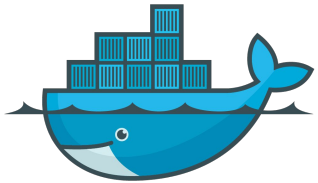


docker Workshop

Von der Installation bis zur Automatisierung

Agenda

neofonie*
Data. Technology. Services.

10:00 - 10:20	Begrüßung & Vorstellung (Neofonie Weg zu Docker, Vorstellung aller)
10:20 -	Einführung in Docker
11:00 -	Installation
11:30	Images & Containers
12:45 - 14:00	- *Mittagspause* -
13:30	Links und Cluster
14:30	Orchestrierung mit docker-compose
15:30	Ausblick docker-machine und Docker Ökosystem
16:00 - 18:00	FAQ und praxisnahe Aufgabenstellungen

Was ist Docker?

neofonie*
Data. Technology. Services.

- zentrales Produkt von Docker Inc.
  - initial von dotCloud entwickelt, erster GitHub-Commit 01/2013
  - von verschiedenen Cloud-Anbietern supportet (Google, Amazon, DigitalOcean usw.)
- DevOps-Tool
- Kapselt Anwendungen und deren Abhängigkeiten in sog. Images
- Versteht Instanzen der Images als sog. Container
- Open Source, geschrieben in Go

Was ist Docker ?

Deployment Schnittstelle

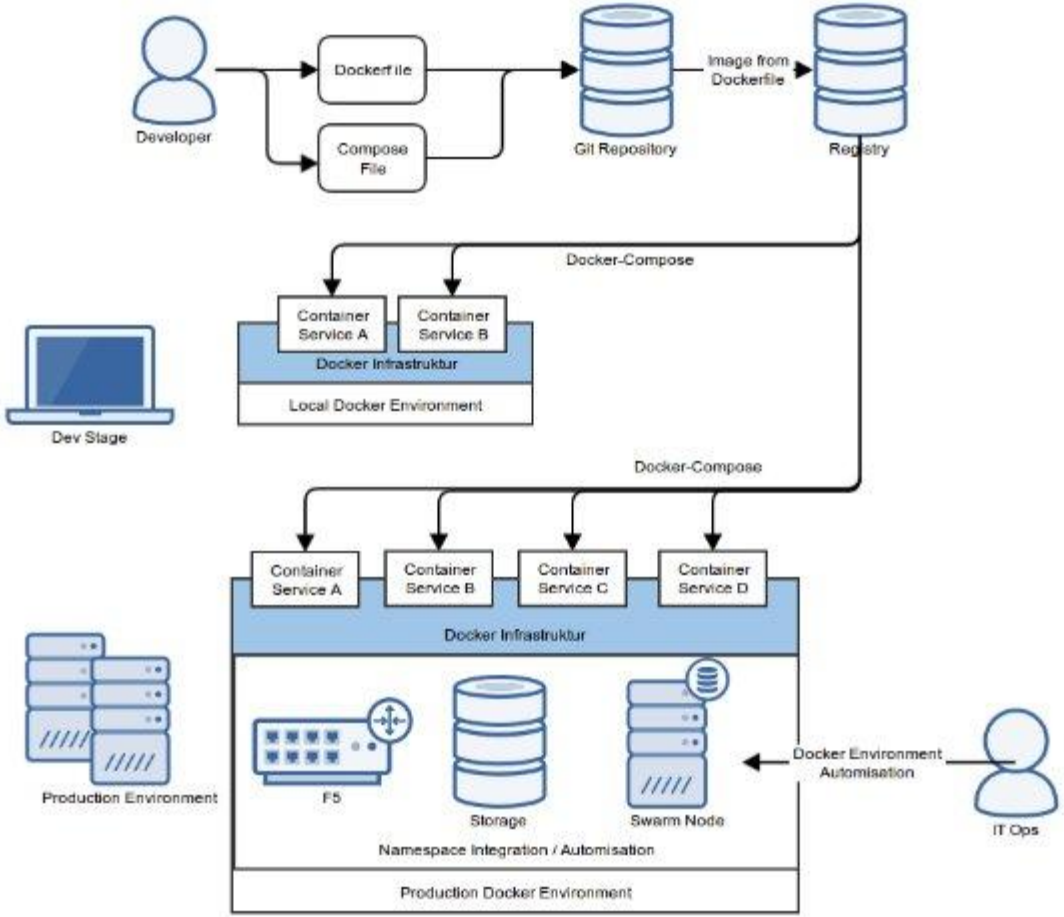
Was ist Docker ?

Deployment Schnittstelle

Anwendungsgebiete

neofonie*
Data. Technology. Services.

- Entwicklung
  - Verwaltung und Verteilung von Tools (IDE, Bash-Toolkits, Environments)
  - Implementierungen unter Live-Bedingungen
- QA / QM
  - (automatische) Tests unter Live-Bedingungen
- Deployment & Betrieb
  - hohe Portierbarkeit (Test-, Staging-, Live-Server, eigenes RZ, Cloud etc.)
  - Virtualisierung ohne Hypervisor
  - Skalierbarkeit


Developer
Dockerfile
Compose File
Git Repository
Image from Dockerfile
Registry
Docker-Compose
Container Service A
Container Service B
Docker Infrastruktur
Local Docker Environment
Dev Stage
Docker-Compose
Container Service A
Container Service B
Container Service C
Container Service D
Docker Infrastruktur
Production Environment
F5
Storage
Swarm Node
Docker Environment Automisation
IT Ops
Namespace Integration / Automisation
Production Docker Environment

Basis-Komponenten

neofonie*
Data. Technology. Services.

Docker besteht aus mehreren Komponenten, die über APIs mit einander kommunizieren:

- Docker Daemon
- Docker Client
- Container
- Images
- Registry

Docker Daemon

neofonie*
Data. Technology. Services.

- Herzstück der Docker Infrastruktur
- Schnittstelle zum Wirt OS
- bietet TCP-, Unix Domain Socket- und Systemd- Schnittstelle
- benutzt Linux Kernel Funktionen (cgroups)
- ist keine Hardware Virtualisierung

Docker Client

neofonie*
Data. Technology. Services.

- CLI zum Docker Daemon
- Container starten, stoppen und löschen
- Images bauen, speichern und löschen
- Informationen über Container und Images ausgeben

Docker Container

neofonie*
Data. Technology. Services.

- ausführbare Instanz eines Image
- ~~ist nicht persistent~~
  sollte als nicht persistent verstanden werden!
- zum Starten eines Containers braucht man mindestens
  - einen Docker Client
  - einen laufenden Docker Daemon
  - ein verfügbares Docker Image

Docker Image

neofonie*
Data. Technology. Services.

- nicht schreibbares Layer Filesystem
- wird aus einem Dockerfile erstellt
- enthält alle Abhängigkeiten einer Anwendung (Libraries, Binaries etc.)
- Basis für den Start von Containern

Docker Registry

- sehr einfache Image Datenbank
- als Docker Image verfügbar

Die Registry verfügt über kein integriertes Browser Interface / GUI. Hierzu gibt es Projekte wie

- Portus
- docker-registry-ui (GitHub)
- ...

Installation

```
~$ git clone https://github.com/Neofonie/DockerBasics.git
```

In `~/DockerBasics/workshop.md` findet ihr die URLs zu den Install-Ressourcen

Sollte der Docker Daemon nicht starten hilft folgender Check meist weiter

```
wget https://raw.githubusercontent.com/docker/docker/master/contrib/check-config.sh
bash check-config.sh
```

neofonie*
Data. Technology. Services.

Erste Container

Erste Container

neofonie*
Data. Technology. Services.

In diesem Abschnitt

- erstellen und starten wir unsere ersten Container
- loggen wir uns in laufende Container ein
- verschaffen wir uns Überblick
- beenden und löschen wir Container

Hello World!

```
~$ docker run busybox echo “hello world”
hello world
~$
```

Starten eines Containers

Wir können einfach Debian in einem brandneuen Container starten

```
~$ docker run -it debian:8
root@6c82121db0fc:/# cat /etc/issue
Debian GNU/Linux 8 \n \l
root@6c82121db0fc:/# exit
exit
~$
```

`-it` veranlasst den Docker Client uns zu stdin zu verbinden (-i) und gibt uns ein tty (-t).

Starten eines Containers

Der Hello World-Container für diesen Workshop ist weltweit verfügbar:

```
~$ docker run -p=7890:80 -it neofonie/hello-world
AH00558: httpd: Could not reliably determine the server's fully qualified domain
name
[...]
[Thu Apr 07 09:48:31.400690 2016] [mpm_prefork:notice] [pid 9] AH00163: Apache/2.4.17
(Unix) configured -- resuming normal operations
[Thu Apr 07 09:48:31.400707 2016] [core:notice] [pid 9] AH00094: Command line: 'httpd
-D FOREGROUND'
```

Starten eines Containers

neofonie*
Data. Technology. Services.

Dockerfile + (~/asp-repos/neofonie/_DockerBasics/neofonie-helloworld) - VIM

Dockerfile + (~/asp-repos/neofonie/_DockerBasics/neofonie-helloworld) - VIM 121x41

```
FROM alpine:latest

MAINTAINER Dennis Winter (dennis.winter@neofonie.de) | Neofonie GmbH

RUN apk update && \
    apk add apache2 bash

EXPOSE 80

RUN mkdir -p /run/apache2

COPY httpd.conf /etc/apache2/httpd.conf
COPY index.html /var/www/localhost/htdocs/

ENTRYPOINT ["httpd", "-D", "FOREGROUND"]
```

Was passiert beim Start eines Containers ?

Container starten mit einem primären Prozess. Dieser wird im *Dockerfile* bei der Erstellung des Images mit den Anweisungen ENTRYPOINT oder CMD definiert.

Auf die beiden Anweisungen gehen wir später im Detail ein.

Starten eines Containers

Container können auch im Hintergrund / “daemon mode” gestartet werden:

```
~$ docker run -p=7891:80 -d neofonie/hello-world
8cadebb85afe9164e67a807f3d09ef1d416b908052b224751a54fb73c544bd4b
~$
```

Docker zeigt lediglich die ID des gestarteten Containers.

Übersicht über Container

neofonie*
Data. Technology. Services.

Wie können wir schauen, ob unser dämonisierter Container noch läuft?

```
~$ docker ps
CONTAINER ID    IMAGE         COMMAND         CREATED            STATUS
f1228909bfa6    debian:8      "/bin/bash"     6 minutes ago      Up 6 minutes
~$
```

Stoppen eines Container

Container können *graceful* gestoppt oder gekillt werden.

```
~$ docker stop f1228909bfa6
```

Hierdurch senden wir ein TERM Signal an den Container und geben ihm 10 Sekunden (default) Zeit, seinen im ENTRYPOINT laufenden Prozess zu beenden. Anschließend sendet Docker ein KILL -9.

```
~$ docker stop -t 600 f1228909bfa6
```

Wir geben dem Prozess 10 Minuten Zeit sich sauber zu beenden. Danach erfolgt ein Kill -9.

```
~$ docker kill f1228909bfa6
```

Wir können auch direkt ein KILL -9 senden. Damit wird umgehend der Prozess im ENTRYPOINT beendet.

Stoppen eines Containers

Mit docker-exec können Befehle an einen Container geschickt werden. So bspw. das Starten einer Bash:

```
~$ docker exec -it 8cadebb85afe bash
bash-4.3# ps fax
[...]
bash-4.3# kill 1
```

Container Heaven?

Was geschieht mit gestoppten Containern?

```
~$ docker ps -a
CONTAINER ID    IMAGE                  COMMAND                  CREATED          STATUS
8cadebb85afe    neofonie/hello-world   "httpd -D FOREGROUND"    12 minutes ago   Exited (137) 1 seconds ago
6c82121db0fc    debian:8               "/bin/bash"              33 minutes ago   Exited (0) 23 minutes ago
~$
```

Sie sind weiterhin auf dem lokalen System vorhanden, und könnten mit `docker start <Container-id>` wieder gestartet werden.

Container Heaven!

```
~$ docker rm <Container-ID>
```

Der Container muß zuvor gestoppt sein.

neofonie*
Data. Technology. Services.

Images

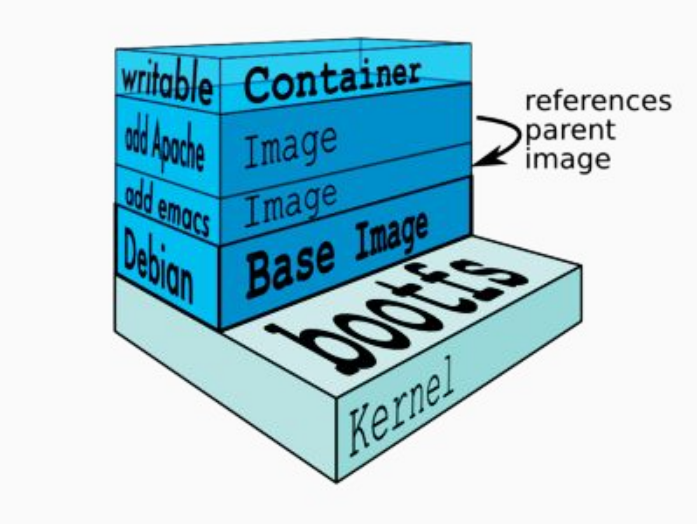
Images

neofonie*
Data. Technology. Services.

In diesem Abschnitt

- betrachten wir den Aufbau von Images
- schauen wir uns die Namensvergabe & Namespaces an
- laden wir Images aus Repositories
- pushen wir Images in Repositories
- verwalten wir das lokale Repo

Was sind Images?



- Eine Sammlung von Dateien & Meta-Daten
- Basiert auf übereinanderliegenden Layers
- In jedem Layer können Dateien ergänzt, geändert und gelöscht werden
- Layer können von mehreren Images verwendet werden

Was sind Images?

Unterschiede zwischen Image und Container sind also:

- ein Image ist ein sortierter Stapel an Layern
- ein Image ist read-only
- `docker run <Image-Name>` startet einen Container von einem bestimmten Image
- der gestartete Container ist eine Instanz eines Images
- von einem Image können beliebig viele Container gestartet werden
- Container laufen mit einem zusätzlichen Layer on-top eines Images, sozusagen als RW-Kopie
- Container starten mit dem Prinzip von *copy-on-write*

Erstellung von Images

neofonie*
Data. Technology. Services.

`docker build`

- baut aus einem Dockerfile ein Image
- sollte immer die bevorzugte Methode sein

`docker commit <container ID>`

- Erstellt eine Kopie eines Containers inklusive aller dort vorgenommenen Änderungen in Form eines Images
- ist schwer reproduzierbar

Wir erklären beides in Kürze.

Erstellung von Images

neofonie*
Data. Technology. Services.

Neue Images leiten in aller Regel von so genannten BASE IMAGEs ab. Diese werden beispielsweise bei der Verwendung von `docker build` im Dockerfile mit der Instruktion `FROM` angegeben.

```
FROM ubuntu:latest
```

oder

```
FROM neofonie/hello-world:latest
```

oder

```
FROM registry.mycompany.com:40000/wild-app:beta
```

Image-Namespaces

Es gibt drei Namespaces

- Basis- oder Root-Namespace
  `ubuntu`
- User und Organisationen
  `neofonie/hello-world`
- Self-Hosted
  `registry.example.com:5000/mein-image`

Root-Namespace

Hier finden sich offizielle Images. Docker Inc. veröffentlicht sie, sie sind aber größtenteils von Dritten gepflegt.

Dazu gehören:

- Mini-Images wie busybox oder alpine (für bspw. Micro-Services)
- Distro-Images wie Debian (zur einfachen Dockerisierung mit allen Möglichkeiten der Distros)
- fertige Komponenten & Dienste wie redis, postgresql etc.

User-Namespace

Der User Namespace enthält die Images für Docker Hub-Users und -Organisationen.

Beispielsweise `neofonie/hello-world`

`neofonie` ist die Organisation, der unsere User-Accounts dort zugewiesen sind

`hello-world` ist der Image Name.

Self-Hosted-Namespace

Dieser Namespace enthält die Images in einer eigenen Registry. Teil der Namen ist dann immer die Adresse und ggf. der Port des Registry Servers.

Beispielsweise `registry.neofonie.de:40000/hello-world`

neofonie*
Data. Technology. Services.

Image Management

Images können auf drei Arten heruntergeladen werden:

- explizit mit `docker pull`
- implizit mit `docker run`, wenn das Image lokal nicht gefunden wird
- implizit mit `docker build`, wenn das Image als BASE IMAGE mit `FROM` im Dockerfile angegeben wurde

Image Management

Images können auf zwei Arten gespeichert werden:

- in einer Registry
- auf dem lokalen System

Mit dem Docker Client können Images von einer Registry sowohl heruntergeladen (`docker pull <Image-Name>`) als auch dorthin übertragen werden (`docker push <Image-Name>`).

Image Management

Ein lokal erstelltes Image können wir mit einer Bezeichnung *taggen* und in eine Registry *pushen*:

```
~$ docker build -f ./Pfad/zum/Dockerfile -t
repository.meine-firma.de:4000/meine-app:latest ./Pfad/zum/Build-Context
~$ docker push repository.meine-firma.de:4000/meine-app:latest
```

Image Management

```
$ docker images
REPOSITORY          TAG         IMAGE ID          CREATED           SIZE
fedora              latest      ddd5c9c1d0f2      3 days ago        204.7 MB
centos              latest      d0e7f81ca65c      3 days ago        196.6 MB
ubuntu              latest      07c86167cdc4      4 days ago        188 MB
redis               latest      4f5f397d4b7c      5 days ago        177.6 MB
postgres            latest      afe2b5e1859b      5 days ago        264.5 MB
alpine              latest      70c557e50ed6      5 days ago        4.798 MB
debian              latest      f50f9524513f      6 days ago        125.1 MB
busybox             latest      3240943c9ea3      2 weeks ago       1.114 MB
```

Image Management

Images können natürlich auch vom lokalen System gelöscht werden.

```
~$ docker rmi <Image-ID>
```

Images ohne Referenz von einem anderen Image (“dangling images”) können auf diese Art entfernt werden:

```
~$ docker rmi $(docker images -f "dangling=true" -q)
```

Image Management

Docker-Hub kann mit dem Docker-Client durchsucht werden:

```
~$ docker search neofonie
NAME                          DESCRIPTION          STARS     OFFICIAL   AUTOMATED
neofonie/aiko                 Image with aiko.     0                    [OK]
neofonie/demo-server2         Test                 0                    [OK]
neofonie/demo-auto-lb                              0
neofonie/demo-loadbalancer                         0
neofonie/hello-world                               0
neofonie/demo-server                               0
```

Image Builds

neofonie*
Data. Technology. Services.

Build mit einem Commit

Von einem laufenden Container kann ein neues Image erstellt werden, das dessen gesamte Änderungen enthält:

```
~$ docker commit
<Container-ID>
123456789abcdef…
```

Build mit einem Dockerfile

Images können anhand einer Reihe von Anweisungen erstellt werden, die in einem sog. *Dockerfile* zusammengetragen werden:

```
~$ docker build -f Pfad/zum/Dockerfile
Pfad/zum/Build-Context
```

Image Builds

Build mit einem Commit

Von einem laufenden Container kann ein neues Image erstellt werden, das dessen gesamte Änderungen enthält:


```
~$ docker commit
<Container-ID>
123456789abcdef
```

So ist der Bau des Images nur schwer zu reproduzieren!

Build mit einem Dockerfile

Images können anhand einer Reihe von Anweisungen erstellt werden, die in einem sog. *Dockerfile* zusammengetragen werden:

```
~$ docker build -f Pfad/zum/Dockerfile
Pfad/zum/Build-Context
```

neofonie*
Data. Technology. Services.

Dockerfile

Dockerfile

In diesem Abschnitt

- finden wir heraus, was ein Dockerfile ist
- schreiben wir ein erstes Dockerfile
- lernen wir ENTRYPOINT und CMD kennen
- bauen wir ein Image
- optimieren wir ein Dockerfile

Image Builds

```
FROM alpine:latest

MAINTAINER Dennis Winter (dennis.winter@neofonie.de) | Neofonie GmbH

RUN apk update && \
    apk add apache2 bash

EXPOSE 80

RUN mkdir -p /run/apache2

COPY httpd.conf /etc/apache2/httpd.conf
COPY index.html /var/www/localhost/htdocs/

ENTRYPOINT ["httpd", "-D", "FOREGROUND"]
```

Dockerfile in a nutshell

Ein Dockerfile ist ein Build-Rezept.
Es enthält eine Serie von Anweisungen die Docker mitteilen, wie ein Image gebaut werden soll.

Image Builds

```
~$ mkdir figlet1
~$ cd figlet1
figlet1 $ vi Dockerfile
```

Das erste Dockerfile

Schreiben wir unser erstes Dockerfile. Es sollte hierzu zunächst in einem eigenen, leeren Verzeichnis sein.

Image Builds

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y figlet
```

Das erste Dockerfile

Wir schreiben diese Zeilen ins Dockerfile.

FROM definiert das Base Image für unseren Build.

RUN führt Befehle während des Builds aus und muss daher non-interactive sein.

Image Builds

```
figlet1 $ docker build -t figlet1 .
```

Das erste Dockerfile

Auf diese Weise bauen wir das Image.

- `-t` tagged das Image
- `.` definiert den Build-Context

Image Builds

```
figlet1 $ docker build -t figlet1 .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM ubuntu
 ---> c9ea60d0b905
Step 2 : RUN apt-get update
 ---> Running in 175beb205b1e
(... OUTPUT DES RUN-COMMANDS ...)
 ---> 74729b434e64
Removing intermediate container 175beb205b1e
Step 3 : RUN apt-get install figlet
 ---> Running in 7a76fc5d73f1
(... OUTPUT DES RUN-COMMANDS ...)
 ---> 45457b13fb57
Removing intermediate container 7a76fc5d73f1
Successfully built 45457b13fb57
```

Das erste Dockerfile

- Jeder Step stellt ein neues Image / einen Layer da.
- vor jedem Schritt schaut Docker, ob ein Image für diese Build-Sequenz schon existiert
- komplette Rebuilds können mit `--no-cache` und `--force-pull=true` erzwungen werden

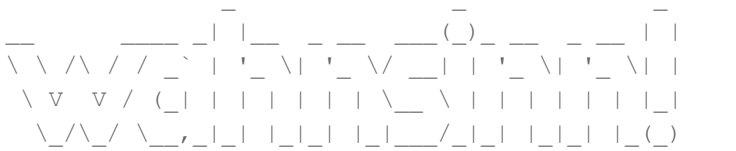
Image Builds

```
~$ docker run figlet1 figlet wahnsinn!
                 _                 _                 _
__      ____ _| |__  _ __  ___(_)_ __  _ __ | |
\ \ /\ / / _` | '_ \| '_ \/ __| | '_ \| '_ \| |
 \ V  V / (_| | | | | | | \__ \ | | | | | | |_|
  \_/\_/ \__,_|_| |_|_| |_|___/_|_| |_|_| |_(_)
```

Erfolg!

ENTRYPOINT & CMD

```
~$ docker run figlet1 figlet wahnsinn!
```



Was läuft da?

- jedes Image hat einen sog. ENTRYPOINT
- CMD funktioniert ähnlich, wird aber primär zur Definition von Default-Values verwendet

ENTRYPOINT & CMD

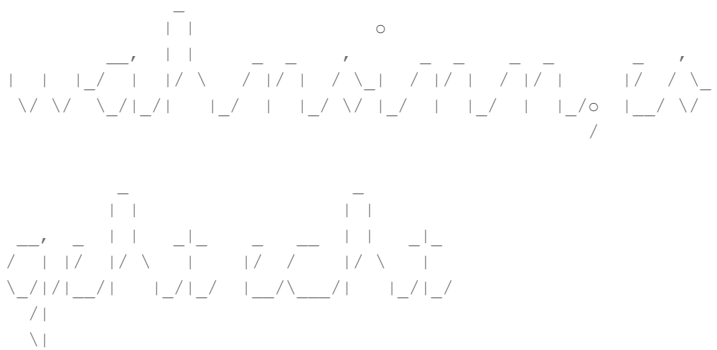
```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y figlet
ENTRYPOINT [“figlet”, “-f”, “script”]
```

```
figlet1 $ docker build -t figlet1 .
```

- ENTRYPOINT and CMD werden im JSON-Format angegeben, andernfalls wird dem String `sh -c` vorangestellt

ENTRYPOINT & CMD

```
~$ docker run figlet1 wahnsinn, es
geht echt
```



Was läuft da?

Hier startet der Container nun mit dem primären Prozess:

```
figlet -f script wahnsinn, es geht
echt
```

Das docker run Kommando interpretiert alles nach dem Image-Namen als Argumente für den ENTRYPOINT. Etwa wie

```
~$ docker run figlet1 $@
```

Dockerfile Referenz

`FROM ubuntu:14.04`	Das Base Image, auf das der Build aufbaut. Auf https://hub.docker.com finden sich die Images, auf GitHub häufig die zugehörigen Projekte.
`MAINTAINER Firma X`	Informationen zum Ersteller. Optional.
`RUN` *`Befehl`*	Führt den anschließenden Befehl im jeweils aktuellen Container des Builds aus. Achtung: hierdurch werden weder Prozess-Zustände gespeichert, noch können auf diese Weise daemons gestartet werden.
`EXPOSE 8080`	Definiert, welche Ports Container dieses Images publishen werden. Per default sind diese Ports nicht extern erreichbar.
`ADD assets/my.conf /etc/my.conf` `COPY assets/mybin /usr/local/bin/`	Instruktionen, um Files aus dem Build-Context in das Image zu kopieren. ADD ist etwas mächtiger, und kann auch mit URLs und Archiven umgehen.

Dockerfile Referenz

`VOLUME /pfad/im/Image`	Erstellt einen Mount Point im definierten Pfad. VOLUMES werden bei docker commit nicht gesichert, und können unterhalb von Containern geteilt werden, sogar VOLUMES gestoppter Container.
`WORKDIR /opt/webapp`	Setzt das Arbeitsverzeichnis für die folgenden Instruktionen.
`ENV WEBAPP_PORT 8080`	Setzt in Containern des zu bauenden Images eine Environment-Variable.
`USER nginx`	Setzt den Usernamen (oder UID), der beim Start des Containers verwendet werden soll.

Dockerfile Effizienz

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y figlet
ADD etc/app.config /opt/webapp/
ADD application/huge.zip
/opt/webapp/
ENTRYPOINT [“figlet”, “-f”,
“script”]
```

```
FROM ubuntu
RUN apt-get update && \
    apt-get install -y figlet && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* /tmp/*
/var/tmp/*

ADD application/huge.zip /opt/webapp/
ENTRYPOINT [“figlet”, “-f”, “script”]
ADD etc/app.config /opt/webapp/
```

Mittagspause!

```
~$ docker stop docker-workshop
```

neofonie*
Data. Technology. Services.

Links & Clusters

Links & Clusters

neofonie*
Data. Technology. Services.

In diesem Abschnitt

- verbinden wir Container auf verschiedene Weisen
- skalieren wir einen einfachen Webservice

Container linking

neofonie*
Data. Technology. Services.

Container sind isoliert voneinander und bekommen ihre Resourcen (IP Adresse) dynamisch zugewiesen. Möchte man innerhalb eines Containers den Webserver (TCP-Port) eines anderen Containers ansprechen muss das explizit konfiguriert werden.

Im Docker Universum geschieht das im einfachsten Fall mit docker-compose. Eine Verbindung zwischen zwei Containern kann aber auch beim Start hergestellt werden.

```
docker run -d --name=loadbalancer --link web01:backend01 ...
```

Container linking

neofonie*
Data. Technology. Services.

Wir starten zwei Webserver

```
docker run -d --name=web01 neofonie/demo-server
docker run -d --name=web02 neofonie/demo-server
```

Und nun einen Loadbalancer

```
docker run -d --name=loadbalancer --link web01:backend01 --link web02:backend02     -p
42080:8080 neofonie/demo-loadbalance
```

Das Ergebnis:

```
$ wget -q -O - "http://127.0.0.1:42080/"
Hello World from 8337b3a48d00
```

Container linking

neofonie*
Data. Technology. Services.

Was passiert dabei nun genau ?

Container linking

Unser Webserver lauscht auf dem TCP-Port 1337. Das ist beim Build des Image bereits in der Konfigurationsdatei festgelegt worden.

Die IP-Adresse wird dynamisch zugewiesen.

```
$ docker inspect web01 | grep -i ipa
          "IPAddress": "172.17.0.3",

$ docker inspect web02 | grep -i ipa
          "IPAddress": "172.17.0.2",
```

Container linking

Da wir für unsere Container kein Netzwerk definiert haben, wird das “default” Netzwerk verwendet:

```
$ docker inspect web01 | grep -i net
          "NetworkMode": "default",
                    "NetworkID": "39074b0c36fa0f74de4f1f33166419dee5a7656316121c1e58
...

$ docker inspect web02 | grep -i net
          "NetworkMode": "default",
                    "NetworkID": "39074b0c36fa0f74de4f1f33166419dee5a7656316121c1e58
..
```

Container linking

Innerhalb einer Webserver Instanz sind andere Container zunächst unbekannt. Befinden sich zwei Container im gleichen Docker Netzwerk können sie sich aber gegenseitig “sehen”.

```
$ docker exec -ti web01 bash -c "ping -c 1 web02"
     ping: unknown host web02

$ docker exec -ti web01 bash -c "ping -c 1 172.17.0.2"
     PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
     64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.072 ms
```

Container linking

Unser Loadbalancer (haproxy) ist ebenfalls bereits beim Erstellen des zugrundeliegenden Images vorkonfiguriert worden. Er möchte die beiden Hostnamen backend01 und backend02 auf dem TCP-Port 1337 ansprechen und ist selbst auf dem TCP-Port 8080 erreichbar.

Diese Konfiguration ist nur innerhalb des Containers nützlich. Außerhalb des Containers muss der Loadbalancer über einen “exposed port” zugänglich gemacht werden.

```
$ docker run … -p 42080:8080 neofonie/demo-loadbalancer
```

Container linking

Mit dem iptables Befehl kann man sich das genauer anschauen:

```
root@kirchhain:~# iptables -t nat -nvL
0 0 DNAT tcp -- !docker0 * 0.0.0.0/0 0.0.0.0/0 tcp dpt:42080 to:172.17.0.4:8080
```

172.17.0.4 ist die dynamisch zugewiesene IP Adresse des Loadbalancers:

```
$ docker inspect loadbalancer | grep IPA
          "IPAddress": "172.17.0.4",
```

Container linking

Der Loadbalancer selbst muss nun noch seine beiden Backend-Server backend01 und backend02 erreichen können. Dafür wird die docker Option “link” verwendet:

```
$ docker run …  --link web01:backend01 … neofonie/demo-loadbalancer
```

Was dabei passieret sieht man im Container. Der Alias backend01 kann jetzt in der Konfigurationsdatei des haproxy verwendet werden

```
$ docker exec loadbalancer bash -c "cat /etc/hosts"
172.17.0.3    backend01 8337b3a48d00 web01
172.17.0.4    1f5548ef510f
```

Container linking

neofonie*
Data. Technology. Services.

Es funktioniert:

```
$ wget -q -O - "http://127.0.0.1:42080/"
     Hello World from 3adb0c334f0f
$ wget -q -O - "http://127.0.0.1:42080/"
     Hello World from 8337b3a48d00
$ wget -q -O - "http://127.0.0.1:42080/"
     Hello World from 3adb0c334f0f
     ...
```

Docker-Compose

Das Loadbalancer Beispiel kann auch über eine Konfigurationsdatei und dem Tool docker-compose gestartet werden:

Installation des docker-compose Tools:

```
$ curl -L
https://github.com/docker/compose/releases/download/1.6.2/docker-compose-`uname
-s`-`uname -m` > /usr/local/bin/docker-compose
$ chmod +x /usr/local/bin/docker-compose
```

Docker-Compose

```
$ cd demo-link-containers
$ cat docker-compose.yml
version: '2'
services:
     web01:
     container_name: web01
     image: demo-server
     networks:
          - default
```

docker-compose verwendet die Konfigurationsdatei **docker-compose.yml**

Das Format der Datei unterscheidet sich ein wenig zwischen version 1 und **version 2**.

Unser Webserver ist ein **Service** mit dem Namen **web01**. Dafür wird eine Instanz des **Image demo-server** gestartet. Der Container bekommt den selben Namen und befindet sich im **Netzwerk “default”.**

Docker-Compose

```
loadbalancer:
     container_name: loadbalancer
     image: demo-loadbalancer
     links:
     - web01:backend01
     - web02:backend02
     ports:
     - 42080:8080
     networks:
          - default
```

Für unseren Loadbalancer wird zusätzlich jeweils ein **Link** auf die beiden Webserver erzeugt.

Der TCP-Port **42080** soll auf dem Host (Wirt) verwendet werden um den Container anzusprechen.

Im Container selbst lauscht der Loadbalancer auf Port **8080.**

Docker-Compose

neofonie*
Data. Technology. Services.

```
$ cd demo-link-containers

$ docker-compose up -d
Creating web02
Creating web01
Creating loadbalancer

$ wget -q -O - "http://127.0.0.1:42080/"
Hello World from cbb95b31d25e
```

Docker-Compose

Mit docker-compose kann man die Anzahl der gestarteten Instanzen dynamisch anpassen. Wir erweitern dafür unser Beispiel um einen “automatisierten” Loadbalancer.

Docker-Compose

```
$ cd demo-link-containers/demo-auto-lb
$ cat docker-compose.yml
version: '2'
services:
     web:
     image: neofonie/demo-server

     loadbalancer:
     image: neofonie/demo-auto-lb
     links:
          - web
     ports:
          - 42080:8080
```

Unser Service **web** bekommt nun keinen Container-Namen. Diese werden beim Aufruf von docker-compose scale einfach numerisch hochgezählt.

Der Loadbalancer wird mit allen Instanzen des Service “verlinkt”

Docker-Compose

Jetzt starten wir einige Webserver und einen Loadbalancer:

```
$ docker-compose up -d
Creating and starting demoautolb_web_1 ... done
Creating and starting demoautolb_loadbalancer_1 ... done

$ docker-compose scale web=3 loadbalancer=1
Creating and starting demoautolb_web_2 ... done
Creating and starting demoautolb_web_3 ... done
```

Docker-Compose

Mit docker-compose kann man sich nun auch die Logfiles der Webserver analog zum Befehl docker logs anschauen:

```
$ docker-compose logs -f web
Attaching to demoautolb_web_2, demoautolb_web_3, demoautolb_web_1
web_2          | Server running at http://0.0.0.0:1337/
web_1          | Server running at http://0.0.0.0:1337/
web_3          | Server running at http://0.0.0.0:1337/
```

Docker-Compose

Die generierten Container-Namen sind im Loadbalancer Container auflösbar.

```
$ docker exec -ti demoautolb_loadbalancer_1 bash
root@e42707366234:/# ping demoautolb_web_2
PING demoautolb_web_2 (172.18.0.4) 56(84) bytes of data.
64 bytes from demoautolb_web_2.demoautolb_default (172.18.0.4): icmp_seq=1 ttl=64
time=0.057 ms
```

Die “link” Option würde man daher eigentlich nicht benötigen. Die Verwendung von Hostnamen in der Konfiguration eines Image wird damit jedoch vereinfacht.

```
root@e42707366234:/# ping web_2
PING web_2 (172.18.0.4) 56(84) bytes of data.
64 bytes from demoautolb_web_2.demoautolb_default (172.18.0.4): icmp_seq=1 ttl=64
time=0.048 ms
```

Docker-Compose

Die DNS Namen der Container sind nicht in der lokalen /etc/hosts Datei der Container zu finden.

```
$ docker exec -ti demoautolb_loadbalancer_1 bash
root@e42707366234:/# cat /etc/hosts
127.0.0.1    localhost
172.18.0.5    e42707366234
```

In der resolv.conf ist nun folgender Eintrag zu finden. Es handelt sich dabei um einen internen Docker Nameserver, der seit Version 1.10 zur Verfügung steht.

```
root@e42707366234:/# cat /etc/resolv.conf
search neofonie.de
nameserver 127.0.0.11
options ndots:0
```

Docker-Machine

Der Docker Daemon läßt sich auch über das Netzwerk ansprechen. Damit kann man seine Container über mehrere Hosts hinweg verwalten.

Der Docker Client bzw. docker-compose verwendet dafür die Umgebungsvariable DOCKER_HOST.

Für den Netzwerkzugriff werden TLS Zertifikate und eine entsprechende Konfiguration des Docker Daemon benötigt. Das lässt sich mit docker-machine leicht konfigurieren.

Docker-Machine

```
$ docker-machine create --driver generic --generic-ip-address=172.42.0.1
--generic-ssh-user=root wirt01
```

Running pre-create checks...
Creating machine...
...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run: docker-machine env wirt01

Docker-Machine

```
$ docker-machine env wirt01
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://172.42.0.1:2376"
export DOCKER_CERT_PATH="/home/trainer/.docker/machine/machines/wirt01"
export DOCKER_MACHINE_NAME="wirt01"
# Run this command to configure your shell:
# eval $(docker-machine env wirt01)
```

Docker-Machine

docker-machine unterstützt neben dem “generic” driver, welcher einen SSH Zugriff auf den Daemon Host benötigt, auch spezielle Umgebungen bekannter Hosting-Anbieter:

- Amazon Web Services
- Microsoft Azure
- Digital Ocean