
docker
advanced

logistics

- Class Hours:
- Start time is 9:30am
- End time is ~4:30pm
- Class times may vary
- Breaks mid-morning and afternoon (15 minutes)

- Lunch:
- Lunch is 12:00pm to 1:15pm
- Yes, 1 hour and 15 minutes
- Extra time for email, phone calls, or simply a walk.

- Telecommunication:
- Turn off or set electronic devices to vibrate / do not disturb
- Reading or attending to devices can be distracting to other students

- Miscellaneous:
- Courseware is available electronically
- Bathroom

Meet the Instructor

Chris Hiestand

Software Engineer Consultant with a linux sysadmin background. Focused on the kubernetes ecosystem.

Docker user since 2014, kubernetes since 2015.

www
https://kistek.consulting/

github
https://github.com/chrishiestand

mail
chris@kistek.consulting

Languages

- javascript/node.js
- python
- golang (in progress)
- php (please don't tell anyone)
- bash

Hello!

```
> docker run busybox echo hello world
hello world
```

- What's your name?
- What do you do?
- What's your experience with docker so far?
- What do you want to learn from this class? Feel free to give a detailed answer.

Course Objectives

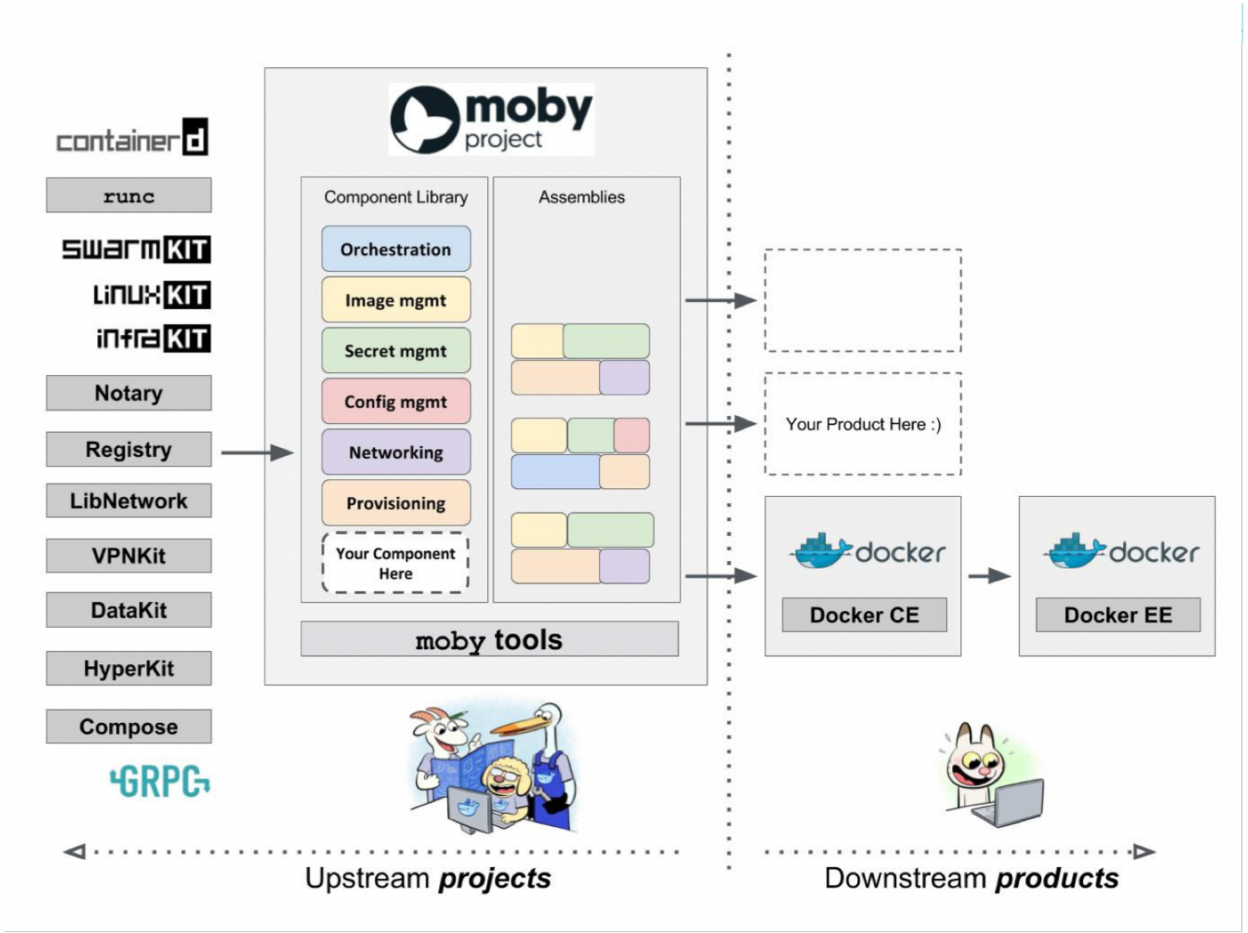
By the end of the course you will be able to:

- Have a broad understanding of the docker ecosystem
- Work with docker plugins, api, events, and networks
- Create size and cache optimized docker images
- Create security-hardened docker containers, images, and hosts
- Select an orchestration system to fit your needs

Part 1. Deep Docker

With some review

What is docker?



Container != VM

- Not a VM, but it has similarities
  - own process space
  - own network
  - can be run as root
  - install or delete programs without changing host
- Containers share the host kernel
- In-container processes are visible from the host
- Container kernel panic = host kernel panic
- Lighter weight than VM

The kernel has no idea what containers are

Containers are a convenience wrapper around kernel features:

- cgroups
- namespaces

Copy-on-write (CoW) is also an important component of running container images. Various filesystems might be used to implement CoW.

It’s layers all the way down

- Container images are composed of a stack of layers
- Layers are shared between containers = shared disk, shared memory
- Layers and images are immutable
- Layers and images have content addressable sha-256 digests
- Image tags are mutable, but might be treated as immutable
- Each command in a Dockerfile is a layer
- Files from previous layers are not removable in a new layer*

*Workarounds:
- multi-stage builds
- Squash (experimental)

Registries

- Docker Images (and therefore layers) are centralized in registries
- hosting options:
  - docker hub
  - docker trusted registry
  - self hosted
  - other cloud hosted (gcp, aws, azure, etc)
- registries may be public or private (requires auth)

Dockerfile

```
FROM golang

ENV GOBIN=/go/bin

CMD ["/go/bin/sys.json"]

RUN git clone https://github.com/EricR/sys.json.git /go/sys.json && \
    cd /go/sys.json && \
    go get && \
    go build
```

12 Factor Apps  Containers

The Twelve Factors

#	Factor	Description
I	Codebase	There should be exactly one codebase for a deployed service with the codebase being used for many deployments.
II	Dependencies	All dependencies should be declared, with no implicit reliance on system tools or libraries.
III	Config	Configuration that varies between deployments should be stored in the environment.
IV	Backing services	All backing services are treated as attached resources and attached and detached by the execution environment.
V	Build, release, run	The delivery pipeline should strictly consist of build, release, run.
VI	Processes	Applications should be deployed as one or more stateless processes with persisted data stored on a backing service.
VII	Port binding	Self-contained services should make themselves available to other services by specified ports.
VIII	Concurrency	Concurrency is advocated by scaling individual processes.
IX	Disposability	Fast startup and shutdown are advocated for a more robust and resilient system.
X	Dev/Prod parity	All environments should be as similar as possible.
XI	Logs	Applications should produce logs as event streams and leave the execution environment to aggregate.
XII	Admin Processes	Any needed admin tasks should be kept in source control and packaged with the application.

https://en.wikipedia.org/wiki/Twelve-Factor_App_methodology

Docker Image Labels

- An image name is made up of slash-separated name components, optionally prefixed by a registry hostname.
- DNS name conventions required
- If hostname empty, registry-1.docker.io is default

Examples

- dockerhubuser/my-image
- us.gcr.io/cloud-project/example-image:v1

Building Images

```
> docker build -t chrishiestand/go-sysjson .
Sending build context to Docker daemon  75.78kB
Step 1/4 : FROM golang
 ---> 1c1309ff8e0d
Step 2/4 : ENV GOBIN=/go/bin
 ---> Using cache
 ---> d901bf31413b
Step 3/4 : CMD ["/go/bin/sys.json"]
 ---> Using cache
 ---> 07156eb9281d
Step 4/4 : RUN git clone https://github.com/EricR/sys.json.git /go/sys.json &&     cd /go/sys.json &&     go get &&     go build
 ---> Using cache
 ---> 377daeefdeb9
Successfully built 377daeefdeb9
Successfully tagged chrishiestand/go-sysjson:latest
```

Tagging Images

```
> docker build .
Sending build context to Docker daemon  75.78kB
Step 1/4 : FROM golang
 ---> 1c1309ff8e0d
Step 2/4 : ENV GOBIN=/go/bin
 ---> Using cache
 ---> d901bf31413b
Step 3/4 : CMD ["/go/bin/sys.json"]
 ---> Using cache
 ---> 07156eb9281d
Step 4/4 : RUN git clone https://github.com/EricR/sys.json.git /go/sys.json &&     cd /go/sys.json &&     go get &&     go build
 ---> Using cache
 ---> 377daeefdeb9
Successfully built 377daeefdeb9
```

```
> docker tag 377daeefdeb9 chrishiestand/go-sysjson
```

Pushing images

```
> docker push chrishiestand/go-sysjson
The push refers to repository [docker.io/chrishiestand/go-sysjson]
6e557742bd8a: Pushing [=========>                                         ]  9.355MB/47.4MB
f8268e3059f2: Layer already exists
6f92d3834a0b: Layer already exists
8f16f42ec6a8: Layer already exists
c8718ae77a2a: Layer already exists
a6afffb1947c: Layer already exists
f7d58f758444: Layer already exists
8568818b1f7f: Layer already exists
```

Pulling images

```
> docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
8176e34d5d92: Downloading [=========>                                         ]  3.909MB/22.5MB
5b19c1bdd74b: Downloading [==========>                                        ]  4.588MB/21.99MB
4e9f6296fa34: Download complete
```

Fast Off the Shelf Development

- docker pull redis:3.2-alpine
- docker pull influxdb
- docker pull nginx
- docker pull nginx:1.13-alpine
- docker pull mysql:8.0.4

Goodbye vagrant

Logging

- Best practice: Log to stdout and stderr
- Log drivers also available

Copy on Write

Common union mount filesystems:

- unionfs
- aufs
- overlayfs

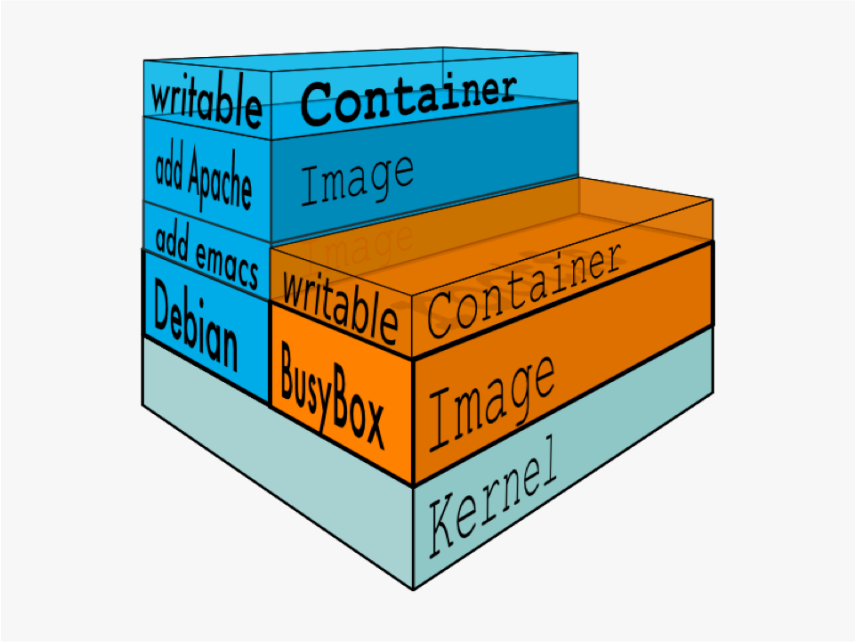


image credit: https://washraf.gitbooks.io/the-docker-ecosystem/content/Chapter%201/Section%203/union_file_system.html

Base Images

- scratch
- alpine
- nixos/nix
- ubuntu
- centos
- debian
- or any docker image

Container Lifecycle

- Container command starts as pid 1
- SIGTERM sent for graceful shutdown

State

- Stateless containers are easy to administer and run just about anywhere
- Stateful containers require coordination but today are well done in many environments

Why Use Containers?

- Dev = Stage = Test = Prod (application-wise)
- Rapid Development
- Standard Transmission, Deployment, and Tooling for any app
- Less Configuration to Manage = Less Configuration Management
- ***Composable***

OCI, runc, etc

An organization to standardize containers

OCI = Open Container Initiative

- runtime-spec (from docker runtime)
- image-spec = (from docker image)

runc = oci compatible container runtime

Lab 1 – composing containers

2. cgroups and namespaces

- Containers aren’t first class objects in linux
- Linux containers are a friendly wrapper around linux namespaces and cgroups

namespaces

What a container can see

- mount
- pid
- uts (hostname)
- ipc (shared memory)
- net
- user
- cgroup

Containers can share namespaces

namespace mnt = mount

- cloned mounts are copied to new namespace
- after creation, mounts in new namespace are typically isolated

https://en.wikipedia.org/wiki/Linux_namespaces

namespace pid = Process IDs

- namespaced pids are independent of other pids
- first process within namespace gets pid 1 – is treated like init
  - this causes problems for some container programs
  - be sure to setup signal handlers
- pid namespaces can be nested (meaning: docker-in-docker is possible)

https://en.wikipedia.org/wiki/Linux_namespaces

namespace uts = hostname

- own hostname
- own domain name

```
[dev@centos ~]$ sudo hostnamectl set-hostname "Dan's Computer"
[dev@centos ~]$
[dev@centos ~]$ hostnamectl status --static
danscomputer
[dev@centos ~]$ hostnamectl status --transient
danscomputer
[dev@centos ~]$ hostnamectl status --pretty
Dan's Computer
[dev@centos ~]$
```

https://en.wikipedia.org/wiki/Linux_namespaces

namespace ipc = Interprocess Communication

- Isolate inter-process communication
- For example, memory cannot be shared across ipc namespaces
- same with semaphores
- and message queues

https://en.wikipedia.org/wiki/Linux_namespaces

namespace net = Network

- virtualized, isolated network stack
- own IPs, routing table, sockets, connections, firewall, etc
- contains only loopback interface by default
- interfaces are pinned to 1 namespace but can be moved between namespaces
- may container physical interfaces

https://en.wikipedia.org/wiki/Linux_namespaces

namespace user = User ID

- user isolation (mostly)
- user namespaces may be nested
- user IDs may be mapped between parent and children namespaces
  - e.g. a container's root might be mapped to a host's nobody

https://en.wikipedia.org/wiki/Linux_namespaces

namespace cgroup = control group

- hides the cgroup identity of the process which is a member of that cgroup

https://en.wikipedia.org/wiki/Linux_namespaces

Not namespaced

- time (syscalls blocked by default)
- kernel keyring (syscalls blocked by default)
- more…

cgroups (control groups)

What a container can utilize

- memory
- cpu
- i/o
- network

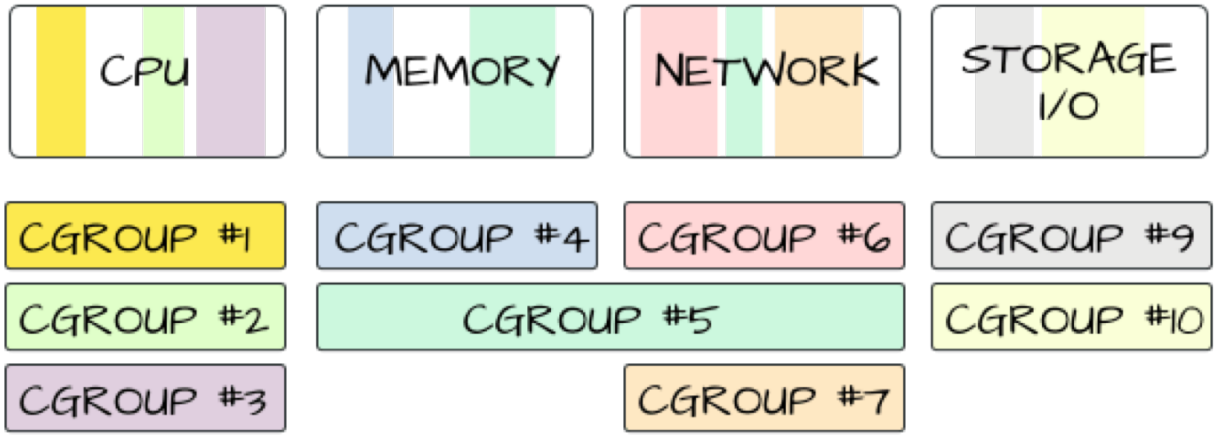


image credit: https://mairin.wordpress.com/2011/05/13/ideas-for-a-cgroups-ui/

cgroup features

- Resource Limitations
- Prioritization
- Accounting
- Control

Example cgroup

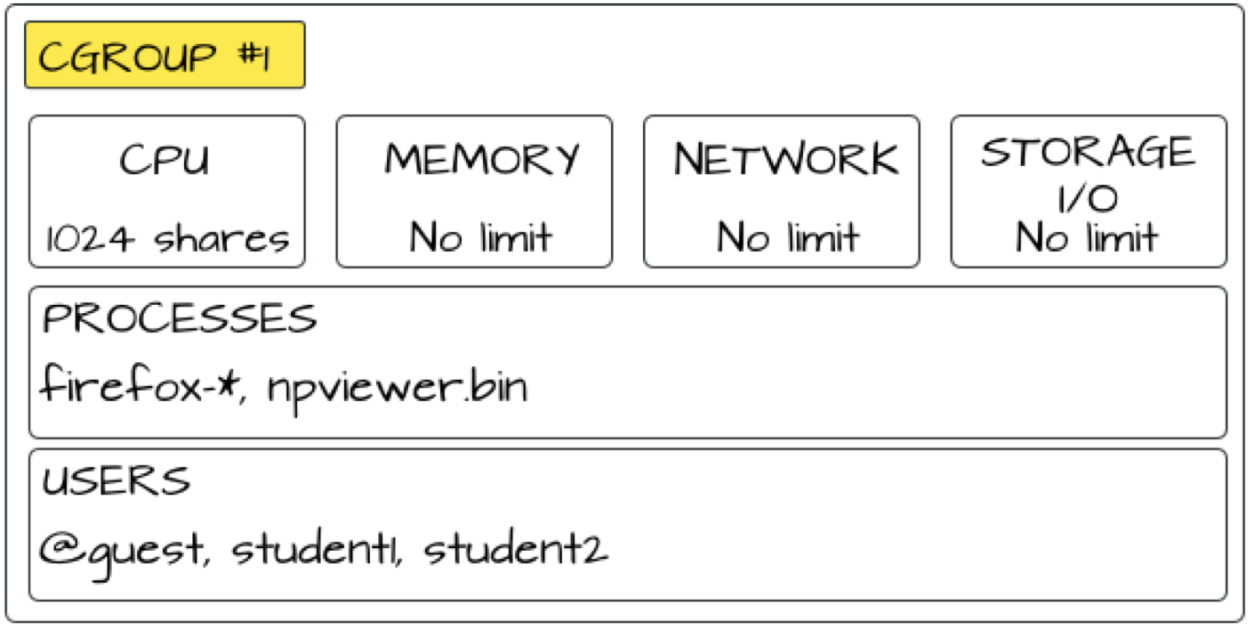


image credit: https://mairin.wordpress.com/2011/05/13/ideas-for-a-cgroups-ui/

docker and cgroups

```
play@lab19:~$ docker run --help |grep -E 'cpu|mem|blk|io'
Options:
      --blkio-weight uint16           Block IO (relative weight), between 10 and 1000, or 0 to disable (default 0)
      --blkio-weight-device list      Block IO weight (relative device weight) (default [])
      --cgroup-parent string          Optional parent cgroup for the container
      --cpu-period int                Limit CPU CFS (Completely Fair Scheduler) period
      --cpu-quota int                 Limit CPU CFS (Completely Fair Scheduler) quota
      --cpu-rt-period int             Limit CPU real-time period in microseconds
      --cpu-rt-runtime int            Limit CPU real-time runtime in microseconds
  -c, --cpu-shares int                CPU shares (relative weight)
      --cpus decimal                  Number of CPUs
      --cpuset-cpus string            CPUs in which to allow execution (0-3, 0,1)
      --cpuset-mems string            MEMs in which to allow execution (0-3, 0,1)
      --device-read-iops list         Limit read rate (IO per second) from a device (default [])
      --device-write-iops list        Limit write rate (IO per second) to a device (default [])
```

```
-m, --memory bytes                    Memory limit
    --memory-reservation bytes        Memory soft limit
    --memory-swap bytes               Swap limit equal to memory plus swap: '-1' to enable unlimited swap
    --memory-swappiness int           Tune container memory swappiness (0 to 100) (default -1)
```

Lab 2 – namespaces and cgroups

3. Advanced Docker Builds

- A great docker image is small e.g. < 20MiB
- The ideal image contains only the application*
- A great image has very few layers*
- The ideal image has 1 layer*
- The ideal build is 100% reproducible
- The ideal build is as fast as possible

*except when it doesn't

Q: When does performance matter?

Q: When does performance not matter?

Typical order of optimizations

1. Correctness
2. Integrity
3. Readability
4. Performance

That said, let’s take performance where it’s free

Why Care?

Smaller Images Mean:

- Faster builds
- Faster pulls
- Faster pushes
- Faster development
- Faster testing
- Faster deployment

Developers Care

Developer Context Switching

Operators Care

- Faster deployments may mean less babysitting
- Faster deployments ship features bugfixes faster

Minimal Base Images

- scratch
- alpine (<5MiB)
- gcr.io/distroless

Application Base Images

- redis:4-alpine
- mysql:5.7

Layer Review

- Images are stacks of layers
- Each Dockerfile line is a layer

```
#Layer n
FROM nixos/nix

#Layer n+1
ENV build=dev

#Layer n+2
CMD /app/bin/start

#Layer n+3
COPY app /app

#Layer n+4
RUN /app/bin/build.sh
```

Add small layers

Might be small, might be HUGE

```
FROM alpine

COPY . .
```

Probably smaller

```
FROM alpine

COPY app /app
```

Independent RUNs at top

- RUNs are now cacheable

```
FROM alpine

RUN dd if=/dev/random of=/tmp/seed bs=1 count=1

# More stuff ...
```

Dependent RUNs immediate after the file they depend on

```
#Layer n
FROM nixos/nix

#Layer n+1
ENV build=dev

#Layer n+2
CMD /app/bin/start

#Layer n+3
COPY app /app

#Layer n+4
RUN /app/bin/build.sh
```

chain your RUNs

Chained commands happen in the same layer

```
FROM node:6-alpine

CMD ["node", "/www/src/index.js"]

COPY package.json /www/

RUN apk add --update git && \
    npm --prefix=/www install && \
    apk del git && \
    rm -rf /var/cache/apk/*

COPY src/ /www/src
```

cleanup your RUNs

This build requires git. Let’s cleanup after the build.

```
FROM node:6-alpine

CMD ["node", "/www/src/index.js"]

COPY package.json /www/

RUN apk add --update git && \
    npm --prefix=/www install && \
    apk del git && \
    rm -rf /var/cache/apk/*

COPY src/ /www/src
```

Add Layers Sparingly

- Each line should have a good reason to exist

Common extraneous layers:
- RUN followed by RUN
- MAINTAINER <in+version+control@example.com>
- EXPOSE 80
- WORKDIR

Sometimes extraneous:
- VOLUME /app/data

Strongly Prefer COPY over ADD

ADD is complex, supports remote URLs and local files

ADD is not as cache friendly as COPY

Only **one** good use case for ADD, unpacking a ***local*** tar archive:

```
ADD rootfs.tar.gz /
```

Which replaces 2 layers with COPY:

```
COPY rootfs.tar.gz /
RUN tar –xvzf /rootfs.tar.gz && \
        rm /rootfs.tar.gz
```

Cache is the Money

Remember, layers are immutable

If time is money
Layers are time
Layers are cached
Cache is the money

- Every image layer is locally cached everywhere the image is stored:
  - On your development machine
  - CI/CD
  - stage nodes
  - production nodes

Order Matters

Anti-Example:

```
COPY . /tmp/
RUN pip install --requirement /tmp/requirements.txt

```

Not bad?:

```
COPY requirements.txt /tmp/
RUN pip install --requirement /tmp/requirements.txt
COPY . /tmp/

```

Order Matters (cont)

Good:

```
COPY LICENSE /tmp/
COPY requirements.txt /tmp/
RUN pip install --requirement /tmp/requirements.txt
COPY *.py /tmp/

```

Best:

```
COPY LICENSE /tmp/
COPY requirements.txt /tmp/
RUN pip install --requirement /tmp/requirements.txt
COPY constants.py /tmp/
COPY main.py /tmp/

```

Decoration Up Top

Image data no-ops are always cacheable, put them up top:

- LABEL
- MAINTAINER
- EXPOSE
- CMD
- ENTRYPOINT

Decoration Up Top (cont)

Anti-Example:

```
FROM nginx
COPY index.html /www/
COPY nginx.conf /etc/nginx/mynginx.conf
EXPOSE 8080
MAINTAINER <useddownstream@example.com>
LABEL com.example.version="0.0.1-beta"
CMD ["nginx", "-p", "/etc/nginx/mynginx.conf"]
```

Decoration Up Top (cont)

```
FROM nginx
EXPOSE 8080
MAINTAINER <useddownstream@example.com>
CMD nginx -p /etc/nginx/mynginx.conf
LABEL com.example.version "0.0.1-beta"
COPY nginx.conf /etc/nginx/mynginx.conf
COPY index.html /www/
```

Layers are immutable, nth reminder

Bad*:

```
COPY . .
```

Probably Worse:

```
COPY . .
RUN rm –rf secrets/
```

*potentially *okay*

.dockerignore - reduce build context

you can use .dockerignore in root of build context to exclude files from the build

.dockerignore makes these better:

```
COPY . .
```

```
COPY src/ /app/src
```

When simple, explicit is still preferable:

```
COPY foo /app/foo
COPY bar /app/bar
```

VOLUME Use Cases

VOLUME /app/data

Use VOLUME if all are true:

- path is hard-coded
- you expect frequent writes
- do not expect path to be bind-mounted at runtime

Examples: database storage, configuration storage, application cache

Squash (experimental)

Squash will convert one or more layers into a single layer

```
docker build --squash
```

Cacheability vs Layers and Readability

We see a tradeoff between values:

- minimizing number of layers and readability
- maximizing cacheability

```
COPY *.py /app/
¬
# VS
¬
COPY constants.py /app/
COPY main.py /app/
```

```
COPY *.py /app/
¬
# VS
¬
COPY a.py /app/
COPY b.py /app/
COPY c.py /app/
COPY d.py /app/
# ...
```

When to **not** minimize an image

- run a debugger inside the container
- run a profiler inside the container
- run a supporting application inside the container (e.g. redis-cli)
- open a shell inside a container

multi-stage build example

```
FROM node:8-alpine AS build
COPY .npmrc /home/user/.npmrc
COPY package.json /app/
RUN npm –prefix=/app install
COPY src /app/src

FROM node:8-alpine
CMD [“node”, ”/app/src/index.js”]
COPY --from=build /app /app
```

Trusting Upstream

FROM nginx:1.13.9

Does project use semver?
If yes are they consistent?
What version of semver?

What happens if upstream breaks my app?
Will tests catch it?

Reliable Rebuilds with Digests

```
#nginx:1.13.9¬
FROM nginx@sha256:4771d09578c7c6a65299e110b3ee1c0a2592f5ea2618d23e4ffe7a4cab1ce5de¬
COPY index.html /www/¬
¬
```

+ Reliable base image for builds

- No automatic upstream updates
- Worse readability

okay, maybe not 100%

Ludicrous (deployment) Speed

For fast rollouts, container startup time matters.

startup time = pull time + start time

Ludicrous (deployment) Speed (cont)

For fast rollouts, container shutdown time also matters!

shutdown time = graceful shutdown after SIGTERM received

Be sure the application (or whatever is PID 1) handles SIGTERM!

Review

- Is the image as small as it can be without sacrificing correctness, integrity, or readability?
- Does the image support troubleshooting needs?
- Use digest if the build need to be 100% reproducible
- Handle SIGTERM

Lab 3