Docker::For::Yaldi::Perl

docker build -t perl:yapc-tpcig .

Wesley Schwengle

- irc: slacker_nl
- CPAN: WATERKIP
- Mail: wesley@schwengle.net / waterkip@cpan.org

Work

- Mintlab BV
  - Zaaksysteem
    - Case management system for governments
    - EUPL
- Mail: wesley@mintlab.nl / wesley@zaaksysteem.nl

This presentation

- Docker
- Perl
- Javascript (well.. maybe)
- Gitlab CI

Why Docker?

At work

- Vagrant
- Installed on five servers
  - Excluding our test platform
  - Configured with Chef
- Different infrastructure
  - VMWare / Bare metal / AWS
- System perl (5.14) from Ubuntu

Problems

- Every OS has its own quirks in Vagrant
- Scaling required loads of work
- JS build was different on every single box
- Not ready for microservices
- Deployments took ages

Docker

- Docker
- Docker Compose

Docker

- Updated all libraries
- Latest Perl
- Every developer has the same box
- Mimic production setup (Docker Compose)
- Deploy to OpenStack (Gitlab CI / Kubernetes)
- Scales better
- Microservice ready

Dockerfile

- One container runs a single service
- Dockerfile describes what you do with a container
- Every command is a layer

```
FROM perl:latest          # layer 1
COPY . .                  # layer 2
RUN cpanm                 # layer 3
```

- No scripting in a Dockerfile
  - Use (shell) scripts instead

Layers

- Building blocks of docker images
- Each layer is read-only, except for the last layer
- `/var/lib/docker/{aufs,overlay2}/diff`
- Maximum amount of layers is around 198

Layers and images

```
# Build
docker build .
# View the images
docker image ls
# View the history
docker history <image>
# Tag a layer as an image
docker tag <image> <tag>
# Or tag it while building
docker build -t <tag> .
```

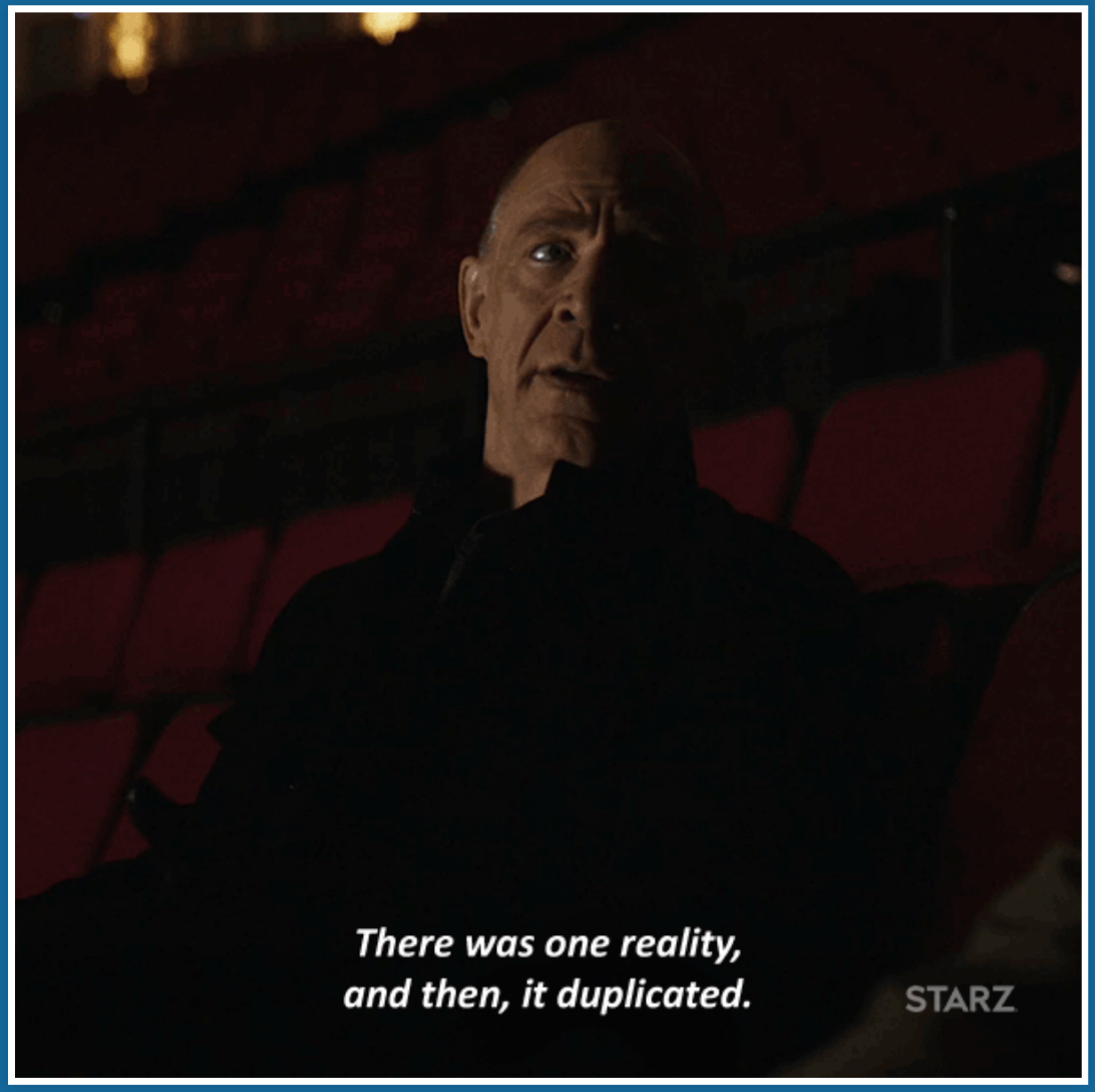
Containers

- Images are NOT containers
- Think of classes and instances
  - Class => image
  - Instance => container

Docker commands

- Similar to git
  - pull, push, tag, commit
- Others include:
  - build, inspect, container, image, etc
- Some can only be used when using Docker Swarm

Docker Compose


There was one reality,
and then, it duplicated.
STARZ

docker-compose.yml

- docker/Dockerfile.backend
- docker/Dockerfile.frontend
- docker-compose.yml

```
services:
  backend:
    build:
      context: "."
      dockerfile: "docker/Dockerfile.backend"
  depends_on:
      database
  frontend:
    build:
      context: "."
      dockerfile: "docker/Dockerfile.frontend"
  depends_on:
      backend
```

External images

External docker images are pulled in like so:

```
database:
  image: "postgres:9.6"
  ports:
    - "5432:5432"
```

Override the defaults

In case a developer needs some additional things he can override it via `docker-compose.override.yml`, eg a project that is outside of the repo:

```
  some-project:
    build:
      context: ../some-project
      dockerfile: Dockerfile

# or override the backend with whatever layer we have on the registry
  backend:
    image: registry.gitlab.com/foo/bar:tag

# or test a new PostgreSQL version
  database:
    image: "postgres:10"
```

File changes

Without volumes in your `docker-compose.yml` (or override) you'll need to rebuild your container if you change code.

Use volumes!

```
services:
  backend:
    build:
      context: "."
      dockerfile: "docker/Dockerfile.backend"
  depends_on:
      database

  volumes:
    - "./bin:/opt/zaaksysteem/bin"
    - "./dev-bin:/opt/zaaksysteem/dev-bin"
    - "./lib:/opt/zaaksysteem/lib"
```

Persistent data

Also use volumes

```
services:
  database:
    image: "postgres:10"
  volumes:
    - "dbdata:/var/lib/postgresql/data"

# somewhere else in your docker-compose.yml
volumes:
  dbdata:
```

Now our database files survive restarts and rebuilds of our container

Recreate containers

Changes in your `docker-compose.yml` or `docker-compose.override.yml`

```
docker-compose rm -s -f <container>
docker-compose up -d <container>
```

Developer tools

- We have a docker maintenance script that rebuilds all the containers.
  - Also updates the docker-compose.override.yml version to reflect the docker-compose.yml version.
- For time consuming layers, are build by one developer on change and pushed to registry.
  - We have a script that deals with rebuilding the base layer, so version management of the base layer is easier.

Do you still follow?

Building images

Dockerfile

```
FROM perl:latest
COPY . .
RUN cpanm .
```

Speeding up builds

- Make use of the `.dockerignore` file
- Make use of the cache

.dockerignore

- Be able to exclude exclude files from a COPY command
- Similar to .gitignore
- Can't have multiple per build context
  `Dockerfile.foo` and `Dockerfile.bar` will share the same file :(

Cache the builds

From

```
FROM perl:latest
COPY . .
RUN cpanm .
```

to

```
FROM perl:latest
COPY cpanfile .
RUN cpanm --installdeps .
COPY . .
RUN cpanm .
```

Multi-stage builds

Multi-stage builds

- Pretty new-ish
- Haven't done much with it in regards to Perl
- Makes builds faster and smaller
  - JS build at work
    - Down from 5 mins to 2 mins
    - Down from 1.02Gb to 176MB
- Not always the best choice

Made possible in

DOCKER VERSION	DOCKER COMPOSE VERSION	MULTI-STAGE SUPPORT
< 17.09.0	3.2	No
>= 17.09.0	3.4	Yes
>=18.02.0	3.6	Yes

Multi-stage

```
FROM buildpack-deps:stretch

COPY *.patch /usr/src/perl/
WORKDIR /usr/src/perl

RUN apt-get update && apt-get install --no-install-recommends less \
    && curl -SL https://www.cpan.org/src/5.0/perl-5.20.3.tar.bz2 -o perl-5.20.3.tar.bz2 \
    && echo '1b40068166c242e34a536836286e70b78410602a80615143301e52aa2901493b *perl-5.20.3.tar.bz2' | sh
    && tar --strip-components=1 -xaf perl-5.20.3.tar.bz2 -C /usr/src/perl \
    && rm perl-5.20.3.tar.bz2 \
    && cat *.patch | patch -p1 \
    && gnuArch="$(dpkg-architecture --query DEB_BUILD_GNU_TYPE)" \
    && archBits="$(dpkg-architecture --query DEB_BUILD_ARCH_BITS)" \
    && archFlag="$([ "$archBits" = '64' ] && echo '-Duse64bitall' || echo '-Duse64bitint')" \
    && ./Configure -Darchname="$gnuArch" "$archFlag" -Duseshrplib -Dvendorprefix=/usr/local  -des \
    && make -j$(nproc) && TEST_JOBS=$(nproc) make test_harness && make install \
    && cd /usr/src && curl -LO http://www.cpan.org/authors/id/M/MI/MIYAGAWA/App-cpanminus-1.7044.tar.gz
    && echo '9b60767fe40752ef7a9d3f13f19060a63389a5c23acc3e9827e19b75500f81f3 *App-cpanminus-1.7044.tar.
    && tar -xzf App-cpanminus-1.7044.tar.gz && cd App-cpanminus-1.7044 && perl bin/cpanm . \
    && rm -fr /root/.cpanm /usr/src/perl /usr/src/App-cpanminus-1.7044* /tmp/*

WORKDIR /root
CMD ["perl5.20.3","-de0"]
```

Multi-stage (cont'd)

```
FROM buildpack-deps:stretch as build
#                           ^^^^^^^^
COPY *.patch /usr/src/perl/
WORKDIR /usr/src/perl

RUN apt-get update && apt-get install --no-install-recommends less \
    && curl -SL https://www.cpan.org/src/5.0/perl-5.20.3.tar.bz2 -o perl-5.20.3.tar.bz2 \
    # Snipped for brevity ..

    && rm -fr ./cpanm /root/.cpanm /usr/src/perl /usr/src/App-cpanminus-1.7044* /tmp/*

WORKDIR /root
CMD ["perl5.20.3","-de0"]

FROM debian:stretch-slim as runtime
#                       ^^^^^^^^^^
WORKDIR /root
COPY --from=0 /usr/local /usr/local
#    ^^^^^^^^

COPY --from=build /usr/local /usr/local
#    ^^^^^^^^^^^^

CMD ["perl5.26.2","-de0"]
```

Selecting a target

docker

```
docker build -f Dockerfile --target build .
```

docker.compose.yml

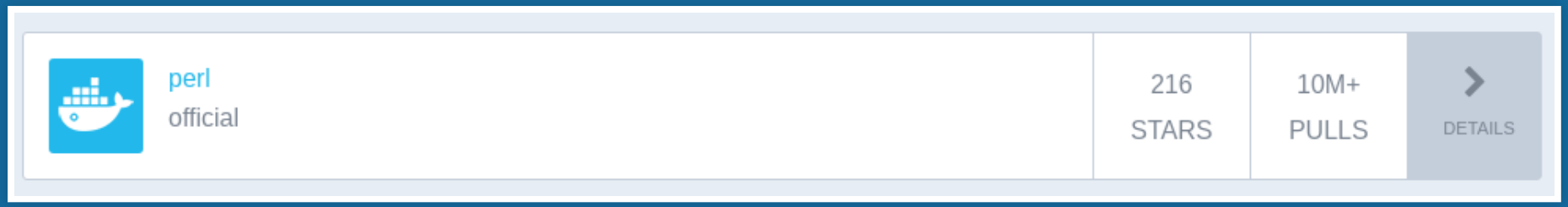
```
perl:
  build:
    context: "."
    dockerfile: "Dockerfile"
    target: "build"
```

docker.override.yml

```
perl:
  build:
    target: "runtime"
```

Multi-stage perl

- Don't want to have test deps hanging around
  - cpanm --installdeps -n
- Don't want to refetch the sources on every stage
  - COPY --from=buildphase /root/.cpanm /root/.cpanm



Docker Hub

- perl:{tag}
  - latest is currently 5.28.0
  - 5 is latest (== 5.28 == 5.28.0)
  - 5.x is 5.x.y (5.26 == 5.26.2)
  - Image size is 800+Mb
  - Good for building
- perl:slim / perl:{tag}-slim
  - Small images of ~100MB
  - Good for your final layer

Caveats

- Volumes cannot be used in the build phase
- cpanm doesn't bail out when something goes wrong
- cpanm doesn't install modules in the same order

Gitlab CI

- On each push it runs a pipeline
- Automatic deployment to kubernetes
- Tags and branches are pushed to a registry
  - Gitlab registry
  - Docker hub

.gitlab-ci.yml

A very basic example

```
image: perl:latest

stages:
  - test

before_script:
  - "apt-get update && apt-get install libmagic-dev"
  - "cpanm --installdeps ."
  - "perl Makefile.PL"
test:
  script:
    - "make test"
```

.gitlab-ci.yml

```
image: docker:stable
services:
  - docker:dind

variables:
  CONTAINER_IMAGE: registry.gitlab.com/$CI_PROJECT_PATH
  DOCKER_DRIVER: overlay2

before_script:
  - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN registry.gitlab.com

stages:
  - build
  - test
```

.gitlab-ci.yml (cont'd)

```
build:
  stage: build
  except:
    - master
  script:
    - docker pull $CONTAINER_IMAGE:$CI_BUILD_REF_NAME || true
    - docker pull $CONTAINER_IMAGE:latest || true
    # Gitlab doesn't support multiline YAML :(
    # but for clarity I do ;)
    - >
      docker build \
      --cache-from $CONTAINER_IMAGE:latest \
      --cache-from $CONTAINER_IMAGE:$CI_BUILD_REF_NAME \
      --tag $CONTAINER_IMAGE:$CI_BUILD_REF_NAME .
    - docker push $CONTAINER_IMAGE:$CI_BUILD_REF_NAME
```

.gitlab-ci.yml (cont'd)

```
test:
  stage: test
  except:
    - master
  script:
    - docker pull $CONTAINER_IMAGE:$CI_BUILD_REF_NAME
    - >
    docker run $CONTAINER_IMAGE:$CI_BUILD_REF_NAME \
    prove -lv -It/lib --formatter TAP::Formatter::Console
```

.gitlab-ci.yml (cont'd)

```
build-latest:
  stage: build
  only:
    - master
  script:
    - docker pull $CONTAINER_IMAGE:latest || true
    - docker build --cache-from $CONTAINER_IMAGE:latest --tag $CONTAINER_IMAGE:latest .
    - docker push $CONTAINER_IMAGE:latest

test-latest:
  stage: test
  only:
    - master
  script:
    - docker pull $CONTAINER_IMAGE:latest
    - docker run $CONTAINER_IMAGE:latest prove -lv -It/lib --formatter TAP::Formatter::Console
```

CPAN modules

My workflow

- Started with Dist::Zilla some time ago
- Dzilla author plugin bundle takes care of all the boilerplate needed
  - Changes
  - Dockerfile
  - .gitlab-ci.yml
  - .dockerignore
  - .gitignore
  - .editorconfig
- Gitlab CI therefore works from the initial commit
- Manually release to CPAN

Gitlab CI

See the previous slide

Dockerfile

```
#
# This is a skeleton Dockerfile.
# It is not intended to be small or super nifty, it tries to cache some,
# but it is intended to be easy to go into an environment and poke
# around and edit and less things
#
FROM perl:latest

RUN apt-get update && apt-get install -y vim-tiny less curl

ENV NO_NETWORK_TESTING=1 \
    DEBIAN_FRONTEND=noninteractive

WORKDIR /tmp/build

COPY dev-bin dev-bin

COPY cpanfile .

RUN ./dev-bin/cpanm --installdeps .

# Install ourselves
COPY . .
RUN ./dev-bin/cpanm .
```

Travis vs Gitlab CI

- Travis let's you test against multiple perls
- Gitlab CI doesn't
- Unless a Dockerfile for each individual version and change your .gitlab-ci.yml to include this in the build
- I only add Dockerfiles for Perl versions which CPAN testers report a failure on.
  - I realize this may not be workable for each CPAN module, especially the highest in the CPAN river.

Summary

- Docker
  - Each command is a layer
  - You want to hit the cache more often than not
    - Things that change often go at the bottom of the Dockerfile
  - Having a .dockerignore is advised
  - Try to make your images as small as possible
    - Get a small base layer helps
    - Multi-stage builds help as well

Summary (cont'd)

- Docker Compose
  - Awesome for recreating your production platform on a dev box
  - Allows local overrides
  - Used by Docker Swarm


I want to ask you some questions.

Yaldi

These slides + examples:

- https://gitlab.com/waterkip/docker-yapc
- https://github.com/waterkip/docker-yapc