AKDB

Docker Workshop

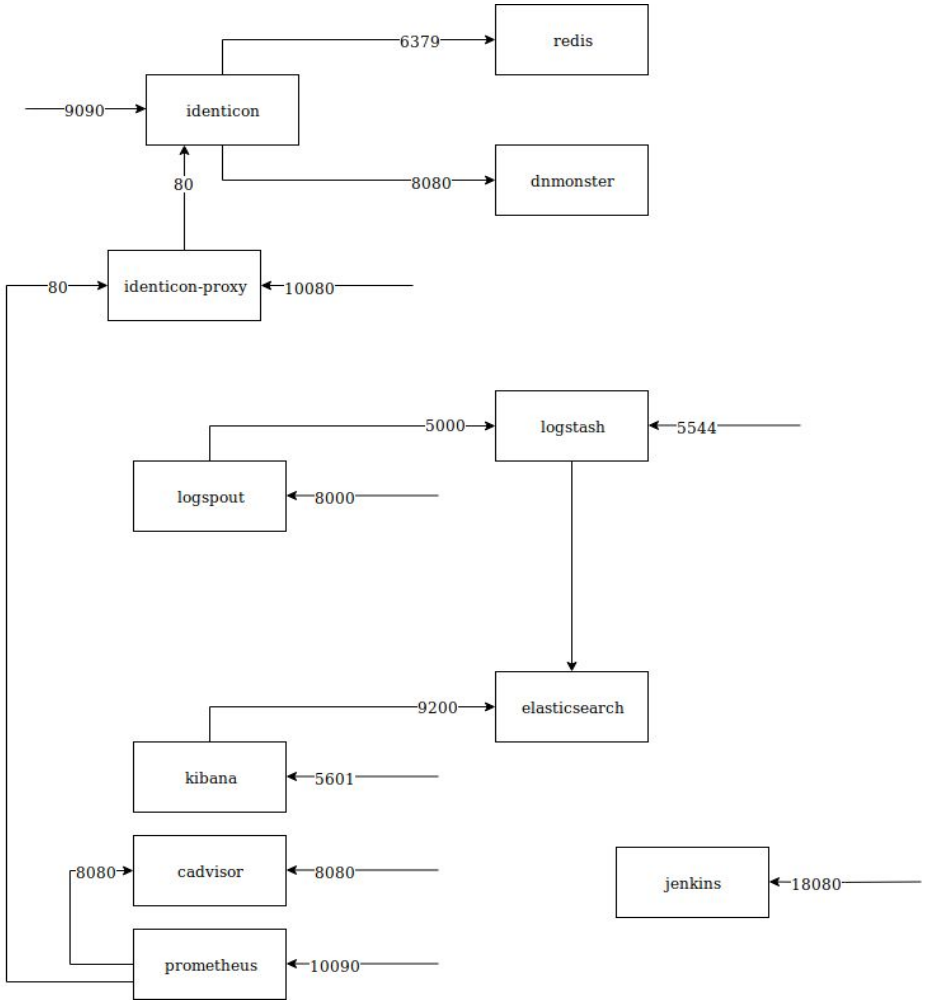
Agenda

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

Großprojekt

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

Großprojekt



Handout

- Cheatsheet
- Aufgabenblätter
- Ausgedruckte Präsentation
- Agenda / Zeitplan

1) Showcases

2) Grundlagen

Grundlagen: Was sind Container? (Docker)

Docker ist ein open-source Paketierungs- und Ausbringungswerkzeug von Software.

Die Paketierung ist vergleichbar mit der Container-Verladung in einem Hafen:

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

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 (vermeidet z.B. JAR-Hell)

Damit läuft ein aus einem solchen Image gestarteter Container auf **jedem** Docker-Host.

Grundlagen: Unterschied Container VS VMs

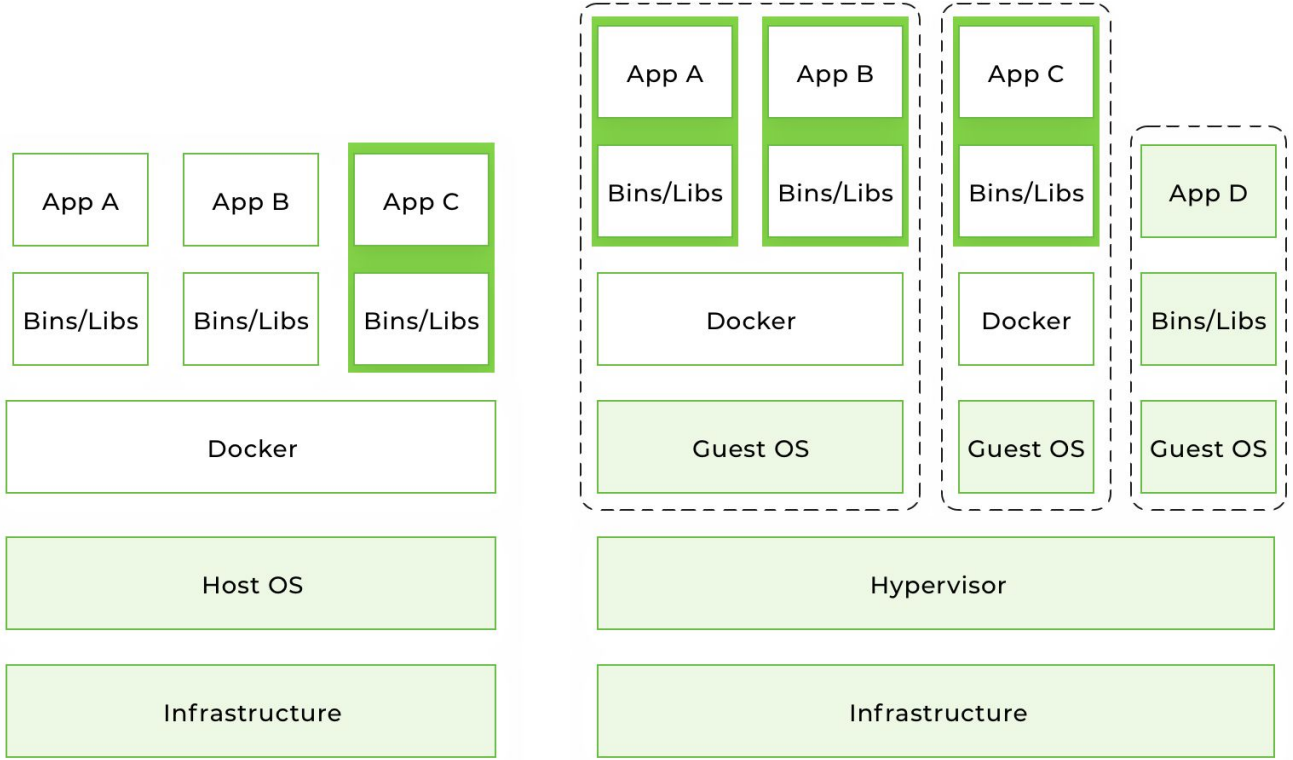
Eine VM simuliert die gesamte PC-Hardware via Software. Dies frisst immense Ressourcen und schlägt sich vor allem in der Performance beim Boot-Vorgang nieder.

Vorteil ist die bessere bzw. sicherere Isolierung von anderen VMs und dem Host selbst.

Docker Container sind dagegen nur durch die Linux **namespace** und **cgroups** Kernel-Technologien voneinander isoliert. Angreifer können hier “leichter” von einem laufenden Container aus das Host-System angreifen.

VMs und Container schließen sich allerdings nicht aus. Gerade wegen dem Sicherheitsaspekt werden Container auch innerhalb einer VM eingesetzt. Einmal gestartet sind dann die Vorteile beider Technologien vereint.

Grundlagen: Unterschied Container VS VMs



Grundlagen: Vorteile von Docker

- Standardisierte Verpackung + Ausbringung
- 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 eine Verzeichnisstruktur beinhaltet, z.B.:

Archiv-Inhalt:

```
folderA/
folderB/
file1
```

Sicht innerhalb des Containers:

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

Terminologie: Image

Docker Layer können gestapelt werden, d.h. ihnen liegt dieselbe Wurzel zugrunde. Oberhalb liegende Layer *verdecken* bzw. *erweitern* die Dateisystem-Inhalte der unter ihnen liegenden Layer, d.h. Dateien werden geshared.
Verändert ein Layer eine unterhalb liegende Datei, so wird die **Copy-on-Write** Strategie angewendet, d.h. die Datei wird zunächst in den oberen Layer kopiert und dort dann verändert.

Ein **Docker Image** besteht aus 1 bis n gestapelten read-only Layern mit jeweils global eindeutiger ID. Diese Vereinigung ist technisch gesehen ebenfalls ein einzelnes Archiv.
Damit kann es z.B. mittels “`scp`” zwischen zwei Computern ausgetauscht werden.

Terminologie: Container

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

Anschaulich gesprochen wird dem read-only Layer-Stapel des Images ein weiterer **read-write** Layer oben aufgelegt und in diesem dann der ausgewiesene Hauptprozess gestartet.

Dieser 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.

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>` beinhaltet die Adresse der Docker-Registry, in der das Image hinterlegt ist.

Terminologie: Docker Registry / DockerHub

Eine Docker Registry ist ein Web-Server, der ähnlich eines File-Servers Docker Images zum Download bzw. auch Upload zur Verfügung stellt.

Es gibt eine offizielle über `hub.docker.com` öffentlich zugängliche Registry (DockerHub), die immer dann referenziert wird, wenn der Host-Anteil der Image-Kennung fehlt, d.h. implizit stellt Docker solchen Image-Kennungen den registry-host "`docker.io/`" voran, z.B:

"`jenkins`" gleichbedeutend mit "`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 (1/2)

Damit das Dateisystems eines Containers nicht völlig von der “Außenwelt” (dem Host-Dateisystems) abgeschottet ist, kann man sogenannte “Volumes” definieren.

Ein Volume ist nichts anderes wie ein gemountetes Host-Verzeichnis bzw. Host-Datei an die vorgesehene Stelle innerhalb des Container-Dateisystems.

Explizit ausgewiesene Volumes (Mountpoints), deren Host-Pfad vom Benutzer angegeben wurden, werden auch “Bind-Mounts” genannt. Implizite Volumes werden von Docker verwaltet und liegen unterhalb von “`/var/lib/docker/volumes`”

Terminologie: Volume (2/2)

Volume Bauart:

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

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

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

Existiert die angegebene Datei oder Verzeichnis bereits im Container, so wird es von dem Mountpoint “überdeckt”, d.h. es ist beim Container-Start **nicht** möglich, Dateien oder Verzeichnisse aus diesem via Volumes auf das Host-System zu übertragen.

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 alternativen Daemons kommunizieren.

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 daraus entstandenen Docker-Images eindeutig beschreibt.

Neben dieser Dokumentation dient es vor allem der Erstellung des darin beschriebenen Images mittels “`docker build`”.

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 als Grundlage hergenommen werden. “FROM scratch” bildet hier eine Ausnahme und bestimmt, dass von einem leeren Dateisystem “/” geerbt wird.

Allgemein wird nun pro nachfolgendem Dockerfile-Schlüsselwort (bis zum Ende oder nachfolgendem FROM) dem aktuell zugrunde liegenden read-only Layer-Stapel ein read-write Layer aufgesetzt, in dem die konkrete Instruktion ausgeführt wird. Dies entspricht einem Container, der vom aktuellen Layer-Stapel gestartet wird und in dem dann die Instruktion ausgeführt wird. Anschließend wird dieser on-top Layer “committed”, d.h. dem Ausgangsstapel als fest “gebrannter” read-only Layer hinzugefügt. Dieser neue Layer-Stapel dient nun allen nachfolgenden Dockerfile-Instruktionen als Ausgangsbasis, d.h. ein Dockerfile beschreibt im Grunde genommen den “additiven Aufbau eines read-only Layer-Stapels”.

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> <Dockerfile>`" 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 hier ein Image names “`myapp`”, das aus einer einzigen ausführbaren Datei “`/myapp`” besteht. Dieses Binary wurde im Vorfeld von dem Hilfsimage “`builder`” gebaut. Bringt man dieses Image mit “`docker run …`” zur Ausführung, so würde ein Container entstehen, der “`/myapp`” unter der PID 1 laufen ließe.

RUN

`RUN <command>` (*shell* form, the command is executed in a shell, which by default is "`/bin/sh -c`" on Linux or "`cmd /S /C`" on Windows)

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

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

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

Damit würde der Paket-Manager aktualisiert und dann das Programm "`telnet`" installiert werden.

Die *exec* Form setzt keine bereits installierte Shell voraus.

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.

ENTRYPOINT (1/2)

Das Schlüsselwort `ENTRYPOINT` hat zwei Ausprägungen:

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

Nur die letzte `ENTRYPOINT` Instruktion der letzten Stage (falls vorhanden) definiert den Prozess, der beim Container-Start ausgeführt wird, z.B. würde ein Container beim Start des folgenden Images den aktuellen Inhalt des Verzeichnisses "`/var`" ausgeben.

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

ENTRYPOINT (2/2)

Beispiel:

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

Aufgabe: Welche Shell-Kommandos würden ausgeführt werden?

1. `docker run -it --rm --name test topImage -H`
2. `docker run topImage`

CMD

Die `CMD` Instruktion hat zwei Ausprägungen:

- `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 Effekt für 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”]
```

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="1.0"

LABEL multi.label1="value1" \
      multi.label2="value2" \
      other="value3"

LABEL maintainer="F.E. <ferdinand.eckhard@x-cellent.com>, S.K. <sandro.koll@x-cellent.com>"
```

LABEL Standards bei der AKDB

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 bei der AKDB

- 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
  b. AKDB: The tag is used

LABEL Standards bei der AKDB: Maven

Demo: AKDB Label-Nutzung mit Maven

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-kennung|image-id>` eingesehen 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.

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

fügt dem Image Umgebungsvariablen hinzu.

Remaining Instructions (2/2)

`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 9” oder “STOPSIGNAL SIGKILL”)

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

`HEALTHCHECK NONE`

setzt den Status eines Containers (starting|healthy|unhealthy) abhängig vom angegebenen Healthcheck.

Containerisieren

Vorgehen:

1. Neues Dockerfile erstellen
2. Basis Image auswählen (kleinstmöglich)
3. Abhängigkeiten installieren
4. ggf. benötigte Daten, Konfigurationen und Applikationen kopieren, z.B. Multi-Staging:
   a. Container 1 compiliert Applikation
   b. Container 2 kopiert nur das Ergebnis
      ⇒ weniger Abhängigkeiten und keine ungenutzten Daten im laufenden Container
5. ggf. Entrypoint setzen

Containerisieren

Beispiel: *Nginx*

Manuell im Container:

```
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 schneller mit offiziellem Nginx-Image:

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

docker build: Proxy nutzen

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

```
docker build . --build-arg
http_proxy=http://proxy.akdb.net:3128
--build-arg
https_proxy=http://proxy.akdb.net:3128
```

Variable	`docker run` Example
HTTP_PROXY	--env HTTP_PROXY="http://proxy.akdb.net:3128"
HTTPS_PROXY	--env HTTPS_PROXY="https://proxy.akdb.net:3128"
FTP_PROXY	--env FTP_PROXY="ftp://proxy.akdb.net:3128"
NO_PROXY	--env NO_PROXY="localhost,127.0.0.1,*.test.example.com"

docker build: Proxy nutzen

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

```
vim ~/.docker/config.json proxy.akdb.net:3128
```

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

- **DRY: Nur 1 mal machen!**

docker build: Proxy nutzen

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

```
sudo mkdir -p /etc/systemd/system/docker.service.d

vim /etc/systemd/system/docker.service.d/http-proxy.conf

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

docker build: Proxy nutzen

Zertifikate in den Container Laden,

...

Aufgabenstellungen: Beschreibung

Alle Aufgabenstellungen sind hier in der Präsentation, im Handout und in dem Git Projekt zu finden:

1) Clonen Sie das Repository:

```
git clone https://github.com/x-cellent/dt.git
```

2) Wechseln Sie in den Aufgaben-Branch

```
git checkout 1-exercise-write-dockerfile
```

3) Lesen Sie die Aufgabenstellung in ./exercise.md

Aufgabenstellung 1: Dockerfile schreiben

Aufgaben:

Befüllen Sie das Dockerfile “./cowsay/Dockerfile”

a) Es soll das Basisimage xcellenthub/whalesay nutzen
b) Die Applikation “cowsay” soll darin ausgeführt werden
c) Optional: Erweitern Sie das Dockerfile so, dass generierter Text ausgegeben wird:
    i) Installieren Sie dafür `fortunes`
    ii) Pipen Sie die Ausgabe von `/usr/games/fortune` als Übergabeparameter an cowsay

Dockerisieren Sie die beiden fehlenden Applikationen

- countdown
- identicon

Docker in der Praxis: Eigene Images nutzen

Auch neue Images beauftragbar!

⇒ Philipp

Oder einfach selbst bauen:

- Imagename anpassen (*camelCase* to *lowercase*)
  - https://172.18.26.73/akdb/egov/bsp/build/spring-boot-build/blob/master/spring-boot-build/pom.xml#L39
  - <repository>${docker.registry.url}/${**env.CI_PROJECT_NAMESPACE**}/${**env.CI_PROJECT_NAME**}</repository>
    - build-arg überschreiben
- Proxy setzen
- Herkunft des Images (Vertrauenswürdig?)
- Inhalte des Images (Installierte Komponenten, Lizenzen?)
- Auf Wartung achten! (Regelmäßige Updates des Images)

Docker in der Praxis: Eigene Base-Images nutzen

Was sind Basis Images?

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

Vorteile:

- Einheitliche Basis für mehrere Systeme
- Es muss nur auf wenige Base-Images geachtet werden (Wartung, immer aktuell)
- Sie sind robust und sicher

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 -it -v $PWD:/pwd:ro hadolint/hadolint hadolint /pwd/Dockerfile
  - docker run --rm -i 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

Aufgaben (2-exercise-linting):

1) Überprüfen Sie die folgenden Dateien mit hadolint:
   a) ./cowsay/Dockerfile
   b) ./jenkins/Dockerfile
   c) ./identicon/Dockerfile
   d) ./countdown/Dockerfile
2) Verbessern Sie diese Dockerfiles

Hinweise:

- hadolint Aufruf: docker run --rm -i hadolint/hadolint < Dockerfile
- Package Versionen mit --version oder `apt-cache policy PACKAGENAME`

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

1. Bauen Sie ein Image aus dem jenkins Dockerfile
   a. ./cowsay/Dockerfile
   b. ./identicon/Dockerfile
   c. ./countdown/Dockerfile
   d. ./jenkins/Dockerfile
2. Geben Sie alle lokal gebauten Images aus
3. Alle für die Aufgaben benötigten Befehle können Sie in die `docker-build.sh` Datei schreiben

Docker Run

Startet einen neuen Container mit einem Befehl

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

Beispiel:

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

docker run xcellenthub/whalesay:1.0 cowsay Hi

Ausgabe:

```
< Hi >
 ----
   \
    \
     \
                    ##         .
              ## ## ##        ==
           ## ## ## ##       ===
       /""""""""""""""""___/ ===
  ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
       \_____ o           __/
        \    \         __/
         \____\_____/
```

Docker Run: Options

Zu jedem kurze erklärung + beispiel + Zeigen

```
docker run -it ubuntu /bin/bash
```

-i, --interactive	Keep STDIN open even if not attached
-t, --tty	Allocate a pseudo-TTY

```
docker run --name my-nginx -d -p 10080:80 -v /nginx:/usr/share/nginx/html:ro nginx
```

--name string	Assign a name to the container
--rm	Automatically remove the container when it exits
-p, --publish list	Publish a container's port(s) to the host
-v, --volume list	Bind mount a volume
--detached, -d	Run detached
-P, --publish-all	Publish all exposed ports to random ports
-e, --env list	Set environment variables
--entrypoint string	Overwrite the default ENTRYPOINT of the image
--privileged	Give extended privileges to this container

Docker: Persistenz

Grundregel: Container sind Wegwerfprodukte!
⇒ das plötzliche Beenden eines Containers sollte nicht zu Datenverlusten führen!

Neben dem Bind-Mounten von Hostsystempfaden (docker run -v path:path ...), ist es auch möglich, Volumes für die Persistenz auf dem Host System zu nutzen.

Das lokale Dateisystem von Containern ist nicht persistent! Dafür werden Volumes oder Bind-Mounts benötigt.

Das Backup der genutzten persistenten Daten (z.B. config files, Datenbanken, logs) auf dem Hostsystem muss durch externe Programme realisiert werden. Diese Programme können wiederum containerisiert sein.

Für Testzwecke kann man sich auch während des Betriebs Daten aus dem Container zur Analyse kopieren:

```
docker cp
```

Readonly Container

Start containers as read only:

```
docker container run --rm --read-only alpine:3.7 touch hello.txt
```

With read only volumes:

```
docker container run --rm --read-only -v /host/path:/container/path:ro alpine:3.7 touch hello.txt
```

Aufgabenstellung 4: Docker Run

1) Starten Sie die bisher erstellten Images
   a) `xcellenthub/whalesay:2.0`
   b) `xcellenthub/jenkins:1.0`
   c) `xcellenthub/identicon:1.0`
   d) `xcellenthub/countdown:1.0`
2) Finden Sie heraus, auf welchen Ports der Jenkins im Container lauscht.
   (Tipp: Boot-Logs, Configfile durchsuchen oder offene ports anzeigen)
3) Mappen Sie diesen Port auf das Hostsystem
4) Verifizieren Sie, dass der Port durchgereicht wird
5) Welche Rechte haben Sie in dem Container? (UID)
6) Welche Rechte hat eine von Ihnen im Container erstellte Datei auf dem Hostsystem?
   (Tipp: Hostsystem-Ordner mounten)
7) Installieren Sie in dem laufenden Container eine Applikation Ihrer Wahl, z.B. “sl”, “vim”, “telnet”...
   Ist dies nach einem Neustart des Containers noch installiert?
8) Optional: Verschaffen Sie sich root-Rechte auf Ihrem Host-System! Führen Sie “chmod o+w /bin/ls” (“chmod o-w /bin/ls”).
   (Tipp: --privileged + Volume)

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.

Usermapping Docker-Hostsystem

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

- Konfiguration beim starten des containers:

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

- Configuration in dem docker-compose.yaml File (später)

```
service
      name
            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 --rm -it ubuntu bash

docker exec -it ubuntu_bash /bin/bash

docker exec -it ubuntu_bash top
```

Aufgabenstellung 5: Docker Exec

1. Starten Sie einen NginX Container
2. Verbinden Sie sich auf den Container
3. Modifizieren Sie die Website (`/usr/share/nginx/html/index.html`) zur Laufzeit
4. Verifizieren Sie die Änderung durch Aufruf der Seite

Docker Start / Stop

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

Requirements:

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

Stop:

```
docker stop $containerID
docker stop simpleApp
```

Start:

```
docker start $containerID
docker start simpleApp
```

Docker CLI: Debugging

`docker inspect <container|image>`

zeigt die Meta-Daten des Containers/Images an

`docker ps [<options>]`

zeigt die ID, Image, PID 1 Command, Port-Mapping, usw. von laufenden/gestoppten Containern an

`docker logs <container|container-id>`

zeigt den Ausgabe- und Errorstream des gegebenen 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.

Befehl:

```
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: Daten austauschen

Mit

```
docker cp <container-name/id>:<container-path> <host-path>
```

können Dateien vom Container auf das Host-System kopiert werden.
Vice-Versa:

```
docker cp <host-path> <container-name/id>:<container-path>
```

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

1) Starten Sie die Komponenten des identicon Systems

- `xcellenthub/identicon:1.0`
- `xcellenthub/dnmonster`
- `xcellenthub/redis`

2) Durchsuchen Sie die Container nach folgenden Inhalten

- `names`
- `port mappings`
- `volumes`
- `IP addresses`
- `labels`
- `container IDs`

3) Speichern Sie das identicon Image als Archiv ab

4) Löschen Sie das identicon Image aus dem lokalen Docker-Cache

5) Laden Sie das Image aus der Archiv-Datei in den lokalen Docker-Cache

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 in der Kennung enthalten

`docker pull <image>`

lädt das angegebene Image aus der Registry in den lokalen Docker-Cache

`docker push <image>`

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

Aufgabenstellung 7: Eigene Registry

1. Starten Sie eine private, lokale Registry
2. Pushen Sie ein Image in diese private Registry
3. Löschen Sie das lokale Image
4. Überprüfen Sie, dass dieses aus dem lokalen Docker-Cache gelöscht wurde
5. Laden bzw. starten Sie das Image aus der privaten Registry
6. Suchen Sie nach Docker-Images, die die Applikationen kibana, logstash und elasticsearch beinhalten. Schreiben Sie hierfür ein `docker-run.sh` Skript, das diese Images startet und miteinander verbindet.

Erfüllen Sie folgende Aufgaben für die nachfolgende Liste typischer Basis-Images:

- Download ins lokale Docker-Cache (Vergleichen Sie die Dauer)
- Vergleichen Sie die jeweiligen Größen
- Starten Sie jeweils einen Container und installieren Sie darin die Programme “telnet” und “vim”

Liste typischer Basis Images:

- centos
- debian
- alpine
- busybox

docker-compose

Aktueller Stand:

Einzelne Container können manuell gestartet werden

Ziel:

Automatisches Starten der gesamten Umgebung mit Abhängigkeiten zueinander

Lösung:

docker-compose

docker-compose

Beschreibung:

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

Starten der Umgebung:

docker-compose up

Stoppen der Umgebung:

docker-compose down

docker-compose: CLI

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

Services are built once and then tagged, by default as `project_service`.

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

Builds, (re)creates, starts, and attaches to containers for a service.

`options: --build`

`docker-compose down [options]`

Stops containers and removes containers, networks, volumes, and images created by `up`.

docker-compose: Keywords

- version: Die Version der docker-compose Datei, bzw. der Umgebung
- volumes: Auflistung eigens benannter impliziter Volumes zur Referenzierung (DRY)
- services: Hier werden alle Container/Services definiert
  - 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 — Hier können Umgebungsvariablen gesetzt werden (z.B. für die Applikation im Container)
  - entrypoint — Entspricht ENTRYPOINT vom Dockerfile
  - command — Entspricht 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 eine Datei `.env` dafür nutzen (muss neben dem verwendeten docker-compose.yaml liegen), die im Dockerfile verwendeten Parameter zu setzen, z.B.:

.env:

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

Dockerfile:

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

Docker: Netzwerk

docker network create myNet

docker run --network=myNet

docker-compose.yaml:

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

(bridge | overlay | macvlan)

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

1. Starten Sie die derzeitige Umgebung (ohne docker-compose).
2. Verifizieren Sie, dass alle drei Container laufen und untereinander verbunden sind
3. Stoppen Sie die Umgebung
4. Erweitern Sie die Umgebung um die fehlenden Komponenten:
    - elasticsearch
    - identicon
    - dnmonster
    - redis
    - jenkins
    - reverse-proxy
    - logspout
    - logstash
    - elasticsearch
5. Verifizieren Sie, dass die gesamte Umgebung korrekt funktioniert:
    - Löschen und stoppen Sie alle Images und Container
    - Starten Sie nun die gesamte Umgebung mittels docker-compose

Übung 9: Weitere Komponenten

Bauen Sie weitere Komponenten in die Umgebung ein, z.B. die folgenden:

- Prometheus (Monitoring)
- cAdvisor (Analyse der Container-Ressourcen)

Alternativ:

Dockerisieren Sie eine (kleine) AKDB Anwendung

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 Provisionierungstool, 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 das/die Kubernetes-Cluster aller Stages (dev/test/preprod/prod)

Kubernetes:

- für die Skalierung und den Betrieb

Docker in der Praxis

Image updaten

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

Update des Base-Images:

- Betrifft meist Patches oder neuere OS-Libraries

Update des Anwendungs-Images:

- Bugfixes bzw. neue Features

Nach den Updates **unbedingt** beachten:

- 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)

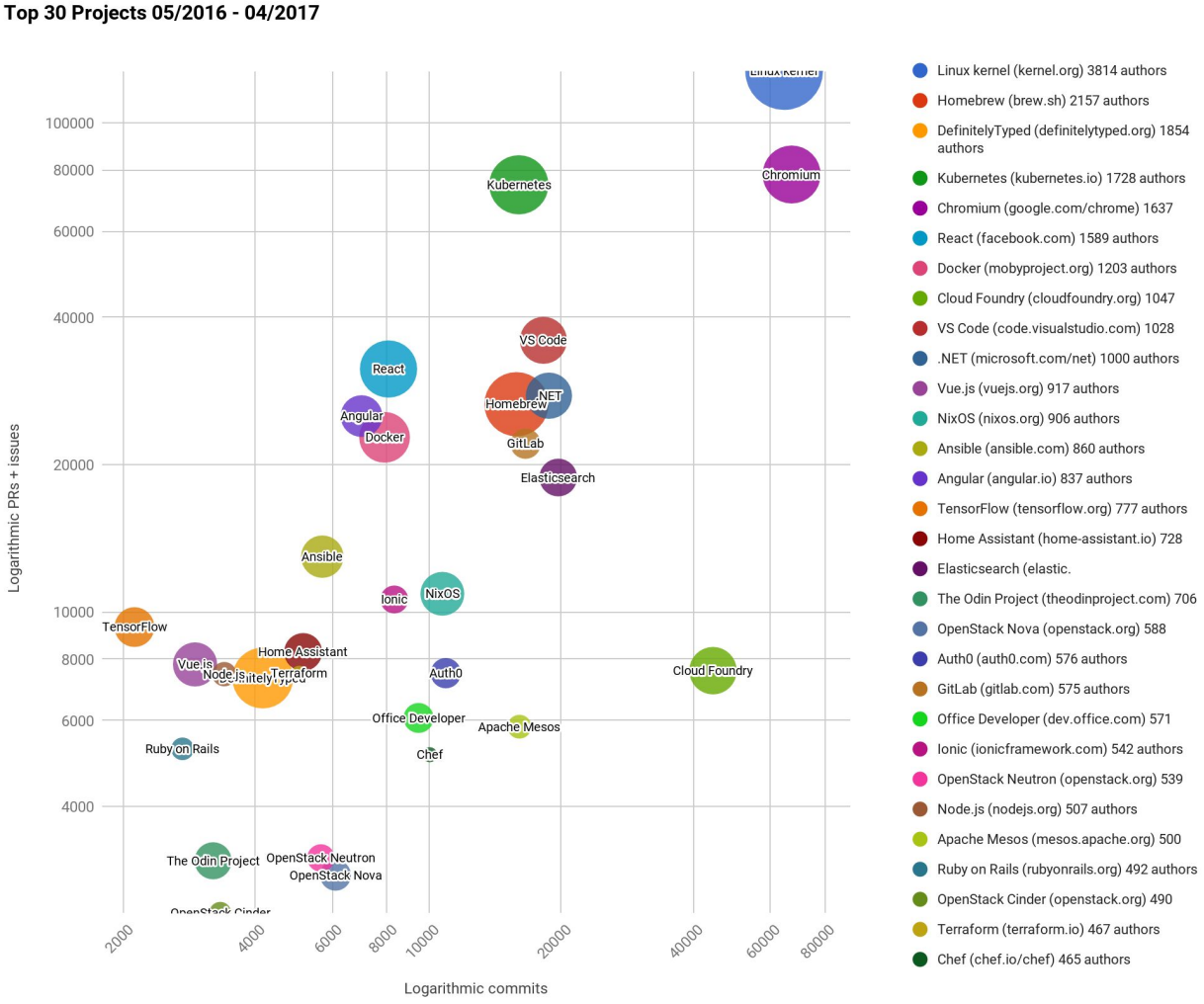
Vorteile:

- 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:

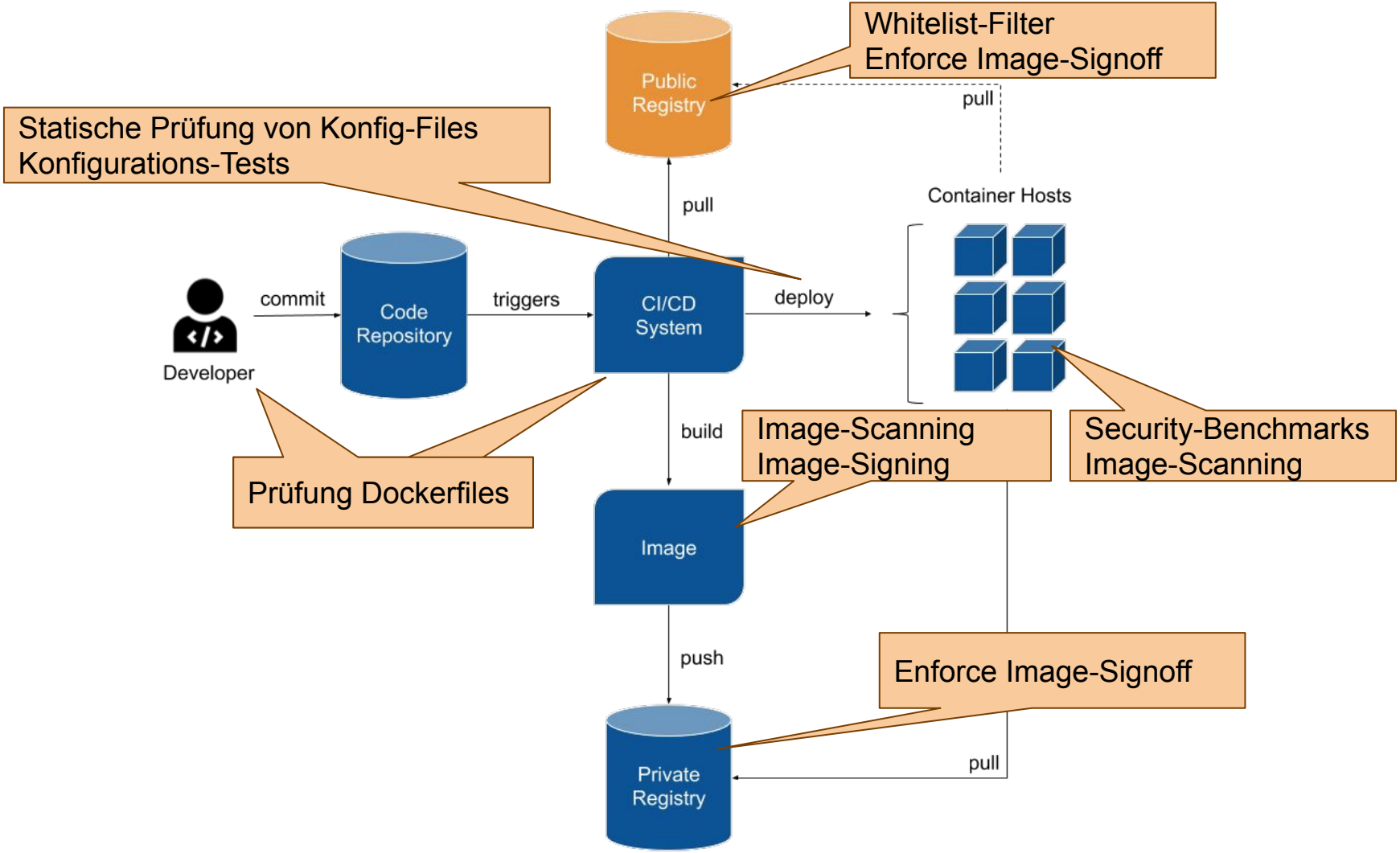
- Die Container-Registrierung mit Atomic Registry oder Docker Registry
- Networking mit OpenvSwitch und intelligentem Edge Routing
- Telemetrie mit Heapster, Kibana, Hawkular und Elastic
- Security mit LDAP, SELinux, RBAC (Role-based Access Control) und OAuth (Open Authorization) mit Multi-Tenancy-Layern,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

Kubernetes Popularität



Ausblick: Automatisierung (CI)

⇒ Consulting, Projektbegleitung, Schulung


Public Registry
Whitelist-Filter
Enforce Image-Signoff
pull
Statische Prüfung von Konfig-Files
Konfigurations-Tests
pull
Container Hosts
Developer
commit
Code Repository
triggers
CI/CD System
deploy
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 DevOps Praktiken und dockerisierter Infrastrukturen
- Migration von Bestandssystemen 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

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

Bild: Docker VS VM *https://github.com/alexryabtsev/docker-workshop*