Docker for Developers

Also maybe some Kubernetes (if we have time)

Docker for Developers

Also maybe some Kubernetes (if we have time)

Lewis Cowper

@lewis on FinLeap Slack

Let’s talk about containers

Containers are not real things

Namespaces & Cgroups

(These are all Linux-y and fun)

Namespaces

Control what a process can see

PID

IPC

Mount

User

Network

Cgroup

UTS

Cgroups

Control what a process can do

Memory	Cpuset
CPU	Devices
Blkio	Net_prio
Cpuacct	Freezer

A container is not virtualisation, it’s just kernel features.

All containers run on the base machine’s kernel, they have no kernel of their own.

Think of containers as processes

Let’s talk about Docker

A containerisation technology that offers process isolation.

Docker provides a runtime for your non-Linux machine to run containers locally (Docker for Mac/Windows).

Docker provides you a way of creating containers and building and running them (the Dockerfile).

Docker provides an abstraction layer to containerisation, written in Go, and called libcontainer. 👋 LXC.

This makes Docker very useful.

Docker images

Images contain some binary state, and layers of other images.

Images are the base for containers

Dockerfiles are basically scripts to run to tell Docker what is going on.

Dockerfile reference

FROM

RUN

COPY

CMD

WORKDIR

ARG

EXPOSE

ENV

ADD

LABEL

ENTRYPOINT

VOLUME

USER

ONBUILD

STOPSIGNAL

HEALTHCHECK

SHELL

Dockerfile reference

FROM

WORKDIR

RUN

COPY

ADD

CMD

ARG

EXPOSE

ENV

FROM

```
FROM mhart:alpine-node
```

```
FROM mhart:alpine-node as builder
```

WORKDIR

WORKDIR /usr/src/app

RUN

```
RUN apt-get update && apt-get install openssh
```

COPY

COPY . .

ADD

```
ADD http://example.com/important_file /
```

CMD

CMD node dist/server.js

EXPOSE

EXPOSE 1337

ENV

ENV TARGET dev

ARG

ARG NPM_TOKEN

Now you know enough to be dangerous (or productive).

#lifehack

```
docker rm $(docker ps -aq)
docker rmi $(docker images -q)
```

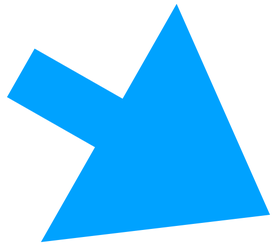
```
alias dockerrm='docker rm $(docker ps -aq)'

alias dockerrmi='docker rmi $(docker images -q)'
```

Let’s put that into practice

Single Stage Dockerfile

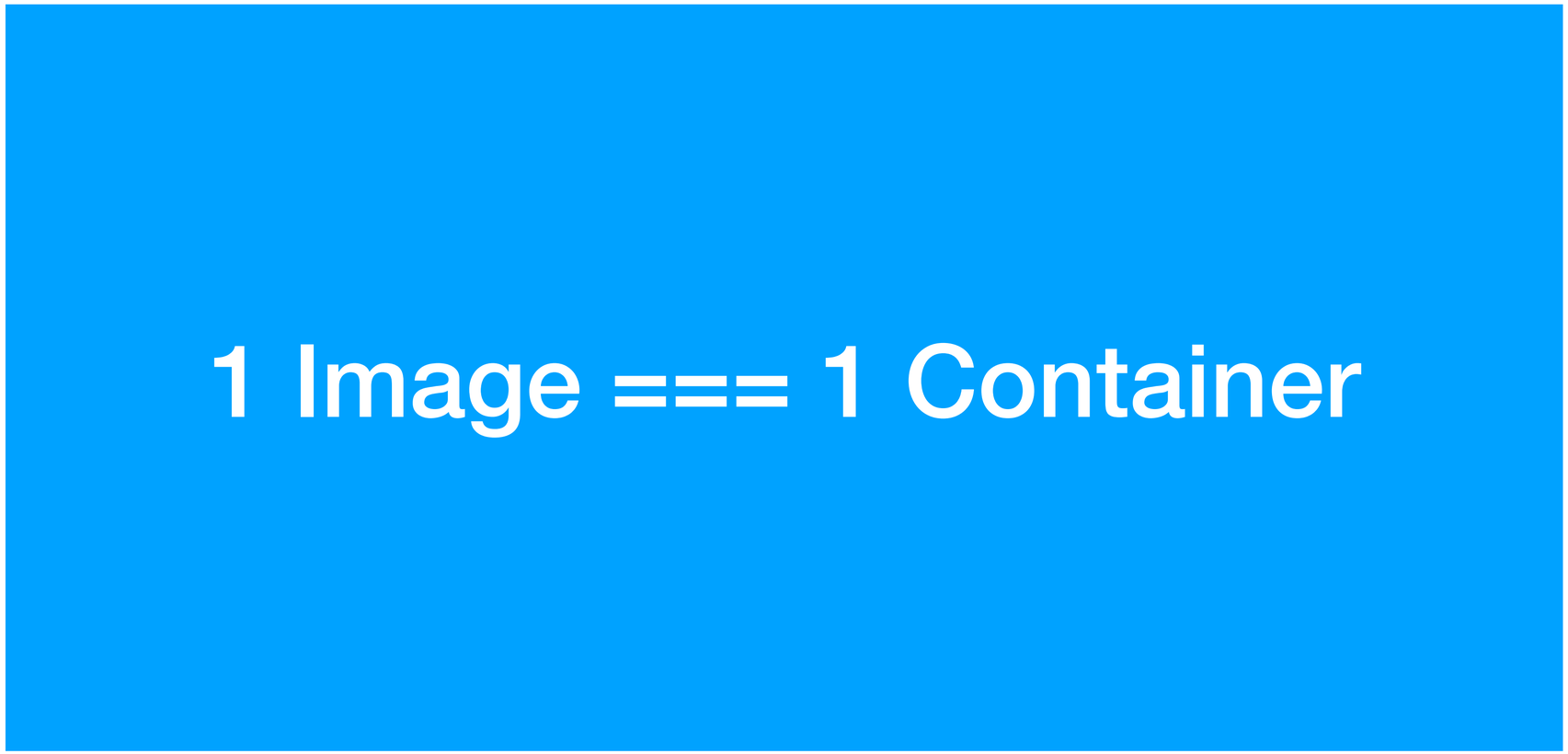
```
FROM rust:1-stretch
# Choose a workdir
WORKDIR /usr/src/app
# Copy sources
COPY . .
# Build app (bin will be in /usr/src/app/target/release/rust-lang-docker-multistage-build)
RUN cargo build --release
# Default command, run app
CMD /usr/src/app/target/release/rust-lang-docker-multistage-build
```

Single Stage Dockerfile



```
FROM rust:1-stretch
# Choose a workdir
WORKDIR /usr/src/app
# Copy sources
COPY . .
# Build app (bin will be in /usr/src/app/target/release/rust-lang-docker-multistage-build)
RUN cargo build --release
# Default command, run app
CMD /usr/src/app/target/release/rust-lang-docker-multistage-build
```

Single Stage Dockerfile



Final image size: 1.57GB

That’s a bit much

Let’s show off some magic

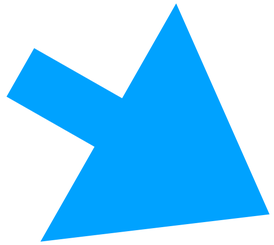
Multi-Stage Builds

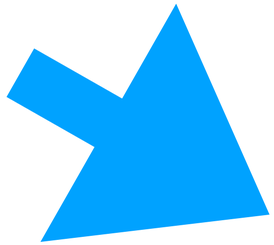
Use your full build environment for builds.
Get a much smaller container for releases.

Multi Stage Dockerfile

```
FROM rust:1-stretch as builder
# Choose a workdir
WORKDIR /usr/src/app
# Copy sources
COPY . .
# Build app (bin will be in /usr/src/app/target/release/rust-lang-docker-multistage-build)
RUN cargo build --release

FROM debian:stretch-slim
# Copy bin from builder to this new image
COPY --from=builder /usr/src/app/target/release/rust-lang-docker-multistage-build /bin/
# Default command, run app
CMD rust-lang-docker-multistage-build
```

Multi Stage Dockerfile





```
FROM rust:1-stretch as builder
# Choose a workdir
WORKDIR /usr/src/app
# Copy sources
COPY . .
# Build app (bin will be in /usr/src/app/target/release/rust-lang-docker-multistage-build)
RUN cargo build --release

FROM debian:stretch-slim
# Copy bin from builder to this new image
COPY --from=builder /usr/src/app/target/release/rust-lang-docker-multistage-build /bin/
# Default command, run app
CMD rust-lang-docker-multistage-build
```

Multi Stage Dockerfile

1 Image (used to build the base for the next image)

1 Image === 1 container (but now we don’t have unnecessary tools)

Let’s talk about layers for a moment

Dockerfiles are made up of read only layers, each of which represent a Dockerfile instruction.

The layers are stacked, and each one is a delta of the changes from the previous layer.

Building a container with a Dockerfile adds a final writeable layer, the “container layer”.

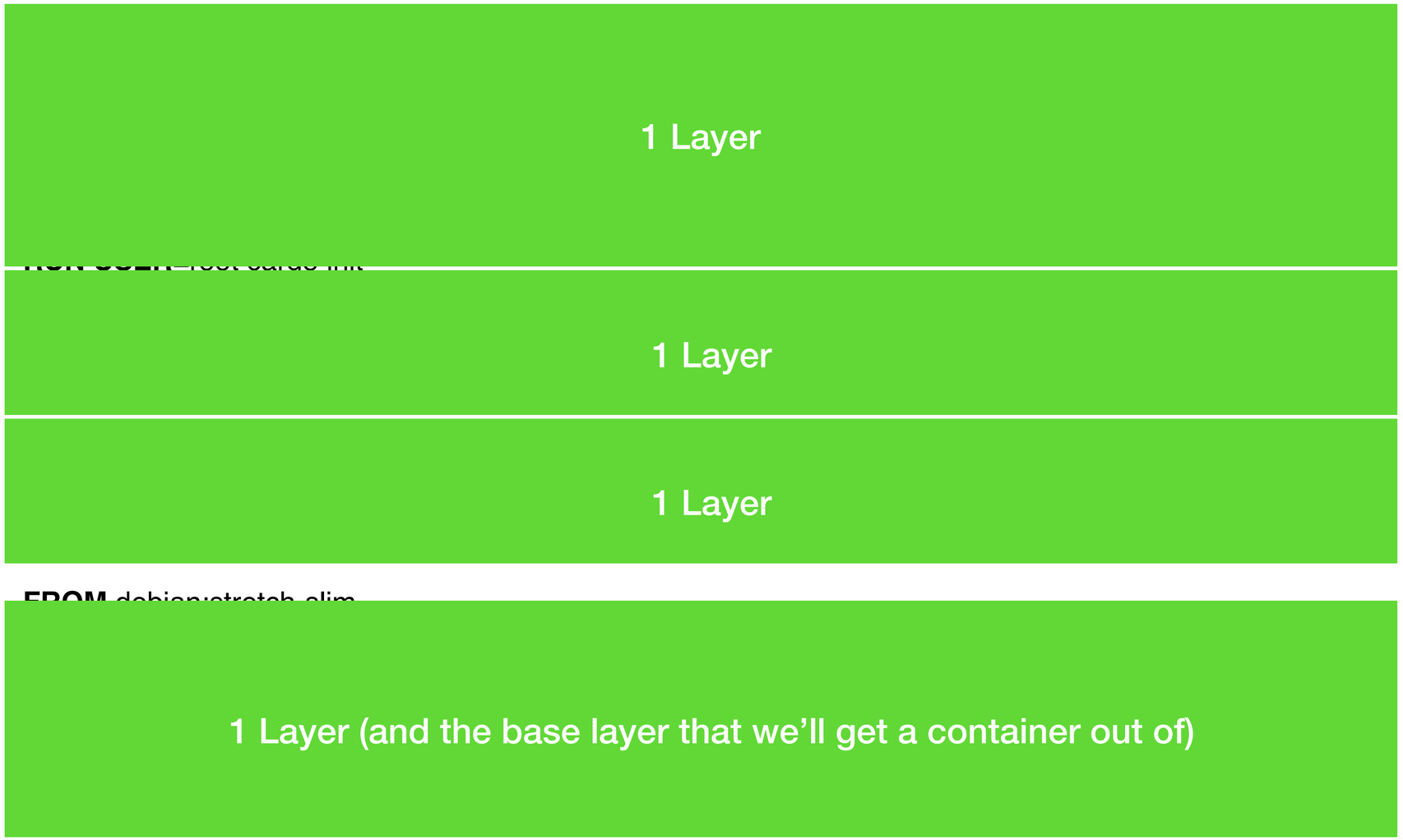
Each layer is cached, so successive builds with the same starting point can use the layers from cache.

Layer caching and speeding up builds

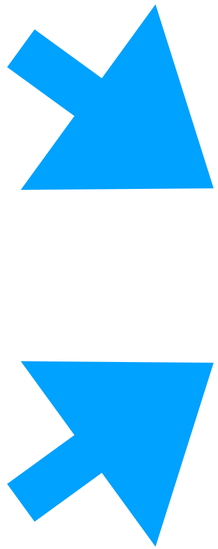
Multi Stage Dockerfile (Better Caching)

```
FROM rust:1-stretch as builder
# Choose a workdir
WORKDIR /usr/src/app
# Create blank project
RUN USER=root cargo init
# Copy Cargo.toml to get dependencies
COPY Cargo.toml .
# This is a dummy build to get the dependencies cached
RUN cargo build --release
# Copy sources
COPY src src
# Build app (bin will be in /usr/src/app/target/release/rust-lang-docker-multistage-build)
RUN cargo build --release

FROM debian:stretch-slim
# Copy bin from builder to this new image
COPY --from=builder /usr/src/app/target/release/rust-lang-docker-multistage-build /bin/
# Default command, run app
CMD rust-lang-docker-multistage-build
```

Multi Stage Dockerfile



Multi Stage Dockerfile



```
FROM rust:1-stretch as builder
# Choose a workdir
WORKDIR /usr/src/app
# Create blank project
RUN USER=root cargo init
# Copy Cargo.toml to get dependencies
COPY Cargo.toml .
# This is a dummy build to get the dependencies cached
RUN cargo build --release
# Copy sources
COPY src src
# Build app (bin will be in /usr/src/app/target/release/rust-lang-docker-multistage-build)
RUN cargo build --release

FROM debian:stretch-slim
# Copy bin from builder to this new image
COPY --from=builder /usr/src/app/target/release/rust-lang-docker-multistage-build /bin/
# Default command, run app
CMD rust-lang-docker-multistage-build
```

Final image size: 60.5MB

Smaller images mean: less costs

Smaller images mean: less security risks

Smaller images mean: less potential for runtime errors

#lifehack

docker exec -it <container_id> sh

Combining Docker Containers

Composing Docker Containers

docker-compose

docker-compose.yml

```
version: '3'
services:
  db:
    image: postgres
    volumes:
      - ./tmp/db:/var/lib/postgresql/data
  web:
    build: .
    command: bundle exec rails s -p 3000 -b '0.0.0.0'
    volumes:
      - .:/myapp
    ports:
      - "3000:3000"
    depends_on:
      - db
```

Really useful in development, but you have to make sure things continue to run when deploying.

Orchestrating Docker containers

Kubernetes (k8s)

i18n, l10n, k8s, a11y, n7m

k8s

Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.

— **the Kubernetes website**

A system for automating deployment, scaling and management of containerized applications.

It’s an abstraction layer for running and scaling docker containers in clusters.

Clusters?

A coupled network of containers connected in such a way they can freely communicate with each other.

Kubernetes lingo bingo

Pods

Services

ReplicaSets

Deployments

Docker for Mac provides k8s as a feature

Have a little play with it.
Orchestrate some containers.

Kubernetes is pretty cool, but doesn’t help us too much with development. It’s really good for deployment though.

Whoa, that was a lot of info

Let’s summarise

Containers are not real things

Dockerfile reference

FROM

WORKDIR

RUN

COPY

ADD

CMD

ARG

EXPOSE

ENV

Multi-Stage Builds

```
docker rm $(docker ps -aq)
docker rmi $(docker images -q)
```

#lifehack

docker exec -it <container_id> sh

#lifehack

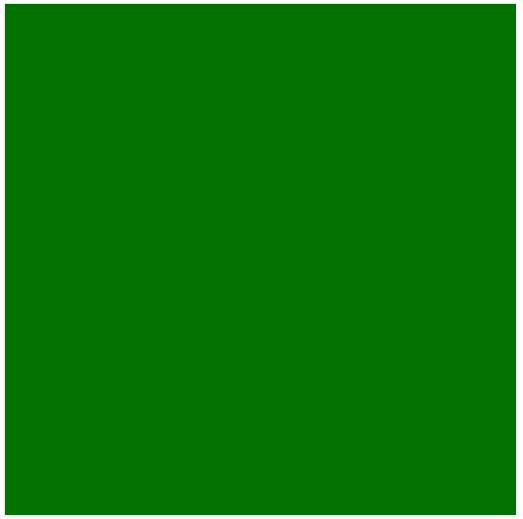
docker-compose

Kubernetes (k8s)



Containerise all the things





I’ve been Lewis, and you’ve been great. Thanks.