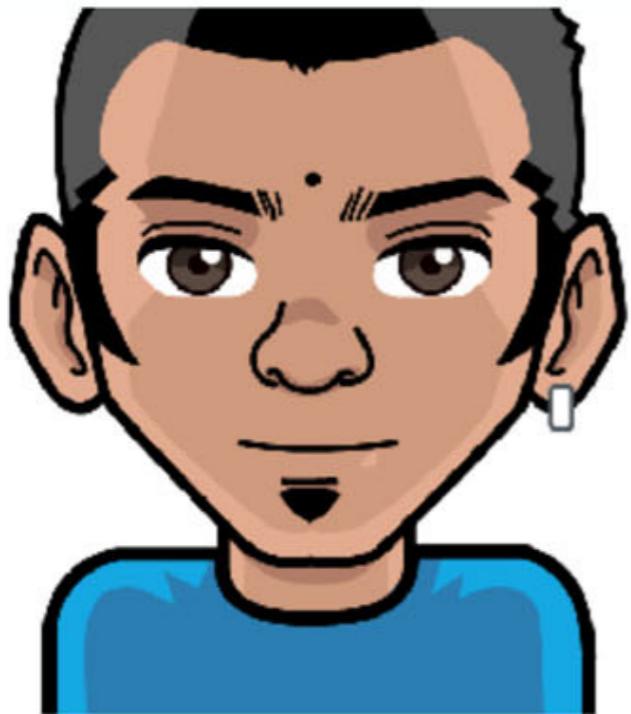


Raju Gandhi

HANDS-ON DOCKER

Learn best practices for building maintainable Dockerfiles



RAJU GANDHI

RAA-JEW ('A' LIKE THE 'A' IN CAR)



@LOOSELYTYPED

FOUNDER - DEFMACRO SOFTWARE

O'REILLY®

Head First

Git

A Learner's Guide
to Understanding Git
from the Inside Out

Raju Ganchi



 A Brain-Friendly Guide

<https://i-love-git.com>

<https://learning.oreilly.com/library/view/head-first-git/9781492092506/>

WHY?

BUILD ONCE, RUN ANYWHERE

WE WANT ...

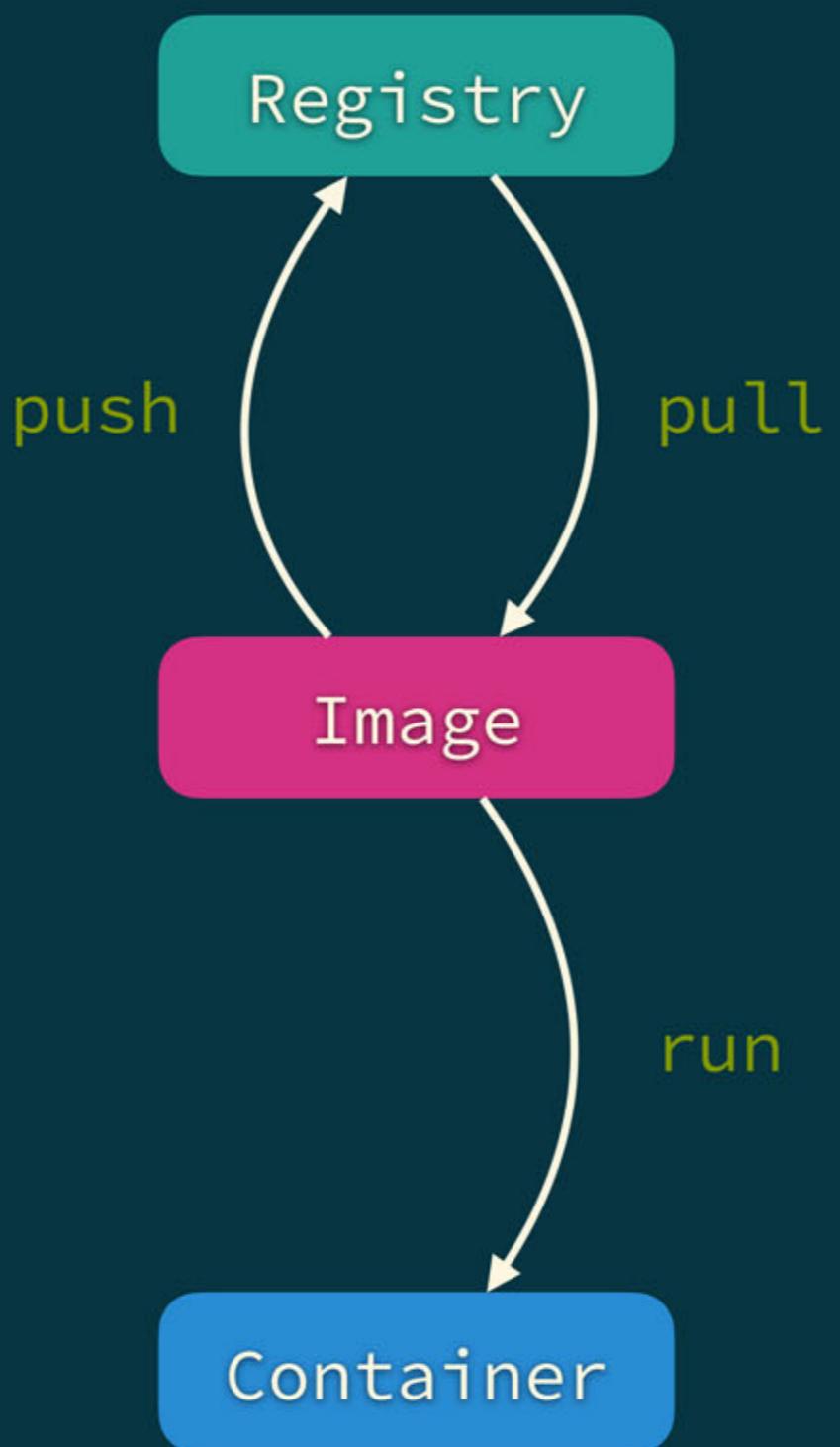
FASTER BUILDS

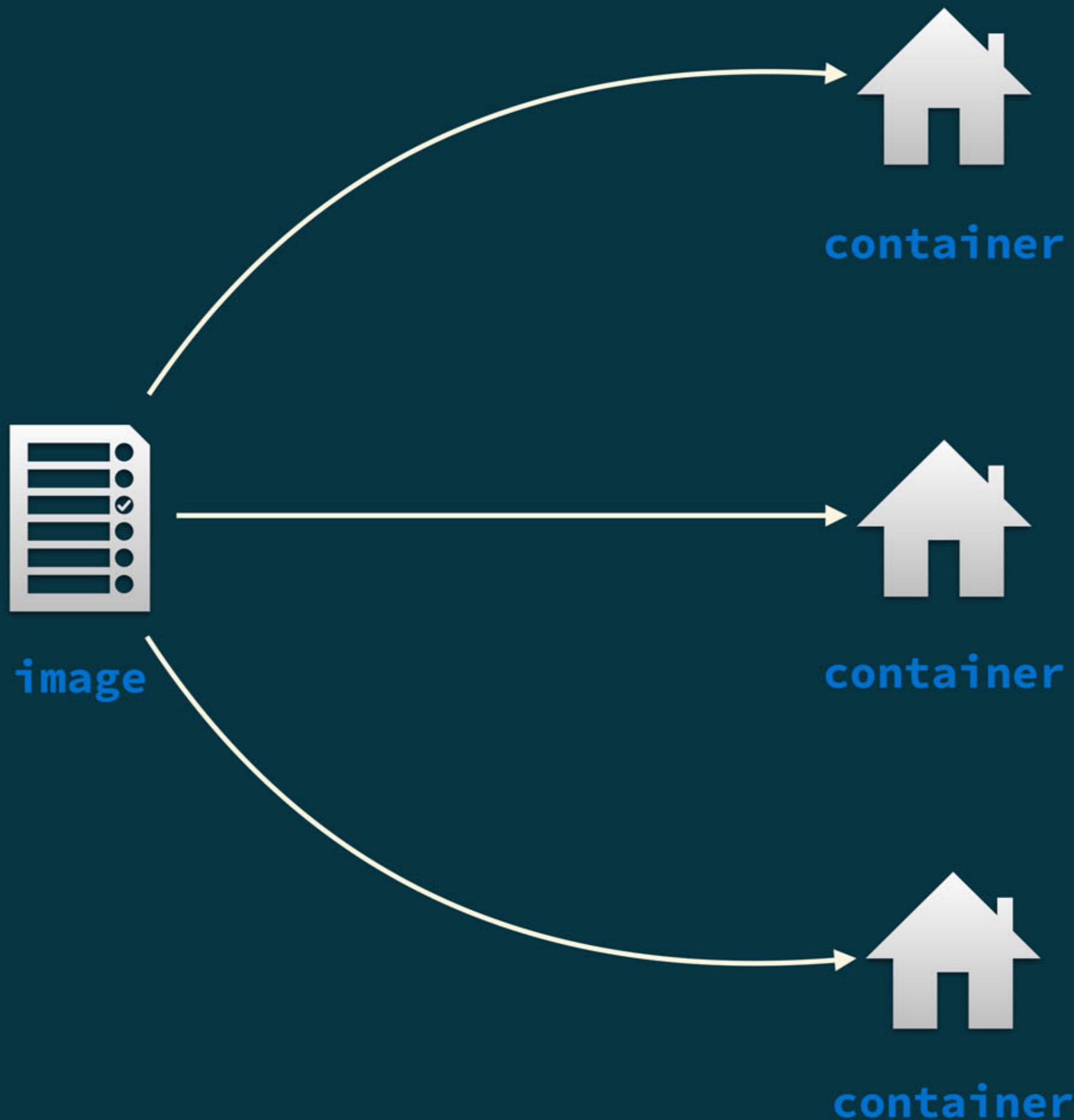
LEANER IMAGES

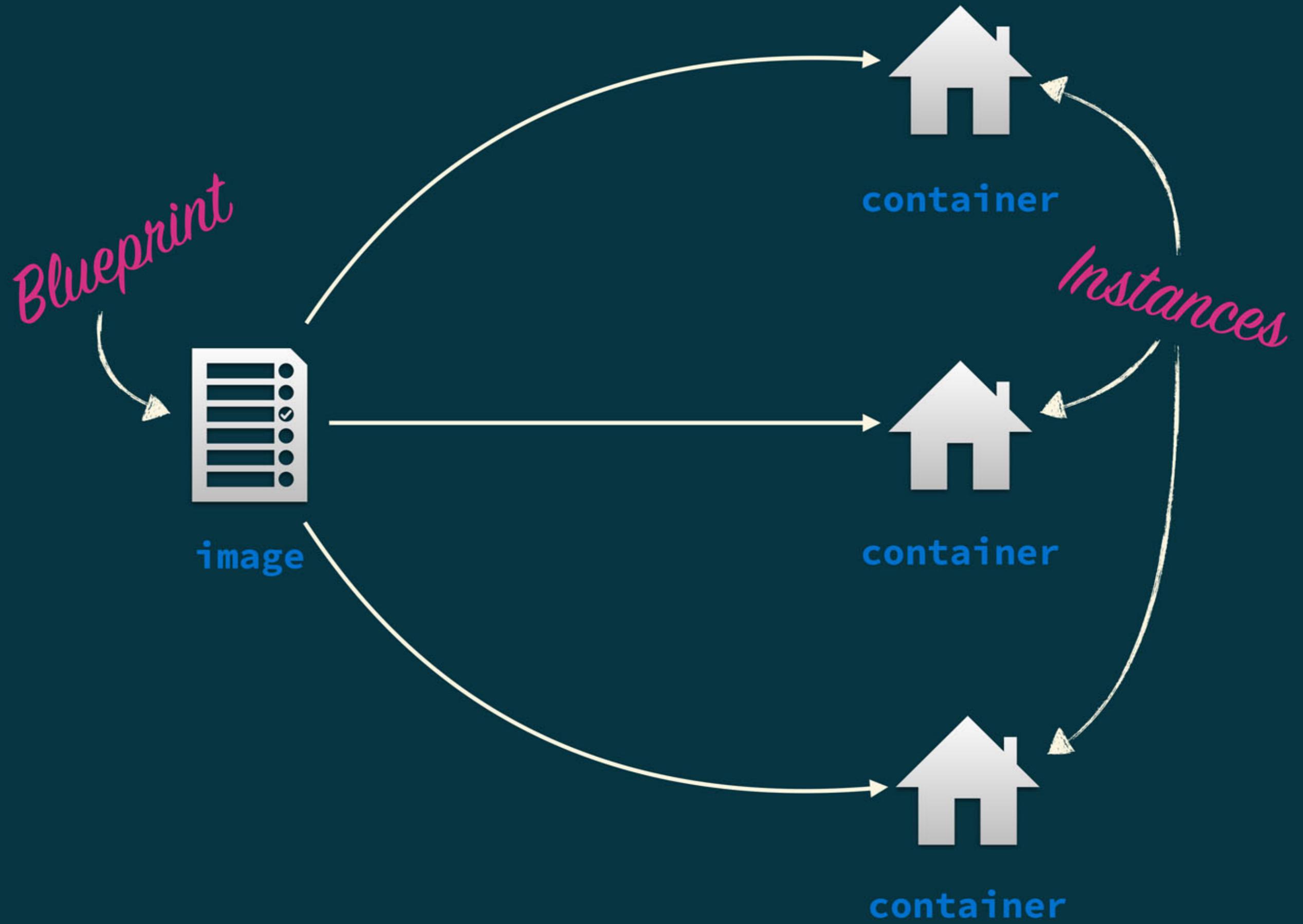
DETERMINISTIC BUILDS

FLEXIBLE RUNTIME

IMAGES







IMAGES AS "CLASSES"

IMAGES

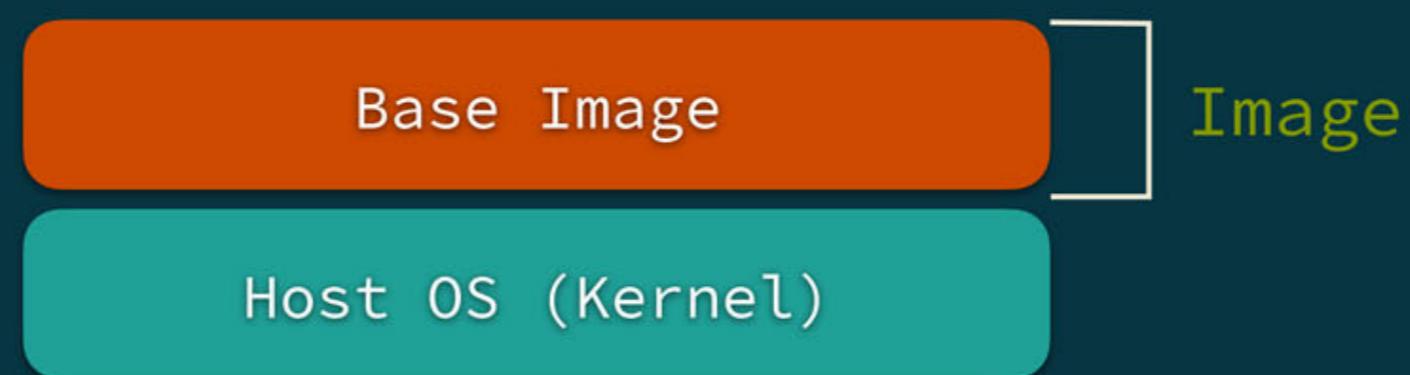
- The docker artifact
- Opaque
- Shared using a registry like <http://hub.docker.com/>

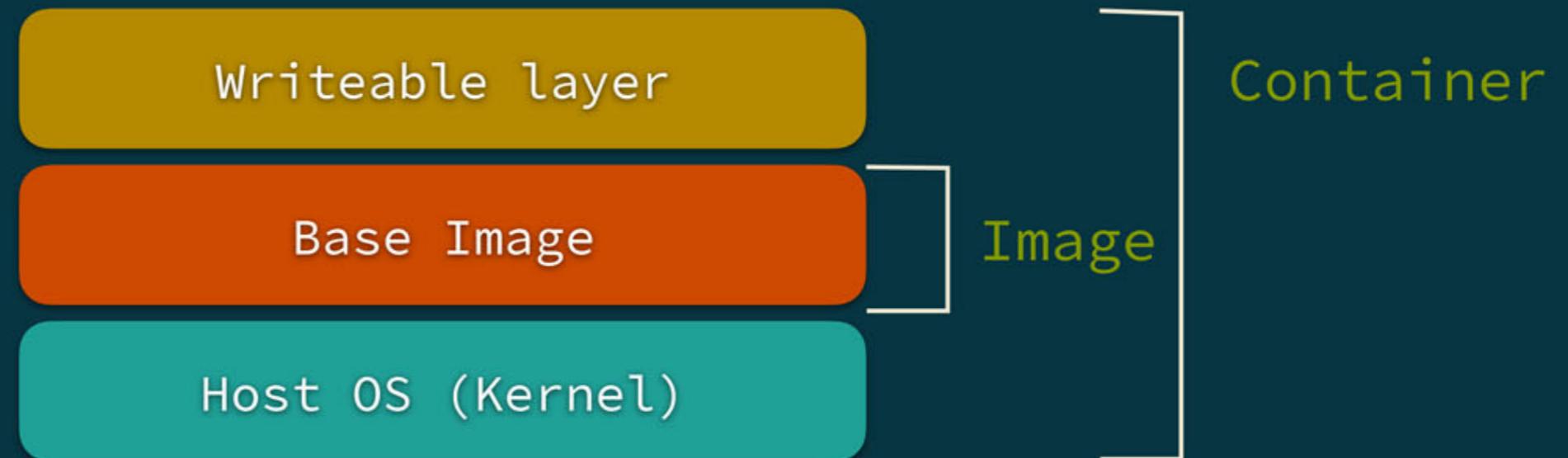
CONTAINERS AND COPY-ON-WRITE

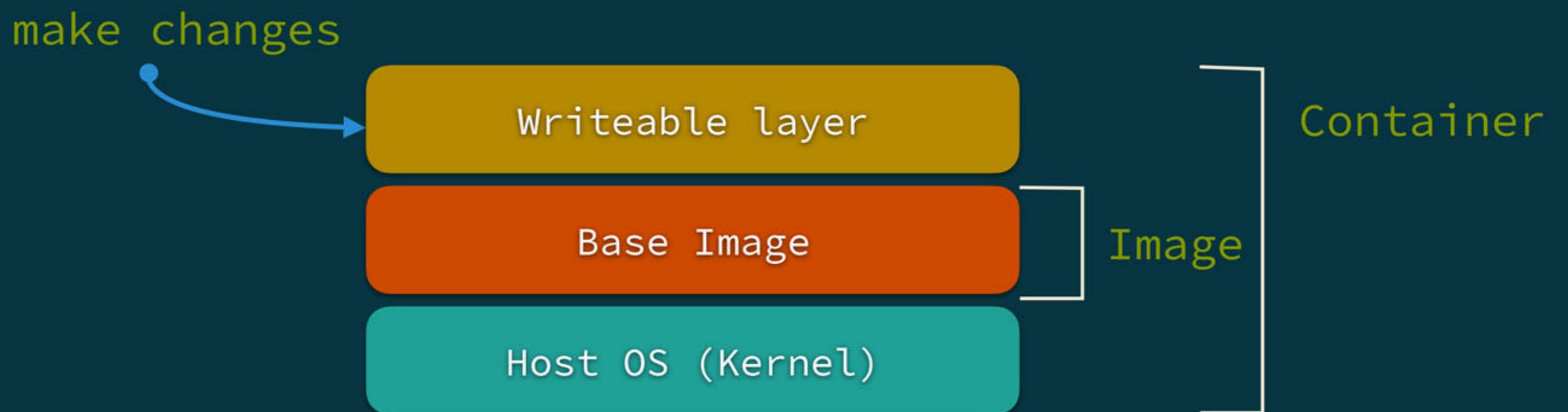
WHAT IS A CONTAINER

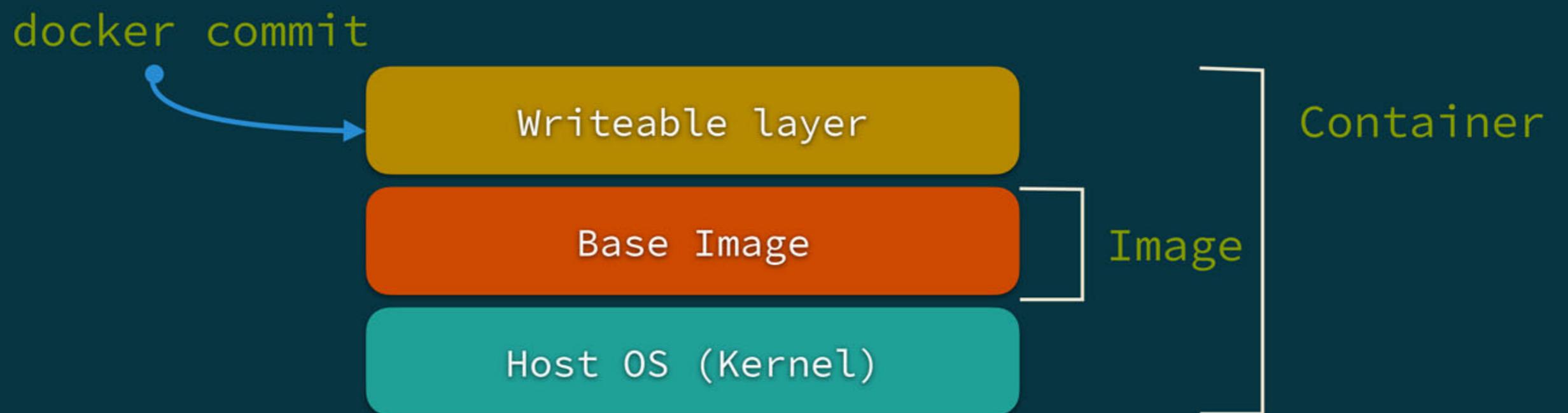
- An instance of an image
- With a r/w layer on top
- Configured with resource limits (cpu/memory), network settings and volume mounts etc on "create"

docker run

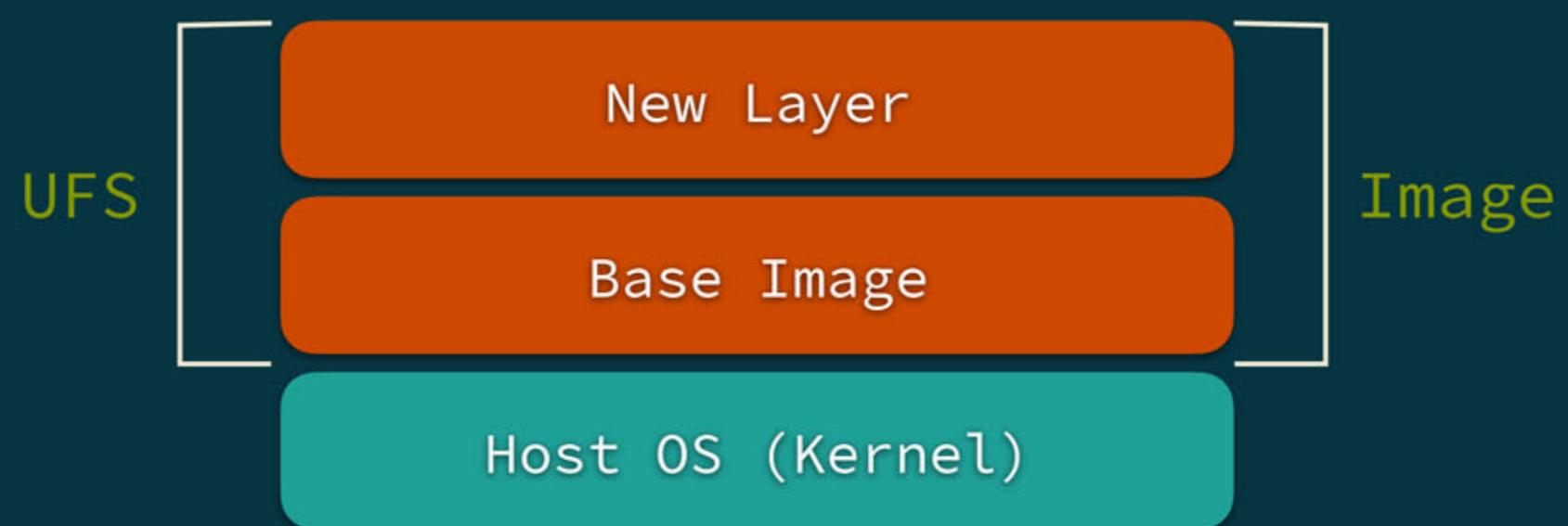




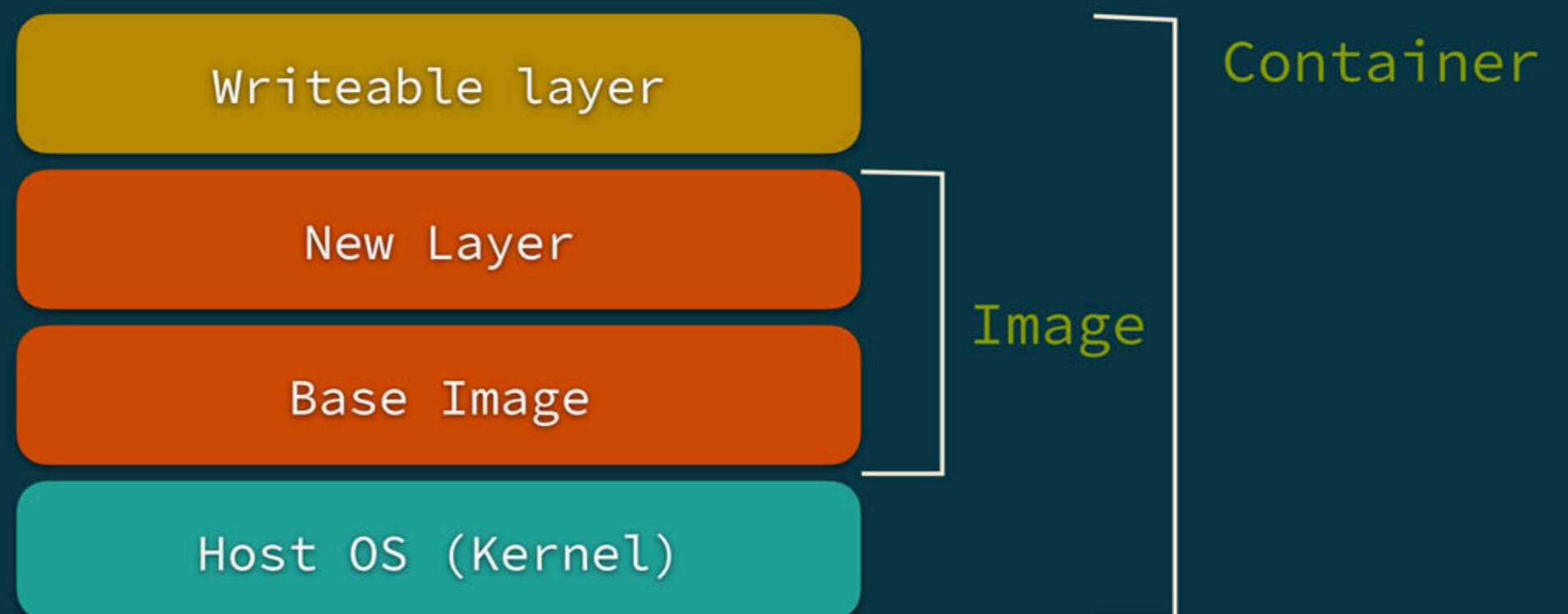


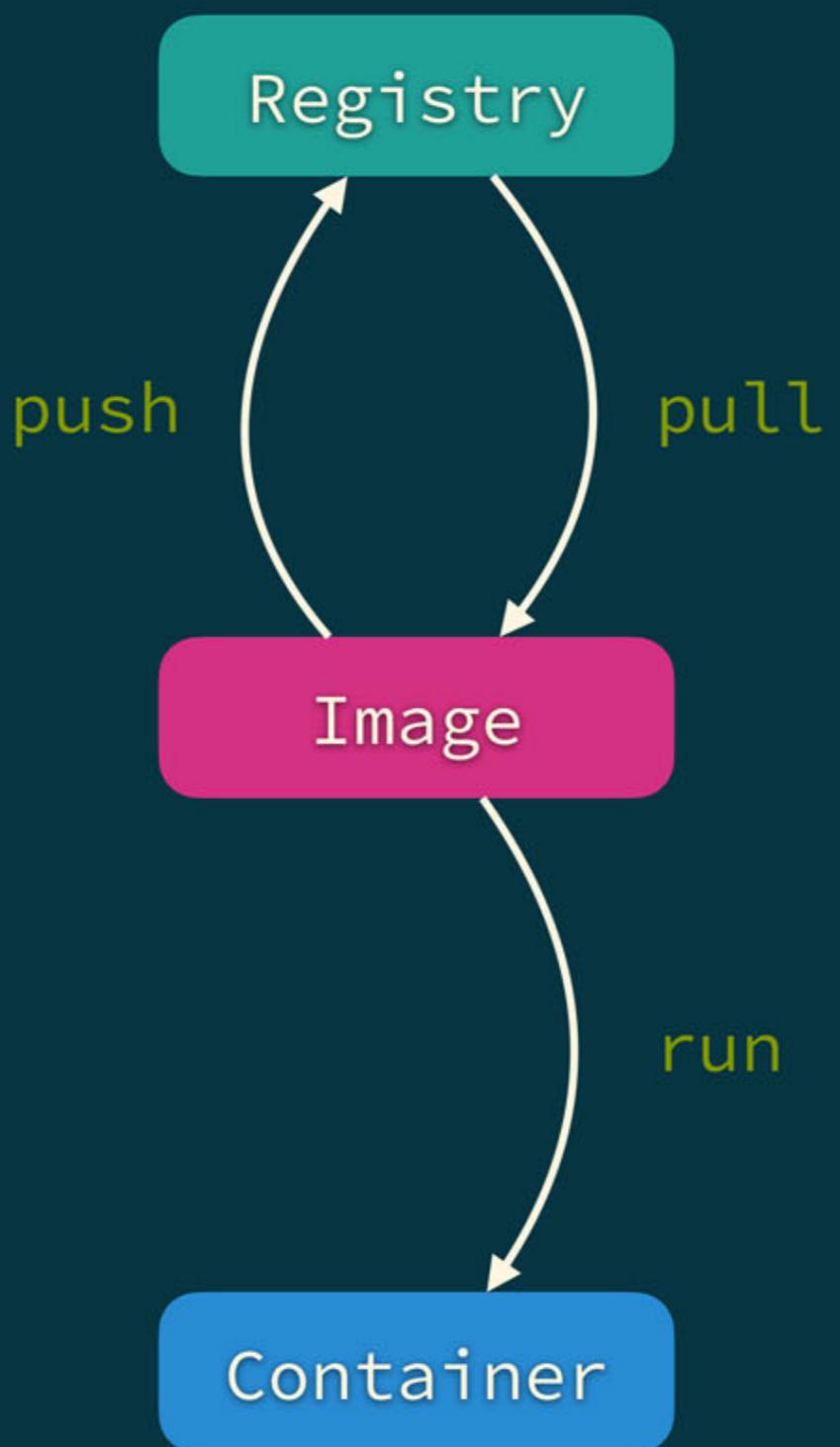


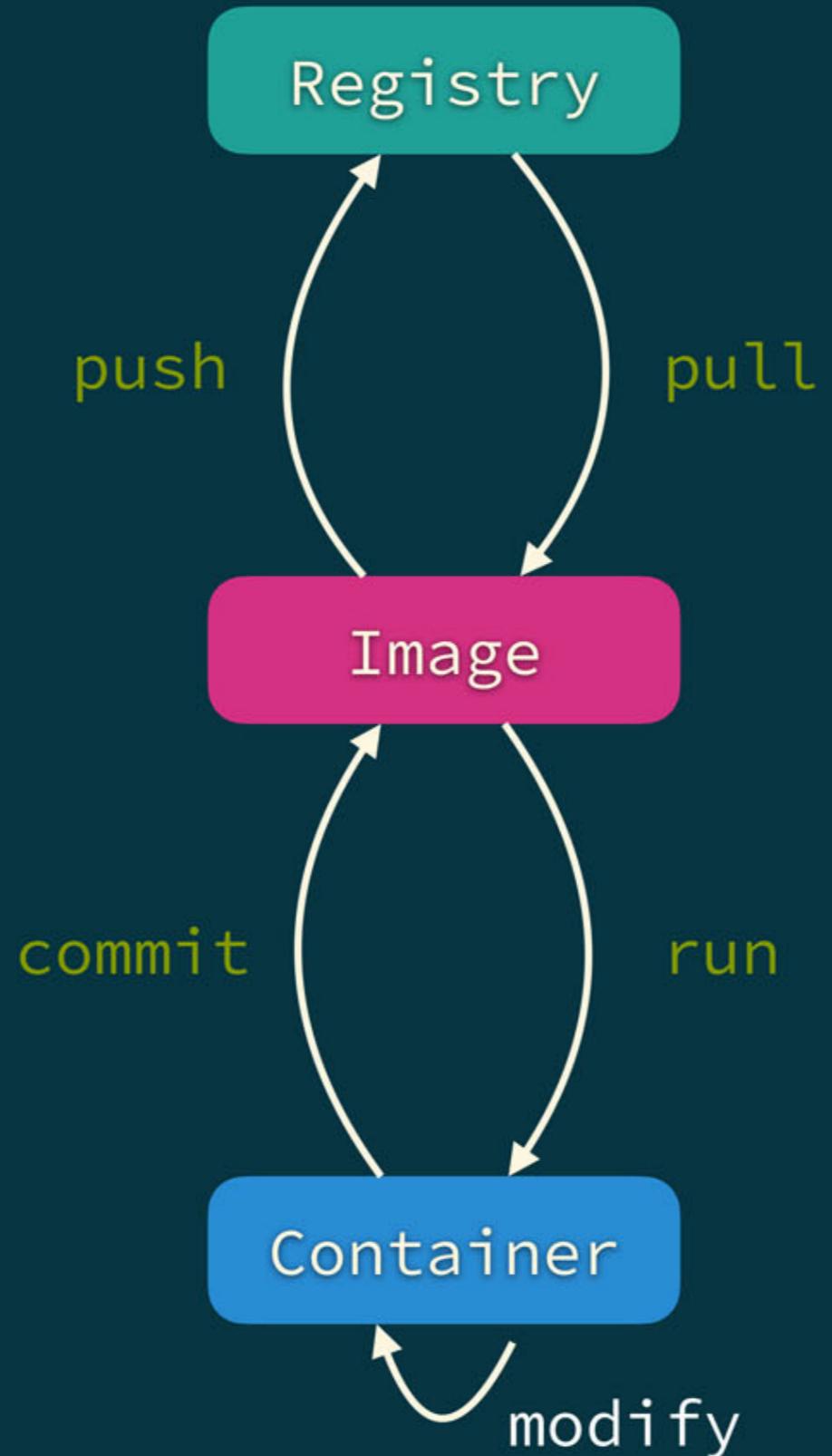




`docker run`







**IF I CAN DO THIS USING
COMMANDS, WHY DO WE NEED
DOCKERFILES?**

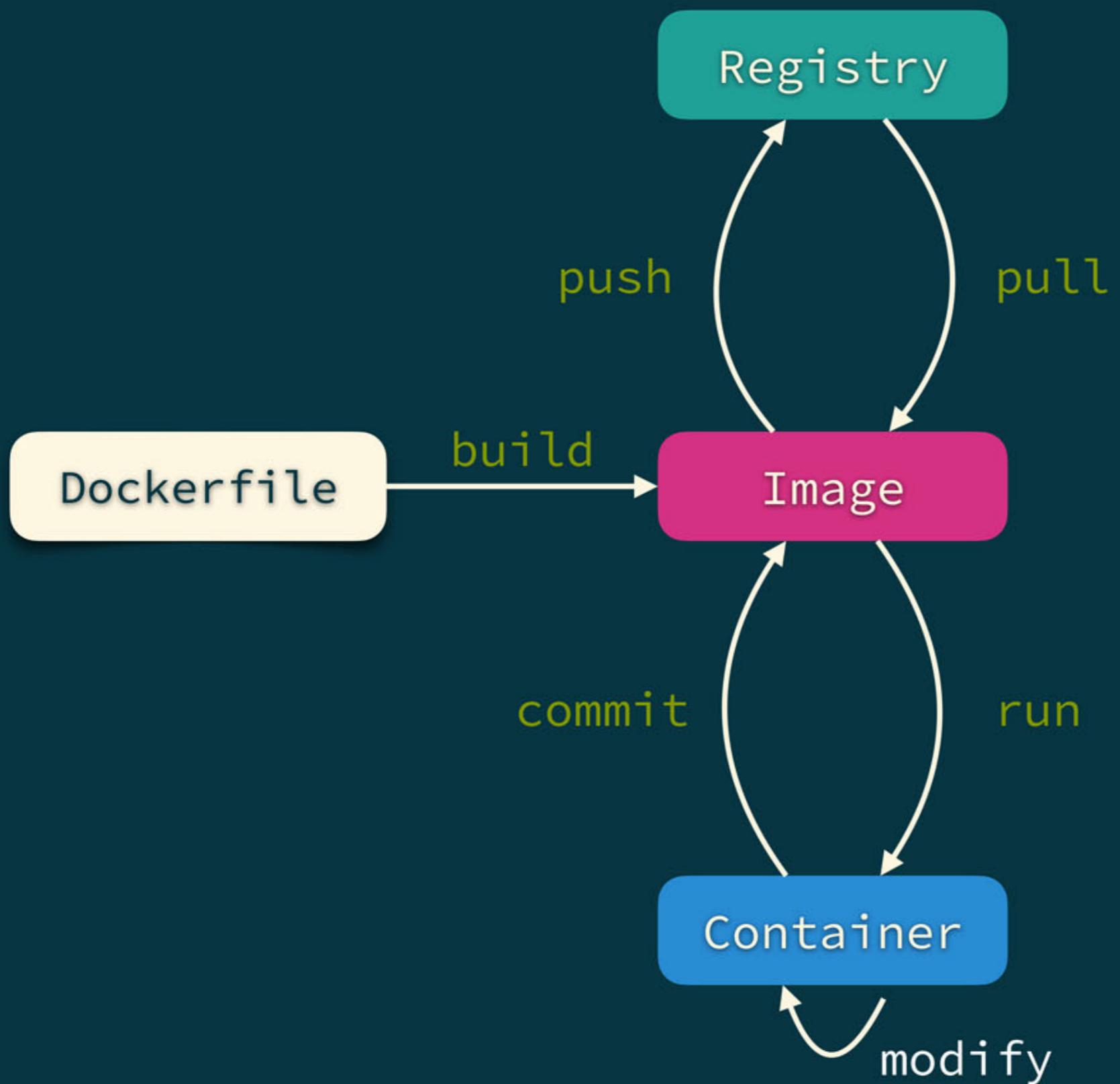
@EXERCISE

EXERCISE

- Start a container with a unique name using `ubuntu:21.04`
- Modify the container in some way – install something or touch some files
- Stop the container, and use "`docker container commit`" to create a new image
- Start a new container from the newly created image
- Check to see if your changes are still around

HINTS

- To install on ubuntu use "`apt update -y && apt install git`"
- To create a file you can use "`touch <filename>`"
- Look up "`docker help container`" and "`docker container commit --help`"



DOCKERFILE

DOCKERFILES

- A set of instructions to build a Docker image
- Plain text, version controlled
- Provides insight into the container needs/capabilities/intents

```
# sample Dockerfile
FROM openjdk:8u131-jre

RUN apt-get update \
    && apt-get install -y netcat

COPY build/libs/app-fat.jar /var/app.jar

CMD ["java", "-jar", "/var/app.jar"]
```

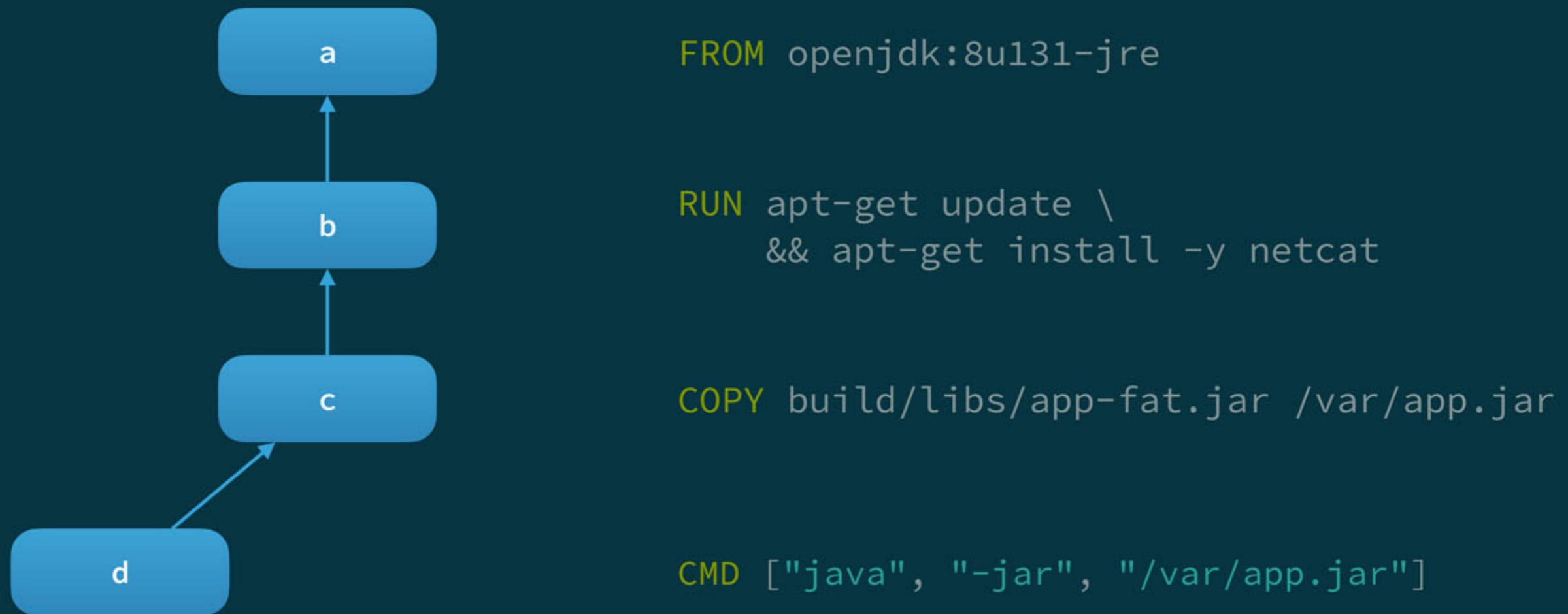
UNION FILE SYSTEM

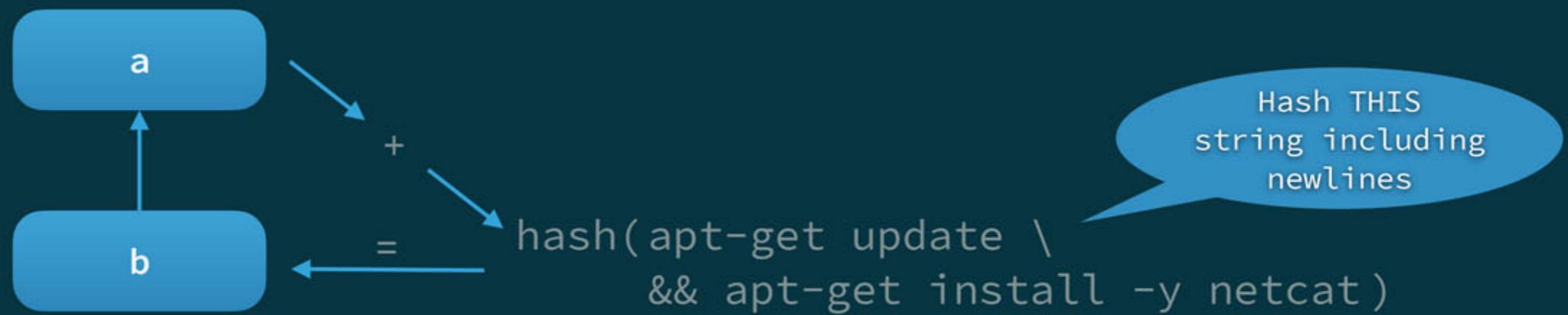
```
# sample Dockerfile
FROM openjdk:8u131-jre

RUN apt-get update \
    && apt-get install -y netcat

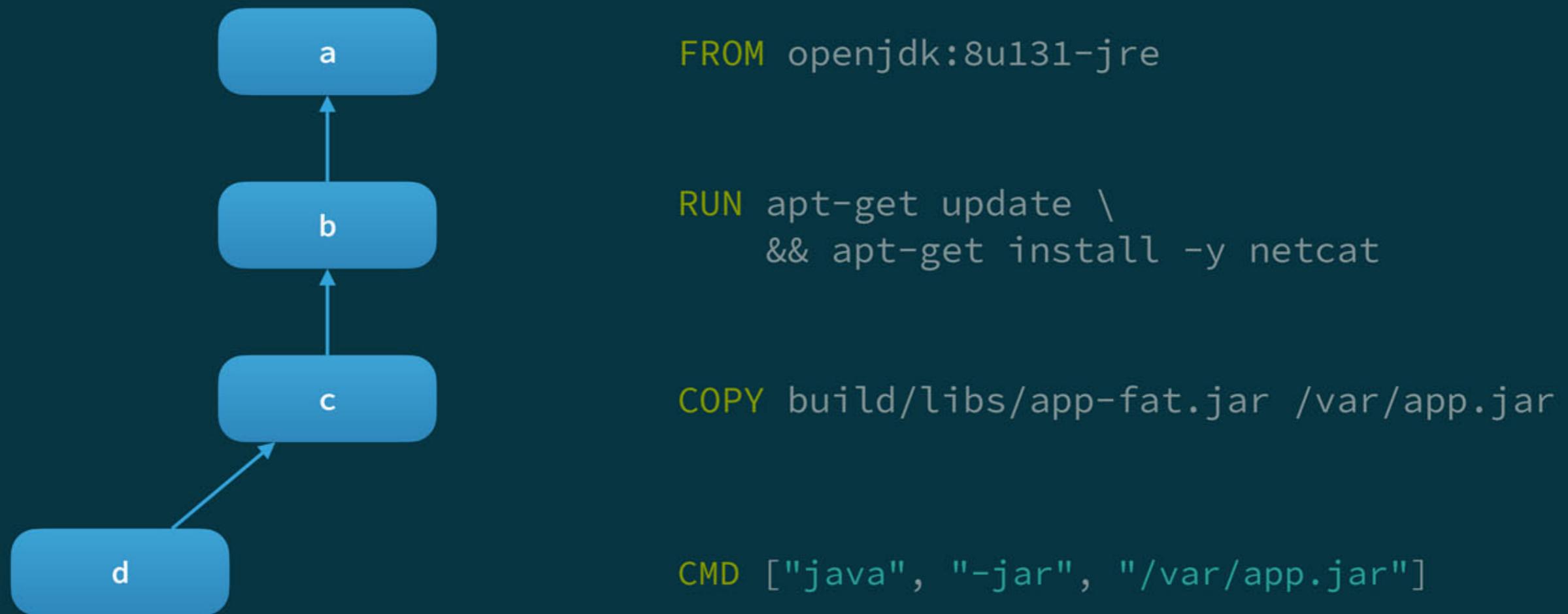
COPY build/libs/app-fat.jar /var/app.jar

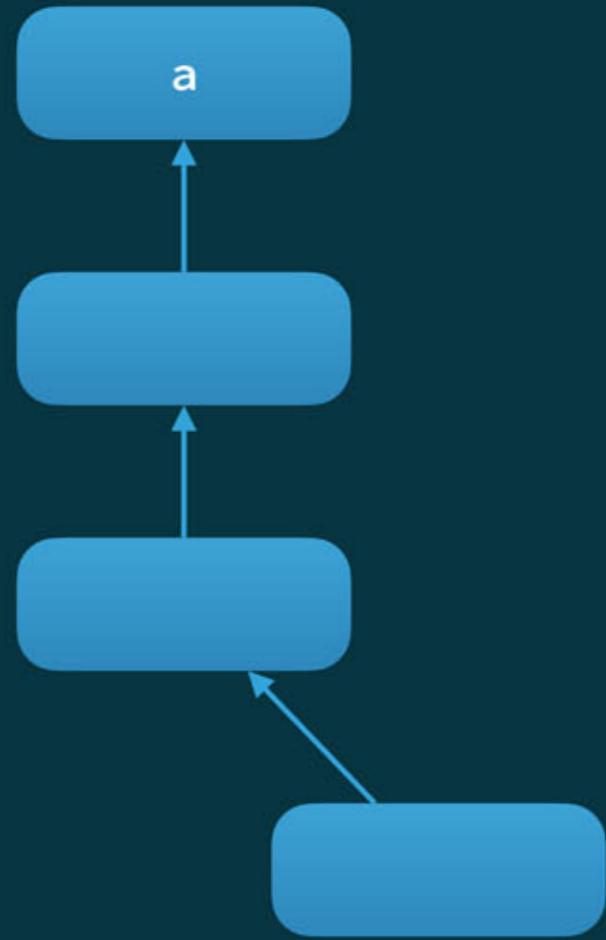
CMD ["java", "-jar", "/var/app.jar"]
```









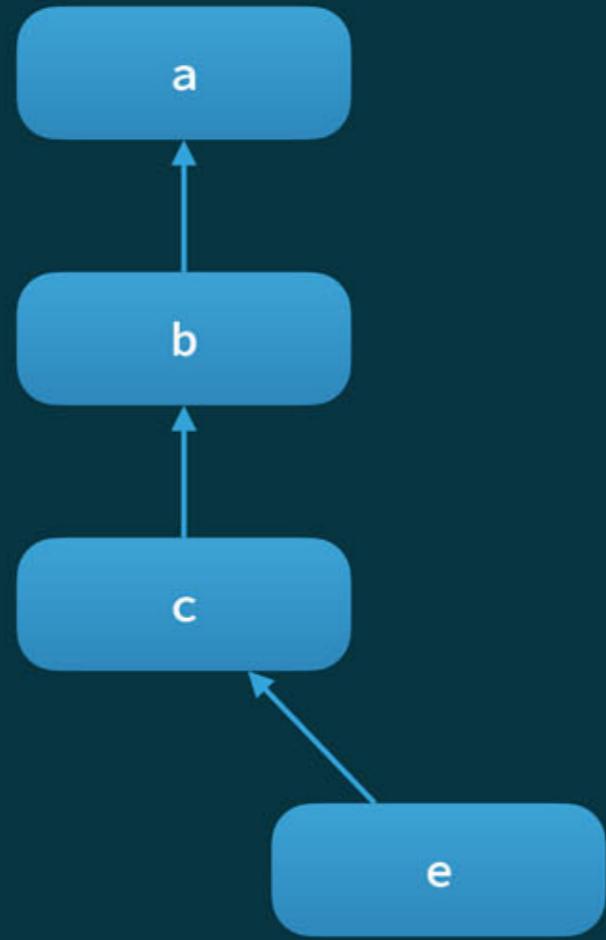


FROM openjdk:8u131-jre

RUN apt-get update \
 && apt-get install -y netcat

COPY build/libs/app-fat.jar /var/app.jar

CMD ["ls", "-al"]

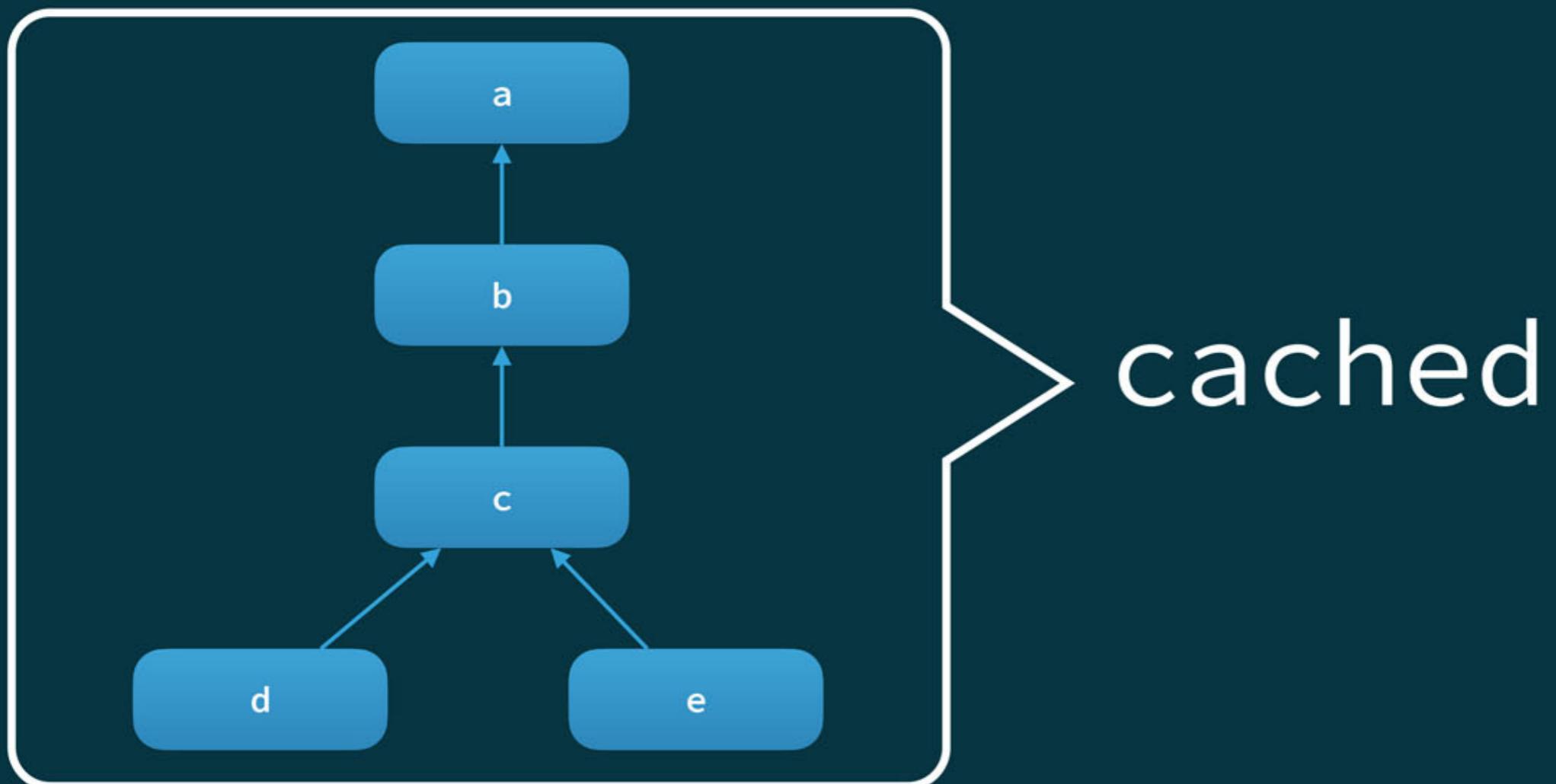


FROM openjdk:8u131-jre

RUN apt-get update \
 && apt-get install -y netcat

COPY build/libs/app-fat.jar /var/app.jar

CMD ["ls", "-al"]



@EXERCISE

EXERCISE

- Start with a simple Dockerfile – You can use one from your projects or use the "Dockerfile" in the repo you cloned for this workshop
- Build it, and pay attention to what "docker build" is doing
- Make some changes to the Dockerfile – reformat the file, introduce some white space, and see what causes the cache to be invalidated

HINTS

- There are many files in the repository you cloned. Use the one named "Dockerfile" if you want something to play with
- Try changing "**RUN** apt-get update" to "**RUN** apt-get update" (double space after "RUN" and "**RUN** apt-get update" (double space after apt-get)



```
#!/usr/bin/env bash

last=alpine:3.8
for i in `seq 200` ; do
    rm -f cid
    docker run --cidfile=cid $last touch file$i;
    docker commit `cat cid` tag$i;
    docker rm `cat cid`;
    last=tag$i;
done
```



```
#!/usr/bin/env bash

last=alpine:3.8
for i in `seq 200` ; do
    rm -f cid
    docker run --cidfile=cid $last touch file$i;
    docker commit `cat cid` tag$i;
    docker rm `cat cid`;
    last=tag$i;
done

# Error response from daemon: max depth exceeded
# Unable to find image 'tag125:latest' locally
```



```
FROM alpine:3.8

RUN touch file1
RUN touch file2
# 125 times more
RUN touch file127
```



```
FROM alpine:3.8

RUN touch file1
RUN touch file2
# 125 times more
RUN touch file127

# Error response from daemon: max depth exceeded
```





ONE CONCERN PER CONTAINER

@LOOSELYTYPED

THE THREE AMIGOS (SORTA)

- Build-time
 - FROM, ARG, RUN, ADD/COPY, USER
- Run-time
 - ENV, ENTRYPOINT/CMD, HEALTHCHECK
- Meta-data
 - LABEL, EXPOSE

FROM

NOTES

- Implies “ancestry”
- **Has** to be the first line (Except if preceded by ARG)
- Has implications on WORKDIR, USER, ENTRYPOINT (and CMD), and ONBUILD, EXPOSE and other commands
 - Use "docker inspect" for this
- Has **HUGE** implications on the final size of your image
- Create a base image with FROM scratch

DO'S



- Pin down the exact tag (or even better the digest)
 - Do not use “latest” tag
- Choose your parent image wisely
 - Vet it!
 - Inspect ancestor images for USERS, PORTs, ENVs, VOLUMEs, LABELs and anything that can be inherited
- Most likely you will build the lineage yourself internal to your team and organization

```
# Don't
FROM alpine

# Do
# Pin the version
FROM alpine:3.8
# OR Use ARG to set it at build time
ARG version=3.8
FROM alpine:${version}
```

@EXERCISE

EXERCISE

- Start with a Dockerfile (If you don't have one, then use Dockerfile in the repository you cloned)
- Build it without supplying a version – Inspect "docker image ls <image-name>" and ensure it has the latest tag
- Make a change to the Dockerfile and now **build it with a version**. Inspect "docker image ls <image-name>"
 - Is the "latest" really the latest?
- Why is FROM a "build time" instruction?

HINTS

- docker build -t demo -f Dockerfile . # build an image with "latest"
- docker build -t demo:1.1.1 -f Dockerfile . # build with a version

ARG

DO'S



- Use ARG for configuring image construction at build times
- Use ARGs to supply dynamic parameters to FROM, ENV, LABEL and RUN
- Default them appropriately

```
# Do
# Use default value
ARG AUTHOR="Raju Gandhi"
ARG BUILD_DATE
ARG VCS_REF

# default author
docker build -t test .
# set at build time
docker build --build-arg AUTHOR="Solomon Hykes" -t test .
```

@EXERCISE

EXERCISE

- Start with the following Dockerfile (Use Dockerfile.sample)

```
FROM alpine:3.12
RUN cat /etc/alpine-release
```

- Introduce a VERSION ARG (with a default of 3.9) and use it in the FROM instruction
- Build the image twice, once with defaults, and once supplying it with an build arg (of 3.9)
- Pay attention to which step invalidates the cache
- Why is ARG a "build time" instruction?

HINTS

- ARG SOME_ARG=VALUE
- To use an ARG use \${SOME_ARG} syntax
- docker help build

RUN

DON'TS



- Do not do OS level upgrades (eg. RUN `dist-upgrade`)

DOS



- Be cognizant of the effects (and drawbacks) of caching
- Group logical operations
 - Clean up as well (reduces image sizes) after installing libraries, unzipping tarballs etc
- Use multiline (\) to make PR / auditing easier
 - Or use the newer "heredoc" syntax (Currently experimental)

```
# Don't
RUN apt-get update
RUN apt-get install -y netcat=1.217-3ubuntu1
RUN apt-get clean

# Do
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
    netcat=1.217-3ubuntu1 \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*
```

EXERCISE

- Why SHOULDN'T you do this?

```
RUN apt-get update \
&& apt-get install -y --no-install-recommends \
netcat=1.217-3ubuntu1
```

```
RUN apt-get clean \
&& rm -rf /var/lib/apt/lists/*
```

@EXERCISE

EXERCISE

- Start with the following Dockerfile (Use Dockerfile.sample)

```
FROM fedora:33  
RUN dnf install -y git
```

- Build this image and inspect it's size
- Follow the instructions in <https://www.redhat.com/sysadmin/tiny-containers> (Step 1) to clean up after installation
- Build the image again, and compare the two sizes
- Why is RUN a "build time" instruction?

HINTS

- docker image ls <image_name>

```
# syntax=docker/dockerfile:1.4

# sample Dockerfile
FROM ubuntu:21.10

RUN <<EOF
apt-get update
apt-get install -y --no-install-recommends \
netcat=1.217-3ubuntu1
apt-get clean
rm -rf /var/lib/apt/lists/*
EOF
```

```
# syntax=docker/dockerfile:1.4

# sample Dockerfile
FROM nginx

COPY <<EOF /usr/share/nginx/html/index.html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Using Heredocs in Docker</title>
  </head>
  <body>
    <h1>What !! </h1>
  </body>
</html>
EOF
```

ADD/COPY

.DOCKERIGNORE

```
# example .dockerignore file
# ignore these folders
.git
build
!build/libs/*.jar # but NOT this

# ignore these files
.project
.gitignore
.dockerignore

# ignore all Docker files
Dockerfile*
docker-compose.yml

# ignore all markdown files (md)
*.md
```

DON'TS



- Avoid ADD
- Do not leave “residual” artifacts
- Be wary of using the array syntax
- Do not copy over all source in one fell swoop
 - Copy over source files separately and later on since they change often

DO'S



- Instead of ADD
 - Combine COPY and RUN
 - OR RUN with wget/curl/tar/unzip
 - See DO'S under RUN
- Be mindful of what you put in the .dockerignore file

@EXERCISE

EXERCISE

- Clear out the `.dockerignore` file in the repository you cloned
- Using `Dockerfile.sample`, `COPY` everything from the root directory into the container
- Build an image from `Dockerfile.sample` paying close attention to the build context size and the time it takes to build
- Now put back the contents of the `.dockerignore` file and repeat the previous exercise – Compare build context sizes and times
- Why are ADD/COPY "run-time" instructions?

HINTS

- It's often a good idea to create a `WORKDIR` – See `Dockerfile.multi` for an example
- Use `COPY . .` to copy everything
- In bash you can time a command by using "`time docker build ...`"

USER

DON'TS



- Do not switch USER often
- Avoid using root

DO'S



- Create a user (if you can) for your service
- Default the container to a non-root user if you can

```
FROM ubuntu:21.04

# Do
RUN groupadd -r app \
    && useradd -r -g app appuser
USER appuser
```

ENV

DON'TS



- Put secrets or sensitive information in ENV variables
- Override parent image ENV's unless absolutely necessary

DO'S



- Use them for documentation and modifying runtime behavior
 - They are baked in the final image
- Use docker run <image-name> env
 - Or docker inspect
- Default then appropriately
- Be cognizant of inherited ENV variables

```
ARG PROJECT_VERSION
# Do
# Default them if set dynamically
ENV PROJECT_VERSION ${PROJECT_VERSION:-2.3}
```

@EXERCISE

EXERCISE

- Start with the following Dockerfile (Use Dockerfile.sample)

```
FROM alpine:3.12
```

- Introduce a few ARGs, and a few ENV
 - Be sure to have at least one ENV use a supplied ARG value
- Build the image, start a container and ensure that you see the ENVs in the container
- Why is ENV a "run-time" instruction?

HINTS

- To set an ENV to an ARG value use "ENV ENV_NAME \${ARG_NAME}"
- You can inspect the envs in a container using "docker run <image_name> env"

ENTRYPOINT/CMD

NOTES

- CMD can be overridden if the user supplies an argument to create or run
- You can (also) override ENTRYPPOINT by explicitly supplying --entrypoint flag
- The default command run in a container is (ENTRYPOINT + CMD)

DON'TS



- Avoid the “shell” form

DO'S



- Use the “exec” form
 - Shell expansion will **not** happen!
- Use ENTRYPPOINT and CMD together
- Use a “entrypoint-script”
 - Allows you to set error (-e) flags and traps
 - Your editor is syntax aware
 - Always “exec”

DO'S



- Make the ENTRYPPOINT be your command
- CMD should be the arguments to the (ENTRYPPOINT) command
- This means your consumers simply pass arguments to your command (docker run image --help)

```
# Do
# Use ENTRYPOINT and CMD together
ENTRYPOINT [ "echo", "hello" ]
CMD [ "world" ]

# > docker build -t entrypoint-cmd .
# > docker run entrypoint-cmd
# hello world
# > docker run entrypoint-cmd raju
# hello raju
```

```
# Do
COPY entrypoint.sh /usr/local/bin/
ENTRYPOINT ["entrypoint.sh"]
CMD ["default"]
```

```
# entrypoint.sh
if [ "$1" = 'default' ]; then
    # do default thing here
    echo "Running default"
    # exec your-default-command
else
    echo "Running user supplied arg"
    # if the user supplied say /bin/bash
    exec "$@"
fi
```

@EXERCISE

EXERCISE

- Start with the following Dockerfile (Use Dockerfile.sample)

```
FROM alpine:3.12
```

- Introduce an ENTRYPOINT (like "ls") and a CMD (like "bin"). Then build an image using this Dockerfile
 - Start the container with no args and observe the behavior
 - Start another container supplying it an argument (like "etc")
 - Start another container supplying it some random word (like "nfjs")
 - Explain this behavior
- Why is ENTRYPOINT and CMD "run-time" instructions?

HINTS

- Alternatively you can use ENTRYPOINT "echo" and CMD "Hello"

@EXERCISE (BONUS)

EXERCISE

- Inspect `080-Dockerfile.entrypoint.sample` in your project repository
 - Can you explain how this works?
 - Can you see any application of this pattern in one of your work projects?

HEALTHCHECK

DON'TS



- Be too aggressive with --interval period
 - Especially if the check itself is expensive
- Use external tools (like curl) if you can

DO'S



- Use a script that leverages the same runtime as your service
 - For example, if you have a node or go service, write a health check using the same runtime
- Be cognizant of the overhead the health check introduces
- Experiment with combinations of interval/timeout/retries

```
# Avoid
HEALTHCHECK CMD curl --fail http://localhost:5000/ || exit 1

# Do
COPY healthcheck ./healthcheck
HEALTHCHECK --interval=1s \
--timeout=1s \
--start-period=2s \
--retries=3 CMD [ "/healthcheck" ]
```

LABEL

DON'TS



- Define individual labels separately
- Leak secrets

DO'S



- Use them liberally
- Labels can see ARG variables. Use this!
 - BUILD_NUMBER, GIT_SHA
- Apply a standard convention to avoid conflicts
 - Build tooling on top of the conventions

```
ARG AUTHORS="Raju Gandhi"
ARG BUILD_DATE
ARG VCS_REF

# Don't
# LABEL org.opencontainers.image.authors=$AUTHOR
# LABEL org.opencontainers.image.created=$BUILD_DATE
# LABEL org.opencontainers.image.revision=$VCS_REF

# Do
LABEL org.opencontainers.image.authors=$AUTHORS \
      org.opencontainers.image.created=$BUILD_DATE \
      org.opencontainers.image.revision=$VCS_REF
```

@EXERCISE

EXERCISE

- Look over the "Build-time labels" in <http://label-schema.org/rc1/>
 - Can you think of any you would add to this list?
- Use "docker inspect" with any image on your computer and see if you can spot the labels?
- How would you categorize the "LABEL" instruction?

EXPOSE

DON'TS



- Avoid “docker run -P”

DO'S



- Document the ports your application needs exposed

COMPILE VS RUNTIME



```
FROM alpine:3.8
```

```
RUN apk add --update \  
tzdata \  
&& rm -rf /var/cache/apk/*
```

```
ARG TZ=America/Los_Angeles
```

```
RUN ln -snf \  
/usr/share/zoneinfo/$TZ \  
/etc/localtime && echo $TZ > /etc/timezone
```



```
FROM alpine:3.8
```

```
RUN apk add --update \  
tzdata \  
&& rm -rf /var/cache/apk/*
```

```
ARG TZ=America/Los_Angeles
```

```
ENV TZ $TZ
```

```
COPY entrypoint.sh /usr/local/bin/  
ENTRYPOINT ["entrypoint.sh"]  
CMD ["default"]
```

```
# entrypoint.sh  
#!/bin/sh  
ln -snf \  
/usr/share/zoneinfo/$TZ \  
/etc/localtime && echo $TZ > /etc/timezone
```

DO'S



- Use a combination of ARG and ENV with ENTRYPOINT/CMD
 - Allows you to express what is at "compile" time versus "runtime"
- ARG/ENV are important parts of your image and containers API

MULTI-STAGE BUILDS



```
#!/usr/bin/env bash

# — BUILD/TEST CODE —
# build/test our code
docker build -t build-img -f Dockerfile.build .

# — CREATE A TEMP DIRECTORY —
mkdir -p target

# — GET THE JAR OUT —
# start a temporary container from our "build-img"
# so we can get to its filesystem
docker create --name builder build-img
docker cp builder:/code/build/libs/docker-olp-0.0.1-SNAPSHOT.jar \
./target/

# — BUILD OUR FINAL IMAGE —
docker build -t final-img -f Dockerfile.final .
```

build



```
FROM openjdk:8u131-jdk as builder  
WORKDIR /code  
ADD . ./  
RUN ["./gradlew", "shadowJar", "--no-daemon"]
```

run



cp



build

```
FROM openjdk:8u131-jre  
RUN apt-get update \  
  && apt-get install -y --no-install-recommends \  
    netcat \  
  && apt-get clean  
COPY docker-workshop-0.0.1-SNAPSHOT-fat.jar \  
      /var/app.jar  
CMD ["java", "-jar", "/var/app.jar"]
```

NOTES

- Allow you to do everything using docker containers
- Separates build environment from runtime, with the net effect of leaner production images
- Keep secrets out of production images
- Image builds are much faster because docker can leverage the cache

```
# — Build/Test image —————
# Name the builder image "as builder"
FROM openjdk:11.0.5-jdk as builder

WORKDIR /code
COPY *.gradle gradle.* gradlew ./
COPY gradle ./gradle
RUN ./gradlew build || return 0

COPY src/ src/
RUN ["./gradlew", "-q", "--no-daemon"]

# — Final production image —————
FROM openjdk:11.0.5-jre

COPY --from=builder \
/code/build/libs/docker-olp-0.0.1-SNAPSHOT.jar \
/var/docker-olp-0.0.1-SNAPSHOT.jar

COPY entrypoint.sh /usr/local/bin/
ENTRYPOINT ["entrypoint.sh"]
CMD ["default"]
```


@EXERCISE

EXERCISE

- Study Dockerfile.multi in your project repository
 - What will you do to improve this? Think along the lines of ARGs, LABELs, additional stages

THANKS!!

RESOURCES

- Best practices for writing Dockerfiles
- Docker Registry V2
- Explaining Docker Image IDs
- Dockerfiles for reference
 - redis
 - Jenkins
 - Postgresql