Technische Hochschule Rosenheim
Technical University of Applied Sciences

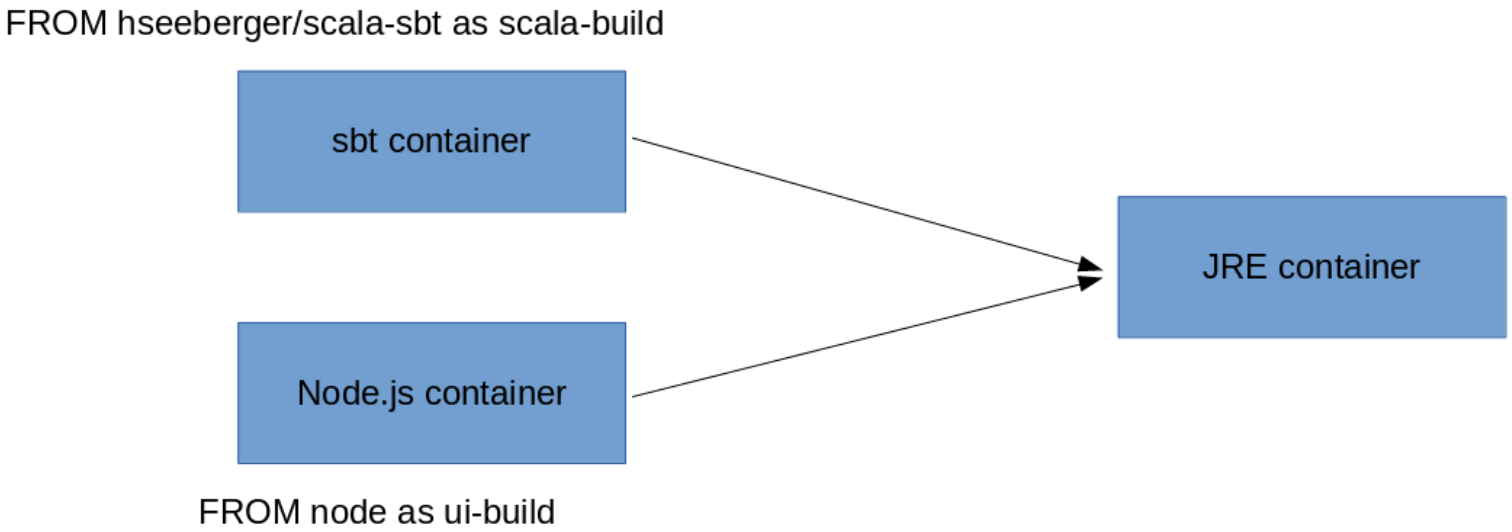
Microservices

Container - Advanced

Content

- Creating better, smaller Docker images
- Logging
- Health checks and monitoring

Multi-stage builds


FROM hseeberger/scala-sbt as scala-build
sbt container
Node.js container
JRE container
FROM node as ui-build

Some facts...

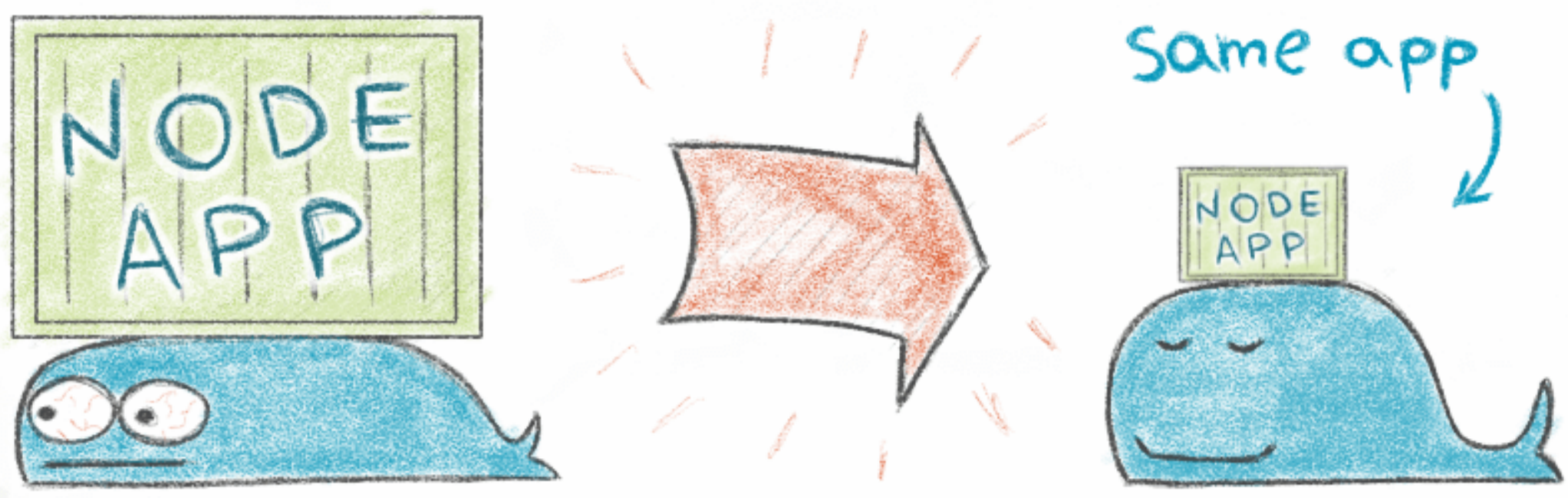
- Introduced with Docker 17.05
- Use multiple pre-built build environments (e.g. Node.js + Golang)
- Create smaller images containing only binaries and required assets

Sample

```
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o ap

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=0 /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

Image minifying


NODE APP
Same app
NODE APP

How to minify?

- Reduce number of layers (remember when new layers are created)
- Exclude development dependencies, test assets,...
- Try to use as small base images as possible (Alpine)
- Use multi stage builds

Uh, that's neat but I want more!

Do you really know which files/libraries/dependencies are necessary? There's a nice 'hack' to see which files ares used.

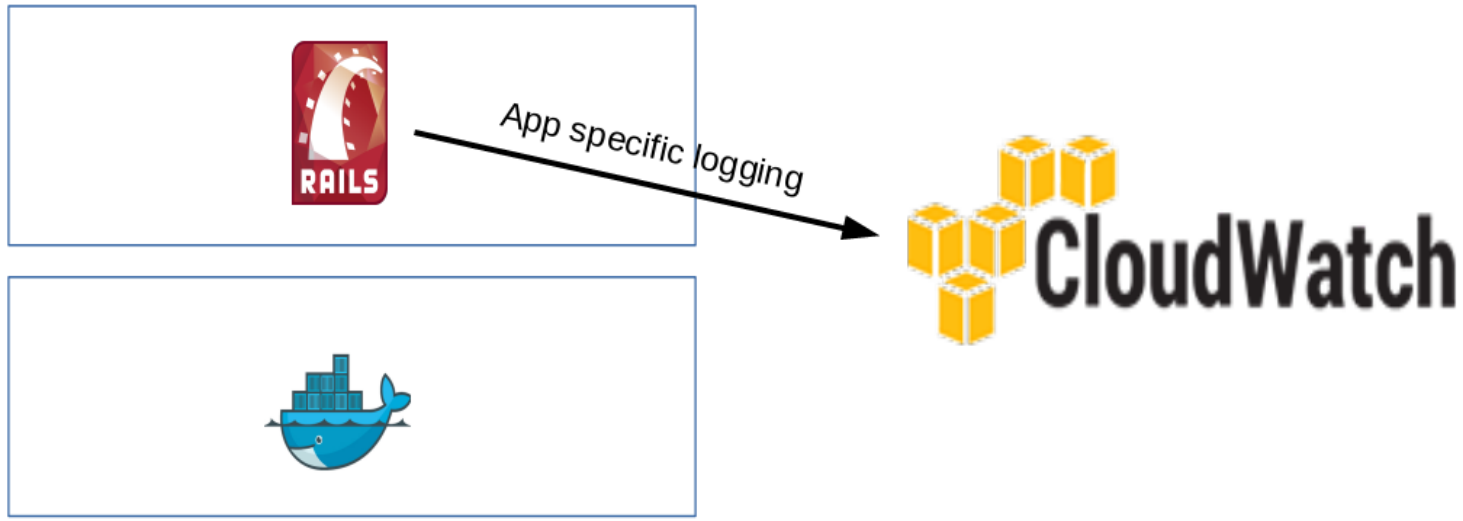
Problem

- the Linux kernel does not update the access times of files when they're read (anymore)
- there are probably special user space tools to monitor file access but requires installation,...

Hack

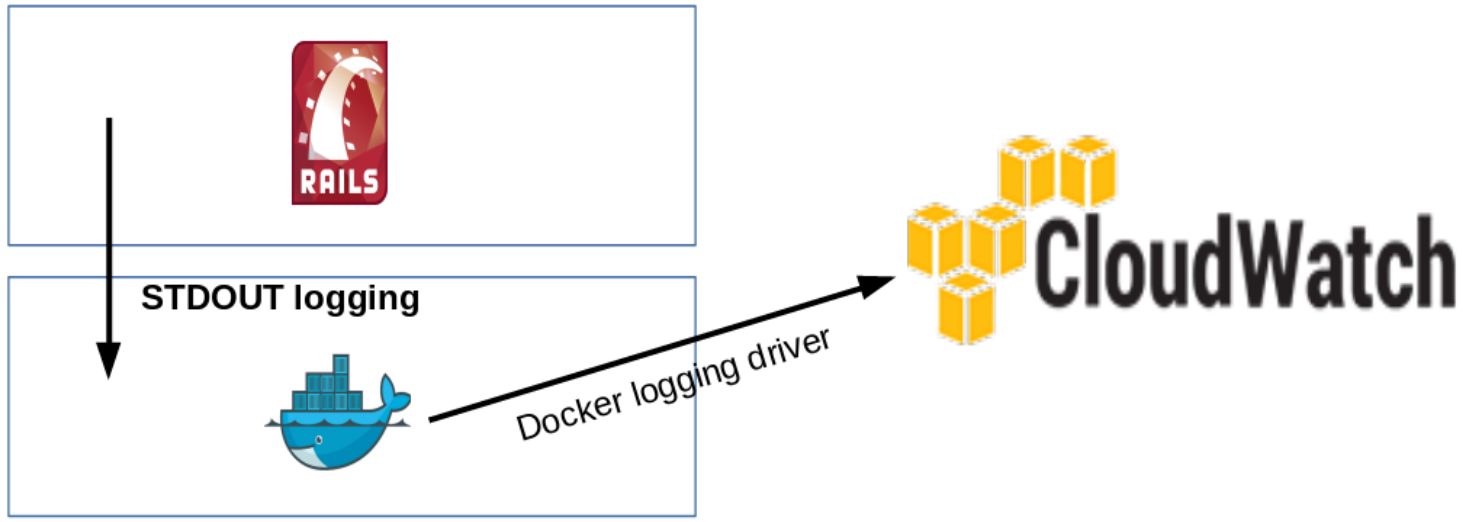
1. deploy your application to a VM
2. move the binaries to a separate partition/virtual disk
3. mount the partiton/virtual disk with the option `strictatime` to force the Linux kernel to update the access times
4. use `find` to query for all accessed files and save the list
5. use `rsync` to collect all required files in another directory structure and move them e.g. to another stage

Logging


RAILS
App specific logging
CloudWatch

Application specific logging

- Logs are directly written to a database or log collector (Logstash, FluentD,... - more on that later)
- Requires probably complex configuration and debugging until logging is working
- Has to be done for all components/services
- Requires probably custom adapters for a specific logging framework if no one else used the combination of logging framework and log storage before
- No matter how many logging protocols are supported the one needed by your customer is always missing


RAILS
STDOUT logging
Docker logging driver
CloudWatch

Docker logging drivers

- Logging to STDOUT and letting the Docker daemon redirect the logs is the preferred way of handling logs within Docker
- Default Docker distribution bundles already a lof of logging drivers, e.g.
  - Syslog
  - GELF
  - FluentD
  - AWS Logs
- Delegates collecting to infrastructure whereas domain specific tasks like what to log is still a task for every developer

Configure logging driver

```
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m"
  }
}
```

/etc/docker/daemon.json

Healthchecks & monitoring

- It is possible to declare healthchecks already in Dockerfiles
- It is also possible to declare healthchecks in a Docker-Compose configuration
- Since version 3 of Docker-Compose healthchecks are not considered in depends_on cases!
- Cluster systems like Kubernetes are also heavily relying on healthchecks

Dockerfile sample

```
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 \
    GOOS=linux \
    go build -a -installsuffix cgo -o app .
HEALTHCHECK \
    --interval=5s \
    --timeout=3s \
    CMD curl -f http://localhost:5000 || exit 1
CMD ["./app"]
```

Docker-Compose sample

```
version: '2.1'

services:
  postgres:
    image: "postgres"
    environment:
    - POSTGRES_PASSWORD=dbpassword
    ports:
    - "5432:5432"
    healthcheck:
      test: ["CMD-SHELL", "psql -U dbuser -d db1 -c 'SELECT 1'"]
      interval: 10s
      timeout: 5s
      retries: 20
```

Kubernetes sample

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 60
    livenessProbe:
      exec:
```

Docker-Compose tricks

Scaling

It's possible to create a Docker-Compose based application and scale it with Docker-Compose. Given the following `docker-compose.yml`:

```
version: '3'

services:
  database:
    image: postgres:alpine
    environment:
      - POSTGRES_PASSWORD=W@c[3~DV>~:]4%+5
  icndb:
    image: baez90/jericho-victim:latest
    depends_on:
      - database
```

```
# start the stack
docker-compose up -d

# scale the icndb
docker-compose scale icndb=3
```

- This way it's possible to validate if an application can be scaled correctly.
- Caveat is that a scaled service cannot publish the same port multiple times. You'd need a load balancer configured by the Docker socket to access the scaled service from outside the host.

Networks

To isolate containers/services of your Docker-Compose stack you can declare custom networks:

```yaml
version: '3.6'
services:
  svc1:
    image: ...
    networks:
    - net1
  svc2:
    image: ...
    networks:
    - net2

networks:
  net1: {}
  net2: {}
```

Linking of containers

To enable cross-container calls on simple hostnames Docker-Compose offers the so called `links` option to set the hostname under which a dependent service will be available. You can define links like shown in the following slide.

```yaml
version: '3.6'
services:
  svc1:
    image: ...
    networks:
    - net1
  svc2:
    image: ...
    networks:
    - net2
    links:
    - svc1:svc1

networks:
  net1: {}
  net2: {}
```