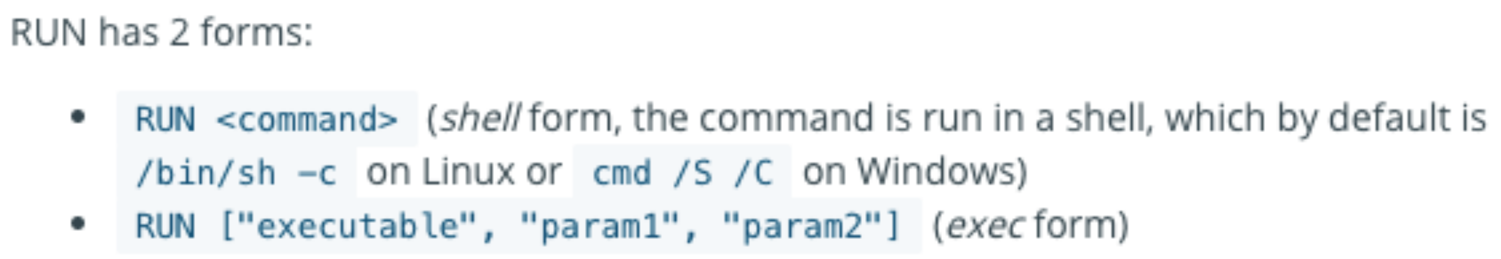
```
ENV myName="John Doe" myDog=Rex\ The\ Dog \
    myCat=fluffy
```

and

```
ENV myName John Doe
ENV myDog Rex The Dog
ENV myCat fluffy
```

RUN

- 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.

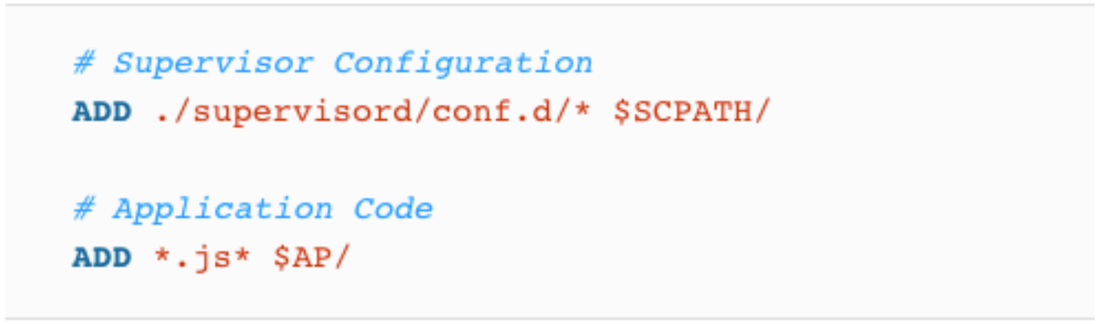


RUN has 2 forms:

- `RUN <command>` (*shell* form, the command is run in a shell, which by default is `/bin/sh -c` on Linux or `cmd /S /C` on Windows)
- `RUN ["executable", "param1", "param2"]` (*exec* form)

ADD

- The **ADD** instruction copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>.
- Multiple <src> resources may be specified but if they are files or directories, their paths are interpreted as relative to the source of the context of the build.
- Each <src> may contain wildcards.

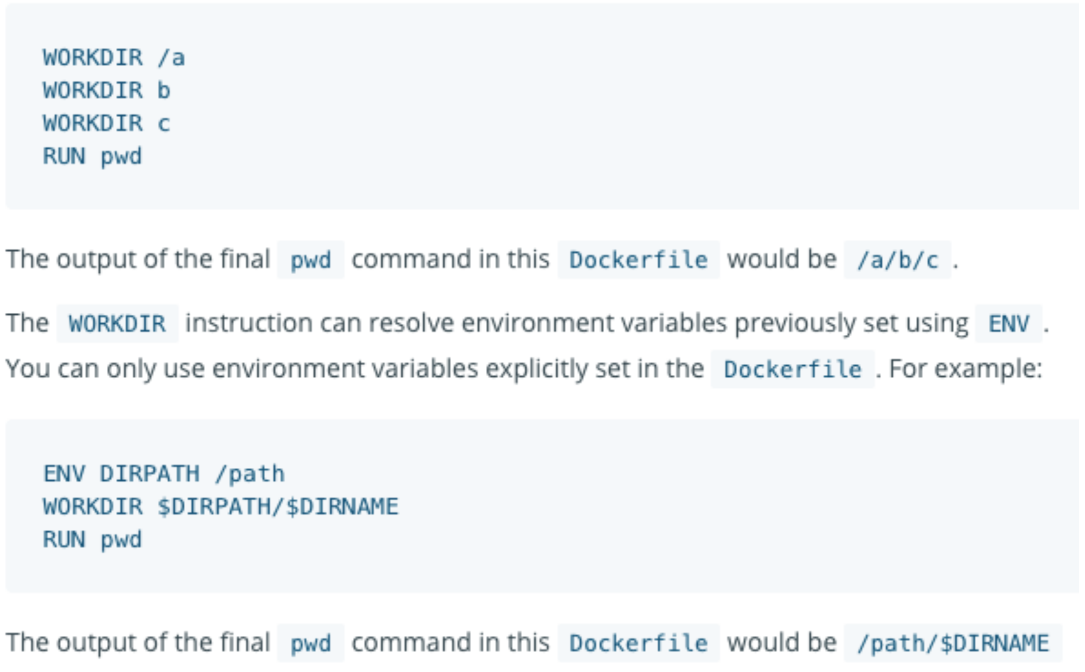


```
# Supervisor Configuration
ADD ./supervisord/conf.d/* $SCPATH/

# Application Code
ADD *.js* $AP/
```

WORKDIR

- The **WORKDIR** instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile.
- If the WORKDIR doesn’t exist, it will be created even if it’s not used in any subsequent Dockerfile instruction.
- The WORKDIR instruction can be used multiple times in a Dockerfile. If a relative path is provided, it will be relative to the path of the previous WORKDIR instruction.
- The WORKDIR instruction can resolve environment variables previously set using ENV. You can only use environment variables explicitly set in the Dockerfile.



```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

The output of the final `pwd` command in this `Dockerfile` would be `/a/b/c`.

The `WORKDIR` instruction can resolve environment variables previously set using `ENV`. You can only use environment variables explicitly set in the `Dockerfile`. For example:

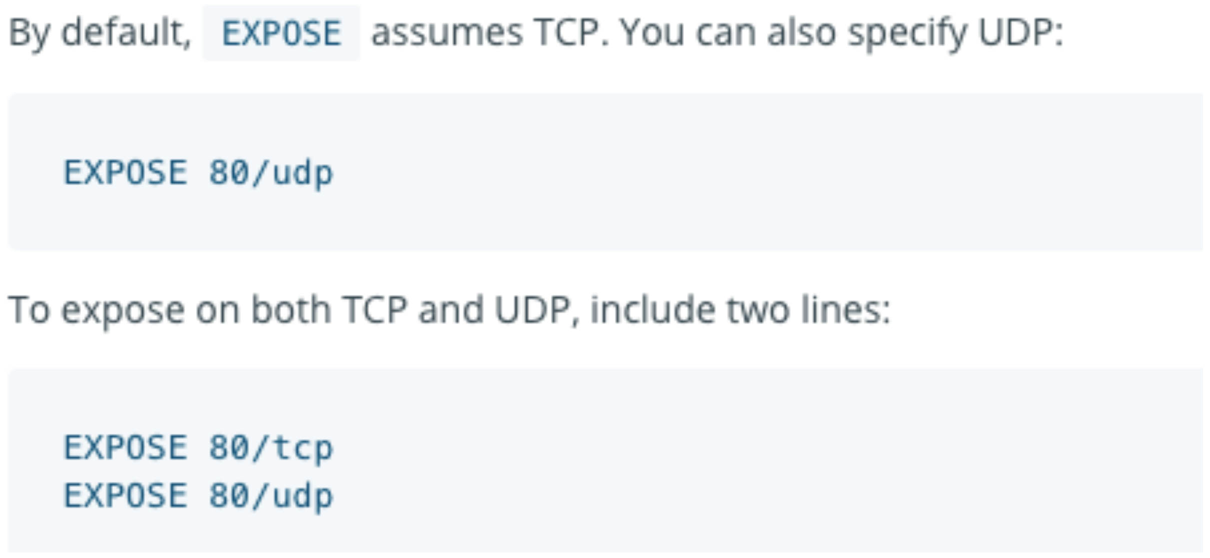
```
ENV DIRPATH /path
WORKDIR $DIRPATH/$DIRNAME
RUN pwd
```

The output of the final `pwd` command in this `Dockerfile` would be `/path/$DIRNAME`

EXPOSE

- The **EXPOSE** instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified.
- The EXPOSE instruction does not actually publish the port. It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published.
- To publish the port when running the container, use the -p flag on docker run to publish and map one or more ports, or the -P flag to publish all exposed ports and map them to high-order ports.

EXPOSE (example)


By default, `EXPOSE` assumes TCP. You can also specify UDP:

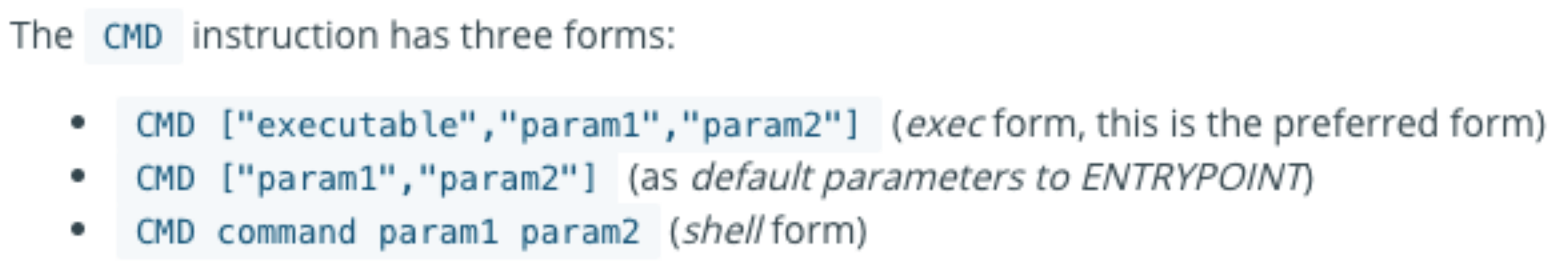
```
EXPOSE 80/udp
```

To expose on both TCP and UDP, include two lines:

```
EXPOSE 80/tcp
EXPOSE 80/udp
```

CMD

- The main purpose of a CMD is to provide defaults for an executing container.
- 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.

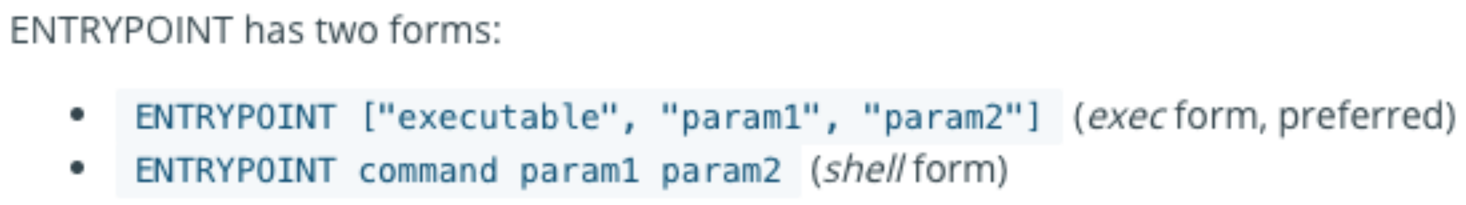


The `CMD` instruction has three forms:

- `CMD ["executable","param1","param2"]` (*exec* form, this is the preferred form)
- `CMD ["param1","param2"]` (as *default parameters to ENTRYPOINT*)
- `CMD command param1 param2` (*shell* form)

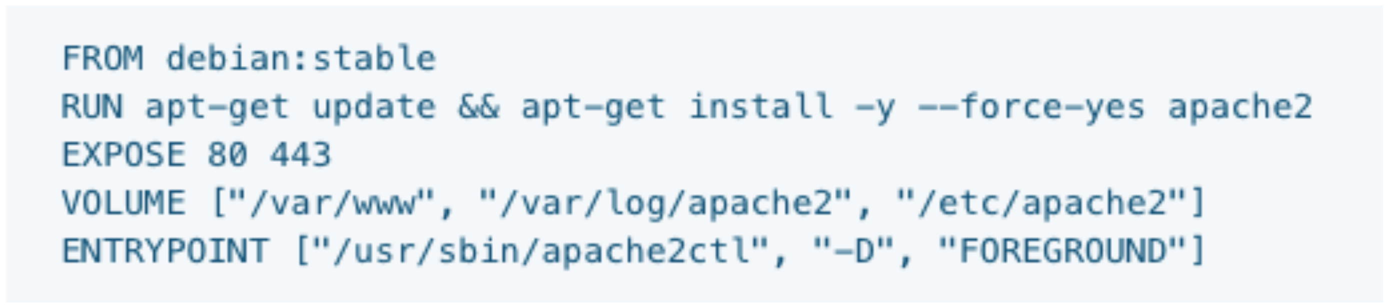
ENTRYPOINT

- An **ENTRYPOINT** allows you to configure a container that will run as an executable.



ENTRYPOINT has two forms:

- `ENTRYPOINT ["executable", "param1", "param2"]` (*exec* form, preferred)
- `ENTRYPOINT command param1 param2` (*shell* form)



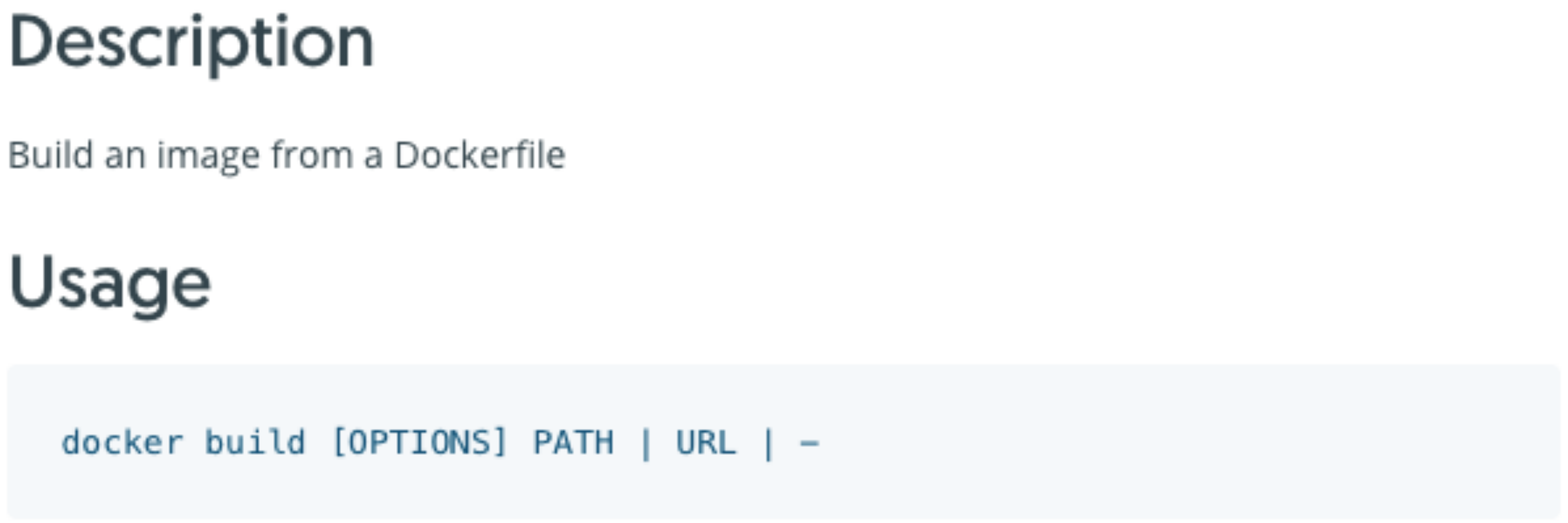
```
FROM debian:stable
RUN apt-get update && apt-get install -y --force-yes apache2
EXPOSE 80 443
VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Understand how CMD and ENTRYPOINT interact

- Both **CMD** and **ENTRYPOINT** instructions define what command gets executed when running a container. There are few rules that describe their co-operation.
  - Dockerfile should specify at least one of CMD or ENTRYPOINT commands.
  - ENTRYPOINT should be defined when using the container as an executable.
  - CMD should be used as a way of defining default arguments for an ENTRYPOINT command or for executing an ad-hoc command in a container.
  - CMD will be overridden when running the container with alternative arguments.

Docker Command Line Interface

Building Docker Images



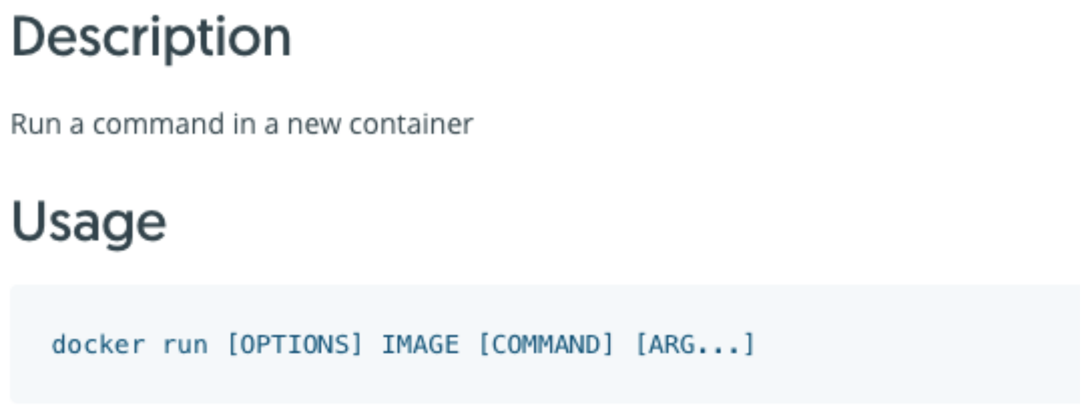
Description

Build an image from a Dockerfile

Usage

```
docker build [OPTIONS] PATH | URL | -
```

Running Docker Container



Description

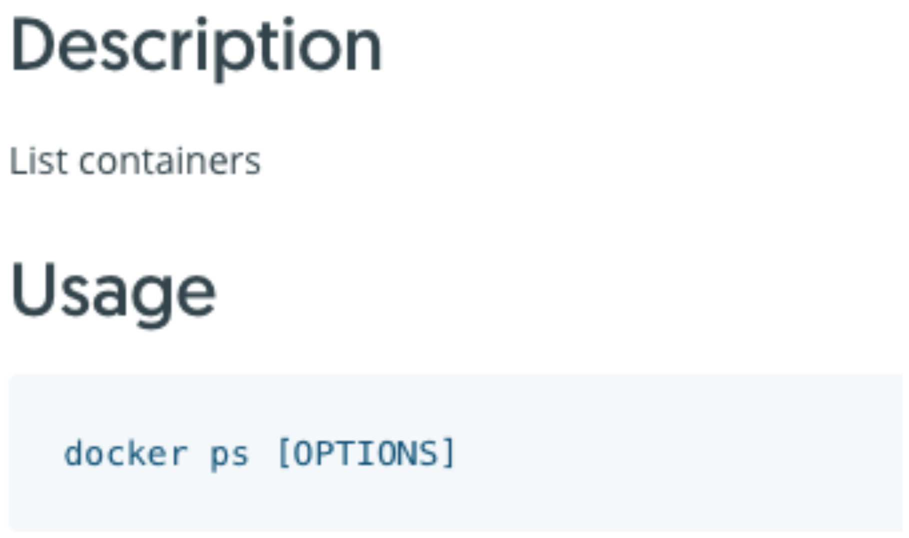
Run a command in a new container

Usage

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

The docker run command first creates a writeable container layer over the specified image, and then starts it using the specified command.

List Docker Containers

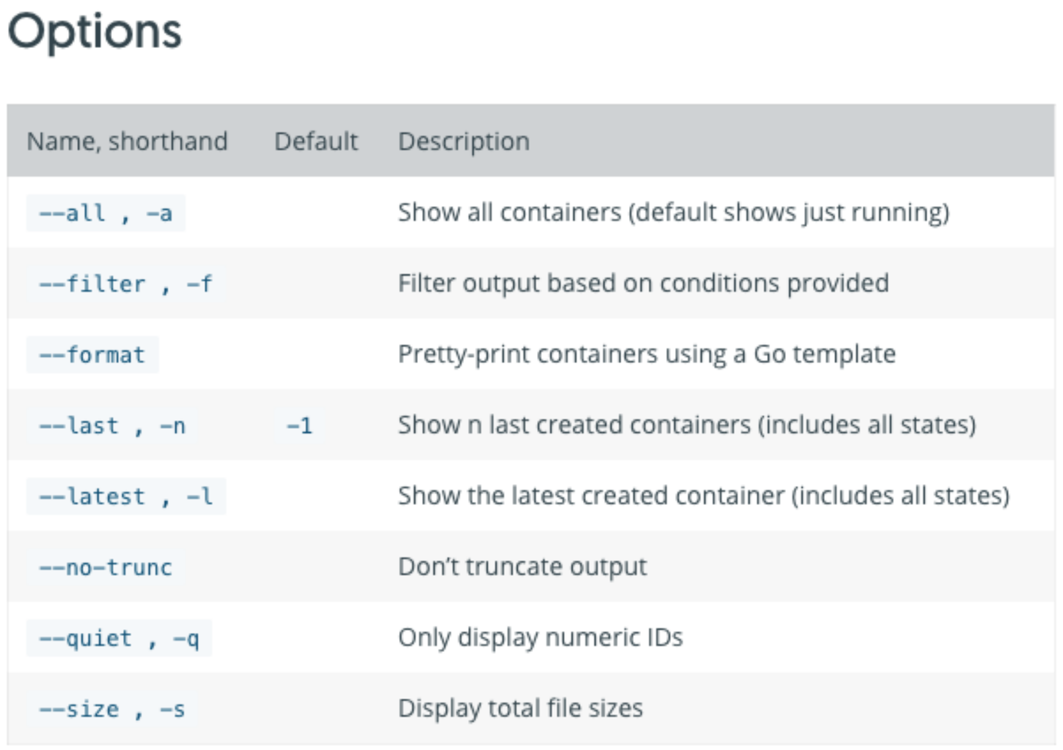


Description

List containers

Usage

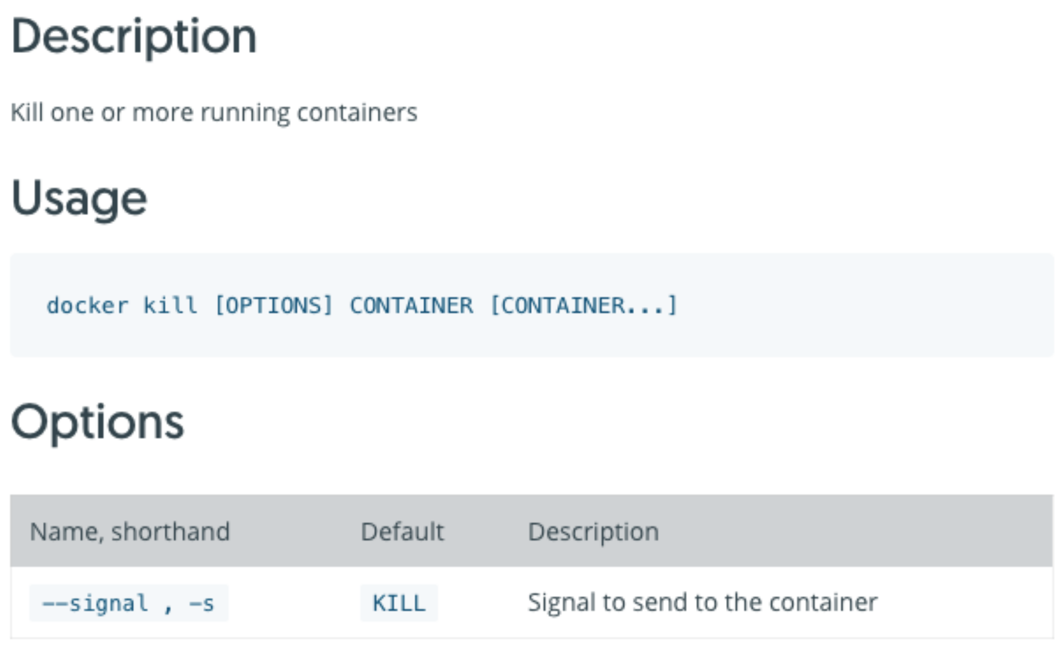
```
docker ps [OPTIONS]
```



Options

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

Kill Running Containers



Description

Kill one or more running containers

Usage

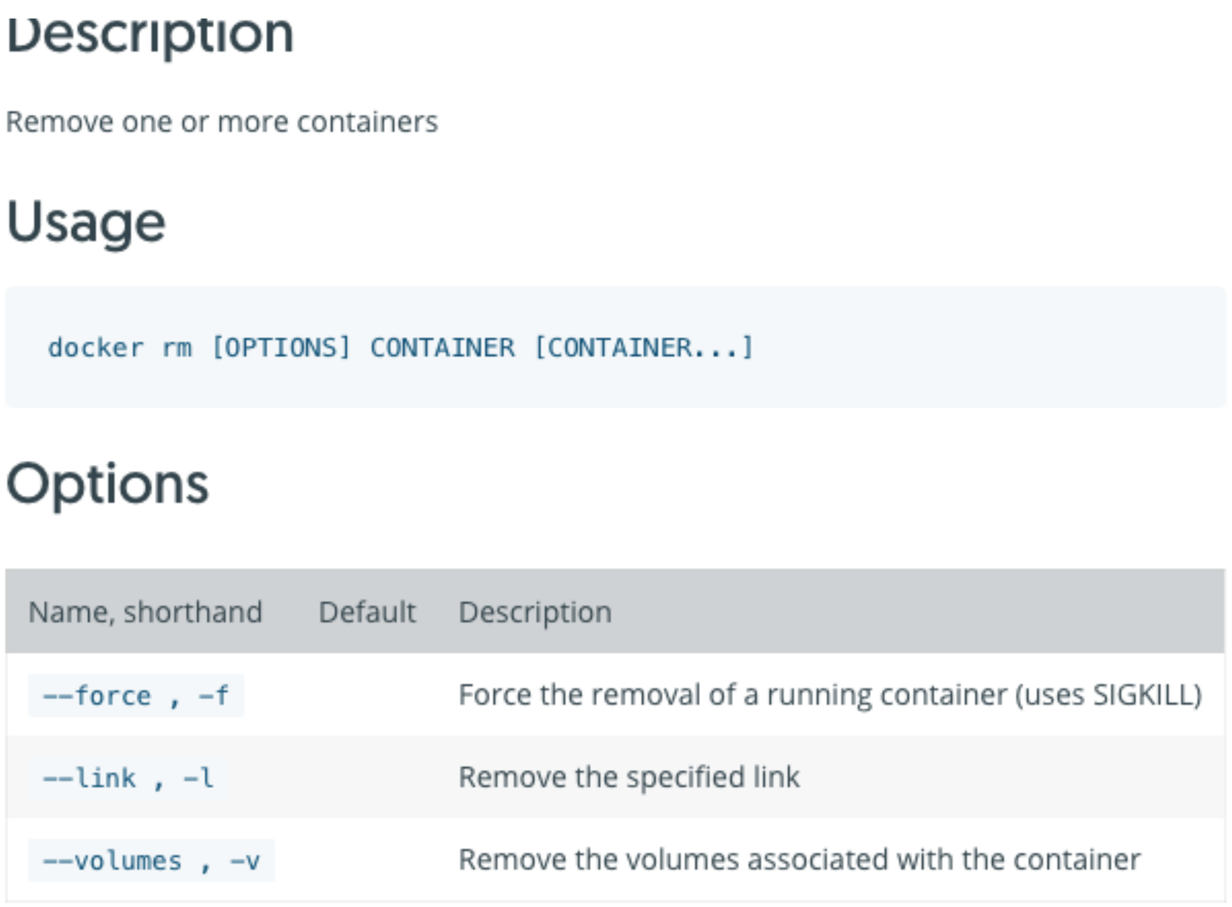
```
docker kill [OPTIONS] CONTAINER [CONTAINER...]
```

Options

Name, shorthand	Default	Description
`--signal , -s`	`KILL`	Signal to send to the container

- The docker kill subcommand kills one or more containers.
- The main process inside the container is sent SIGKILL signal (default), or the signal that is specified with the --signal option.
- You can kill a container using the container's ID, ID-prefix, or name.

Remove Containers



Description

Remove one or more containers

Usage

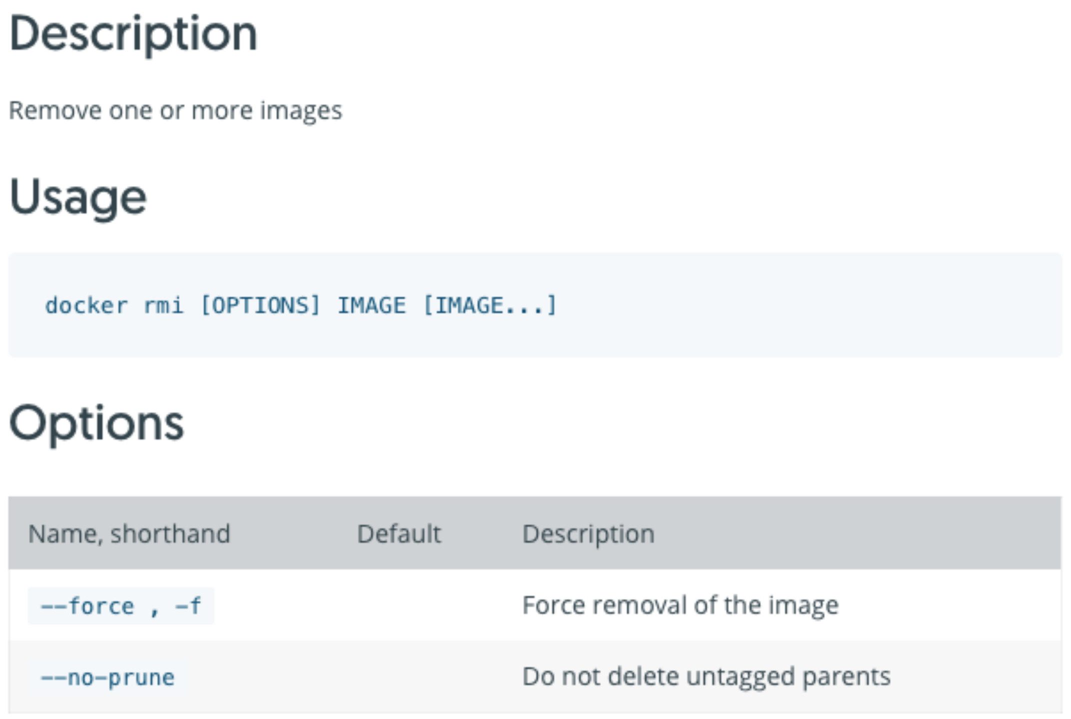
```
docker rm [OPTIONS] CONTAINER [CONTAINER...]
```

Options

Name, shorthand	Default	Description
`--force , -f`		Force the removal of a running container (uses SIGKILL)
`--link , -l`		Remove the specified link
`--volumes , -v`		Remove the volumes associated with the container

Remove Container Images

- Removes (and un-tags) one or more images from the host node.
- If an image has multiple tags, using this command with the tag as a parameter only removes the tag.
- If the tag is the only one for the image, both the image and the tag are removed.
- This does not remove images from a registry.
- You cannot remove an image of a running container unless you use the -f option.



Description

Remove one or more images

Usage

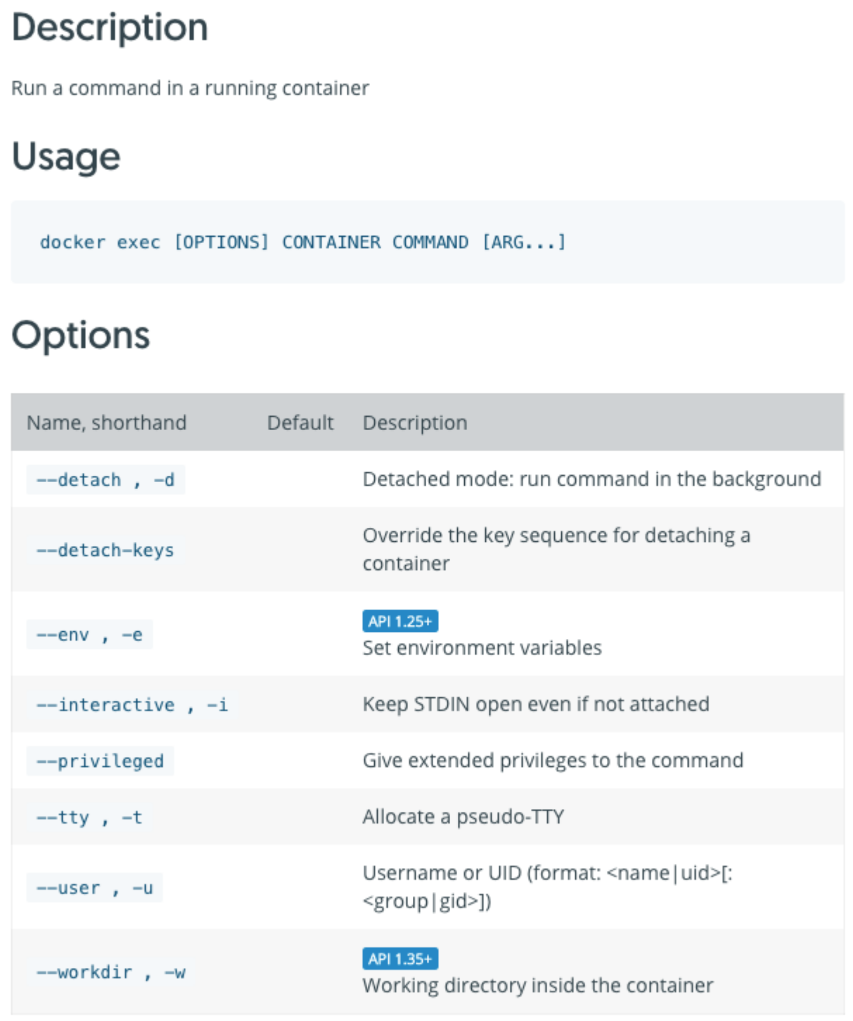
```
docker rmi [OPTIONS] IMAGE [IMAGE...]
```

Options

Name, shorthand	Default	Description
`--force , -f`		Force removal of the image
`--no-prune`		Do not delete untagged parents

Run Command In Running Container

- The docker **exec** command runs a new command in a running container.
- COMMAND will run in the default directory of the container.
- COMMAND should be an executable, a chained or a quoted command will not work.



Description

Run a command in a running container

Usage

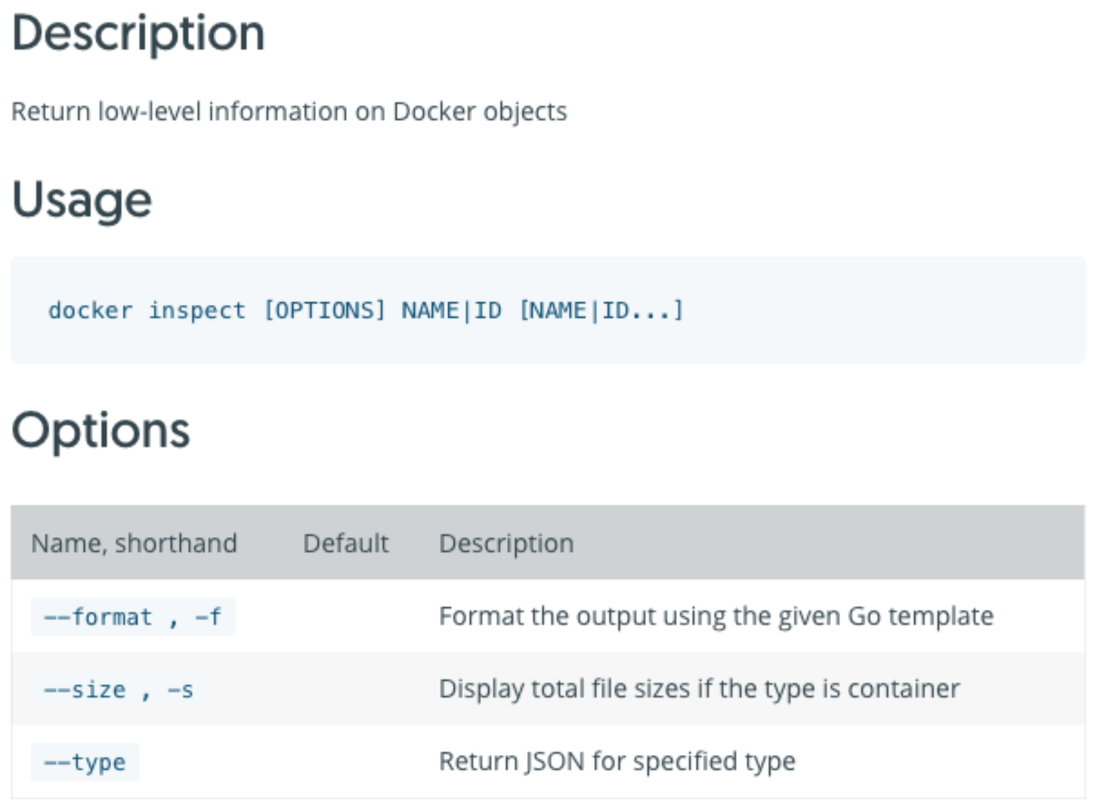
```
docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

Options

Name, shorthand	Default	Description
`--detach , -d`		Detached mode: run command in the background
`--detach-keys`		Override the key sequence for detaching a container
`--env , -e`		API 1.25+ Set environment variables
`--interactive , -i`		Keep STDIN open even if not attached
`--privileged`		Give extended privileges to the command
`--tty , -t`		Allocate a pseudo-TTY
`--user , -u`		Username or UID (format: <name|uid>[:<group|gid>])
`--workdir , -w`		API 1.35+ Working directory inside the container

Inspect Container Images

- Docker inspect provides detailed information on constructs controlled by Docker.
- By default, docker inspect will render results in a JSON array.



Description

Return low-level information on Docker objects

Usage

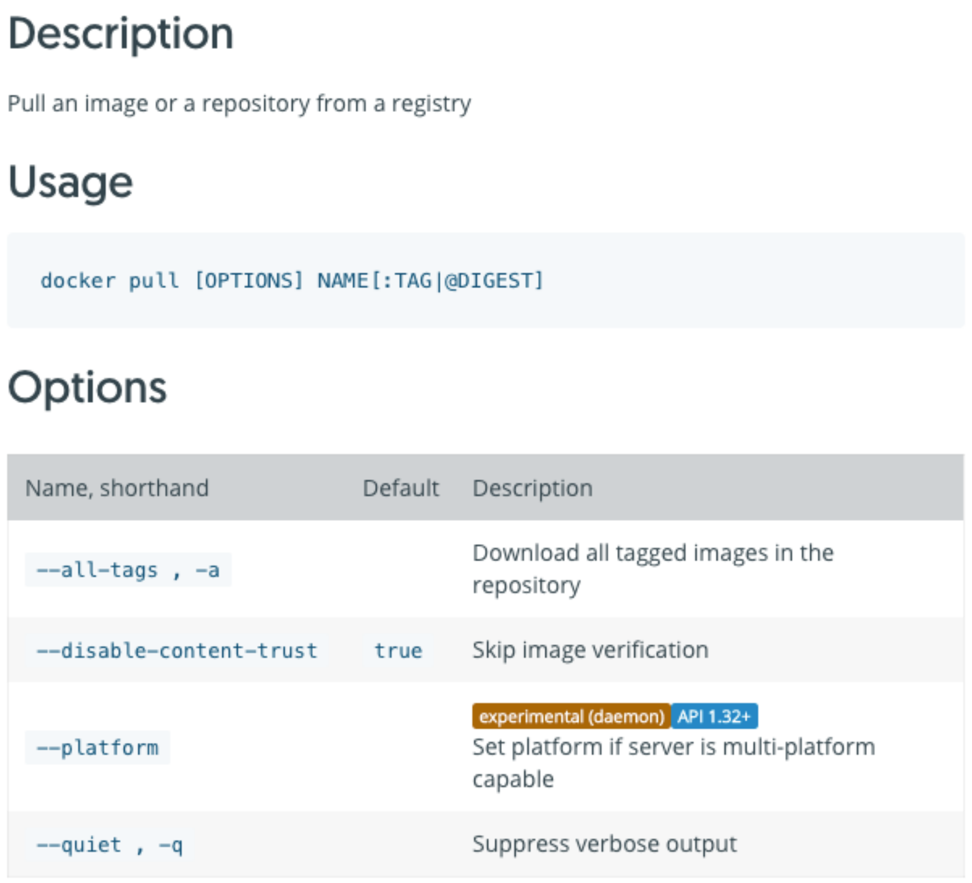
```
docker inspect [OPTIONS] NAME|ID [NAME|ID...]
```

Options

Name, shorthand	Default	Description
`--format , -f`		Format the output using the given Go template
`--size , -s`		Display total file sizes if the type is container
`--type`		Return JSON for specified type

Pull Container Image From Registry

- Most of your images will be created on top of a base image from the container registry.
- Container registry such as [Docker Hub]() contains many pre-built images that you can pull and try without needing to define and configure your own.
- To download a particular image, or set of images (i.e., a repository), use docker pull.



Description

Pull an image or a repository from a registry

Usage

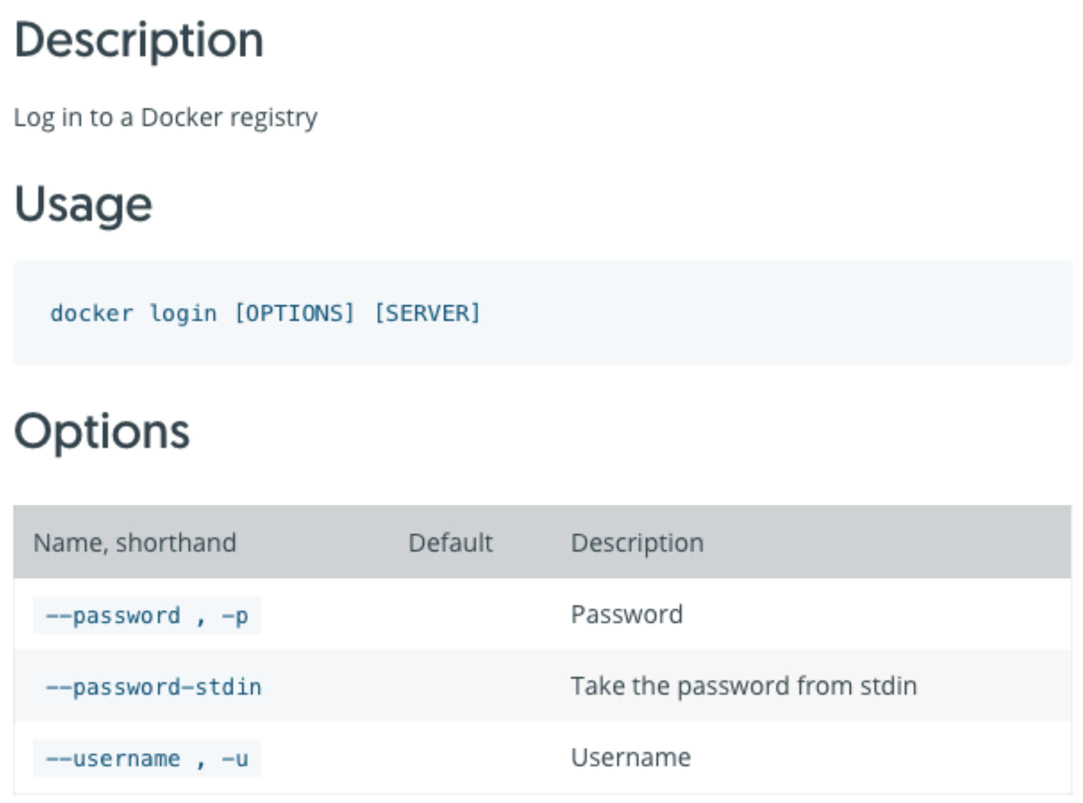
```
docker pull [OPTIONS] NAME[:TAG|@DIGEST]
```

Options

Name, shorthand	Default	Description
`--all-tags , -a`		Download all tagged images in the repository
`--disable-content-trust`	`true`	Skip image verification
`--platform`		experimental (daemon) API 1.32+ Set platform if server is multi-platform capable
`--quiet , -q`		Suppress verbose output

Login to Container Registry

Login to a registry.



Description

Log in to a Docker registry

Usage

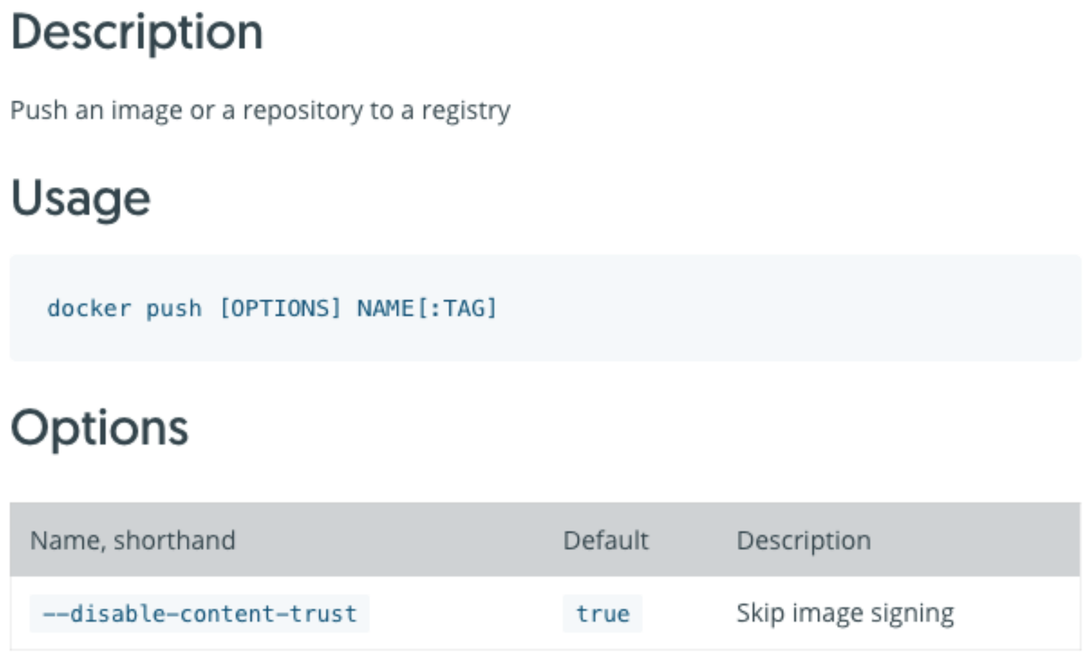
```
docker login [OPTIONS] [SERVER]
```

Options

Name, shorthand	Default	Description
`--password , -p`		Password
`--password-stdin`		Take the password from stdin
`--username , -u`		Username

Push Image to Registry

- Use docker push to share your images to the container registry.
- Killing the docker push process, for example by pressing CTRL-c while it is running in a terminal, terminates the push operation.



Description

Push an image or a repository to a registry

Usage

```
docker push [OPTIONS] NAME[:TAG]
```

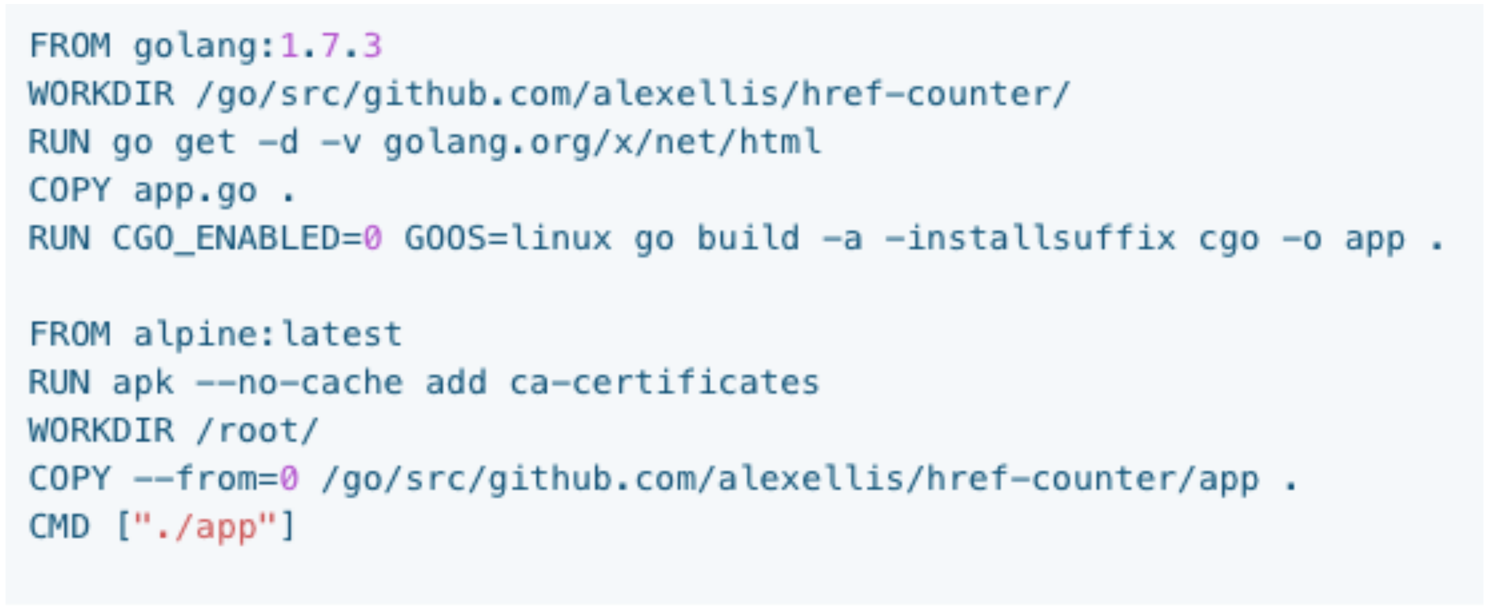
Options

Name, shorthand	Default	Description
`--disable-content-trust`	`true`	Skip image signing

Multistage Builds

- One of the most challenging things about building images is keeping the image size down.
- Each instruction in the Dockerfile adds a layer to the image, and you need to remember to clean up any artifacts you don’t need before moving on to the next layer.
- To write an efficient Dockerfile, you have traditionally needed to employ shell tricks and other logic to keep the layers as small as possible and to ensure that each layer has the artifacts it needs from the previous layer and nothing else.

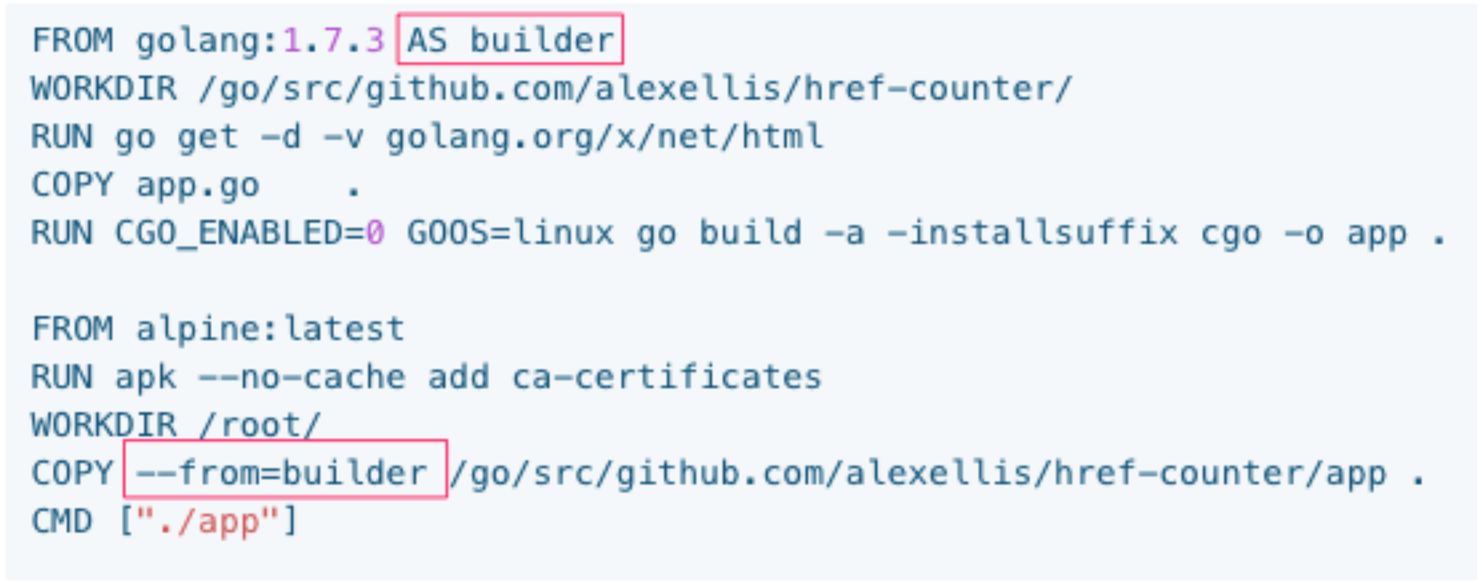
Multistage Builds



```
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=0 /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

Name Build Stages

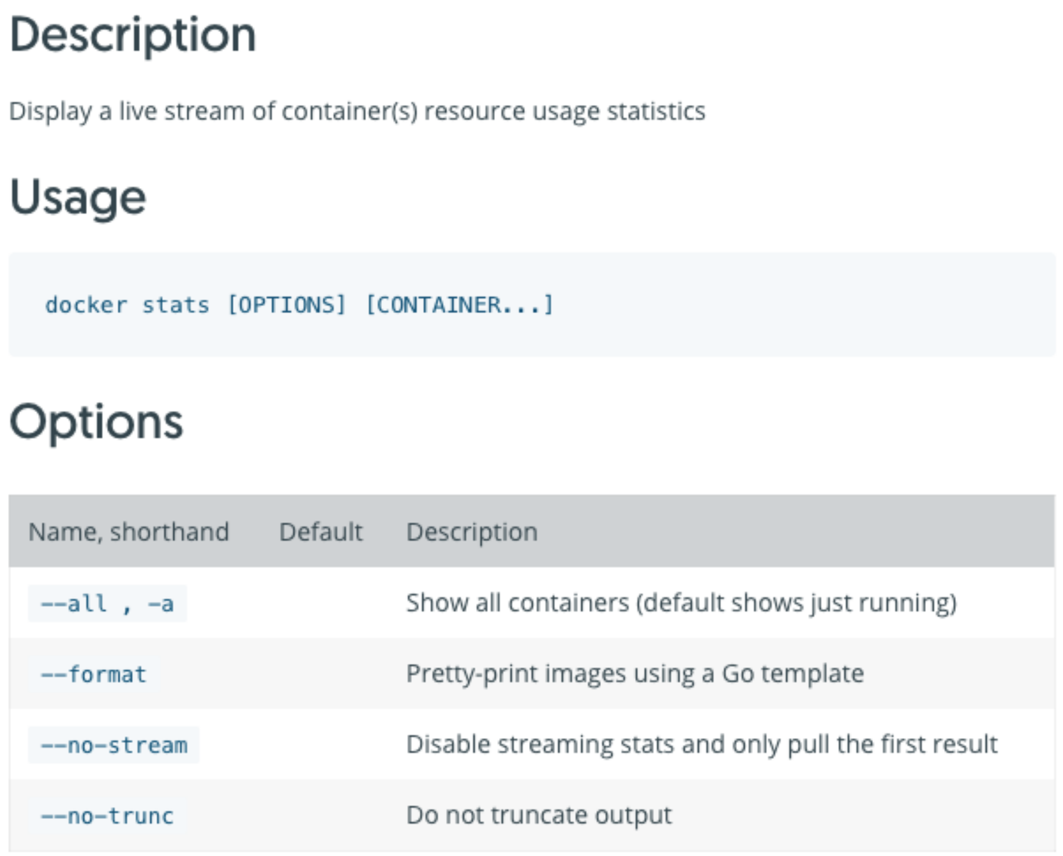


```
FROM golang:1.7.3 AS builder
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go    .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

Docker Stats

- The docker stats command returns a live data stream for running containers.
- To limit data to one or more specific containers, specify a list of container names or ids separated by a space.
- You can specify a stopped container but stopped containers do not return any data.



Description

Display a live stream of container(s) resource usage statistics

Usage

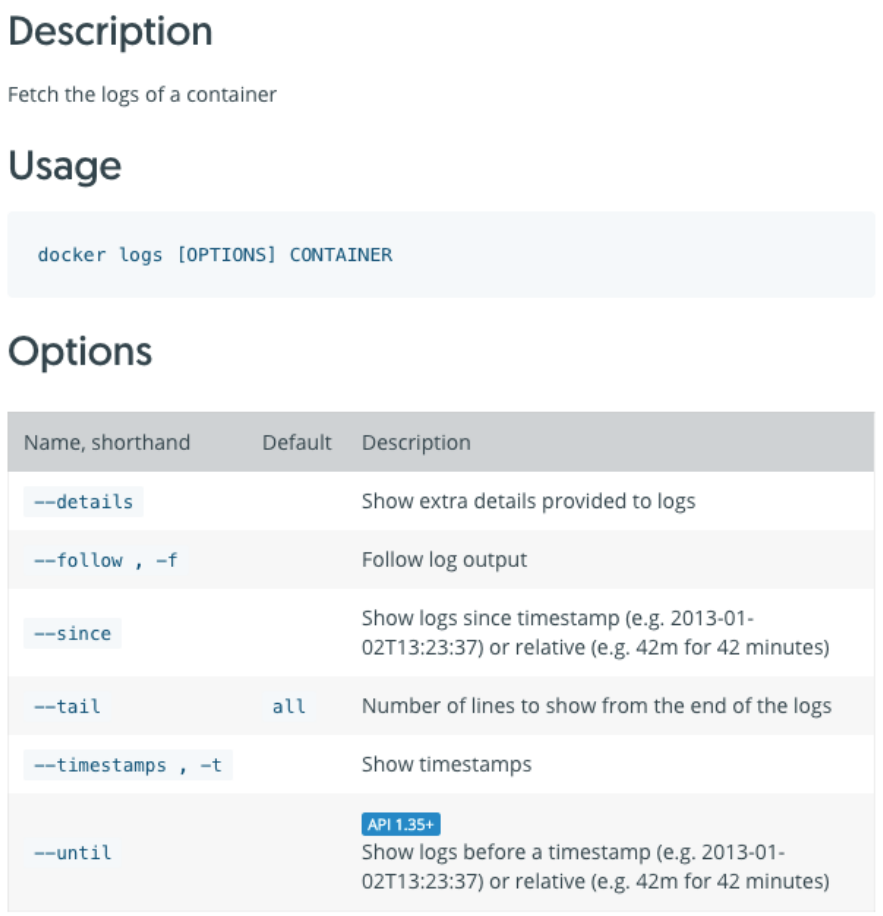
```
docker stats [OPTIONS] [CONTAINER...]
```

Options

Name, shorthand	Default	Description
`--all , -a`		Show all containers (default shows just running)
`--format`		Pretty-print images using a Go template
`--no-stream`		Disable streaming stats and only pull the first result
`--no-trunc`		Do not truncate output

Docker Logs

- The docker logs command batch-retrieves logs present at the time of execution.



Description

Fetch the logs of a container

Usage

```
docker logs [OPTIONS] CONTAINER
```

Options

Name, shorthand	Default	Description
`--details`		Show extra details provided to logs
`--follow , -f`		Follow log output
`--since`		Show logs since timestamp (e.g. 2013-01-02T13:23:37) or relative (e.g. 42m for 42 minutes)
`--tail`	`all`	Number of lines to show from the end of the logs
`--timestamps , -t`		Show timestamps
`--until`		API 1.35+ Show logs before a timestamp (e.g. 2013-01-02T13:23:37) or relative (e.g. 42m for 42 minutes)

Additional Resources

See Lecture Page