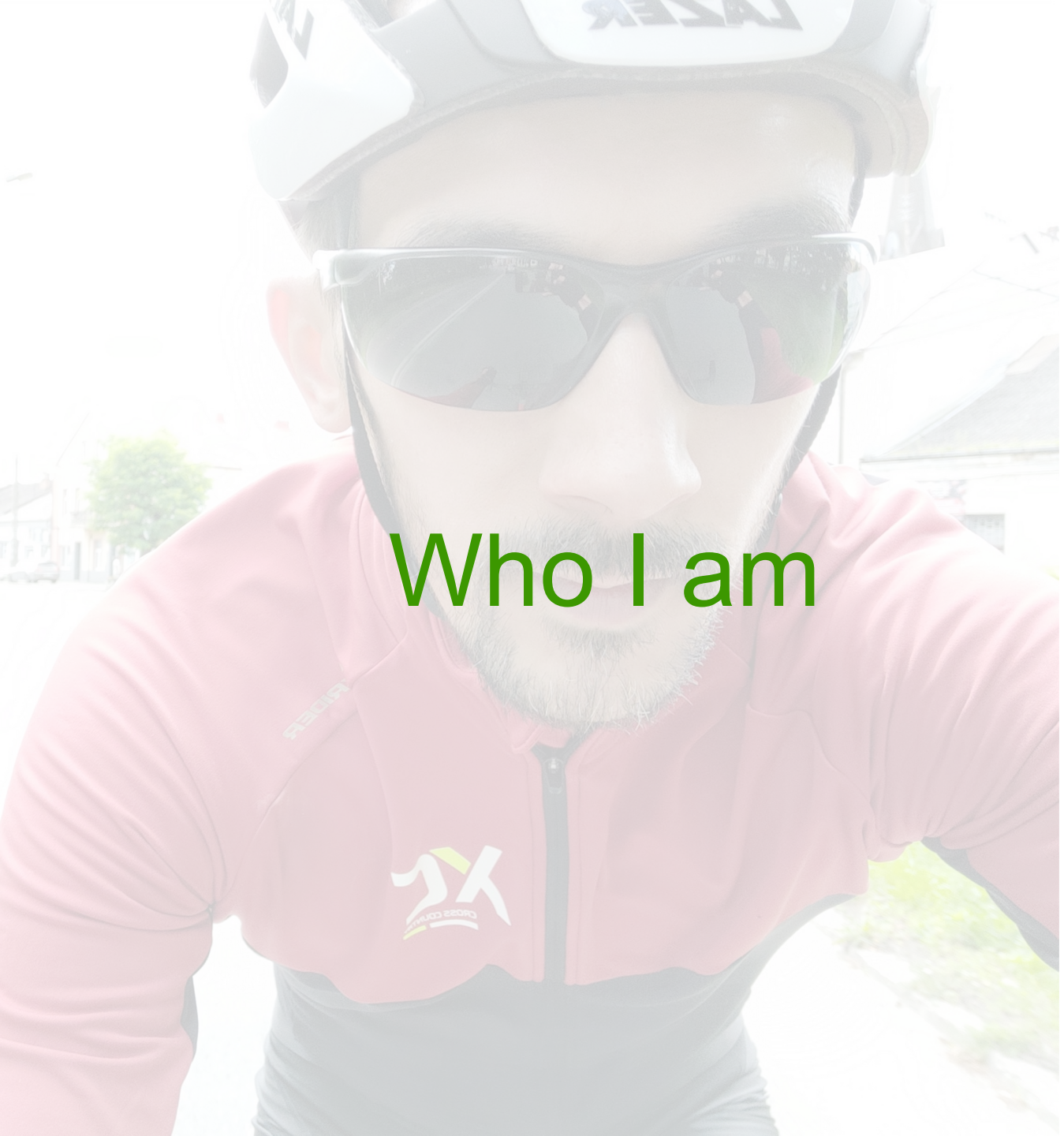
zooplus

Docker base images

Ideas how to manage them at scale

TOMASZ GĄGOR | PEOPLE LEAD/DEVOPS ENGINEER | DEVOXX 2022



Who I am

Tomasz Gągor

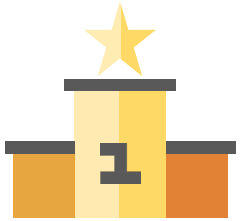
~15 years of experience in IT

Working with Docker for ~8 years

3+ years at **zooplus**

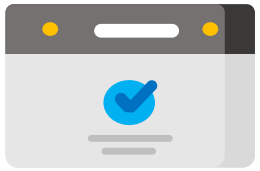
Part of “**C**ore **A**pplication & **T**echnology **S**ervices” (CATS) Team

zooplus



Nr 1

European online
pet supplies retailer



Business Model
implemented in

30

European countries

LOCATIONS

21 locations

around Europe

10 Entities / Offices

11 Fulfilment Centers

+ **1000** employees,

60 different nationalities

Over 9m

active customers

More than

8000

SKU

100k

Orders / Day

Growing from **1999**
across Europe

Who are you?

Agenda

What's the scale?

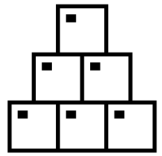
Pain points

This is how we do it

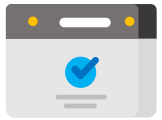
Q & A

So, what's the scale?

~ 40 product teams
Distributed around Europe



300+ products
On production



~ 1.500 (70%)
Self-developed Applications

400+ accounts
Dev + Prod

300+
Developers

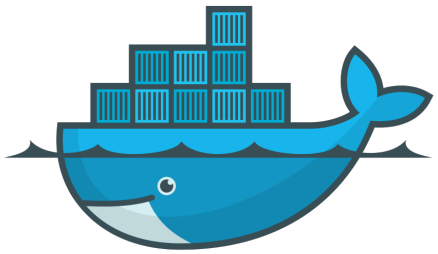
~ 150
Releases / Day

1mln
Technical Metrics

1bn
Logs / Day (20k / sec)

And Docker?

50+ base images

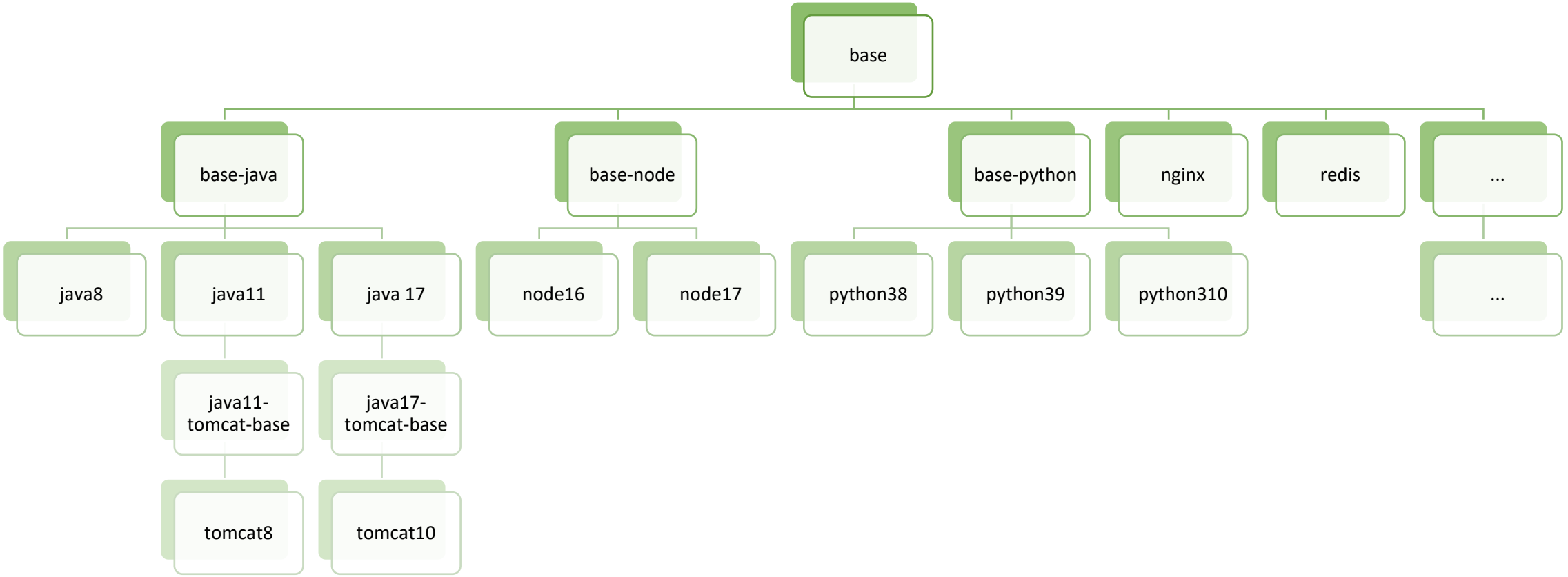


4 flavours

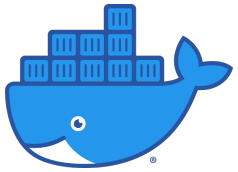
- unprivileged
- privileged
- rc.sh
- telegraf

200+ images on each build

Base images hierarchy



Our build stack



Docker in Docker

What a surprise…

Jenkins

~80% of apps runs in JVM, no surprise either

GNU Make

Sorry… What?

Why make?

It’s hard to run Jenkinsfile locally

It’s possible, docker-compose, Vagrant... But heavy

One command to rule them all

Our build command is dense – tags, labels, etc
But it’s easy to call: make 0000 or make 0000-centos8

Make handles a lot out of the box

It’s easy to add parallelism, to track file modifications or add new stage ex. security-scan and add it to test stage
Variable assignments with ?= are sooo COOL

```
make                # build all
make 0000           # build images from stage 0000
make 0100-centos8-java17      # just one image
make test           # prepare venv, test images
make clean          # clean test build dirties
make prune          # docker prune ...
```

```
.EXPORT_ALL_VARIABLES:
SHELL=/bin/bash
REPOSITORY_REMOTE ?= XXX.private.XXX/base
REPOSITORY_BUILD ?= base.local
BUILD_NUMBER ?= 0

# BUILD_DATE ?= $(shell date -u +"%Y-%m-%dT%H:%M:%SZ")
BUILD_DATE ?= $(shell date -u +"%Y-%m-%d")
GIT_COMMIT ?= $(shell git rev-parse HEAD)
GIT_BRANCH ?= $(shell git rev-parse --abbrev-ref HEAD)
GIT_URL ?= $(shell git config --get remote.origin.url)
MAINTAINER ?= "Technology Core Team <XXX@zooplus.com>"

DOCKER_IMAGES ?= $(shell find . -mindepth 1 -maxdepth 1 -type d -name "????-*" | grep -v "^\./\.git$$" | sed "s|^\
DOCKER_COMMON_TAGS ?= $(shell find tags -mindepth 1 -maxdepth 1 -type d | sed "s|^tags/||" | sort)
DOCKER_TAG ?= $(shell echo $(GIT_BRANCH) | sed -E 's/[/:]/-/g' | sed 's/main//' )
BUILD_STAGES ?= $(shell find . -mindepth 1 -maxdepth 1 -type d -name "????-*" | sed -E "s|.*([0-9]{4}).*|\1|" | so

DOCKER_HOST ?= unix:///var/run/docker.sock
DOCKER_CMD ?= docker -H $(DOCKER_HOST)

PYTEST_ADDITIONAL_PARAMS ?= --verbose

# will automatically use all CPUs for build
PARALLEL_JOBS ?= $(shell $(DOCKER_CMD) info -f "{{.NCPU}}")
```

```
all: build deprecate-images summary

# everything
build: $(BUILD_STAGES)

# here we build specific stages (like 0000, or 0100)
$(BUILD_STAGES):
    $(call stage_status,"Building",$@)
    $(MAKE) -j$(PARALLEL_JOBS) $(shell echo $(DOCKER_IMAGES) | xargs -n1 | grep $@)

# here we build specific images (like 0000-centos8)
$(DOCKER_IMAGES):
    $(call stage_status,"Building",$@)
    DOCKER_IMAGE_DIR=$@ && \
    DOCKER_IMAGE=$(shell echo $@ | sed -E 's#^(./)?[0-9]{4}-##') && \
    $(DOCKER_CMD) build \
        --label maintainer=$(MAINTAINER) \
        --label org.opencontainers.image.version=$(DOCKER_TAG) \
        --label org.opencontainers.image.source=$(GIT_URL) \
        --label org.opencontainers.image.revision=$(GIT_COMMIT) \
        --label org.opencontainers.image.branch=$(GIT_BRANCH) \
        --label org.opencontainers.image.created=$(BUILD_DATE) \
        --build-arg REPOSITORY=$(REPOSITORY_BUILD) \
        --squash \
        -t $(REPOSITORY_BUILD)/$$DOCKER_IMAGE \
        $$DOCKER_IMAGE_DIR || \
```

Agenda

What's the scale?

Pain points

This is how we do it

Q & A

State when we started

Root by default 😎

Many images used root by default

No proper versioning

Minor version change, but…
Breaking changes + few tool upgrades
Surprises, surprises…

Irregular updates

Upgrades happen accidentally, usually when someone was changing stuff and decided to upgrade this and that

Writing Dockerfiles is easy (but tricky)

No need to read the docs

I will just change this bash script and install a package… Added 5 COPY and 3 RUN steps

Layers easily collect garbage

You add 100MB of files with COPY/ADD then change permissions and end up with image bigger by 200MB

No metadata

From which branch/commit/app version this image was build? Who knows…

Does anyone recognize these issues?

Follow “Best practices” – they say…

Best practices for writing Dockerfiles

Estimated reading time: 33 minutes

This document covers recommended best practices and methods for building efficient images.

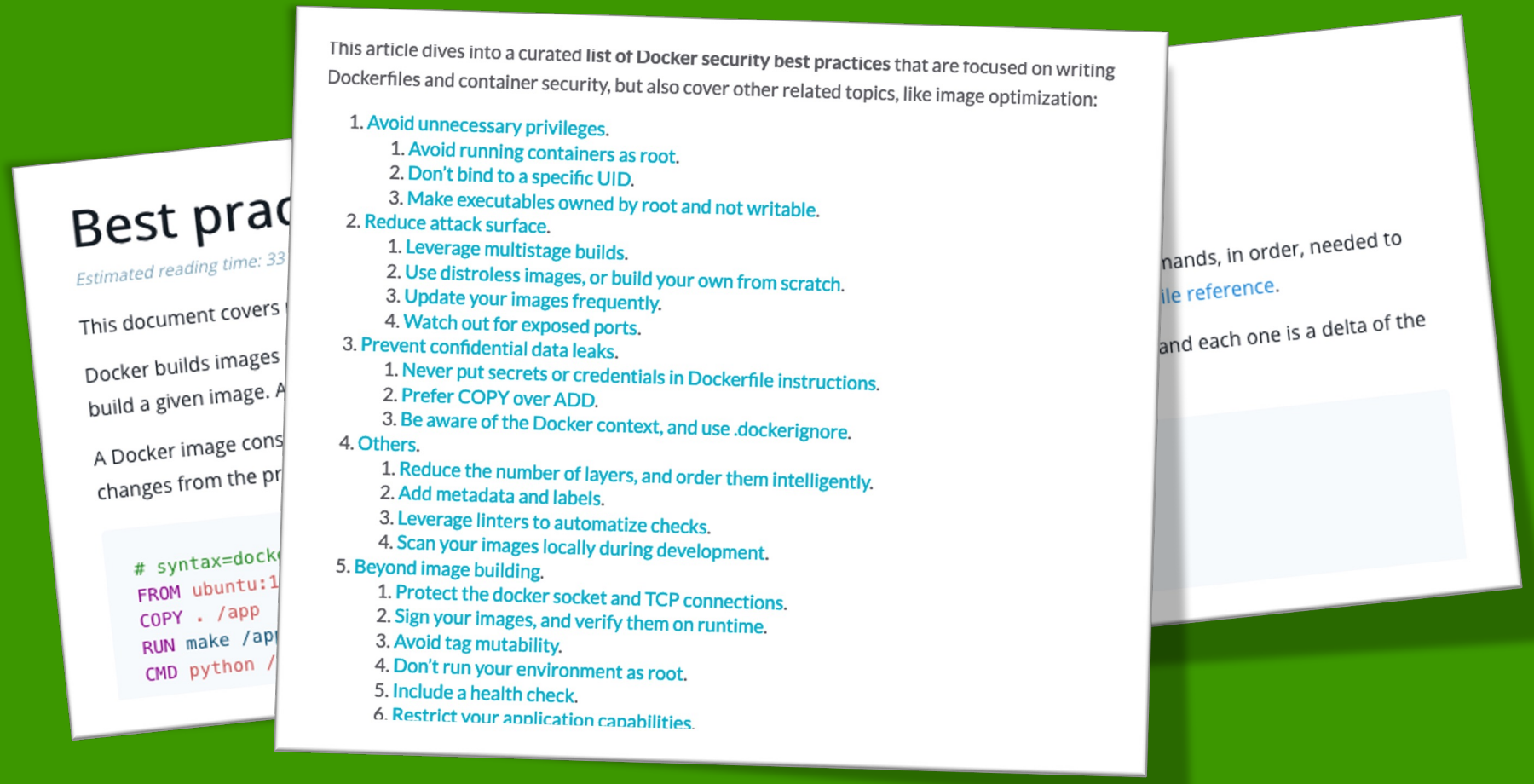
Docker builds images automatically by reading the instructions from a `Dockerfile` -- a text file that contains all commands, in order, needed to build a given image. A `Dockerfile` adheres to a specific format and set of instructions which you can find at Dockerfile reference.

A Docker image consists of read-only layers each of which represents a Dockerfile instruction. The layers are stacked and each one is a delta of the changes from the previous layer. Consider this `Dockerfile`:

```
# syntax=docker/dockerfile:1
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Follow “Best practices” – they say…



https://sysdig.com/blog/dockerfile-best-practices/

Follow “Best practices” – they say…

https://testdriven.io/blog/docker-best-practices/

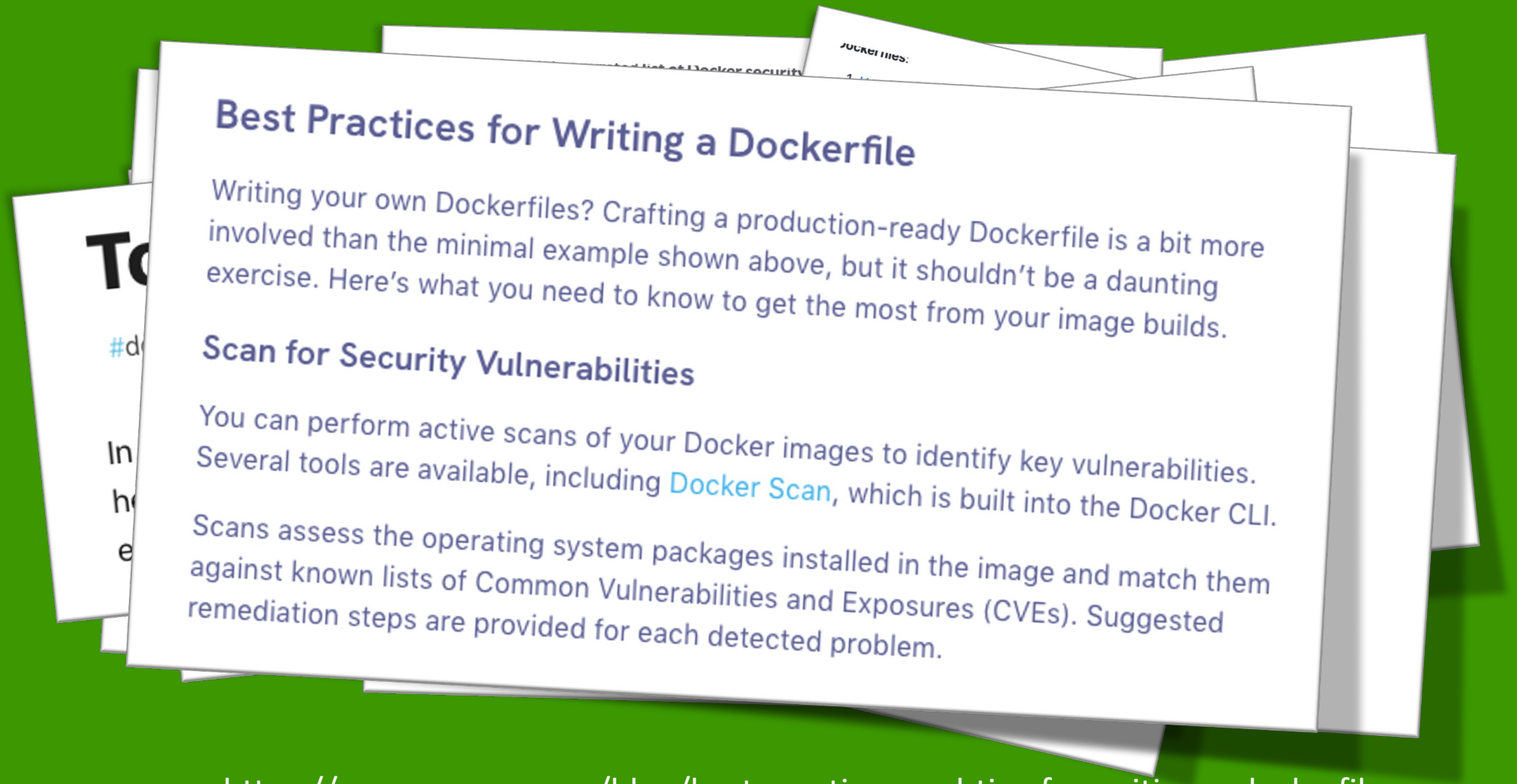
Follow “Best practices” – they say…

https://dev.to/techworld_with_nana/top-8-docker-best-practices-for-using-docker-in-production-1m39

Follow “Best practices” – they say…

https://dev.to/karanpratapsingh/top-5-docker-best-practices-57oh

Follow “Best practices” – they say…



Best Practices for Writing a Dockerfile

Writing your own Dockerfiles? Crafting a production-ready Dockerfile is a bit more involved than the minimal example shown above, but it shouldn’t be a daunting exercise. Here’s what you need to know to get the most from your image builds.

Scan for Security Vulnerabilities

You can perform active scans of your Docker images to identify key vulnerabilities. Several tools are available, including Docker Scan, which is built into the Docker CLI.

Scans assess the operating system packages installed in the image and match them against known lists of Common Vulnerabilities and Exposures (CVEs). Suggested remediation steps are provided for each detected problem.

https://www.qovery.com/blog/best-practices-and-tips-for-writing-a-dockerfile

We really tried to follow them…

Agenda

What's the scale?

Pain points

This is how we do it

Q & A

Bye, bye to root, but…

People get used to the easiness of “root”

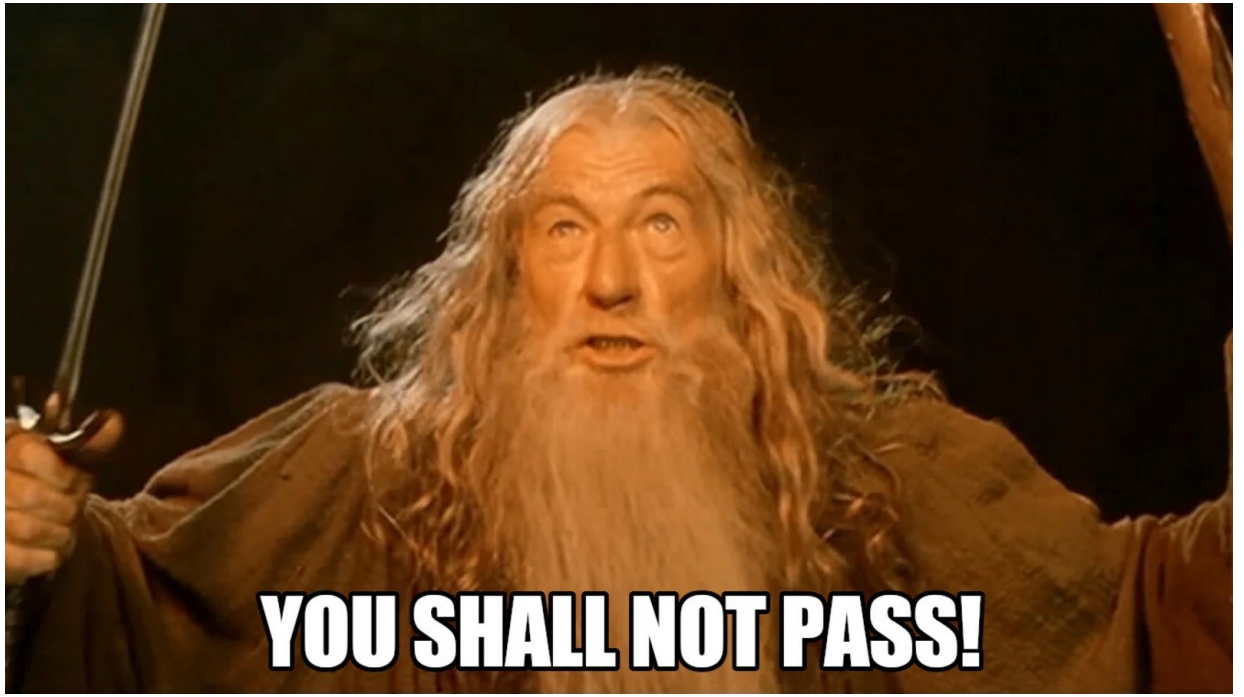
Just switching to USER will cause us a lot of issues

How do people use USER?

Think about JBoss or Tomcat

Gosu to the rescue

Unprivileged but more elastic



https://www.polygon.com/lord-of-the-rings/2021/11/10/22772790/lotr-you-shall-not-pass-ian-mckellen-gandalf

PRACTICE

Use Gosu instead of USER

OUTCOME

- *root* permissions during build
- Secure, unprivileged runtime
- Harder to break by our users than USER

Metadata – OCI Image Spec

```
BUILD_DATE ?= $(shell date -u +"%Y-%m-%d")
GIT_COMMIT ?= $(shell git rev-parse HEAD)
GIT_BRANCH ?= $(shell git rev-parse --abbrev-ref HEAD)
GIT_URL ?= $(shell git config --get remote.origin.url)
```

```
    $(DOCKER_CMD) build \
        --label maintainer=$(MAINTAINER) \
        --label org.opencontainers.image.version=$(DOCKER_TAG) \
        --label org.opencontainers.image.source=$(GIT_URL) \
        --label org.opencontainers.image.revision=$(GIT_COMMIT) \
        --label org.opencontainers.image.branch=$(GIT_BRANCH) \
        --label org.opencontainers.image.created=$(BUILD_DATE) \
        --build-arg REPOSITORY=$(REPOSITORY_BUILD) \
        --squash \
        -t $(REPOSITORY_BUILD)/$$DOCKER_IMAGE \
        $$DOCKER_IMAGE_DIR || \
```

PRACTICE

Metadata + SemVer versioning

OUTCOME

- You know how big the change is from the version number
- If you’re unsure what changed, it’s easy to spot

Use the “latest” tag

We encourage to use “latest”

We tell people to always use “latest” version of our images

Incremental changes

As our changes are usually small, the same apply to bugs – they’re usually easy to spot and fix

Provide versioned images

Not just providing SemVer versions but also strictly comply to the rules

What if the latest tag breaks stuff?

What if..?

We accept the risk that using “latest” tags might cause issues from time to time

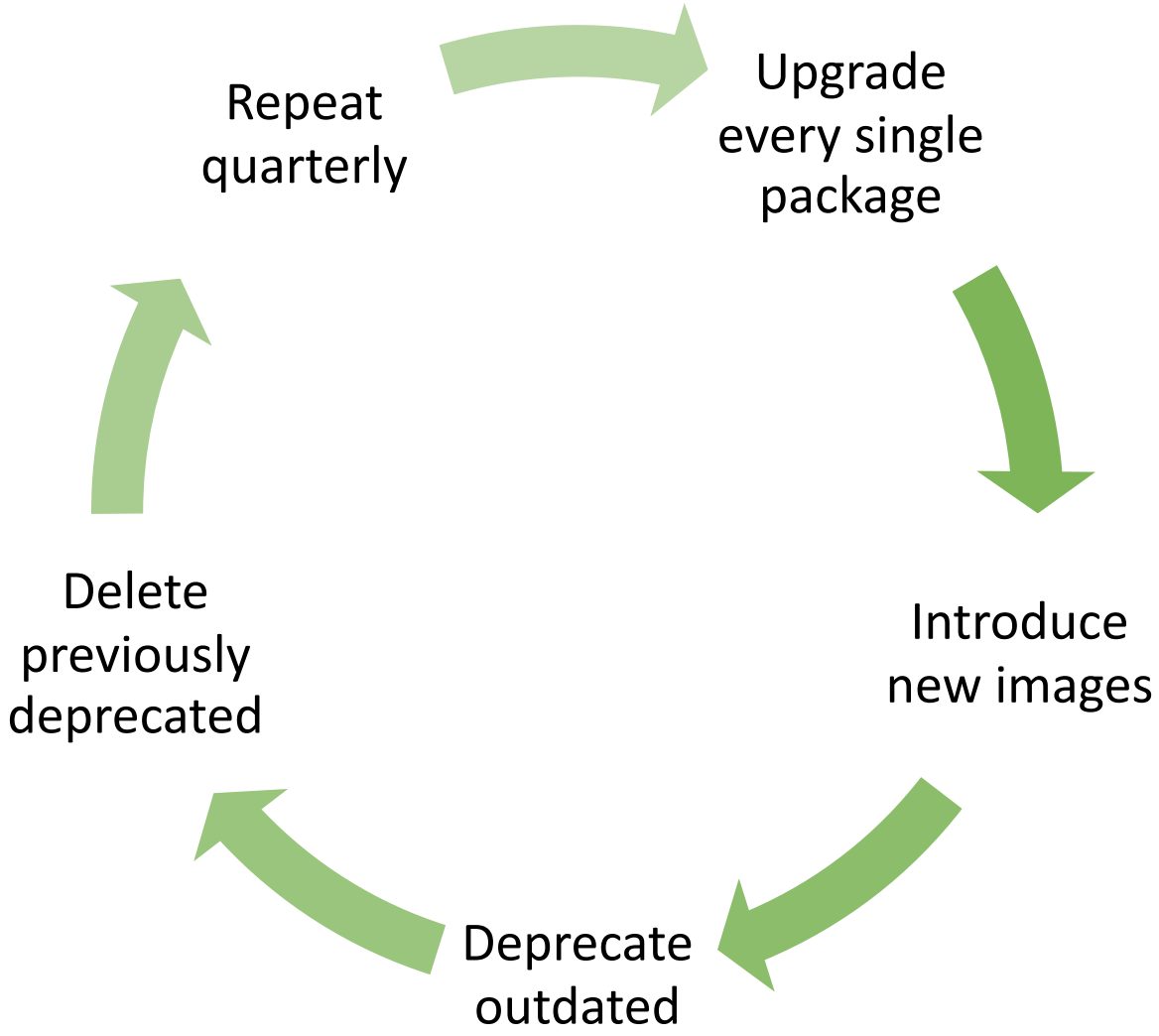
I can’t use latest

That’s ok - use what works for now…

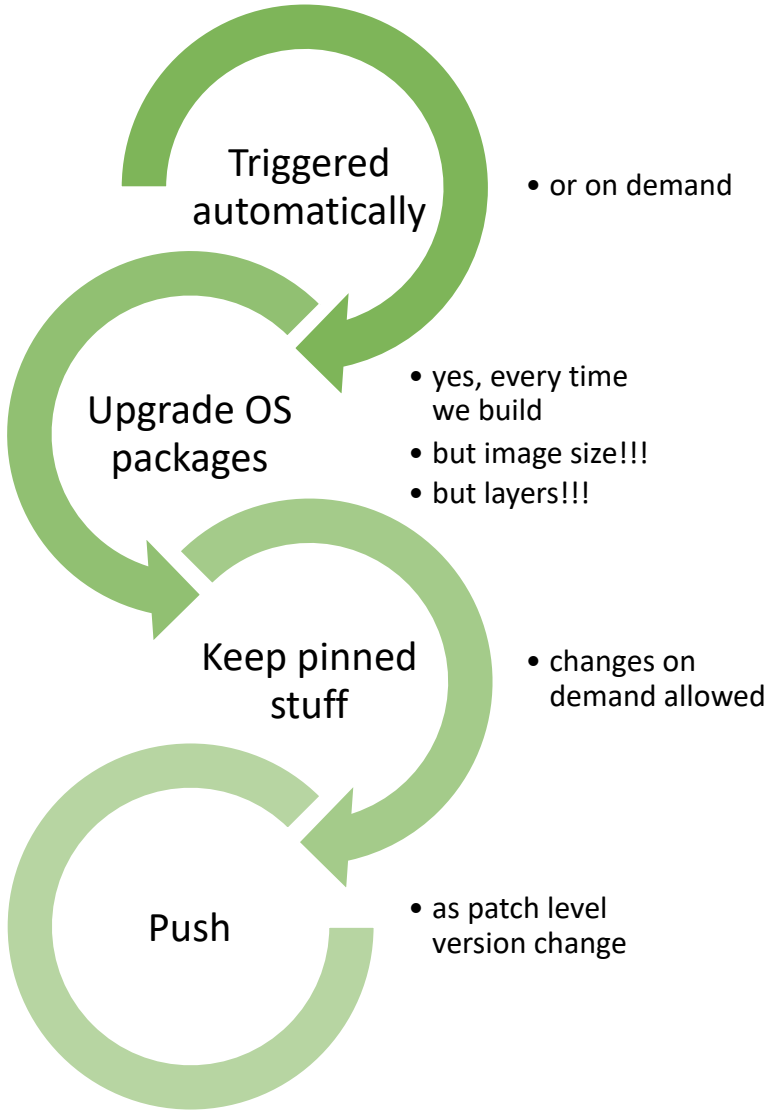
Pinning as a fall-back

If we break anything, you’re allowed to pin to an older version, **but always follow up with a fix soon!**

Quarterly Patch Day



Monthly upgrades



Don’t you push too hard?

Let them know

- short reminder
- at beginning of sprint
- predictions on what's coming

- what's new
- what's deprecated
  - marked for deletion for next QPD
- what was deleted

Communication is key

People don’t like surprises

When people know what to expect, it’s business as usual

We didn’t know…

Seriously? 2 emails withing 2 weeks, slack messages, etc.

We didn’t have enough time…

Usually, if you don’t progress in 3 months, 6 wouldn’t have changed anything

https://www.facebook.com/thedigitalprojectmanager/posts/good-communication-is-key-for-remote-teamsfind-out-how-to-create-a-project-commu/2920653528042580/

PRACTICE

Use *latest* by default + QPD
+ smaller upgrades

OUTCOME

- Updates propagate faster
- People get used to small, iterative changes
- We move forward, rather than backward

To cache or not to cache?

Build cache – no thank you!

It’s harder to keep images updated with build cache
In distributed environments it’s also harder to keep it in sync

Builds are slower

They are, but most of our app images are simple – they’re fast enough

It might be valuable, but…

It’s nice for NPM installs, huge package installs, etc.

To cache or not to cache?

Causing “it was working yesterday”…

Relying too much on build cache was causing issues
when cache was lost
Small incremental changes are easy to spot

Alternatives?

We prefer to cache things with Artifactory
or Jenkins archiveArtifacts

PRACTICE

Don’t use build cache

OUTCOME

- Easier to manage
- Less issues when cache is gone
- People use proper pinning to stabilize builds

Use VOLUMEs more!

Do you do this?

```
FROM ubuntu:22.04
VOLUME ["/tmp", "/var/tmp", "/var/cache/apt"]
```

```
FROM centos:8
VOLUME ["/tmp", "/var/tmp", "/var/cache/yum", "/var/cache/dnf"]
```

“

Changing the volume from within the Dockerfile: If any build steps change the data within the volume after it has been declared, those changes will be discarded.

https://docs.docker.com/engine/reference/builder/#volume

Without VOLUME

```
FROM debian
RUN apt-get update && \
    apt-get install -y nginx && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
```

With VOLUME

```
FROM Debian
VOLUME ["/tmp", "/var/tmp", "/var/cache/apt"]
RUN apt-get update && \
    apt-get install -y nginx
```

Layers, ehh, layers…

Too many layers hurt

Oh, really?
Download speeds are slower (but from our tests extraction faster)

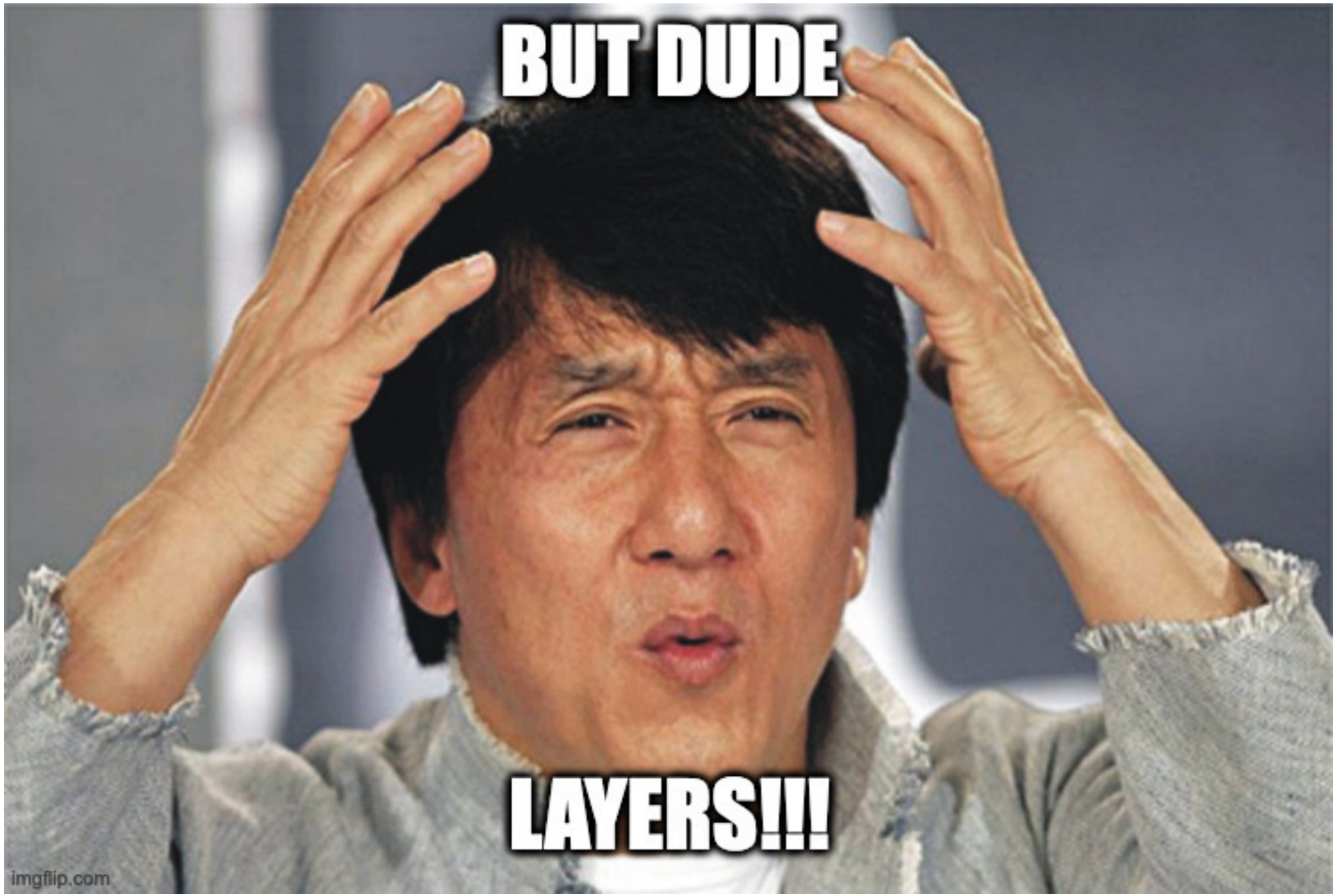
Reduce amount of RUN/COPY/ADD

For image size, better readability, performance…
hmm

```
FROM debian:bullseye-slim

LABEL maintainer="NGINX Docker Maintainers <docker-maint@nginx.com>"

ENV NGINX_VERSION   1.21.6
ENV NJS_VERSION     0.7.2
ENV PKG_RELEASE     1~bullseye

RUN set -x \
# create nginx user/group first, to be consistent throughout docker variants
    && addgroup --system --gid 101 nginx \
    && adduser --system --disabled-login --ingroup nginx --no-create-home --home /no
    && apt-get update \
    && apt-get install --no-install-recommends --no-install-suggests -y gnupg1 ca-ce
    && \
    NGINX_GPGKEY=573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62; \
    found=''; \
    for server in \
        hkp://keyserver.ubuntu.com:80 \
        pgp.mit.edu \
    ; do \
        echo "Fetching GPG key $NGINX_GPGKEY from $server"; \
        apt-key adv --keyserver "$server" --keyserver-options timeout=10 --recv-keys
    done; \
    test -z "$found" && echo >&2 "error: failed to fetch GPG key $NGINX_GPGKEY" && e
    apt-get remove --purge --auto-remove -y gnupg1 && rm -rf /var/lib/apt/lists/* \
    && dpkgArch="$(dpkg --print-architecture)" \
```

https://github.com/nginxinc/docker-nginx

We add RUN/COPY as many times as we need 😎


BUT DUDE
LAYERS!!!
imgflip.com

Layers, ehh, layers…

Does it really matter for base images?

They're usually already in the image cache as they're the base for many other images…

Too many layers, squash it!

We squash base images, so we don't care anymore ☺

```
cat /etc/docker/daemon.json
{
    "experimental": true
}
```

```
docker build --squash -t base/something .
```

Layers, ehh, layers…

After squashing, we end up with just 2 real layers

```
docker history base/ubuntu20
IMAGE          CREATED BY                                      SIZE
8c68691ed07c                                                   211MB
<missing>      /bin/sh -c #(nop)  LABEL org.opencontainers.…   0B
<missing>      /bin/sh -c #(nop)  LABEL org.opencontainers.…   0B
...
<missing>      /bin/sh -c #(nop)  VOLUME [/var/lib/apt/list…   0B
<missing>      /bin/sh -c #(nop)  ENV DEBIAN_FRONTEND=nonin…   0B
<missing>      /bin/sh -c #(nop)  CMD ["bash"]                 0B
<missing>      /bin/sh -c #(nop) ADD file:b83df51ab7caf8a4d…   72.8MB
```

Layers, ehh, layers…

Is squashing for everybody?

It makes sense only in a few situations, like base images
Or where the installation process produces a lot of rubbish (IBM WebSphere…)

Don’t do this at home

Seriously, measure gains over costs

Housekeeping

Squashing produces a lot of intermediate layers and impacts build times
Remember to clean after each build!

PRACTICE

Use VOLUME for temp file locations

+

Squash base images

OUTCOME

- Less rubbish in images
- Dockerfiles easy to read
- Just 1 layer
- Faster downloads

Security of images

- simple & fast
- up to date
- dedicated to Docker images

Scan

- whole platform
- specific images

Address issues

- as Security Team
- directly by dev teams

Who tests docker images?

Tests

Testinfra + pytest – our choice

It provides unit tests for configuration management tools like Ansible… But not only
Python based + supports Docker

Alternatives

Serverspec – but it's 
Ansible – I saw it, it's possible, doesn't feel like real tests

Tests

It supports Docker, right?

Supported: container as a test environment not a test case

Matching test case to container

We tried RegExps… Team didn’t like it… ¯_(ツ)_/¯
We used pytest “marks” to group tests
Dammit, we still need a config to match marks to images

Tests

Example code

Memcached tests

```python
@pytest.mark.memcached
@pytest.mark.parametrize(
    "container", **get_containers(in_loop=False, params=["-m", "128m"]), indirect=True
)
@pytest.mark.usefixtures("container")
class TestMemcached:
    def test_memcached_user(self, container):
        assert container.user("memcached").exists

    def test_memcached_command(self, container):
        assert container.run("memcached -V").succeeded

    def test_memcached_process_user(self, container):
        assert container.process.get(comm="memcached").user == "memcached"

    def test_memcached_listeners(self, container):
        # install prerequisites
        if not (container.package("net-tools") or container.package("iproute")):
            container.run("yum install -y net-tools iproute")

        assert container.socket("tcp://0.0.0.0:11211").is_listening

    def test_memcached_wrapper(self, container):
        assert container.file("/usr/local/bin/memcached-wrapper.sh").exists
        assert container.file("/usr/local/bin/memcached-wrapper.sh").mode == 0o755

    def test_memcached_wrapper_tuning(self, container):
        """
        As we're running memcached container with only 128MB of RAM,
        it should limit memcached memory to only 102MB, leaving some space for OS.
        """
        assert "-m 102" in container.process.get(comm="memcached").args
```

PRACTICE

Tests

OUTCOME

- Stability
- Confidence when moving forward
- Makes other practices work

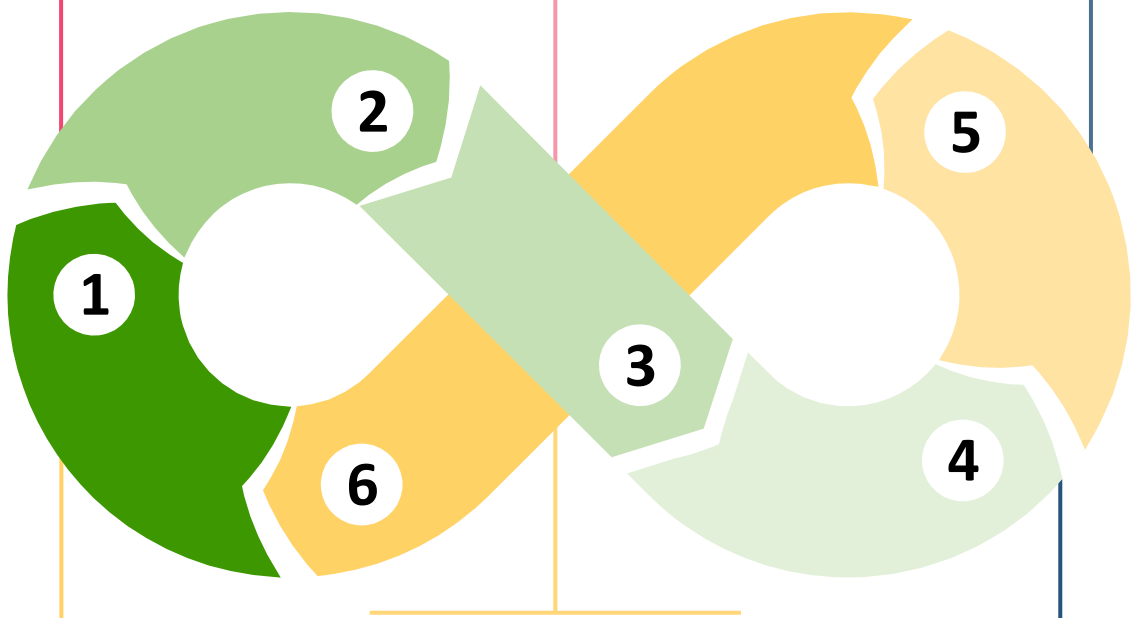
Don’t be afraid to challenge best practices!

A DevOps approach to “Best practices”

BEST PRACTICE
Promise of solution

PoC
Test implementation

ADAPT & SCALE UP
If it’s not perfect, make it such



PROBLEM
That’s where we usually start

REPEAT
One at a time

VALIDATION
Does it work for us?
(Measure!)

zooplus

Thank you!

TOMASZ GĄGOR | PEOPLE LEAD/DEVOPS ENGINEER | DEVOXX 2022

Q & A

linkedin.com/in/tgagor