AKDB

Basic Docker Workshop

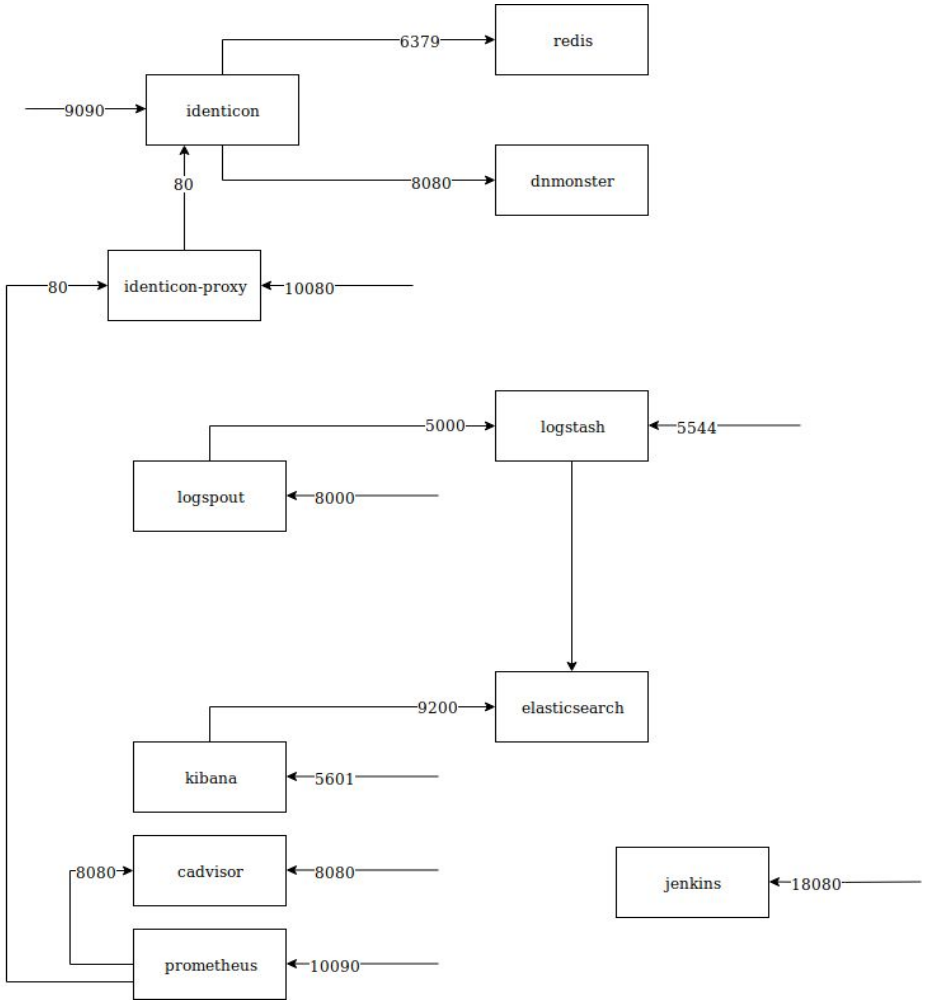
Agenda

1) **Showcases**
2) **Grundlagen**
3) **Dockerfile**
4) **Docker Command Line Interface**
5) **Docker Registry**
6) **Großprojekt (Teil 1)**
7) **Anwendungen Containerisieren**
8) **Großprojekt (Teil 2)**
9) **Ausblick: Security, CI, Kubernetes**

Großprojekt

Im Zuge der Schulung bauen wir gemeinsam Schritt für Schritt eine Demo-Applikation, bestehend aus mehreren, miteinander agierenden Docker Containern.

Großprojekt



Handout

- Cheatsheet
- Ausgedruckte Präsentation
- Agenda / Zeitplan

1) Showcases

2) Grundlagen

Grundlagen: Was sind Container? (Docker)

Docker ist ein OCI standardisiertes Open-Source **Paketierungswerkzeug von Software**, das gleichzeitig eine Runtime zum Ausführen der Software bietet.

Die Paketierung ist vergleichbar mit der Be- und Verladung von ISO-Containern in einem Hafen:

- Verpackung verschiedenartiger Produkte in standardisierten Containern
- Temperierung / Polsterung von Containern möglich (Chemikalien, Medikamente, Lebensmittel)
- Einheitliche Schiffe für den Transport aller Container
- Umladung mittels Kran auf die Schiffe
- Zentrale Hafenaufsicht

Bzgl. Docker heißt das, dass Anwendungen in Images verpackt werden, die gezielt auf diese Software zugeschnitten sind:

- Minimales Betriebssystem
- Nur die Abhängigkeiten, die die Software für den Betrieb benötigt, sind in den passenden Versionen vorinstalliert

Damit lässt sich ein soches Image **auf jedem Docker-Host “ausführen”** (Docker-Container).

Grundlagen: Unterschied Container vs. VMs

Virtuelle Maschienen: simuliert die gesamte PC-Hardware via Software.

⇒ frisst immense Ressourcen, Langsamer start des systems, hoher speicherplatzverbrauch

Vorteil: Isolierung (sichergestellt durch Hypervisor)

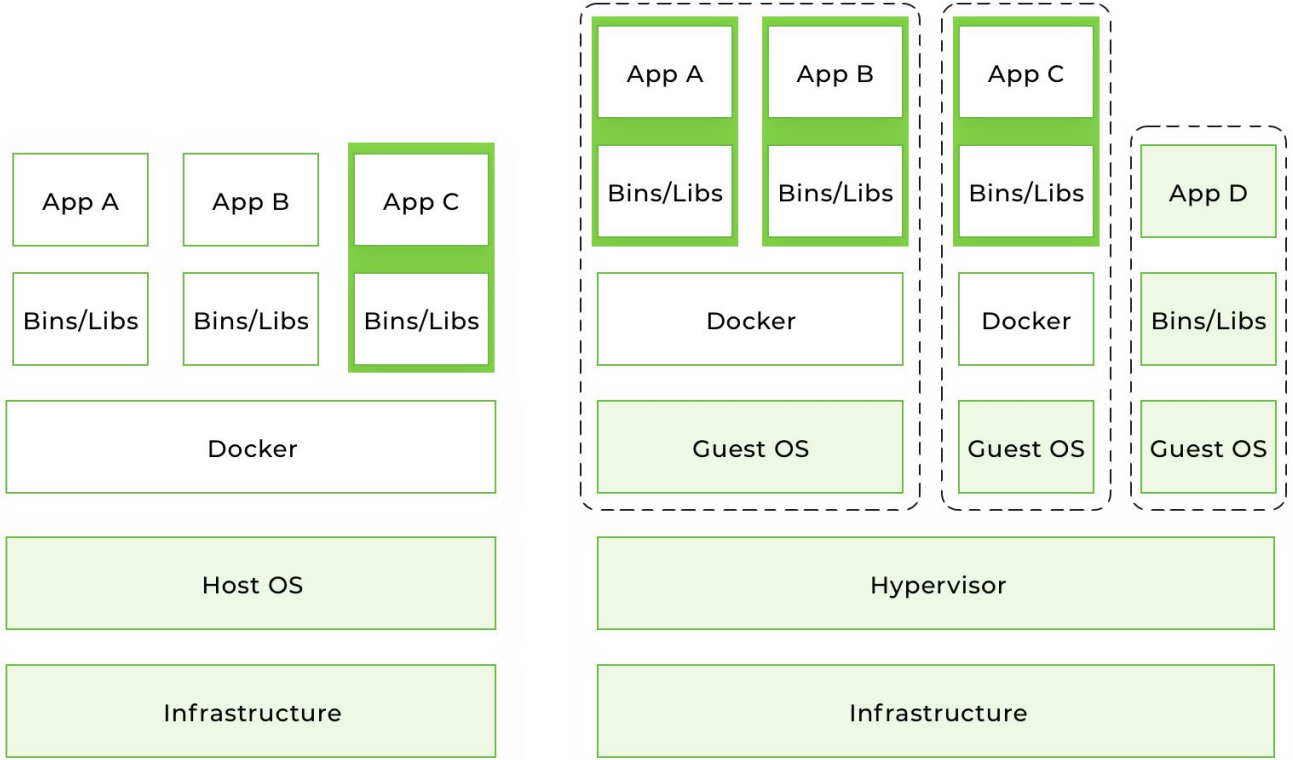
Docker Container: **Isolierung nur durch namespace** Kernel-Technologie (leichtgewichtig)

⇒ Angreifer können hier “leichter” von einem kompromittierten Container aus das Host-System angreifen.

VMs & Container:

- Können gemeinsam betrieben werden (Container auf VM Host)
- Verbindet Vorteile & Nachteile Beider! (performance, isolierung, Standardisierung, …)

Grundlagen: Unterschied Container VS VMs



Grundlagen: Vorteile von Docker

- Standardisiertes Format + Starten / Stoppen der Container
- Damit schnellerer, reproduzierbarer Deployment-Prozess
- Gleiches Verhalten auf allen Stages (dev/test/preprod/prod)
- Einfach zu verwalten / debuggen / monitoren
- Sehr gut skalierbar (Basis für Orchestrierungstools wie z.B. Kubernetes)

Terminologie

- Layer
- Image
- Container
- Image-Kennung
- Docker Registry / DockerHub
- Volume

Terminologie: Layer

Ein Layer ist technisch gesehen ein Archiv (`/var/lib/docker/<storage-driver>/layers/<layer-id>`), das über eine global eindeutige ID gekennzeichnet ist und eine Verzeichnisstruktur beinhaltet, z.B.:

Archiv-Inhalt:

```
folderA/
folderB/
file1
```

Sicht innerhalb des Layers:

```
/
|-- folderA/
|-- folderB/
|-- file1
```

Terminologie: Image

Docker Layers können gestapelt werden, ihnen liegt dieselbe Wurzel zugrunde. Oberhalb liegende

- Layers *verdecken* bzw. *erweitern* die Dateisystem-Inhalte der unterhalb liegenden Layers

Verändert ein Layer eine Datei, die in einem unterhalb liegenden Layer beheimatet ist, wird die **Copy-on-Write** Strategie angewendet, d.h. die Datei wird zunächst in den oberen Layer kopiert (Verdeckung der Ursprungs-Datei) und dort dann verändert.

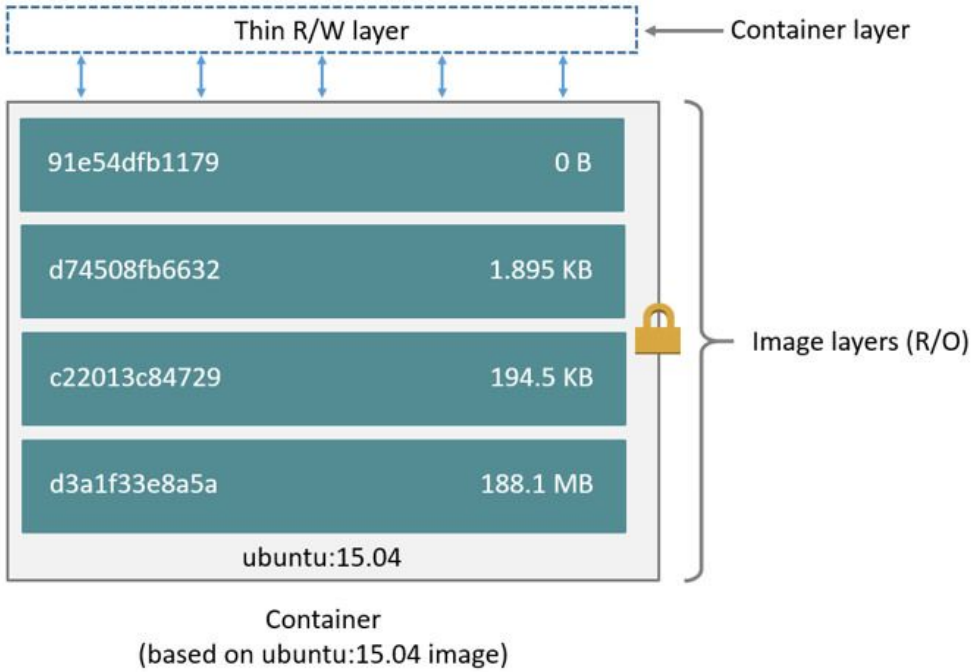
Ein **Docker Image** ...

➔ ... besteht aus 1 bis n gestapelten read-only Layern
➔ ... hat eine global eindeutiger ID.
➔ ... kann als einzelnes Archiv gepackt (und kopiert) werden

Terminologie: Container

Ein Container ist ein “zur Ausführung gebrachtes Docker Image”.

Anschaulich gesprochen wird dem read-only Image (Layer-Stapel) ein leerer **read-write** Layer oben aufgelegt und in diesem dann der ausgewiesene Hauptprozess gestartet.



Terminologie: Container

- Dieser Prozess unterscheidet sich auf dem Host-System nicht von anderen laufenden Prozessen (PID xyz).
- Im Container wird ihm allerdings die PID 1 zugewiesen.
- Sobald der Hauptprozess endet, “stirbt” der Container (auch alle weiteren Container-Prozesse werden gestoppt).

```
user@user-Inspiron-5770:~$ docker run ubuntu sleep 30 &
[1] 10215
user@user-Inspiron-5770:~$ ps -aux | grep sleep
user     10215  0.6  0.3 1080756 63292 pts/1   Sl   14:25   0:00 docker run ubuntu sleep 30
root     10284  3.5  0.0   4532   768 ?        Ss   14:25   0:00 sleep 30
user     10338  0.0  0.0  21536  1012 pts/1    S+   14:25   0:00 grep --color=auto sleep
user@user-Inspiron-5770:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
b32658ccbad7        ubuntu              "sleep 30"          18 seconds ago      Up 13 seconds                           heuristic_goldberg
user@user-Inspiron-5770:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
[1]+  Fertig                  docker run ubuntu sleep 30
user@user-Inspiron-5770:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
user@user-Inspiron-5770:~$ ps -aux | grep sleep
user     10590  0.0  0.0  21536  1088 pts/1    S+   14:26   0:00 grep --color=auto sleep
user@user-Inspiron-5770:~$
```

Terminologie: Image-Kennung

Jedem Image wird eine eindeutige Kennung zugeordnet, die folgender Bauart ist:

`<registry-host:port>/<namespace>/<name>:<tag>`

Dabei ist nur `<name>` mandatory, d.h. “**`jenkins`**” wäre z.B. eine valide Image-Kennung.

`<registry-host:port>` beinhaltet die Adresse der Docker Registry, in der das Image hinterlegt ist.

Terminologie: Docker Registry / DockerHub

Was ist das?

- Eine Docker Registry ist ein Web-Server, der
- Stellt Docker Images zum Download / Upload zur Verfügung
- absicherung mittels Authentifizierung bzw. Autorisierung

Offizielle Docker Registry: DockerHub (`docker.io`)

`Fehlende Registry Addresse ⇒ prefix = docker.io/`

⇒ `docker pull` “`jenkins`” ist gleichbedeutend mit docker pull “`docker.io/jenkins`”

Private Registry

- Private Registry (abgesichert)
- Kann im Firmennetz betrieben werden
- Vorteile:
  - Nur freigegebene Images können genutzt werden (Schadprogramme / Versionen / Vulnerabilities)
  - Nur die Firma hat Zugriff auf die Images
  - Firma besitzt die Datenhoheit

Terminologie: Volume

Ein Docker-Volume ist ein gemountetes Host-Verzeichnis bzw. Host-Datei an eine bestimmte Stelle innerhalb des Container-Dateisystems.

Es wird zwischen expliziten und impliziten Volumes (oder auch Bind-Mounts) unterschieden:

Explizit: `<host-path>:<container-path>:<attribute>`

Implizit: `<container-path>:<attribute>`

Gültige Attribute sind: `ro`, `rw` (default) bzw. `z`, `Z` unter SE-Linux

Existiert das Zielverzeichnis (-Datei) im Container bereits, wird es von dem Mountpoint “überdeckt”.

Bei impliziten Volumes mountet Docker einen intern verwalteten Host-Pfad (irgendwo unter /var/lib/docker/volumes/...) oder erstellt einen solchen, falls noch nicht vorhanden.

Docker Installation (CentOS)

```
sudo yum install -y yum-utils device-mapper-persistent-data lvm2
sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
sudo yum install docker-ce
sudo usermod -aG docker $(whoami)
sudo systemctl enable docker.service
sudo systemctl start docker.service
```

Links:

https://docs.docker.com/install/
https://docs.docker.com/install/linux/docker-ce/centos/

Docker Daemon / Docker Client

Docker liegt eine Client/Server Architektur zugrunde.

Docker-Daemon/Docker-Engine = Server
Docker-CLI = Client

Kommunikation standardmäßig über Unix-Socket (`/var/run/docker.sock`)

Client kann via DOCKER_HOST Umgebungsvariable mit anderen Daemons kommunizieren (remote).

Daemon konfigurierbar über:

1. `/etc/docker/daemon.json`
2. via -H (default “`-H unix://`”) in `/[etc|lib]/systemd/system/docker.[service|socket]`

3) Dockerfile

Dockerfile

Ein Dockerfile ist eine Datei, die den Inhalt eines Docker-Images beschreibt.

Neben dieser Dokumentation dient es vor allem der Erstellung dieses Images mittels “`docker build -t <tag> <Dockerfile-dir>`”.

Dabei können folgende Schlüsselwörter verwendet werden:

siehe auch:
https://docs.docker.com/engine/reference/builder/

FROM (1/3)

```
FROM <image> [AS <name>]
```

```
FROM <image>[:<tag>|@<digest>] [AS <name>]
```

Das FROM Schlüsselwort ist immer der Einstiegspunkt für den Bau eines neuen Images und zeigt auf ein Image, dessen Layer-Stapel als Grundlage hergenommen werden soll. “FROM scratch” bildet hier eine Ausnahme und bestimmt, dass von einem leeren Dateisystem “/” geerbt wird.

FROM (2/3)

Das zuletzt gebaute Image, d.h. das Image, das nach dem im Dockerfile zuletzt aufgeführten `FROM` entsteht, wird entsprechend dem “`docker build -t <tag> .`” getaggt. Alle vorangehenden Images werden damit wieder verworfen.

Sollte mehr als ein FROM Schlüsselwort im Dockerfile enthalten sein, spricht man von einem “Multi-Stage Build”. Die vorangehenden Images dienen hier nur als Hilfsimages für den Bau des finalen Images. Ihnen können mit “AS <name>” Namen gegeben werden, mittels derer sie von den nachfolgenden Images über das `COPY`-Schlüsselwort referenziert werden können, um darin enthaltene Dateien zu übernehmen.

FROM (3/3)

Beispiel multi-stage:

```
FROM golang:1.11 AS builder
COPY myapp.go /app/
WORKDIR /app
RUN go build -o /myapp .

FROM scratch
COPY --from=builder /myapp /
CMD [“/myapp”]
```

```
FROM golang:1.11
COPY myapp.go /app/
WORKDIR /app
RUN go build -o /myapp .

FROM scratch
COPY --from=0 /myapp /
CMD [“/myapp”]
```

Über “`docker build -t myapp .`” entstünde ein Image names “`myapp`”, das aus einer einzigen ausführbaren Datei “`/myapp`” besteht. Dieses Binary wird vom Hilfs-Image “`builder`” gebaut. Bringt man dieses Image mit “`docker run …`” zur Ausführung, so würde ein Container entstehen, der unter der PID 1 “`/myapp`” laufen ließe.

RUN

`RUN <command>` (*shell* form)

`RUN ["executable", "param1", "param2"]` (*exec* form)

Die *exec* Form setzt keine vorinstallierte Shell voraus (/bin/bash).

Die Standard-Shell, die bei der *shell* Form verwendet wird, kann mit der `SHELL` Instruktion verändert werden.

In der *shell* Form können Zeilen mit einem Backslash gebrochen werden.

Über `RUN` Instruktionen können beliebige Shell-Kommandos abgesetzt werden, z.B.

```
RUN apt update && apt install -y telnet
```

ENTRYPOINT

- `ENTRYPOINT ["executable", "param1", "param2"]` (*exec* form, preferred)
- `ENTRYPOINT command param1 param2` (*shell* form)

Die letzte `ENTRYPOINT` Instruktion der letzten Stage definiert den Prozess, der beim Container-Start ausgeführt werden soll:

```
FROM ubuntu
WORKDIR /var
ENTRYPOINT [“ls”]
```

Container-Start gibt den Inhalts von “`/var`” aus, ehe er wieder beendet wird.

CMD

- `CMD ["executable","param1","param2"]` (*exec* form, preferred)
- `CMD command param1 param2` (*shell* form)

Nur die letzte `CMD` Instruktion (falls vorhanden) der letzten Stage hat einen Einfluss auf das finale Image:

Wenn keine `ENTRYPOINT` Instruktion gegeben ist, dient die Anweisung unter `CMD` als Ausführungsanweisung für jeden Container (sofern nicht mittels “docker run …” überschrieben).

Im anderen Fall werden die unter `CMD` angegebenen Argumente dem `ENTRYPOINT` als Argumente übergeben, z.B. käme “`ls -la /lib`” dem Start des folgenden Images gleich:

```
FROM ubuntu
ENTRYPOINT [“ls”]
CMD [“-la”, “/lib”]
```

ENTRYPOINT vs. CMD

Beispiel:

```
FROM ubuntu
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
```

```
docker build -t topImage .
```

Aufgabe: Welche Shell-Kommandos würden ausgeführt werden?

1. `docker run -t topImage -H`
2. `docker run -t topImage`

ENTRYPOINT vs. CMD

Beispiel:

```
FROM ubuntu
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
```

```
docker build -t topImage .
```

Aufgabe: Welche Shell-Kommandos würden ausgeführt werden?

1. `docker run -t topImage -H` → `top -b -H`
2. `docker run -t topImage` → `top -b -c`

LABEL

```
LABEL <key>=<value> <key>=<value>...
```

Mit Hilfe des `LABEL` Schlüsselwortes können dem Image Metadaten in Form von key-value Paaren hinzugefügt werden, die z.B. mittels "`docker inspect <image-kennung|image-id>`" eingesehen werden können.

Beispiele:

```
LABEL com.example.vendor="ACME Incorporated"

LABEL version="1.1-alpha" \
      DEBUG="1"

LABEL maintainer="John Doe <john.doe@acme.com>"
```

LABEL Standards

Label Standards Opencontainer: https://github.com/opencontainers/image-spec/blob/master/annotations.md

- org.opencontainers.image.url
  a. URL to find more information on the image (string)
  b. build repo url
  c. automatically By maven
- org.opencontainers.image.authors
  a. contact details of the people or organization responsible for the image (freeform string)
  b. automatically By maven
- org.opencontainers.image.created
  a. date and time on which the image was built (string, date-time as defined by RFC 3339).
  b. automatically By maven
- org.opencontainers.image.revision
  a. source control revision identifier for the packaged software.

LABEL Standards

- org.opencontainers.image.description
  a. Human-readable description of the software packaged in the image (string)
- org.opencontainers.image.title
  a. Human-readable title of the image (string)
- org.opencontainers.image.vendor
  a. Name of the distributing entity, organization or individual.
- org.opencontainers.image.version
  a. version of the packaged software // Tag name
    i. The version MAY match a label or tag in the source code repository
    ii. version MAY be Semantic versioning-compatible

COPY

```
COPY [--from=<name|index>] [--chown=<user>:<group>] <src>... <dest>
```

```
COPY [--from=<name|index>] [--chown=<user>:<group>] [“<src>”, ..., “<dest>”]
```

Mit Hilfe des `COPY` Schlüsselwortes können dem Image Metadaten in Form von key-value Paaren hinzugefügt werden, die z.B. mittels “`docker inspect <image-name|image-id>`“ ausgelesen werden können.

Beispiele:

```
COPY index.html /usr/local/nginx/html/index.html
```

```
COPY index.html index.htm /usr/local/nginx/html/
```

```
COPY --from=builder /build/my-binary /usr/bin/my-app
```

Remaining Instructions (1/2)

`ONBUILD <instruction>`

wird in jedem abgeleiteten Image zuerst ausgeführt.

`EXPOSE <port>[/<protocol>] <port>[/<protocol>]...`

dokumentiert vom Container verwendete Ports (samt Protokoll UDP/TCP, defaults to TCP).

`VOLUME <path> <path>...`

erstellt die angegebenen Mountpoints beim Start des Containers.

`ARG <key>[=<value>] <key>[=<value>]...`

parametrisiert das Dockerfile. Uninitialisierte Argumente müssen per “--build-arg <key>=<value>” dem “docker build” mitgegeben werden.

Remaining Instructions (2/2)

`ENV <key>=<value> <key>=<value>...`

fügt dem Image Umgebungsvariablen hinzu.

`USER <user>`

wechselt zum angegebenen Benutzer.

`WORKDIR <path>`

wechselt ins angegebene Verzeichnis. Wird von RUN, COPY,

`STOPSIGNAL <number|signal>`

sendet dem Container das angegebene Signal, wenn er gestoppt wird (z.B. “STOPSIGNAL SIGKILL”)

`HEALTHCHECK [<options>] CMD <command>`

setzt Container-State (starting|healthy|unhealthy) abhängig vom Ergebnis des angegebenen Healthchecks

Containerisieren

Vorgehen:

1. Neues Dockerfile erstellen
2. Basis Image auswählen (kleinstmöglich)
3. Abhängigkeiten installieren
4. ggf. benötigte Daten und Applikationen kopieren, z.B. via Multi-Staging:
   a. Container 1 compiliert Applikation
   b. Container 2 übernimmt das Kompilat von Container 1
      ⇒ weniger Abhängigkeiten und keine ungenutzten Daten
5. Entrypoint und/oder CMD setzen

Containerisieren

Beispiel: *Nginx*

Manuell verproben:

```
docker run -it -v $PWD/nginx:/usr/share/nginx/html ubuntu /bin/bash
$ apt-get install nginx
$ nginx start
```

Dockerfile:

```
FROM ubuntu:18.10
RUN apt-get update \
 && apt install -y nginx
ENTRYPOINT [“/usr/bin/nginx”]
```

oder einfach mit dem offiziellen Nginx-Image:

```
docker run -d -v $PWD/nginx:/usr/share/nginx/html nginx
```

Proxy (1/3)

Docker Client Env: https://docs.docker.com/network/proxy/#configure-the-docker-client

```
docker build -t <tag> . \
    --build-arg http_proxy=http://proxy.akdb.net:3128 \
    --build-arg https_proxy=http://proxy.akdb.net:3128
```

Proxy (2/3)

Docker Client: https://docs.docker.com/network/proxy/#configure-the-docker-client

```
cat ~/.docker/config.json
```

```
{
  "proxies":
  {
    "default":
    {
      "httpProxy": "http://proxy.akdb.net:3128",
      "httpsProxy": "http://proxy.akdb.net:3128",
      "noProxy": "localhost,127.0.0.1,*.test.example.com"
    }
  }
}
```

Proxy (3/3)

Docker daemon: https://docs.docker.com/config/daemon/systemd/

```
cat /lib/systemd/system/docker.service.d/http-proxy.conf

[Service]
Environment="HTTP_PROXY=http://proxy.akdb.net:3128/"
Environment="NO_PROXY=localhost,127.0.0.1,*.test.example.com"
```

Aufgabenstellungen

Alle Aufgabenstellungen und Lösungen sind in folgendem Git Projekt zu finden:

https://github.com/x-cellent/basic-docker-workshop

Clonen Sie das Repository mittels Git:

```
git clone https://github.com/x-cellent/basic-docker-workshop.git
```

Aufgabenstellung 1: Dockerfile schreiben

`1-exercise-write-dockerfile/exercise.md`

Docker in der Praxis: Eigene Images bauen

- Auf Imagename achten: *camelCase* to *lowercase*
- Proxy konfigurieren
- Herkunft des Basis-Images: Vertrauenswürdig?)
- Inhalte des Images: Libraries, App, Config, Lizenzen, …
- ENTRYPOINT, CMD, HEALTHCHECK, ...
- Auf Wartung achten: Regelmäßige Updates des Images)

Docker in der Praxis: Eigene Basis-Images nutzen

Was sind Basis Images?

- Vom Unternehmen/Geschäftsbereich freigegeben
- Gehärtet, je nach Anwendungsfall Basisfunktionalitäten enthalten (z.B. JDK/JRE)

Vorteile:

- Einheitliche Basis für mehrere Systeme
- Es müssen weniger Images gepflegt werden (Wartung, immer aktuell)
- Erhöhte Sicherheit, robuster

Bereits vorhandene Base-Images der AKDB:

- `akdb/egov/devops/base/springboot:8-jre-alpine`
- `akdb/egov/devops/base/nginx:1-alpine`
- `builder/maven:3.5.0-jdk8`

Linting Tool

Hadolint: https://github.com/hadolint/hadolint

- Analysiert Dockerfiles anhand von best-practices
- Regel-Set in Anlehnung an https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices
- Validiert inline Shell-Kommandos
- Aufruf:
  - hadolint ./Dockerfile
  - docker run --rm -t -v $PWD:/pwd:ro hadolint/hadolint hadolint /pwd/Dockerfile
  - docker run --rm -t hadolint/hadolint < Dockerfile
- Weitere:
  - Project Atomic Dockerfile Lint: https://github.com/projectatomic/dockerfile_lint
  - Replicated HQ Dockerfilelint: https://github.com/replicatedhq/dockerfilelint

Linting Tool

hadolintoutput:

/dev/stdin:1 DL3007 Using latest is prone to errors if the image will ever update. Pin the version explicitly to a release tag

/dev/stdin:2 DL4000 MAINTAINER is deprecated

/dev/stdin:3 DL3008 Pin versions in apt get install. Instead of `apt-get install <package>` use `apt-get install <package>=<version>`

/dev/stdin:3 DL3009 Delete the apt-get lists after installing something

/dev/stdin:3 DL3015 Avoid additional packages by specifying `--no-install-recommends`

/dev/stdin:4 DL3025 Use arguments JSON notation for CMD and ENTRYPOINT arguments

Dockerfile (vorher):

```
FROM xcellenthub/whalesay:latest
MAINTAINER Ferdinand.Eckhard@x-cellent.com
RUN apt-get -y update && apt-get install -y fortunes
CMD cowsay
```

Dockerfile (nachher):

```
FROM xcellenthub/whalesay:1.0
LABEL maintainer=”Ferdinand.Eckhard@x-cellent.com”
RUN apt-get -y update \
  && apt-get install -y --no-install-recommends \
       fortunes=1:1.99.1-7 \
  && apt-get clean \
  && rm -rf /var/lib/apt/lists/*
CMD ["/usr/local/bin/cowsay”]
```

Aufgabenstellung 2: Linting

`2-exercise-linting/exercise.md`

Docker Build

- Baut aus einem Dockerfile ein Image
- Legt dieses zunächst lokal ab

Aufruf:

```
docker build [OPTIONS] PATH | URL | -
```

- `docker build http://server/context.tar.gz`
- `docker build - < Dockerfile`
- `docker build -t xcellenthub/whalesay:2.0 .`
- `docker build -f Dockerfile.debug ./debug`

Beispiel:

```
docker build -t xcellenthub/whalesay:2.0 .
```

Output:

```
Sending build context to Docker daemon  3.584kB
Step 1/4 : FROM xcellenthub/whalesay:latest
 ---> 6b362a9f73eb
Step 2/4 : MAINTAINER Ferdinand.Eckhard@x-cellent.com
 ---> Using cache
 ---> 90efea02079a
Step 3/4 : RUN apt-get -y update && apt-get install -y fortunes
 ---> Using cache
 ---> b1f700e2d5ab
Step 4/4 : CMD /usr/games/fortune -a | cowsay
 ---> Using cache
 ---> 028704965be8
Successfully built 028704965be8
Successfully tagged xcellenthub/whalesay:2.0
```

Docker Images

Befehl:

- docker image list
- docker image ls
- docker images

Beispielausgabe:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
xcellenthub/whalesay	2.0	028704965be8	13 minutes ago	278MB
xcellenthub/whalesay	1.0	6b362a9f73eb	3 years ago	247MB
xcellenthub/whalesay	latest	6b362a9f73eb	3 years ago	247MB

Aufgabenstellung 3: Docker Build

`3-exercise-docker-build/exercise.md`

Docker Run

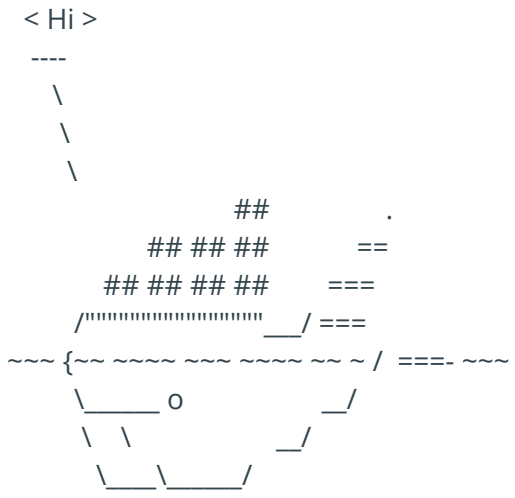
Startet einen neuen Container mit einem Befehl

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Beispiel:

```
docker run xcellenthub/whalesay:1.0 cowsay Hi
```

Ausgabe:



Docker Run: Options

-i, --interactive	Mit STDIN verbinden
-t, --tty	Pseudo-TTY anfordern
--name string	Container-Name setzen
--rm	Container nach Stopp löschen (read-write Layer)
-p, --publish list	Portweiterleitung vom Host in den Container
-P, --publish-all	Portweiterleitung von zufälligen, freien Host-Ports auf alle “exposed” Ports des Containers
-v, --volume list	Bind-Mount
--detached, -d	Container im Hintergrund starten
-e, --env list	Setzt Umgebungsvariablen im Container
--entrypoint string	Überschreibt den ausgewiesenen Entrypoint
--privileged	Erlaubt alle Syscalls

Docker: Persistenz

Grundregel: Container sind Wegwerfprodukte!
⇒ Das plötzliche Beenden eines Containers sollte nicht zu Datenverlusten führen!
⇒ Das lokale Dateisystem von Containern ist **nicht persistent**!
Dafür werden **Volumes** bzw. **Bind-Mounts** benötigt:

```
docker run -v <host-path>:<container-path> ...
```

Für Testzwecke kann man sich Dateien aus dem laufenden Container zur Analyse kopieren:

```
docker cp <container-id:file-path> <target-host-path>
```

oder vice-versa

```
docker cp <host-path> <container-id:target-path>
```

Read-Only Container

Container können im Read-Only Modus gestartet werden:

```
docker container run --rm --read-only nginx
```

Auch Volumes können als Read-Only eingebunden werden:

```
docker container run -v /host/path:/container/path:ro nginx
```

Aufgabenstellung 4: Docker Run

`4-exercise-docker-run/exercise.md`

Usermapping: Docker-Hostsystem

- Default user im Container ist root, wenn nicht durch USER geändert
- Berechtigungen sind eingeschränkt:
  - Kein Hostfileystem-Zugriff (ohne explizite Mountpoints)
  - Syscalls eingeschränkt

Regel: Möglichst neue User nutzen, die nicht auf dem Host System vorhanden sind.

(Default Mapping auf höhere Benutzer IDs (ID = 20001 ff.))

Wenn ein Host-User benötigt wird, um auf bestimmte Dateien auf dem Hostsystem zuzugreifen (z.B. Log Verzeichnis), so kann im Container unter diesem Benutzer gearbeitet werden (UID-Mapping).

Usermapping Docker-Hostsystem

Es gibt 2 Möglichkeiten, einen User auf das Hostsystem zu mappen:

- Konfiguration beim Starten des Containers:

```
docker run -u “999:999” <image>
```

- Konfiguration via docker-compose.yaml File:

```
service:
  name: my-app
  configs:
    uid: “999”
    gid: “999”
```

Docker Exec

Damit lässt sich ein Befehl in einem laufenden Container ausführen:

```
docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

Anwendungsfälle:

- Debugging (z.B. env, logs, top)
- in den Container springen (bash)

Beispiel:

```
docker run --name ubuntu-bash ubuntu sleep 100

docker exec -it ubuntu-bash /bin/bash

docker exec -it ubuntu-bash top
```

Aufgabenstellung 5: Docker Exec

`5-exercise-docker-exec/exercise.md`

Docker Start / Stop

Docker Container lassen sich stoppen und wieder starten. Hierdurch gehen Änderungen auf dem Dateisystem im Gegensatz zu “docker run --rm” bzw. “docker rm” nicht verloren.
Dies kann über deren Namen (docker run --name=my-name …) oder Container-ID erfolgen (siehe “docker ps”)

Requirements:

```
containerID=$(docker run -d --name=simpleApp -p 8080:80 nginx)

docker stop $containerID
docker stop simpleApp

docker start $containerID
docker start simpleApp
```

Docker CLI: Debugging

`docker inspect <container|image>`

zeigt die Metadaten des Containers/Images an

`docker ps [<options>]`

zeigt ID, Image, PID 1 Command, Port-Mapping, etc. aller Container an

`docker logs <container|container-id>`

zeigt den Ausgabe- und Errorstrom des angegebenen Containers an

Docker CLI: Image Speichern/Laden

`docker save`: Speichern Sie ein Image als Datei ab.

`docker load`: Laden Sie das Image aus einer Datei.

```
docker save IMAGE --output|-o > NAME

docker load --input|-i < NAME
```

`Beispiel:`

```
docker save busybox > busybox.tar

docker save --output busybox.tar busybox

docker load < busybox.tar

docker load --input busybox.tar
```

Docker CLI: Docker Inspect

```
docker inspect [OPTIONS] IMAGE-NAME|IMAGE-ID
docker inspect [OPTIONS] CONTAINER-NAME|CONTAINER-ID
```

liefert die Meta-Daten zu einem Image bzw. Container.
U.a. finden sich dort die Volumes, Port-Mappings, Netzwerke, IPs, Labels, usw.

Die Option `-f “{{ TEMPLATE-SELECTOR }}”` kann als Ausgabefilter dienen, z.B.:

```
docker inspect -f “{{ .NetworkSettings.Ports }}” my-container
```

Aufgabenstellung 6: Docker CLI

`6-exercise-docker-cli/exercise.md`

Docker Login / Logout

```
docker login [<options>] <registry-host>
```

```
docker logout <registry-host>
```

Beispiel:

```
docker login localhost:8080
```

```
docker login
```

```
docker logout localhost:8080
```

```
docker logout
```

Docker CLI: search / pull / push

`docker search <name>`

durchsucht DockerHub nach Images, die den angegebenen Namen enthalten

`docker pull <image>`

lädt das angegebene Image von DockerHub in den lokalen Docker-Cache

`docker push <registry-host[:registry-port]/image>`

kopiert das angegebene Image aus dem lokalen Docker-Cache in die angegebene Registry

Aufgabenstellung 7: Eigene Registry

`7-exercise-registry/exercise.md`

docker-compose

Problem:

Jeder Container einer Anwendungslandschaft muss einzeln via “docker run” gestartet werden.
Dabei muss auf Volumes, Umgebungsvariablen, Port-Mappings, etc. geachtet werden.

Ziel:

Einheitliches Starten der gesamten Umgebung mit nur einem Befehl

Pseudo-Lösung: Bash-Script (schlecht)

Lösung:

docker-compose

docker-compose

Beschreibung:

- docker-compose (minimalistisches Orchestrierungstool)
- Deklaration einer Anwendungslandschaft (Dokumentation)
- Deklaration mittels einer einzigen Datei (docker-compose.yaml)
- Einfaches Starten der Umgebung:

```
docker-compose up
```

- Einfaches Stoppen der Umgebung:

```
docker-compose down
```

docker-compose: CLI

`docker-compose build [options] [--build-arg key=val...] [SERVICE...]`

Baut die angegebenen Images

`docker-compose up [options] [--scale SERVICE=NUM...] [SERVICE...]`

Startet die angegebenen Images.

`option: --build`

`docker-compose down [options]`

Stoppt und löscht die Container, die via “docker-compose up” gestartet wurden.

docker-compose: Keywords

- version: Die Version der docker-compose Datei, bzw. der Umgebung
- volumes: Auflistung eigens benannter impliziter Volumes zur Referenzierung (DRY)
- services: Definiert alle Container/Services
  - image Der Name des zu verwendenden Images
  - container_name Container-Name für den gestarteten Container
  - ports Port-Mappings “Host <-> Container”
  - volumes Die Verzeichnisse/Dateien/Volume-Refs, die in den Container gemountet werden
  - environment Setzt Umgebungsvariablen (z.B. für die Applikation im Container)
  - entrypoint Überschreibt den ENTRYPOINT vom Dockerfile
  - command Überschreibt den CMD vom Dockerfile
  - build Verzeichnis, in dem das Dockerfile liegt und das dem Daemon zur Verfügung gestellt wird
- networks: Hier können zusätzlich eigene Netze definiert werden.

Beispiel: docker-compose.yaml

```
---
version: '3.5'

services:
 kibana:
   image: kibana:7.0.0
   container_name: kibana
   environment:
     LOGSPOUT: ignore
     ELASTICSEARCH_URL: http://elasticsearch:9200
   build:
     context: ./kibana
     Dockerfile: ./kibana/Dockerfile
   links:
     - elasticsearch
   ports:
     - "5601:5601"
…
```

```
---
version: ‘3.5’

services:
  cowsay:
    image: xcellenthub/whalesay:1.0
    container_name: whalesay
    build: ./cowsay
```

docker-compose: Parametrisierung

Sie können die Datei `.env` dafür nutzen, die im Dockerfile verwendeten Variablen zu setzen, z.B.:

.env:

```
BASE=ubuntu:18.10
TEMP_DIR=/tmp
```

Dockerfile:

```
FROM ${BASE}
COPY tmp ${TEMP_DIR}
```

`.env` muss neben dem verwendeten docker-compose.yaml liegen

Docker: Netzwerk

```
docker network create akdb
docker run --network=akdb ...
```

docker-compose.yaml:

```
networks:
	akdb:
		name: akdb
		driver: bridge
		ipam:
			driver: default
			config:
				- subnet: 10.0.0.0/24
services:
	reverse-proxy:
		image: nginx
		networks:
			-  myNet
```

docker-compose: Installieren

Dokumentation:

https://docs.docker.com/compose/install/

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.24.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

```
sudo chmod +x /usr/local/bin/docker-compose
```

Verify:

```
docker-compose --version
```

Docker CLI: Clean environment

Container stoppen:

```
docker kill [OPTIONS] CONTAINER [CONTAINER...]
```

```
docker rm -f CONTAINER [CONTAINER...]
```

Images löschen:

```
docker rmi [OPTIONS] IMAGE [IMAGE...]
```

Gesamte Docker Umgebung bereinigen:

```
docker system prune [OPTIONS]
```

```
docker system prune -f (Löscht alle gestoppten Container und nicht verwendete Images)
```

```
docker system prune -af (Löscht alle Container und Images)
```

Aufgabenstellung 8: docker-compose

`8-exercise-docker-compose/exercise.md`

Aufgabenstellung 9: Weitere Komponenten

`9-exercise-add-components/exercise.md`

Docker in der Praxis: Einheitliche Umgebungen

docker-compose:

- Sehr empfehlenswert für den lokalen Bau und Test von Anwendungslandschaften (POC)
- Orchestrierung und Betrieb mittels Docker Swarm möglich (nicht mehr empfehlenswert)

Ansible:

- universelles Provisionierungs-Tool, das auch für den Aufbau von Container-Landschaften verwendet werden kann
- bedingte, sehr aufwändige Orchestrierung möglich (nicht mehr empfehlenswert)
- bedingter Betrieb möglich z.B. mittels systemd (nicht mehr empfehlenswert)

Helm:

- für das Deployment in die Kubernetes-Cluster aller Stages (dev/test/preprod/prod)

Kubernetes:

- u.a. für die Skalierung, Verteilung und den Betrieb

Docker in der Praxis

Image updaten

Image Updates betreffen entweder das Basis-Image oder das daraus abgeleitete Anwendungs-Image.

Update des Basis-Images:

- Betrifft meist Patches oder neuere OS-Libraries

Update des Anwendungs-Images:

- Bugfixes und neue Features

Nicht vergessen:

- Anderen, eindeutigen Image-Tag verwenden (nicht *latest*)
- Zunächst nur in den Dev- und Test-Stages verproben (Rollbacks schnell und einfach möglich)
- Dann erst nach Prep bzw. Prod deployen

Images lokal nutzen

Ich habe eine Anwendung, die ich immer und überall brauche, wie mache ich mir das Leben am einfachsten? Z.B. Java-IDE, kubectl, ...

Nutze Docker!

1) Image suchen: *docker search <app-name>*
2) Wenn es kein passendes gibt, Anwendung selbst containerisieren (Dockerfile)
3) Evtl. docker-compose.yaml schreiben (Image-Name, Port-Mappings, Volumes, etc.)

Vorteile:

- Einheitlicher Start: *docker-compose up -d*
- Installiert die Anwendung beim ersten Start automatisch (keine Verunreinigung)
- Einheitliche Haptik (Color-Schemes, Preferences, etc.)

Ausblick

Ausblick

- Kubernetes
- Automatisierung
- IT-Sicherheit
- Weiterführende Quellen

Ausblick: Kubernetes (Orchestrierung)

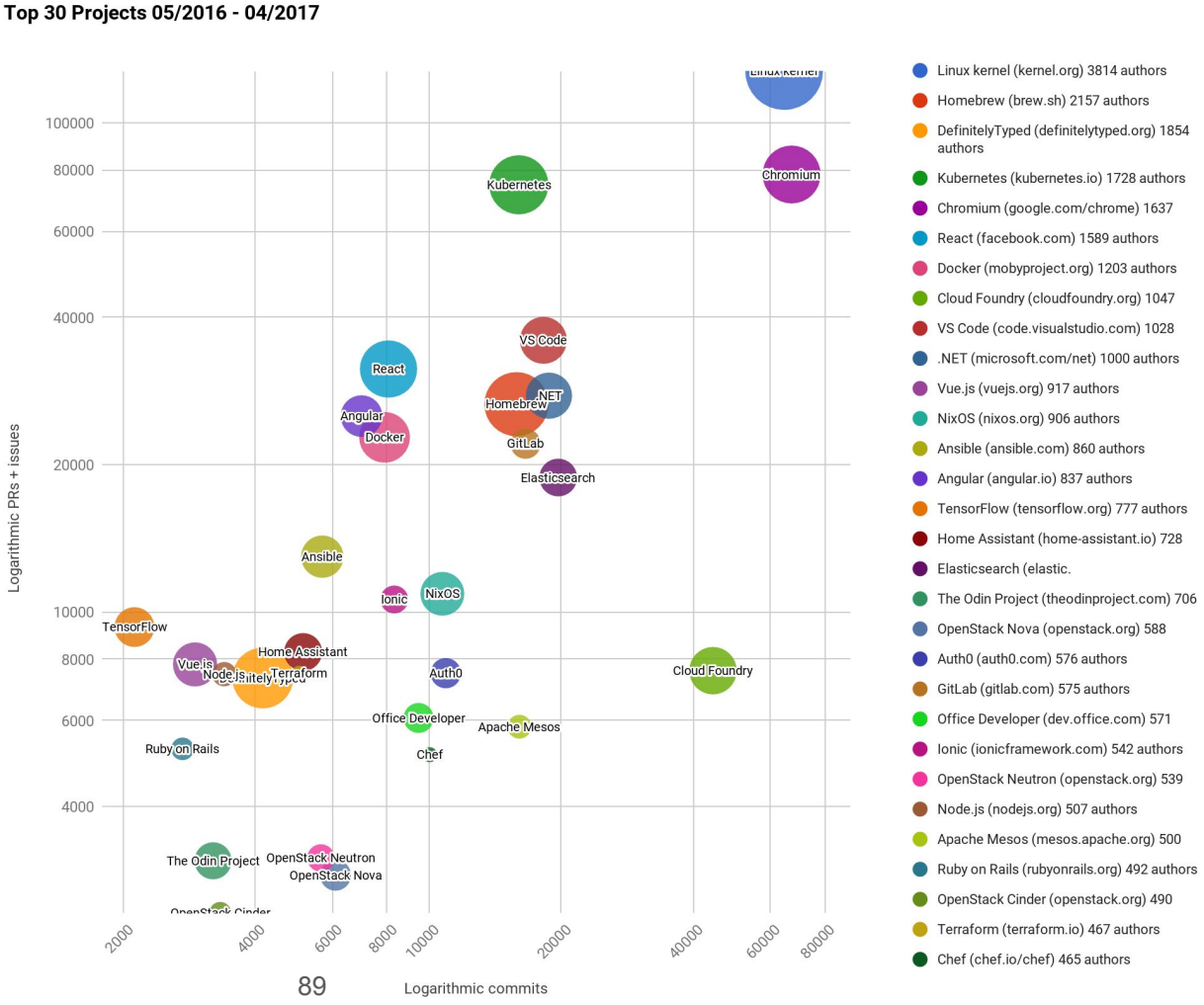
Vorteile:

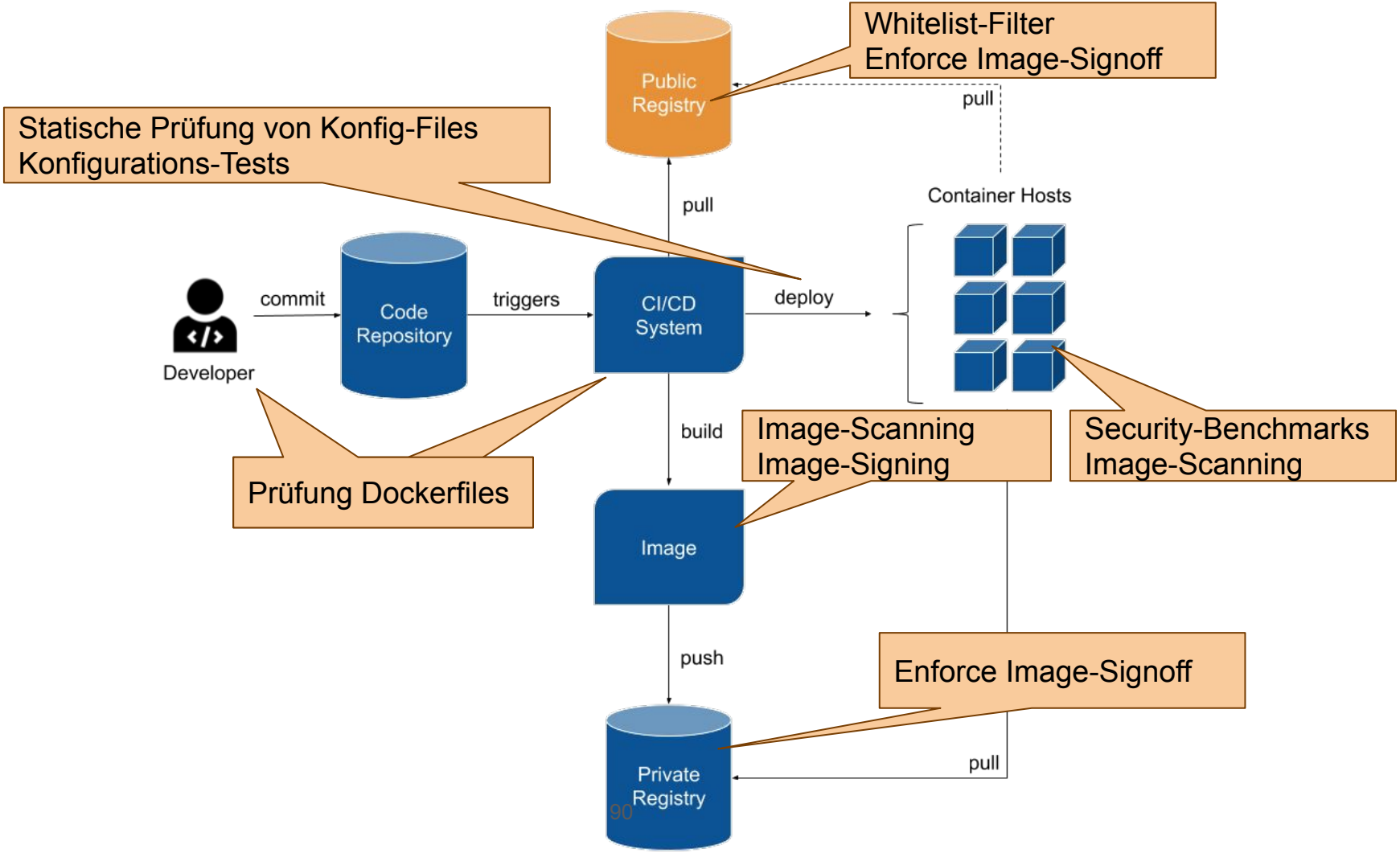
- Container über mehrere Hosts hinweg orchestrieren (⇒ Ausfallsicherheit, autoscaling)
- Hardware-Ressourcen effizienter nutzen
- Die Bereitstellung und Aktualisierung von Applikationen steuern und automatisieren (0% downtime)
- Storage mounten und Speicherkapazitäten hinzufügen, um zustandsbehaftete Applikationen auszuführen
- Applikations-Container und deren Ressourcen skalieren (automatisch, je nach Ressourcenverbrauch)

In Verbindung damit sind folgende Komponenten sinnvoll:

- Die Container-Registrierung mit Atomic Registry oder Docker Registry
- Networking mit OpenvSwitch und intelligentem Edge Routing
- Telemetrie mit Heapster, Kibana, Hawkular und Elastic
- Security mit LDAP, SELinux, RBAC (Role-based Access Control) und OAuth (Open Authorization) mit Multi-Tenancy-Layern, cilium.io
- Automatisierung mit Ansible-Playbooks für die Installation und das Cluster-Lifecycle-Management

Kubernetes Popularität




Public Registry
Whitelist-Filter
Enforce Image-Signoff
pull
Statische Prüfung von Konfig-Files
Konfigurations-Tests
pull
Container Hosts
commit
Code Repository
triggers
CI/CD System
deploy
Developer
build
Image-Scanning
Image-Signing
Security-Benchmarks
Image-Scanning
Prüfung Dockerfiles
Image
push
Enforce Image-Signoff
Private Registry
pull

Ausblick: Security in Container-Umgebungen

- Automatische Vulnerability Scans:
  - git push ⇒ CI ⇒ Docker build ⇒ Scan the created image ⇒ Report
- Automatische Schwachstellen, Überblick der laufenden Umgebung:
  - Liste aller laufenden Container mit Vulnerabilities
- Best Practices:
  - Dockerfile, User-Rechte, ...
- Security Richtlinien für Container:
  - BSI SYS 1.6 Container
  - Benchmarks
- Container Hardening

Portfolio-Überblick DevOps

- Einführung von DevOps Praktiken und dockerisierter Infrastrukturen (CI/CD)
- Migration von Bestand-Systemen auf Container-Plattformen
- Aufbau (Pilot-)Umgebungen mit Kubernetes
- Security- und Compliance-Scanning in Container-Umgebungen
- Qualitätssicherung von Docker-Containern
- Container-Betrieb (24x7 mit Rufbereitschaft)
- Architektur-Beratung zum Aufbrechen von Monolithen
- Coaching / Workshops / Schulungen für Entwickler / DevOps / OPs zu:
    - DevOps-Praktiken
    - Containerisierung
    - Anwendungsentwicklung nach 12-Factor Guideline
    - Docker, Ansible, Kubernetes

Offene Fragen?

Kontakt

Georg Baumgartner (Geschäftsführer)

- Email: georg.baumgartner@x-cellent.com
- Mobil: +49 170 / 566 0099

x-cellent technologies GmbH

- Rosenkavalierplatz 10
- 81925 München
- Tel. 089/929274-0

Quellen

https://github.com/alexryabtsev/docker-workshop