



# docker **Workshop**

Von der Installation bis zur Automatisierung

# Agenda

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	- <i>Mittagspause</i> -
13:30	Loadbalancing, DNS, Network
14:30	Volumes und Logging
15:30	docker-machine und Docker Ökosystem
16:00 - 18:00	FAQ und praxisnahe Aufgabenstellungen

## Was ist Docker?

- 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

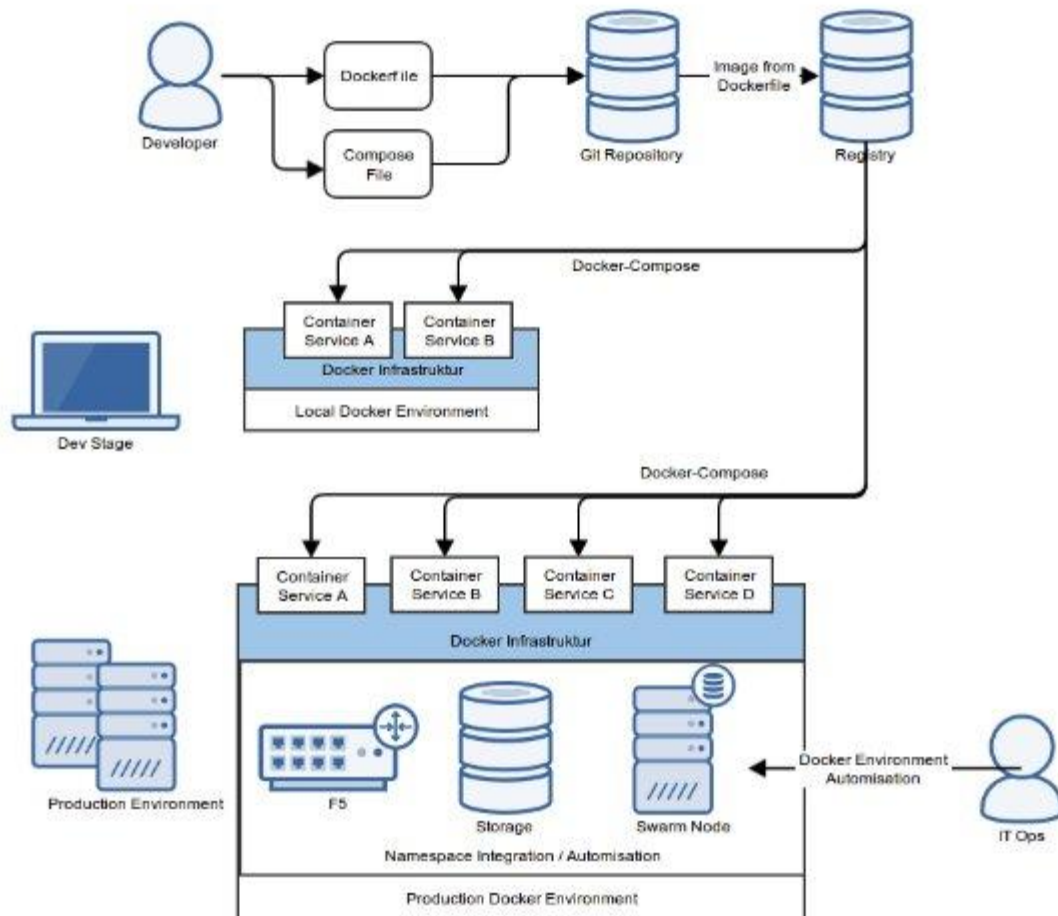
## Was ist Docker ?



Deployment Schnittstelle

## Anwendungsgebiete

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



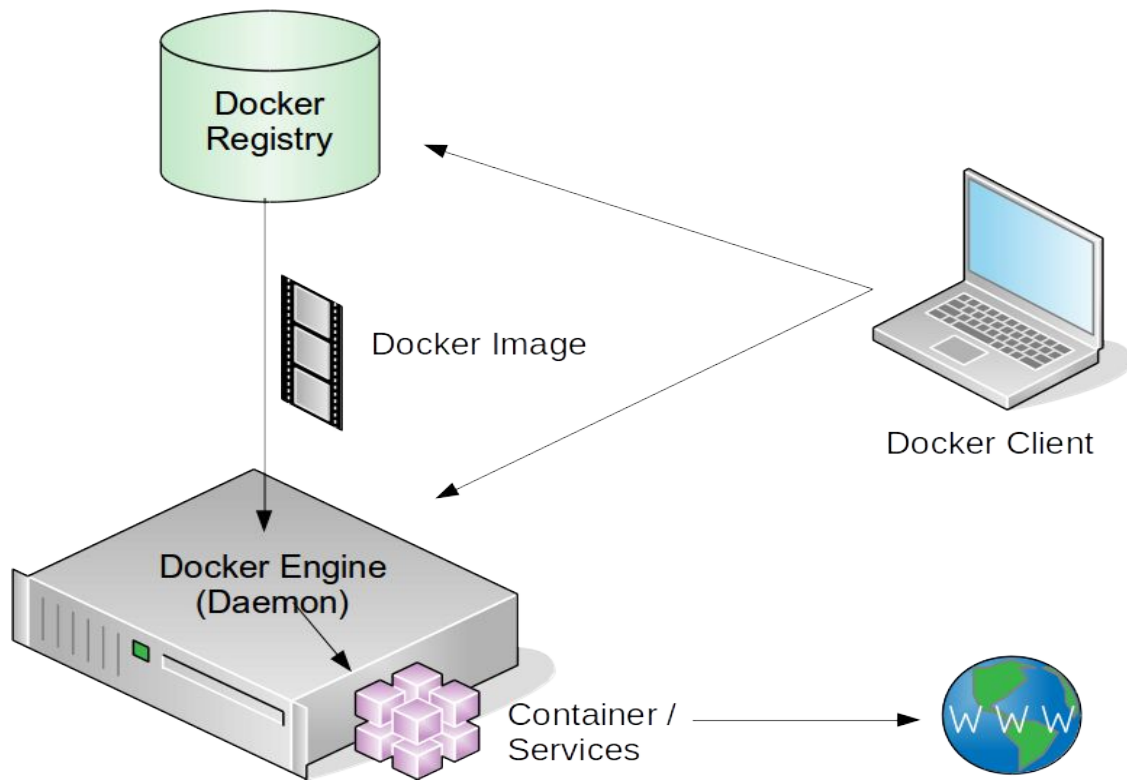
# Basis-Komponenten

Docker besteht aus mehreren Komponenten, die wichtigsten Begriffe sind:

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



## Basis-Komponenten



## Docker Daemon

- 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
- dockerd (core-engine) und docker-containerd (container runtime)

## Docker Client

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

# Docker Container

- ausführbare Instanz eines Image -> ein Prozess
- ~~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

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

## Docker Registry

- sehr einfache Image Datenbank
- als Docker Image verfügbar
- öffentliche Registry als Service auf [docker.io](https://docker.io)

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

- Portus
- [docker-registry-ui](#) (GitHub)
- [konradkleine/docker-registry-frontend:v2](#)

# 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
```

Erste Container



# Erste Container

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

```
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

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 getötet 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.

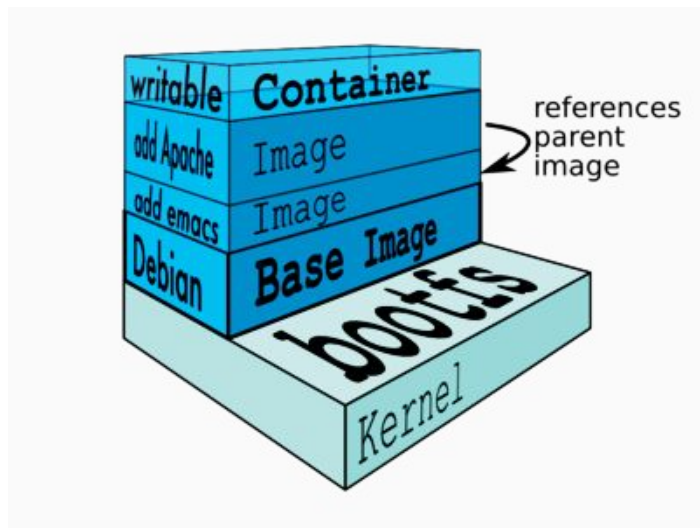
Images

# Images

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

```
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

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`

# 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

## 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
```

# 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!
```

[illegible]

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

<code>FROM ubuntu:14.04</code>	Das Base Image, auf das der Build aufbaut. Auf <a href="https://hub.docker.com">https://hub.docker.com</a> finden sich die Images, auf GitHub häufig die zugehörigen Projekte.
<code>MAINTAINER Firma X</code>	Informationen zum Ersteller. Optional.
<code>RUN Befehl</code>	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.
<code>EXPOSE 8080</code>	Definiert, welche Ports Container dieses Images publishen werden. Per default sind diese Ports nicht extern erreichbar.
<code>ADD assets/my.conf /etc/my.conf COPY assets/mybin /usr/local/bin/</code>	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

<code>VOLUME /pfad/im/Image</code>	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.
<code>WORKDIR /opt/webapp</code>	Setzt das Arbeitsverzeichnis für die folgenden Instruktionen.
<code>ENV WEBAPP_PORT 8080</code>	Setzt in Containern des zu bauenden Images eine Environment-Variable.
<code>USER nginx</code>	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
```

Kommunikation zwischen Containern  
-  
Links und Docker Network

## Links & Docker Network

In diesem Abschnitt

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

- Container bekommen ihre IP Adresse(n) dynamisch zugewiesen
- Netzwerkverbindungen zwischen Containern müssen explizit konfiguriert werden
  - Verbindung zwischen Containern, Option "--link"
  - Erstellen eines Docker Network, Option "--network"

## Container linking

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-loadbalancer
```

Das Ergebnis:

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



## Container linking

Was passiert dabei nun genau ?

## Container linking

Die IP-Adresse wird dynamisch zugewiesen.

Sie läßt sich mit docker inspect ermitteln.

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

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

## Container linking

Unsere beiden Demo-Webserver sind ohne weitere Netzwerk-Optionen gestartet worden. Andere Container sind für sie zunächst unbekannt.

Befinden sich zwei Container im gleichen Docker Netzwerk können sie sich aber gegenseitig über die IP Adresse erreichen.

```
$ 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

Da wir für unsere Container kein Netzwerk definiert haben, wird das “default” Netzwerk verwendet. In unserem Beispiel mit der Option “docker --link” spielt das Docker Netzwerk erst einmal keine weitere Rolle.

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

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

## Container linking

Unser Loadbalancer (haproxy) ist 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.

Außerhalb des Containers muss der Loadbalancer über einen “exposed port” zugänglich gemacht werden.

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

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

Was dabei passiert sieht man im Container. Der Alias backend01 kann jetzt in der Konfigurationsdatei des haproxy verwendet werden. Er wird über die Datei /etc/hosts im Container aufgelöst.

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

## Container linking

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/1.9.0/docker-compose-`uname  
-s`-`uname -m` > /usr/local/bin/docker-compose  
$ chmod +x /usr/local/bin/docker-compose
```



# Docker-Compose

Was ist docker-compose:

- ein Orchestration Tool für Docker
- basiert auf einer zentralen Definitionsdatei (docker-compose.yml)
- verwaltet ein Bundle aus mehreren Services
- steuert den Docker Daemon
  - startet Container
  - legt Docker Netzwerke an
  - Image push, pull und build
- kennt auch Docker CLI Befehle wie z.B. docker logs -> docker-compose logs

# Docker-Compose

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

**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 "web01".

## Docker-Compose

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

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

```
$ 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

Der Aufruf von docker-compose erzeugt nun zusätzlich ein lokales Netzwerk. Alle Container in diesem Netzwerk können sich per Container-Namen erreichen.

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
3e45c9eeb0d0	bridge	bridge	local
189691eb5ab6	demolinkcontainers_default	bridge	local
db44e3bbe4ab	host	host	local
c1049589097a	none	null	local

## Docker-Compose

**Docker build-in DNS:** Seit der Docker Version 1.10 findet man den Eintrag des internen DNS Servers in der `/etc/resolv.conf`.

```
$ docker exec -ti web01 bash -c "nslookup loadbalancer"
```

```
Server:          127.0.0.11
```

```
Address:         127.0.0.11#53
```

```
Non-authoritative answer:
```

```
Name:           loadbalancer
```

```
Address: 172.18.0.4
```

```
$ docker exec -ti web01 bash -c "cat /etc/resolv.conf"
```

```
search neofonie.de
```

```
nameserver 127.0.0.11
```

```
options ndots:0
```

## 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
    ports:
      - 42080:8080
```

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

Die docker-compose default Einstellungen sorgen bereits dafür, daß alle Container in einem Projekt per DNS "sichtbar" sind, da sie sich im selben "default" Netzwerk befinden.



## 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/
```

Wo werden im Container Daten geschrieben ?

- Im Layer-Filesystem
  - schreibbarer Layer zusätzlich zu den Image-Layern des Containers
  - beim Erstellen von Docker Images
  - für Konfigurationen, die einen Neustart “überleben” sollen
- Im Docker Volume
  - Dockerfile: VOLUME [“/var/lib/mysql”]
  - Mountpoint auf dem Host: `docker run -v /mnt/nfs_volume01:/data ...`
  - Docker Volume Treiber/Plugin: z.B. REX-Ray
  - Für Daten, die unabhängig von Containern persistent sein müssen (DB usw.)

Was ist ein Docker Volume ?

- dynamisch bereitgestelltes Docker Volume, Dockerfile -> VOLUME ["/var/lib/mysql"]
  - ist Standard für Images wie mysql oder wordpress
  - wird vom docker daemon unter /var/lib/docker/volumes angelegt
  - ist ein reguläres Verzeichnis im Filesystem des Node
  - bereits im Image definiert, VOLUME ["/var/lib/mysql"]
- Bind-Mount auf dem Node
  - Mapping zwischen Node und Container Filesystem
  - ergibt eine zusätzliche Abhängigkeit, Verzeichnis muß auf dem Node existieren
  - Verzeichnisname ist statisch vorbestimmt

- Docker Volume Plugins
  - direkte Anbindung einer Storage Lösung
  - viele verschiedene Treiber vorhanden
  - EMC REX-Ray
  - [https://docs.docker.com/engine/extend/plugins\\_volume](https://docs.docker.com/engine/extend/plugins_volume)

Logging Informationen aus einem Container:

- sollten über stdout geschrieben werden.
- nicht im Layer-Filesystem des Containers geschrieben werden
- Fehlermeldungen (stderr) sollten auf stdout umgeleitet werden.

## Container Logging:

- zusätzlich kann ein Logging Volume verwendet werden.
  - Webserver Access-Logs
- für die Weiterverarbeitung gibt es Docker Logging Treiber
  - fluentd
  - syslog
  - splunk

# Docker-Machine

Was ist docker-machine ?

- ein Provisioning Tool
- installiert eine Docker Umgebung zum Betrieb von Containern
- verwaltet mehrere Hosts
- generiert TLS Zertifikate



## 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](#)

## Docker Swarm Mode

Mit Docker Swarm kann man seine Container in einem Cluster aus mehreren Hosts betreiben. Es gibt eine "ältere" Variante der Docker Swarm Implementierung, die im Gegensatz zum Docker Swarm Mode, Docker Swarm Classic genannt wird.

## Docker Swarm Mode

Docker Swarm CLI Komponenten:

- docker swarm
- docker service
- docker node
- docker stack/deploy <- experimentell bzw. in der Entwicklung

## Docker Swarm Mode

Docker Swarm einrichten: Wir starten einen Swarm Manager Node.

```
$ docker swarm init
```

```
Swarm initialized: current node (01r60w2ixzwrgr2fce7gwi6qp) is now a manager.
```

To add a worker to this swarm, run the following command:

```
docker swarm join \
  --token
  SWMTKN-1-3yhmfl1tg4ni9kich8hezo3ebgf0dne2oul5pwvvebo6ihcom2-ccae85nvsq8xn8sxc8ta6fnk3
  \
  1.2.3.4:2377
```

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

## Docker Swarm Mode

Docker Swarm Worker hinzufügen::

```
$ docker swarm join --token \  
SWMTKN-1-3yhmfl1tg4ni9kich8hezo3ebgf0dne2oul5pwvvebo6ihcom2-ccae85nvsq8xn8sxc8ta6fnk3  
<Manager IP Adresse hier eintragen>:2377
```

## Docker Swarm Mode

Swarm Token ausgeben lassen:

```
workshop@node01:~$ docker swarm join-token manager
```

To add a manager to this swarm, run the following command:

```
docker swarm join \
  --token
  SWMTKN-1-3nfxn8s3tbfyp3u35y26odx8sactj9u0w6636mvswtutykpwwu-ddumcn5dx1sc5jooqlu6f4m83
  \
  138.68.83.108:2377
```



## Docker Swarm Mode

Liste aller Swarm Nodes ausgeben:

```
workshop@node01:~$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
4c35vpyt3icxjfwyzkk578ij *	node01	Ready	Active	Reachable
84dpg6b1i14crty343jxvhdt	node03	Ready	Active	Reachable
9jrbvyv33uv896b8bs745qtcs	node02	Ready	Active	Leader

## Docker Swarm Mode

### Einen Service starten

```
workshop@node01:~$ docker service create --name ping debian bash -c "ping  
www.neofonie.de"
```

### Auf welchem Swarm Node wurde der Service gestartet ?

```
workshop@node01:~$ docker service ps ping
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE
ERROR					
65cicbdpcl4ap1	ping.1	debian	node01	Running	Preparing 1 seconds ago

## Docker Swarm Mode

Die Logausgabe des Containers anschauen. Im Swarm Mode müssen wir dafür auf dem selben Node eingeloggt sein auf dem auch der Container läuft.

```
ssh node01
```

```
workshop@node01:~$ docker logs ping.1.65cicbdpc14ap1
```

```
64 bytes from 81.17.211.34: icmp_seq=50 ttl=57 time=15.436 ms
```

```
64 bytes from 81.17.211.34: icmp_seq=51 ttl=57 time=15.415 ms
```

```
64 bytes from 81.17.211.34: icmp_seq=52 ttl=57 time=15.669 ms
```

# Monitoring

- Service Monitoring statt Host Monitoring
  - nicht CPU, RAM, SWAP
  - sondern Antwortzeiten, Anwendungs URLs
- Jeder Service braucht eine Healthcheck URL
- Healthcheck URL von außen zugänglich machen

# Monitoring

## Tools:

- Icinga
  - dynamische Konfiguration
- Prometheus.io
  - stellt Metriken dar und erzeugt Alerts
  - Soundcloud Projekt
- Rancher
  - Cloud Management
  - Ressourcenverbrauch der Container

## eigene Registry betreiben

- docker registry bietet kaum Features
  - image push
  - image pull
- zusätzlich erforderlich
  - GUI und Suchfunktion
  - Cleanup Job
  - LDAP Anbindung / User Authentifizierung

## eigene Registry betreiben

- GUI und Suchfunktion
  - Docker Datacenter Tool
  - docker-registry-ui
  - Portus
- LDAP Anbindung / User Authentifizierung
  - Docker Datacenter Tool
  - Proxy davorschalten (haproxy, apache)
  - Portus