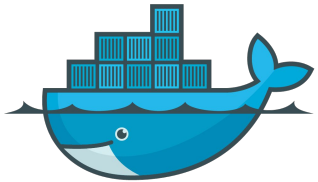


docker Workshop

From installation to automation

neofonie*
Data. Technology. Services.

Workshop Schedule

10:00	Welcome
10:20	Introduction to Docker
11:00	Installation
11:30	Images and containers
12:45	Lunch break
13:30	Linking and clustering
14:30	Orchestration with docker-compose
15:30	docker-machine and the Docker ecosystem
16:00	FAQ and real world tasks (until 17:00)

What is Docker?

- The core product of Docker Inc., also called Docker Engine
- Started by dotCloud, first GitHub commit in January 2013
- Supported by a couple of cloud providers
  (Google, Amazon, DigitalOcean, etc.)
- DevOps tool
- Puts applications and there dependencies in so called images
- Instances of images are called containers
- Open-source, written in Go

Put it into a container.

neofonie*
Data. Technology. Services.

Use the same tool for all deployments

Fields of application

- Development
  - Mangement and distribution of tools (IDE, Bash tool kits, environments)
  - Programming in a production env
- QA / QM
  - (automatic) Tests on a system that is like production
- Deployment & Operation
  - high portability (test, staging, production server, in-house data center, cloud etc.)
  - virtualization without hypervisor (no extra hardware layer)
  - scalability
  - quick testing of new components without native installation

Base components

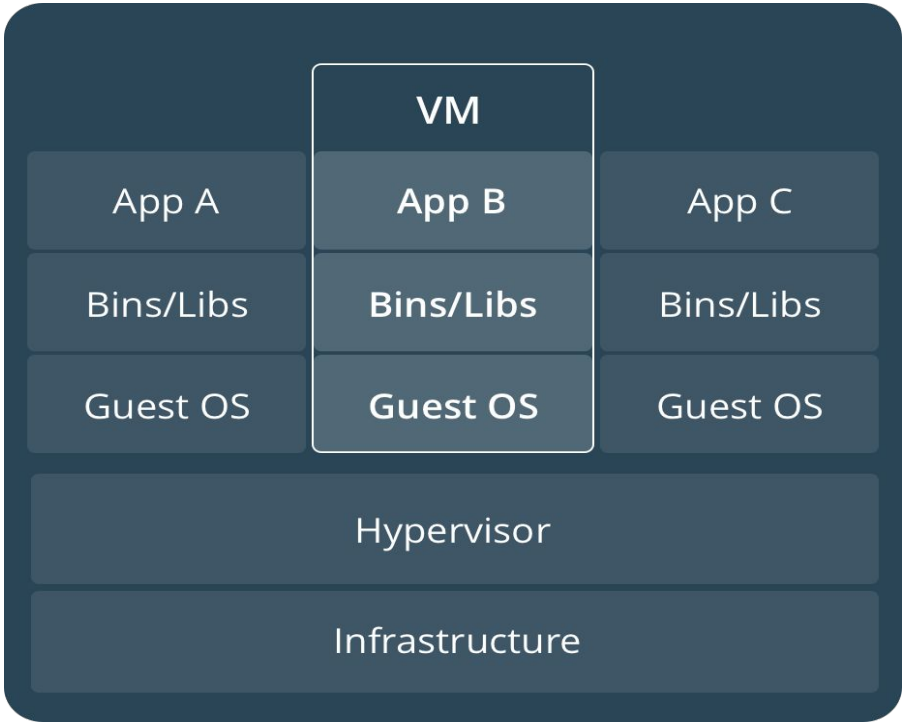
Docker is made from a couple of components that collaborate via an API:

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

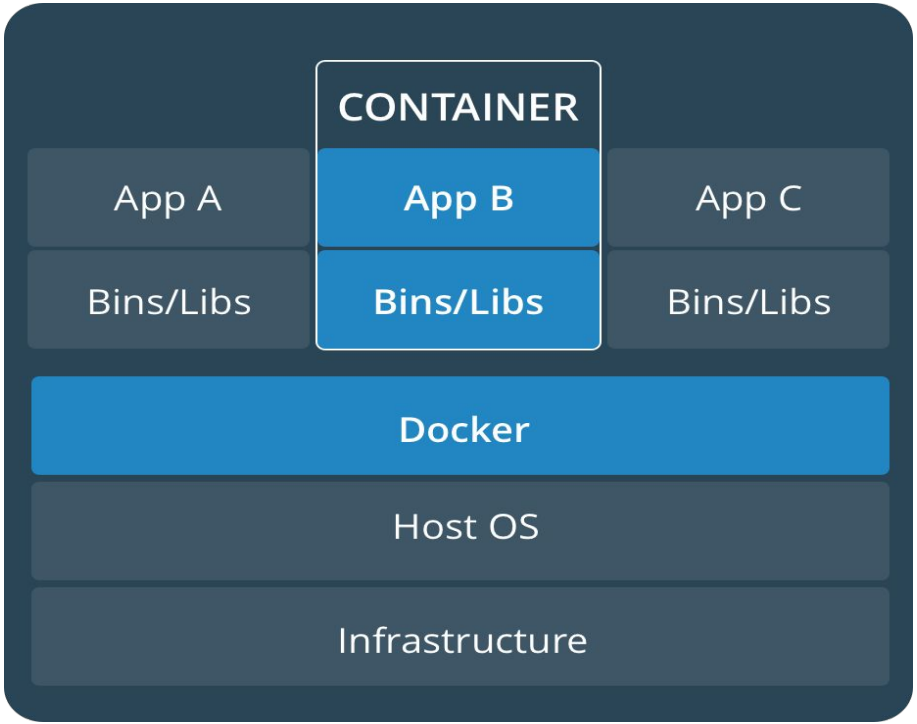
Docker Daemon (+ container daemon)

- Center of the Docker Infrastructure
- Interface to the host OS
- has TCP, Unix Domain Socket and Systemd interface
- uses Linux kernel functions (cgroups)
- it is not a hardware virtualisation

Docker vs. VM



Docker vs. VM



Docker Client

- CLI tool to talk with the Docker daemon
- start, stop, delete containers
- build, save, delete images
- print information about containers and images

Docker Container

- executed instance of an image
- is not persistent
- shouldn't be considered persistent
- to start a container you need at least
  - a Docker client
  - a running Docker daemon
  - an available Docker image

Docker Image

- files in a non-writeable layer filesystem
- is made from a Dockerfile
- contains all dependencies of an app (libs, binaries, etc.)
- base for the start of a container

Docker Registry

- simple image database
- distributed as a Docker image
- Docker Hub registry, public image repository

The registry has no browser interface, but there are projects to fill this gap …

- Portus
- docker-registry-ui (GitHub)

How to install

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

In `~/DockerBasics/workshop.md` you find the URLs of the install manual

If the Docker daemon is not starting, try this script to debug:

```
$ ./check-config.sh
```

Our first container

Our first container

In this section we ...

- create and start our first container
- "log in" to a running container
- get an overview
- terminate and delete a container

Hello World

```
$ docker run busybox echo "hello world!"
hello world!
```

Start a container

We can start Debian in a brand new container:

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

`-it` tells Docker Client to attach us to stdin (`-i`) and gives us a tty (`-t`).

Start a container

Start apache webserver container:

```
$ docker run -p=7890:80 -it httpd
```

```
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'
```

Start a container

```
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"]
```

What happens when a container starts?

Containers start with a primary process (PID 1). It is defined in the *Dockerfile* with the commands ENTRYPOINT or CMD.

We'll see these commands in detail later.

Start a container

You can start containers in the background (daemon mode):

```
$ docker run -p=7891:80 -d httpd
8cadebb85afe9164e67a807f3d09ef1d416b908052b224751a544bd4b
```

Docker is only showing the ID the started container.

Overview of the running container

How can we see if our daemonized container is still running?

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

Stopping a container

Container can be stopped gracefully or be killed.

```
$ docker stop f1228909bfa6
```

With this we are sending a TERM signal to process 1 in the container. The process has 10 seconds to terminate, otherwise it will be killed by sending a KILL signal.

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

We grant the process 10 minutes to terminate.

```
$ docker kill f1228909bfa6
```

Stopping a container

With docker exec you can run commands in the container like a shell:

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

Container Heaven

What happens to a stopped container?

```
$ 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
```

It is this present on the local system and can be restarted with docker start <container id>.

Container Heaven

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

The container must not run in order to do this.

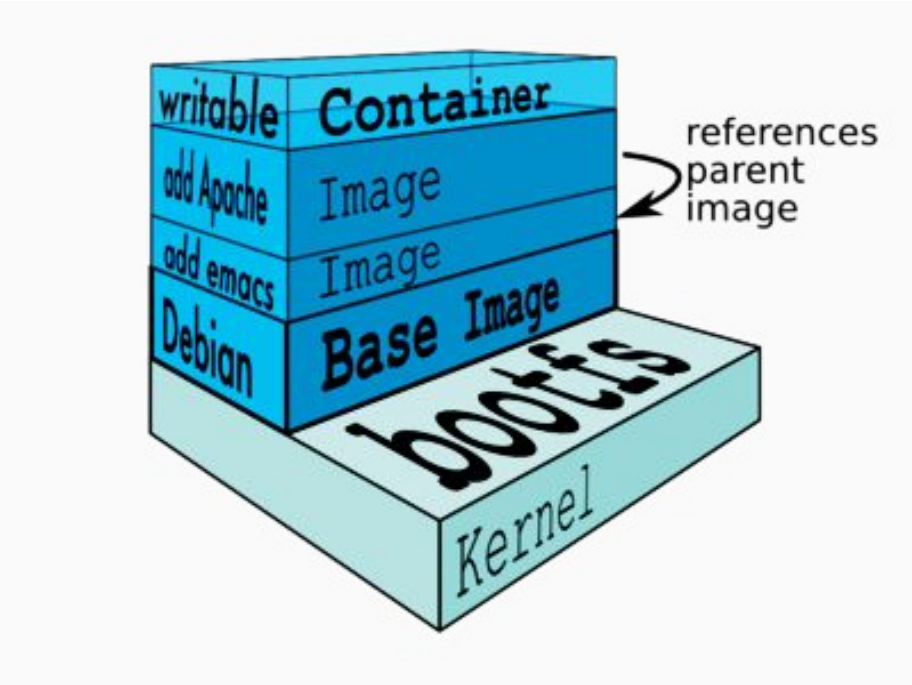
Images

Images

In this section we ...

- look at the structure of images
- look at naming conventions and name spaces
- are loading images from Docker repos
- pushing images to Docker repos
- manage the local Docker repo

What are images?



- a collection of files and meta data
- based on layers
- in each layer you can add, change, delete files
- Layers can be shared between images

What are images?

The diffs between an image and a container are:

- an image is a ordered stack of layers
- an image is read-only
- `docker run <Image-Name>` starts a container of the given image
- the started container is an instance that image
- you can start as many containers as you like from the same image
- Containers run with an additional layer on top of the image, you can call it a RW copy
- Containers use *copy-on-write*

Create images

`docker build`

- builds an image from a Dockerfile
- prefered method to build images

`docker commit <container ID>`

- create an image as a copy of a container
- hard to reproduce (do not use it!)

We'll expain both ways later.

Create images

New images are usually derived from a base image. The base image is defined with a FROM instruction in the Dockerfile.

```
FROM ubuntu:latest
```

or

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

or

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

Image namespaces

There are three namespaces

- Base or root namespace

  `ubuntu`
- User and Org

  `neofonie/hello-world`
- Self-Hosted

  `registry.example.com:5000/mein-image`

Root namespace

In this namespace you find the official images. Docker Inc. publishes them, but they are maintained by 3rd-partys.

These are:

- Light weight distros like busybox or alpine
- Distros like Debian
- ready-to-use components like Redis, Postgres, Mysql, Elasticsearch

User namespace

The user namespace contains images of Docker Hub users.

Like `neofonie/hello-world`

`neofonie` is the org.

`hello-world` is the image name.

Self-hosted namespace

Adresses images in a custom registry. It contains hostname and port.

Like this: `registry.neofonie.de:40000/hello-world`

Image Management

There is three ways to download an image:

- `docker pull`
- implicitly with `docker run`, when there is no image in the local cache
- implicitly with docker build, when the image is mentioned in the FROM statement

Image Management

Images are stored in two locations:

- in a registry
- on the local system

With the Docker Client you can download images from a registry (`docker pull <Image-Name>`) or upload to a registry (`docker push <Image-Name>`).

Image Management

A locally built image can be tagged and pushed to a registry:

```
$ docker build -f ./path/to/Dockerfile \
    -t repository.meine-firma.de:4000/meine-app:latest \
    ./path/to/builddir
$ docker push repository.meine-firma.de:400/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

You can delete images from the local cache:

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

Layers w/o reference (dangling images) can be deleted:

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

Image Management

Docker Hub can be searched with Docker Client:

```
$ 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 with commit

You can create an image by copying a running container:

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

Build with a Dockerfile

Images can be built with a couple of commands written in a Dockerfile:

```
$ docker build -f Dockerfile .
```

Image Builds

Build with commit

You can create an image by copying a running container:

```
$ docker commit <Container-ID>
12
```

Build with a Dockerfile

Images can be built with a couple of commands written in a Dockerfile:

```
$ docker build -f Dockerfile .
```

Dockerfile

Dockerfile

In this section we ...

- learn what a Dockerfile is
- write your first Dockerfile
- learn about ENTRYPOINT and CMD
- build an image
- optimize a Dockerfile

Image Builds

```
FROM alpine:latest

MAINTAINER Dennis Winter (dennis.winter@neo

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"]
```

Docker in a nutshell

A Dockerfile is a build receipe.
It contains a couple of instructions how to build an image.

Image Builds

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

1st Dockerfile

Let's write your first Dockerfile.

Put it in a new empty directory.

Image Builds

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

1st Dockerfile

Copy these lines in the Dockerfile.

FROM defines the base image.

RUN runs commands during the build they must be non-interactive.

Image Builds

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

1st Dockerfile

That's how we build the image.

- `-t` tags the image
- `.` defines the build context

Image Builds

```
$ 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
```

1st Dockerfile

- Every step is a new image / layer
- before each step Docker checks if there is a layer already for the next step.
- complete rebuilds can be forced with `--no-cache` and `--force-pull=true`

Image Builds

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

Success!

ENTRYPOINT & CMD

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

What is running there?

- each image has an ENTRYPOINT
- CMD works similar to ENTRYPOINT but is used to define default values

ENTRYPOINT & CMD

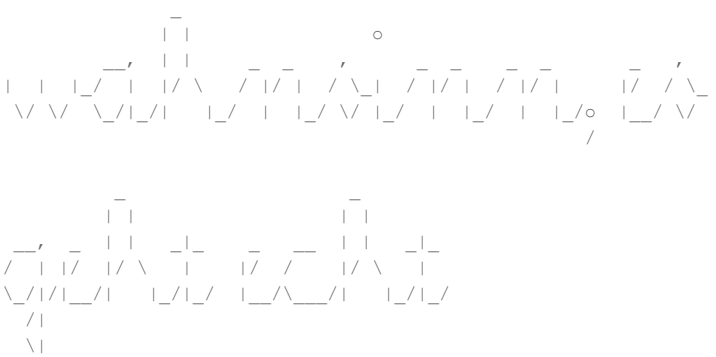
```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y figlet
ENTRYPOINT ["figlet", "-f", "script"]
```

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

ENTRYPOINT and CMD are defined in JSON format here otherwise `sh -c` is used to execute the entrypoint

ENTRYPOINT & CMD

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



What is running?

Here we start a container with this process:

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

docker run interprets everything after the image name as args to entrypoint, like

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

Dockerfile Referenz

`FROM ubuntu:14.04`	The image we are use as a base. On https://hub.docker.com you can find the images. The code is usualy on GitHub.
`MAINTAINER Firma X`	Maintainer. Optional.
`RUN` *`Befehl`*	Runs command in the current container. Keep in mind that process state is not maintained, i.e. you cannot start applications this way.
`EXPOSE 8080`	The ports that are exposed by this container.
`ADD assets/my.conf /etc/my.conf` `COPY assets/mybin /usr/local/bin/`	Instructions to copy files from the build context into the image. ADD is more powerful and can handle URLs and Tar archives.

Dockerfile Referenz

`VOLUME /pfad/im/Image`	Creates a mount point. VOLUMES can be shared between containers even when they are stopped.
`WORKDIR /opt/webapp`	Set the working directory for the following instructions.
`ENV WEBAPP_PORT 8080`	Set an env var.
`USER nginx`	Change the username, uid, gid.

Dockerfile Efficiency

```
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/
```

Lunch break!

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

Links und Clusters

Links & Clusters

In this section we ...

- connect containers using different methods
- are scaling a simple webservice

Container linking

Container are isolated from each other and the resources (like IP addresses) are dynamically assigned. If a container wants to talk to another container it must explicitly configured.

The easiest way to do it is docker-compose but it can be done at start time using nothing but docker:

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

Container linking

We start two webservers

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

And a load balancer

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

result:

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

Container linking

What's happening?

Container linking

Our webserver is listening on TCP port 1337. We defined that in a config file at build time.

The IP address is assigned dynamically.

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

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

Container linking

Since we defined no network, the "default" network is used:

```
$ docker inspect web01 | grep -i net
        "NetworkMode": "default",
                "NetworkID": "39074b0c36fa0f74de4f1f33166419dee1e58...

$ docker inspect web02 | grep -i net
        "NetworkMode": "default",
                "NetworkID": "39074b0c36fa0f74de4f1f331316121c1e58...
```

Container linking

Within a webserver instance other containers are unknown but if they are in the same network they can see each other.

```
$ 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

Our load balancer (haproxy) was also configured at build time. It expects the hostnames backend01 and backend02 to listen on TCP port 1337 and is listening on port 8080 itself.

The config is only useful in a container. Outside of a container it must be made available with an "exposed port":

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

Container linking

Docker is doing some iptables magic:

```
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 is the assigned IP address of the load balancer:

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

Container linking

The load balancer itself needs to talk to the backends. It can be done with "link":

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

backend01 can be used in the config of haproxy now:

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

Container linking

It works:

```
$ 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

The load balancer example can be started with docker-compose:

Installation of the docker-compose tool:

```
$ 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 used a config file called **docker-compose.yml**

The format differs between version 1 und **version 2**.

Our webserver is a **service** with the name **web01**. An instance of **image demo-server** is started for this. The container has the same name and is in **network “default”.**

Docker Compose

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

Haproxy needs a link for each backend.

Port **42080** of the host is used to talk to the container.

In the container the load balancer is listening on 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

With docker-compose you can change the number of running instances all the time. We extend our example with an automatic load balancer.

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
```

Our service **web** is not getting a name. docker-compose scale just increases a number.

Haproxy is linked with all instances.

Docker Compose

We start a couple of services and a load balancer:

```
$ 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

To see the log files of the webserver with docker-compose:

```
$ 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

The generated name are resolvable in the load balancer container:

```
$ 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
```

You don't require "link" but the usage of hostnames in config files makes stuff simpler:

```
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

The hostnames are not in /etc/hosts:

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

There is an internal name server since version 1.10:

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

Docker Machine

You can talk to the Docker daemon over the network. With this feature you can manage your containers across multiple hosts.

Docker Client and docker-compose use the env var DOCKER_HOST to do this.

For remote access you need TLS certs and a properly configured Docker daemon. docker-machine can do the hard work for you.

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 not only supports the “generic” driver, for SSH access to the daemon host, but also an impressive list of cloud hosters:

- Amazon Web Services
- Microsoft Azure
- Digital Ocean