DockerHandsOn-Admincamp2018

September 25, 2018

1 Docker Hands On - Admincamp 2018

Author: Christoph Stoettner
Mail: christoph.stoettner@panagenda.com

Dieses Dokument ist ein Jupyter Notebook mit Bash Kernel! Es kann direkt in Python gestartet und die Kommandos mit Strg+Return ausgeführt werden.

Jupyter Notebook ist eine Open-Source-Webanwendung, mit der Sie Dokumente erstellen und freigeben können, die Live-Code, Gleichungen, Visualisierungen und FlieSStext enthalten.

Zu den Anwendungen gehören:

- Datenreinigung und -transformation
- numerische Simulation
- statistische Modellierung
- Datenvisualisierung
- maschinelles Lernen

und vieles mehr.

1.1 Einleitung

Ich verwende in diesem Dokument meistens die lange Form von Kommandos, da ich finde daSS diese sprechender sind. Man kann in Docker z.B. die Kommandos

```
docker ps
docker container ls
```

synonym verwenden. Da es den Befehl `docker image ls` ebenfalls gibt, fällt aber die Abgrenzung leichter. Ausserdem habe ich versucht soweit wie möglich immer `docker container ...` bzw. `docker image ...` zu verwenden.

Wenn Sie wie hier im Beispiel den Code

```
...
```

sehen, soll damit ein Command Prompt dargestellt werden. Wenn Sie also das Kommando in einem Terminal eingeben, lassen Sie bitte den Pfeil weg!

1.2 Erste Alpine Linux Container

```
In [3]: docker container run hello-world
```

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

1.2.1 Was passiert hier?

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
```

Das Image hello-world ist lokal nicht verfügbar, daher wird es von https://hub.docker.com heruntergeladen. Es wird automatisch der Tag `latest` ergänzt und benutzt. Wenn eine spezielle Version benötigt wird, kann hello-world:1.0 verwendet werden.

Weitere Informationen zum Image, sowie die zugehörigen Dockerfiles können unter https://hub.docker.com/_/hello-world/ heruntergeladen werden.

Wenn die Meldung `Unable to find image 'hello-world:latest' locally` nicht erscheint, ist das Image lokal bereits vorhanden.

1.2.2 Welche Tags gibt es?

```
In [ ]: wget -q https://registry.hub.docker.com/v1/repositories/hello-world/tags -O - | jq -r
```

```
In [ ]: docker container ls
```

```
In [ ]: docker container ls -a
```

1.3 Docker Images

Alpine Linux ist eine sicherheitsorientierte, Linux-Distribution mit geringen Resourcenanforderungen. Sie basiert auf * musl * libc * busybox

Daher wird sie sehr häufig als Grundlage für Container Images verwendet.

```
In [4]: docker image pull alpine

Using default tag: latest
latest: Pulling from library/alpine
Digest: sha256:621c2f39f8133acb8e64023a94dbdf0d5ca81896102b9e57c0dc184cadaf5528
Status: Image is up to date for alpine:latest
```

Das `docker pull` Kommando holt das alpine Image von der Docker Registry und speichert es lokal. Default für die Docker Registry ist hub.docker.com, kann aber geändert werden.

Um die aktuell verwendete Registry zu sehen, verwenden sie das folgende Kommando:

```
In [5]: docker info | grep Registry

WARNING: No swap limit support
Registry: https://index.docker.io/v1/
```

https://index.docker.io leitet den Browser weiter auf https://hub.docker.com.

```
In [6]: docker image ls

REPOSITORY                      TAG                 IMAGE ID            CREATED             SIZ
admincamp/hello                 latest              915fa307573f        2 days ago          4.4
admincamp/commit                latest              f0329cfb5677        3 days ago          4.4
admincamp/vim-optimized         latest              d5d59d05f003        3 days ago          201
admincamp/vim                   latest              81ff3f410667        3 days ago          242
admincamp/curl                  latest              445b6b03a6f5        3 days ago          7.2
admincamp/alpine                latest              36600b1d7064        3 days ago          5.7
alpine                          latest              196d12cf6ab1        6 days ago          4.4
hello-world                     latest              4ab4c602aa5e        10 days ago         1.8
ibmcom/websphere-traditional    latest              f786ba8916fb        12 days ago         1.7
ubuntu                          18.04               cd6d8154f1e1        12 days ago         84.
nginx                           latest              06144b287844        13 days ago         109
```

1.3.1 Details zu einem Image anzeigen

```
 docker image inspect alpine
```

```
In [7]: docker image inspect alpine
```

```
[
    {
        "Id": "sha256:196d12cf6ab19273823e700516e98eb1910b03b17840f9d5509f03858484d321",
        "RepoTags": [
            "alpine:latest"
        ],
        "RepoDigests": [
            "alpine@sha256:621c2f39f8133acb8e64023a94dbdf0d5ca81896102b9e57c0dc184cadaf5528"
        ],
        "Parent": "",
        "Comment": "",
        "Created": "2018-09-11T22:19:50.322783064Z",
        "Container": "cdce86c23e71b41aab0cdb15c010945deadaf59b296f216b0f9f43c331730049",
        "ContainerConfig": {
            "Hostname": "cdce86c23e71",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) ",
                "CMD [\"/bin/sh\"]"
            ],
            "ArgsEscaped": true,
            "Image": "sha256:836dc9b148a584f3f42ac645c7d2bfa11423c1a07a054446314e11259f0e59b7"
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {}
        },
        "DockerVersion": "17.06.2-ce",
        "Author": "",
        "Config": {
            "Hostname": "",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
```

```
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/sh"
            ],
            "ArgsEscaped": true,
            "Image": "sha256:836dc9b148a584f3f42ac645c7d2bfa11423c1a07a054446314e11259f0e59b7"
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": null
        },
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 4413370,
        "VirtualSize": 4413370,
        "GraphDriver": {
            "Data": {
                "MergedDir": "/var/lib/docker/overlay2/df9e9c4979acfde358bdf49828dd1c74d9164df
                "UpperDir": "/var/lib/docker/overlay2/df9e9c4979acfde358bdf49828dd1c74d9164dfd
                "WorkDir": "/var/lib/docker/overlay2/df9e9c4979acfde358bdf49828dd1c74d9164dfde
            },
            "Name": "overlay2"
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:df64d3292fd6194b7865d7326af5255db6d81e9df29f48adde61a918fbd8c332"
            ]
        },
        "Metadata": {
            "LastTagTime": "0001-01-01T00:00:00Z"
        }
    }
]
```


1.4 Starten eines Docker containers

```
docker container run alpine ls -l
```

oder

```
docker run alpine ls -l
```

Zuerst als Vergleich die Ausgabe von `ls -l` im aktuellen Verzeichnis.

```
In [ ]: ls -l
```

```
In [8]: docker run alpine ls -l
```

```
total 52
drwxr-xr-x    2 root     root          4096 Sep 11 20:23 bin
drwxr-xr-x    5 root     root           340 Sep 18 12:36 dev
drwxr-xr-x    1 root     root          4096 Sep 18 12:36 etc
drwxr-xr-x    2 root     root          4096 Sep 11 20:23 home
drwxr-xr-x    5 root     root          4096 Sep 11 20:23 lib
drwxr-xr-x    5 root     root          4096 Sep 11 20:23 media
drwxr-xr-x    2 root     root          4096 Sep 11 20:23 mnt
dr-xr-xr-x  373 root     root             0 Sep 18 12:36 proc
drwx------    2 root     root          4096 Sep 11 20:23 root
drwxr-xr-x    2 root     root          4096 Sep 11 20:23 run
drwxr-xr-x    2 root     root          4096 Sep 11 20:23 sbin
drwxr-xr-x    2 root     root          4096 Sep 11 20:23 srv
dr-xr-xr-x   13 root     root             0 Sep 18 12:36 sys
drwxrwxrwt    2 root     root          4096 Sep 11 20:23 tmp
drwxr-xr-x    7 root     root          4096 Sep 11 20:23 usr
drwxr-xr-x   11 root     root          4096 Sep 11 20:23 var
```

Sie sehen es wird das Directory Listing aus dem Container zurückgeliefert.

Die Anzeige von `ls` ist natürlich keine Raketenwissenschaft, aber das Kommando an sich ist interessant um die Technik dahinter zu verstehen.

`docker container run alpine` startet einen Container auf Basis des Images alpine. Das darauffolgende Kommando `ls -l` führt das Kommando im Container aus. Nach erfolgreicher Ausführung wird der Container **beendet**.

1.4.1 Anzeige laufender Container

```
docker ps
docker container ls
```

```
In [9]: docker ps
```

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
```

```
In [11]: docker container ls -a
```

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
0768144bd498        alpine              "ls -l"             48 seconds ago      Exited (0) 46 s
0679366c186c        hello-world         "/hello"            20 minutes ago      Exited (0) 18 m
c0945ea9d317        hello-world         "/hello"            20 minutes ago      Exited (0) 20 m
f99d66d9c0f4        hello-world         "/hello"            27 minutes ago      Exited (0) 27 m
891f6ba23883        hello-world         "/hello"            28 minutes ago      Exited (0) 28 m
```

1.4.2 Weitere Container starten

```
In [ ]: docker run alpine echo "Hallo vom Admincamp"
```

Probieren wir einen Container mit Shell zu starten.

```
docker container run alpine /bin/sh
```

```
In [ ]: docker run alpine /bin/sh
```

Wir sehen, daSS wir nichts sehen. :)
Nach der Ausführung der Shell wird der Container beendet.

Starten des Containers im interactive Mode Das Kommando hierzu:

```
docker run -it alpine /bin/sh
```

müssen Sie in einem Terminalfenster starten! Im Jupyter Notebook ist ein interaktives Arbeiten mit der Shell leider (noch) nicht möglich.
Starten Sie bitte das Kommando und testen dann den Code der nächsten Zelle:

```
In [ ]: docker container ls
```

Anzeige aller Container auf dem Rechner Um alle, also laufende und gestoppte Container anzuzeigen, können wir die Kommandos:

```
docker container ls -a
docker ps -a
```

verwenden.

```
In [ ]: docker container ls -a
```

docker run --help

```
In [ ]: docker container run --help
```

1.4.3 Vorhandenen Container starten

Tip:
Anstatt die volle Container-ID zu verwenden, können Sie nur die ersten Zeichen verwenden, solange sie ausreichen, um einen Container eindeutig zu identifizieren.
Wir könnten also einfach "ba8" verwenden, um die Containerinstanz im obigen Beispiel zu identifizieren, da keine anderen Container in dieser Liste mit diesen Zeichen beginnen.
In den folgenden Beispielen müssen Sie die in Ihrem Terminal angezeigten IDs verwenden!

```
In [ ]: docker start ba8
```

```
In [ ]: docker container ls
```

```
In [ ]: docker exec ba8 ls
```

1.4.4 Container Isolation

In den obigen Schritten haben wir mehrere Befehle über Containerinstanzen mit Hilfe von `docker run` ausgeführt. Das `docker ps -a` Kommando zeigte uns, dass mehrere Container aufgelistet waren. Warum sind so viele Container aufgelistet, wenn sie alle auf dem `Alpine` Image basieren?

Dies ist ein Sicherheitskonzept in der Welt der Docker-Container!

Obwohl jeder `docker run` das gleiche Alpine Image verwendete, war jede Ausführung ein separater, isolierter Container. Jeder Container hat ein eigenes Dateisystem und läuft in einem anderen Namensraum; standardmäSSig hat ein Container keine Möglichkeit, mit anderen Containern zu interagieren, auch nicht mit denen aus demselben Image.

In einem Terminal ausführen:

```
docker container run -it alpine /bin/ash

echo "Hello World" > hello.txt

ls
```

```
In [ ]: docker container run alpine ls
```

```
docker ps -a

CONTAINER ID        IMAGE               COMMAND             CREATED
d515b951ee09        alpine              "/bin/ash"          About a minute ago
```

```
In [ ]: docker container ls -a
```

1.5 Eigene Container erstellen

Es gibt bei https://hub.docker.com bereits über 700.000 Images, die wir verwenden können. Diese sind natürlich nicht für unsere Umgebungen, Wünsche und Anforderungen optimiert. Wir müssen also selbst Images erstellen.

Als ersten Schritt sehen wir die Verwendung von `commit` an einem unserer Docker Container.

1.5.1 Image aus einem Container erstellen

Starten wir mit einer interaktiven Shell in einem Ubuntu Container (in einem seperaten Terminal):

```
docker container run -ti ubuntu bash
```

Als ersten Schritt installieren wir:

```
apt update
apt install -y cowsay
/usr/games/cowsay "Hello Admincamp"

 -----------------
< Hello Admincamp >
```

```
 -----------------
    \   ^__^
     \  (oo)\_______
        (__)\       )\/\
            ||----w |
            ||     ||

exit

 docker ps -a

 docker container commit CONTAINER_ID admincamp/cowsay
```

Beispiel

```
 docker commit 7ed114d35192 admincamp/cowsay
sha256:dfd8e66f098f86343e2ef939a6118d810969da19c36e97ac5585049ce550308e

 docker image ls

REPOSITORY                      TAG                 IMAGE ID            CREATED             SI

In [ ]: docker image ls

In [ ]: docker image tag dfd8e66f098f admincamp18/cowsay

In [ ]: docker image ls

REPOSITORY                      TAG                 IMAGE ID            CREATED             SI
admincamp18/cowsay              latest              dfd8e66f098f        4 minutes ago       20
```

Verwenden des Images Das neue Image kann nun wieder als Grundlage für neue interaktive Container genutzt werden.

```
 docker run -it admincamp18/cowsay
root@8284b8096225:/# /usr/games/cowsay "Hello Admincamp"
 -----------------
< Hello Admincamp >
 -----------------
    \   ^__^
     \  (oo)\_______
        (__)\       )\/\
            ||----w |
            ||     ||
```

Wir haben hier auch die gleiche `history` wie in dem Container, den wir als Grundlage benutzt haben.

Natürlich kann er auch direkt auf der Kommandozeile verwendet werden.

```
In [ ]: docker run admincamp18/cowsay /usr/games/cowsay "Hello Admincamp"
```

1.5.2 Image mit einem Dockerfile erstellen

Anstatt Images über Container und `commit` zu erstellen, kann auch ein `Dockerfile` verwendet werden.

Der Vorteil der Verwendung eines `Dockerfile` ist die Reproduzierbarkeit. Jeder kann das Dockerfile einsehen und es überprüfen!

Cowsay Das Dockerfile für unser cowsay-Image sieht folgendermaSSen aus. Siehe im Verzeichnis dieser Jupyter-Datei unter `cowsay/Dockerfile`:

```
FROM ubuntu:latest

RUN apt update
RUN apt install -y cowsay

CMD ["/usr/games/cowsay", "Hello Admincamp"]
```

Erstellen des Images mit:

```
 docker image build -t admincamp18/cowsay:v2 cowsay
```

```
 docker image ls -a
REPOSITORY                      TAG                 IMAGE ID            CREATED              S
admincamp18/cowsay              v2                  edd79094cfdf        About a minute ago   1
<none>                          <none>              5289a1c322b0        About a minute ago   1
<none>                          <none>              746d3acf046a        About a minute ago   1
```

Wir sehen daSS unsere beiden `RUN` und das `CMD` Kommando drei Layer erstellt haben.

```
In [ ]: docker run admincamp18/cowsay:v2
```

Beispiel aus training.play-with-docker.com Die folgenden Dateien liegen im Unterverzeichnis `hello`.

index.js

```
var os = require("os");
var hostname = os.hostname();
console.log("hello from " + hostname);
```

Dockerfile

```
FROM alpine

RUN apk update && apk add nodejs
COPY . /app
WORKDIR /app

CMD ["node","index.js"]
```

Image erstellen

```
docker image build -t hello:v0.1 hello
```

```
docker image ls -a
```

```
REPOSITORY                          TAG                 IMAGE ID            CREATED              S

<none>                              <none>              725b2ab2a84b        About a minute ago   3
hello                               v0.1                882c41bc5f3e        About a minute ago   3
<none>                              <none>              88d643094431        About a minute ago   3
<none>                              <none>              59ad92da2980        About a minute ago   3
```

Wir sehen also, daSS es während der Ausführung so aussieht, wie wenn der Container ein einzelnes Paket aus OS und Applikation ist. Aber das Image ist aus **Layern** aufgebaut.

```
In [ ]: docker run hello:v0.1
```

Was ist gerade passiert?
Orginal: https://training.play-with-docker.com/ops-s1-images/
Wir haben zwei Dateien erstellt:
Unser Anwendungscode (`index.js`) ist ein einfaches Stück Javascript-Code, das eine Nachricht ausgibt. Und das Dockerfile ist die Anleitung für die Docker-Engine, um unseren benutzerdefinierten Container zu erstellen. Diese Dockerdatei führt folgende Schritte aus:

1. Gibt ein Basisbild an (das alpine Image), das wir in früheren Schritten auch verwendet haben.
2. Dann führt es zwei Befehle (apk update und apk add) innerhalb des Containers aus, der den Node.js-Server installiert.
3. Dann haben wir ihm gesagt, dass er Dateien aus unserem Arbeitsverzeichnis in den Container kopieren soll. Die einzige Datei, die wir im Moment haben, ist unsere `index.js`
4. Als nächstes spezifizieren wir das WORKDIR - das Verzeichnis, das der Container beim Start verwenden soll.
5. Und schlieSSlich gaben wir unserem Container einen Befehl (CMD), um ihn beim Start des Containers auszuführen.

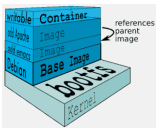
Mit einer Dockerdatei können wir präzise Befehle angeben, die für jeden ausgeführt werden, der diesen Container verwendet.
Die Dockerdatei ermöglicht es uns, festzulegen, wie ein Container erstellt werden soll, so dass wir diese Schritte jederzeit genau wiederholen können, und wir können festlegen, was der Container tun soll, wenn er läuft.

1.5.3 Layer

Wir überprüfen das von uns erstellte Image (`cowsay` oder `hello`) mit:

```
In [ ]: docker image history hello:v0.1
```

Im Ordner hello2 ist eine neue Version verfügbar. Es wurde nur eine Zeile in `index.js` hinzugefügt.



Layer

```
docker image build -t hello:v0.2 hello2
```

```
In [ ]: docker image build -t hello:v0.2 hello2
```

```
In [ ]: docker image history hello:v0.2
```

Sie sehen, daSS die untersten drei Layer für beide Images gleich sind. Erst beim `COPY` unterscheiden sich die Images.

1.6 Löschen von Images und Containern

Mit den folgenden Kommandos sehen sie, daSS wir inzwischen schon eine groSSe Anzahl Container und Images erstellt haben. Da wir die meisten von Docker Hub oder über `Dockerfile` erstellt haben, können wir sie problemlos löschen und bei Bedarf neu genieren.

```
docker image ls -a
docker container ls -a
```

1.6.1 Container löschen

Gestoppte Container können gelöscht werden.

```
docker container rm CONTAINER_ID
```

Beispiel

```
docker container rm 3cf8a423c10b
3cf8a423c10b
```

Alle Container löschen

```
docker container rm $(docker container ls -a | grep -v CONTAINER |awk '{print $1}')

# Alternativ und kürzer:
docker container rm $(docker container ls -a -q)
```

1.6.2 Image löschen

```
docker image rm IMAGE_ID
```

Beispiel

```
 docker image rm ffe8239a147c
Untagged: ruby:2.3.1
Untagged: ruby@sha256:a5ebd3bc0bf3881258975f8afa1c6d24429dfd4d7dd53a299559a3e927b77fd7
Deleted: sha256:ffe8239a147c666621476db01004e6d525b9a3b0db2deace8861afba2aef3001
Deleted: sha256:5983b1164ccb0e6ac7a2dab6a6e29efaa11fcdf1350e91ff2aab919c31d63934
Deleted: sha256:6049f38e8c033dea5e5b7ee7abd5b1af20e5c77f570752b560e49e51c35cc4df
Deleted: sha256:d2987ea1ae6dfc8f2f3b07fd4e5802feab53e95c1590f12453166d2f57d2f301
Deleted: sha256:8f3411639785500fb0133ca55d2639e0661ceb439417574adf084341905dd3d0
Deleted: sha256:b7b7fbb077245d146b83458cb5099b22cf667f82342d7eed0b16f8e5f641149f
Deleted: sha256:d0a6c2235a31bee1812d9db1afdf5ec5ff5eec53218e05d328db9c09bfded893
Deleted: sha256:be2878c078e2df267e68687181f8373a31853f79086cac36edbf81ce07b40b28
Deleted: sha256:fe4c16cbf7a4c70a5462654cf2c8f9f69778db280f235229bd98cf8784e878e4


 docker image rm 3fd9065eaf02
Error response from daemon: conflict: unable to delete 3fd9065eaf02 (cannot be forced) - image
```

Im letzteren Fall ist das Image also einer der Layer, der für ein weiteres Image, oder einen Container verwendet wird.

Alle lokalen Images löschen

```
 docker image rm $(docker image ls -a | grep -v 'IMAGE' | awk '{print $3}')

# Alternativ und kürzer
 docker image rm $(docker image ls -a -q)
```

Bzw. mit Force:

```
 docker image rm -f $(docker image ls -a | grep -v 'IMAGE' | awk '{print $3}')
```

Vorsicht bei der Verwendung von `-f|--force` es werden damit alle Images gelöscht.

```
In [ ]: docker image ls -a

In [ ]: docker container ls -a
```

1.7 Docker Netzwerk

Bisher haben wir Container nur lokal verwendet. Interessant wird es v.a. dann, wenn die Container die Services von anderen Containern nutzen können, bzw. wenn wir Services aus Containern im Netz verfügbar machen können.

```
 docker network COMMAND

In [ ]: docker network --help
```

1.7.1 Netzwerke auflisten

```
In [ ]: docker network ls
```

Die Ausgabe zeigt die Netzwerke die nach einer Standardinstallation vorhanden sind.

1.7.2 Details zum Docker Netzwerk

Um die Konfiguration eines Netzwerks anzusehen, kann der Befehl

```
In [ ]: docker network inspect bridge
```

verwendet werden.

1.7.3 Docker Bridge

Docker Netzwerke sind als Bridge angelegt und können auch mit Linux Bordmitteln `brctl` oder `brctl show` konfiguriert werden.

Hierzu muss eventuell das Paket `bridge-utils` installiert werden.

```
apt install bridge-utils
```

```
In [ ]: brctl show docker0
```

```
In [ ]: docker container run -dt ubuntu sleep infinity
```

Da beim Kommando `docker run` kein Netzwerk angegeben wurde, wird der Container zum default Bridge Netzwerk `docker0` hinzugefügt.

```
In [ ]: docker container ls
```

```
In [ ]: brctl show docker0
```

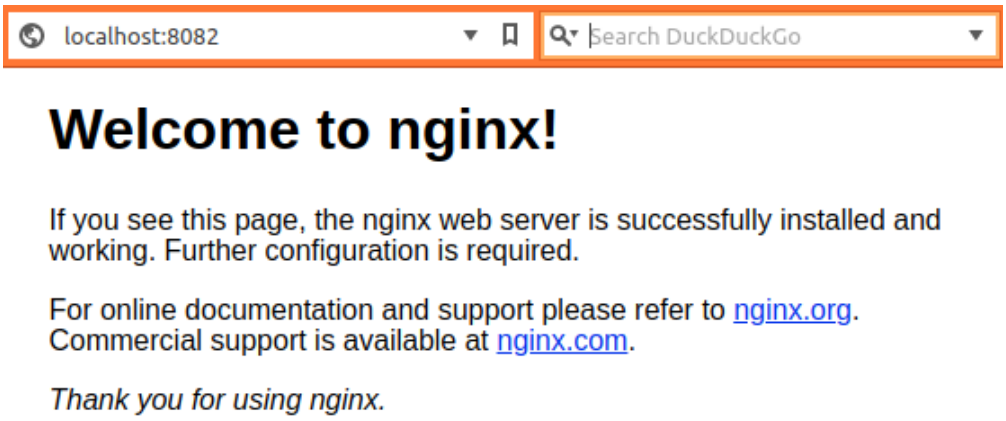
Überprüfen Sie das Bridge Netzwerk erneut, wir sehen daSS jetzt ein Container hinzugefügt wurde.

```
docker network inspect bridge
```

```
...
"Containers": {
        "bf5b8d67401e66f9b9e3aa4211ee9b59aad8820a9cb30b089c8a1cdc1a5df6f9": {
            "Name": "optimistic_shirley",
            "EndpointID": "0641b3b9d67544c7f72e5817bca5cbbc26c98b4566779b2fe0e29640f5215db2",
            "MacAddress": "02:42:ac:11:00:02",
            "IPv4Address": "172.17.0.2/16",
            "IPv6Address": ""
        }
```

```
In [ ]: docker network inspect bridge
```

Docker verwendet also ein Netzwerk, das von ausserhalb Ihres Servers nicht erreichbar ist.



nginx default page

1.7.4 Container von aussen erreichbar machen

Für den nächsten Abschnitt verwenden wir nginx.

> nginx [engine x] is an HTTP and reverse proxy server, a mail proxy server, and a generic TCP/UDP proxy server, originally written by Igor Sysoev.

Auf dem Docker Hub können Sie sich über das Image informieren: https://hub.docker.com/_/nginx/

> Hosting some simple static content
> `$ docker run --name some-nginx -v /some/content:/usr/share/nginx/html:ro -d nginx`
>
> Exposing external port
> `$ docker run --name some-nginx -d -p 8080:80 some-content-nginx`
> Then you can hit http://localhost:8080 or http://host-ip:8080 in your browser.

Wir starten einen Container auf Basis des Default nginx Images und mappen den Container-Port 80 auf den lokalen Port 8082:

```
 docker run -d -p 8082:80 nginx
```

```
In [12]: docker run -d -p 8082:80 nginx

0f2e76e92f5f5ec5c2c52c855f1be0ef7b36e46deb1914108ef9ed224674913b
```

Wenn wir mit dem Browser die Url http://localhost:8082 öffnen erhalten wir:

```
In [ ]: docker ps
```

Der weitergeleitete Port ist auch beim Kommando

```
docker container ls
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
37e23c769972        nginx               "nginx -g 'daemon of"   5 minutes ago       Up 5 minut
```

sichtbar. Ausserdem sehen wir hier sehr gut den Namen den Docker automatisch an Container vergibt. In diesem Fall `zen_ritchie`. Bei allen Docker Kommandos in denen die CONTAINER_ID verwendet wird, kann auch der Name verwendet werden!

1.8 Docker Advanced

1.8.1 Namen von Docker Containern

Sie können den Namen auch beim Erstellen des Containers selbst vergeben:

```
docker container run --name cool-nginx -d -p 8083:80 nginx
```

```
docker container ls
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
1cec6dc0c37e        nginx               "nginx -g 'daemon of"   8 seconds ago       Up 7 second
37e23c769972        nginx               "nginx -g 'daemon of"   8 minutes ago       Up 8 minut
```

```
In [ ]: docker container run --name cool-nginx -d -p 8083:80 nginx

In [ ]: docker container ls

In [ ]: docker stop cool-nginx

In [ ]: docker container ls
```

1.8.2 Kopieren von lokalen Dateien in laufende Container

Um z.B. die Default index.html unseres Webservers auszutauschen, können wir das Kommando `docker cp` verwenden.

```
docker container cp nginx/index.html zen_ritchie:/usr/share/nginx/html/
```

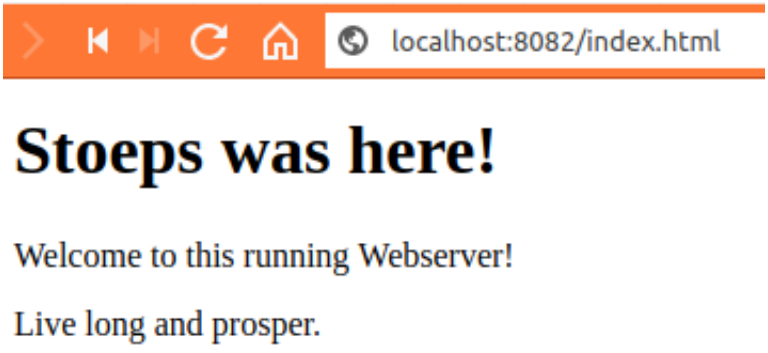
`docker container cp` gibt keine Bestätigung auf der Konsole aus!

Wenn Sie jetzt die URL http://localhost:8082/index.html im Browser neu laden, wird der neue Inhalt ausgeliefert.

Sollten wir die neue Webseite im Image wollen, muss diese über `commit` oder einem entsprechenden `Dockerfile` ins Image geschrieben werden!

```
In [ ]: docker container cp nginx/index.html zen_ritchie:/usr/share/nginx/html/

In [ ]: curl http://localhost:8082/index.html
```



nginx geanderte Seite

1.8.3 Lokale Ordner in Container mappen

Praktischer ist ein lokales Verzeichnis zu verwenden, dann können direkt vom Host Dateien geändert oder dazukopiert werden.

```
docker container run --name nginx-localfolder -v $(pwd)/nginx:/usr/share/nginx/html/ -d -p 808
```

```
In [ ]: docker container run --name nginx-localfolder -v $(pwd)/nginx:/usr/share/nginx/html/ -
```

```
In [ ]: docker container ls
```

```
In [ ]: curl http://localhost:8084/index.html
```

Im `nginx` Ordner befindet sich eine 2. Datei `index2.html`, diese ist in unseren Testservern nur über `nginx-localfolder` aber nicht im `zen_ritchie` zu finden.

```
In [ ]: curl http://localhost:8082/index2.html
```

```
In [ ]: curl http://localhost:8084/index2.html
```

1.8.4 Volume Container verwenden

Ein Docker Volume oder ein Container Volume ist ebenfalls ein Container!

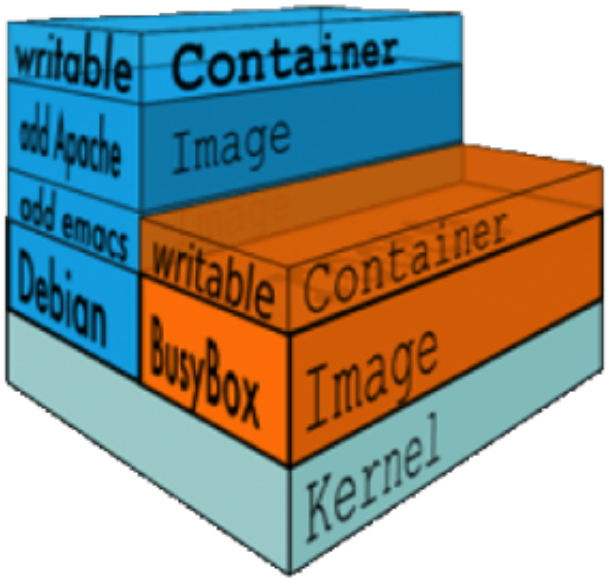
Im Prinzip neben der Verwendung eines lokalen Ordners die einzige Möglichkeit, um Daten persistent zu speichern, wenn ein Container zerstört wird. Da das Prinzip von Docker die Trennung von Applikation und Daten ist, um Updates schneller verfügbar zu machen, sollte man sich mit dem Thema Volumes näher befassen.

Volume bei der Ausführung von `docker container run`

```
docker container run -it --name vol-test -h my-new-container -p 8085:80 -d -v /usr/share/nginx
```

Volumes können auch im Dockerfile mit `VOLUME /usr/share/nginx/html` definiert werden. Oder mit dem Kommando `docker volume create --name my-volume` angelegt werden.

Wir starten wieder einen nginx Container, diesmal mit dem Hostnamen `my-new-container`. Der `/usr/share/nginx/html` Ordner wird als Volume, also unabhängig vom Unionfs definiert.



volume

```
In [ ]: docker container run -it --name vol-test -h my-new-container -p 8085:80 -d -v /usr/shar

In [ ]: docker container ls

In [ ]: docker inspect -f "{{json .Mounts}}" vol-test | jq .
```

In einem Terminal können wir uns den Inhalt des Verzeichnisses ansehen:

```
sudo ls -l /var/lib/docker/volumes/1a7a5096599a274c5fd4828fd51352183901bc9f512adcc4662a414ac32c
total 8
-rw-r--r-- 1 root root 494 Aug 28 15:32 50x.html
-rw-r--r-- 1 root root 612 Aug 28 15:32 index.html
```

Wir sehen hier einen groSSen Vorteil der Verwendung von Volumes gegenüber dem mapping von lokalen Ordnern. Dateien die sich im Image in diesem Bereich befunden haben, werden automatisch in das Volume kopiert!

Man kann auch anderen Containern Zugriff auf Container Volumes geben, dazu gibt es den `--volumes-from`! Dies kann z.B. verwendet werden, um Daten zu sichern, oder Daten/Datenbanken nach Updates zu aktualisieren.

Beispiel

```
 docker run -it -h newcontainer --volumes-from vol-test alpine ash

/ # ls /usr/share/nginx/html/
50x.html    index.html  newfile
/ # hostname
newcontainer
```

Sie sehen der neue Container `newcontainer` hat das gleiche Volume wie unser `vol-test` Container. Dateien können geändert werden. Man kann z.B. mit dieser Technik zusätzliche Programme für das Update installieren, das aber im Image für die Ausführung nicht notwendig ist. Dadurch hält man die Container/Images kleiner.

Oder installiert `unzip` kopiert ein Archiv in den temporären Container und entpackt diesen in das Volume. Dadurch wird der Container nicht grösSer und der temporäre Container kann wieder zerstört werden.

Um ein Volume eines anderen Containers zu verwenden, muss der Container nicht laufen!

Zurück zu unserem Volume:

Wir können auch mit

```
sudo touch /var/lib/docker/volumes/1a7a5096599a274c5fd4828fd51352183901bc9f512adcc4662a414ac3
```

eine Datei erstellen.

1.8.5 Shell in laufendem Container starten

Um zu sehen ob die Datei vorhanden ist, können wir einfach ein Shell im laufenden Container `vol-test` starten.

```
docker exec -it vol-test bash
```

```
root@my-new-container:/# ls /usr/share/nginx/html/ -al
total 16
drwxr-xr-x 2 root root 4096 Sep 11 17:25 .
drwxr-xr-x 3 root root 4096 Sep  5 00:56 ..
-rw-r--r-- 1 root root  494 Aug 28 13:32 50x.html
-rw-r--r-- 1 root root  612 Aug 28 13:32 index.html
-rw-r--r-- 1 root root    0 Sep 11 17:25 newfile
```

Wir sehen also die Datei und ausserdem am Konsolenprompt, daSS der Hostname auf `my-new-container` gesetzt ist.

```
In [ ]: docker volume ls
```

Genau überlegen! Damit werden persistente Daten gelöscht!

```
In [ ]: docker volume rm $(docker volume ls -q)
```

1.8.6 Docker restart policy

Um abgestürzte Container wieder zu starten, oder Container nach reboots zu starten kann die Restart Policy verwendet werden. Hier ein Beispiel mit einem Shellskript.

crash.sh

```
#!/bin/bash
sleep 20
exit 1
```

Dockerfile

```
FROM ubuntu:14.04
ADD crash.sh /
CMD /bin/bash /crash.sh
```

```
 docker build -t admincamp18/crash crash
```

```
In [ ]: docker build -t admincamp18/crash crash
```

```
In [ ]: docker run -d --name crashtestdummy admincamp18/crash
```

```
In [ ]: docker container ls
```

Nach 20 Sekunden beendet sich der Container. `Exit 1` ist `not successful`.

```
 docker run -d --name crashtestdummy2 --restart always admincamp18/crash
```

```
In [ ]: docker run -d --name crashtestdummy2 --restart always admincamp18/crash
```

```
In [ ]: docker container ls
```

```
In [ ]: docker container ls
```

`docker --restart` kennt folgende Optionen:

- no [default]
- on-failure
- unless-stopped
- always

1.9 Domino Dockerfile von Eknori als detailliertes Beispiel

```
FROM centos

ENV DOM_SCR=resources/initscripts
ENV DOM_CONF=resources/serverconfig
ENV NUI_NOTESDIR /opt/ibm/domino/

RUN yum update -y && \
    yum install -y which && \
    yum install -y wget && \
    yum install -y perl && \
    useradd -ms /bin/bash notes && \
    usermod -aG notes notes && \
    usermod -d /local/notesdata notes && \
    sed -i '$d' /etc/security/limits.conf && \
    echo 'notes soft nofile 60000' >> /etc/security/limits.conf && \
    echo 'notes hard nofile 80000' >> /etc/security/limits.conf && \
    echo '# End of file' >> /etc/security/limits.conf
```

```
COPY ${DOM_CONF}/ /tmp/sw-repo/serverconfig

RUN mkdir -p /tmp/sw-repo/ && \
    cd /tmp/sw-repo/ && \
    wget -q http://YOUR_HOST/DOMINO_9.0.1_64_BIT_LIN_XS_EN.tar && \
    tar -xf DOMINO_9.0.1_64_BIT_LIN_XS_EN.tar &&\
    cd /tmp/sw-repo/linux64/domino && \
    /bin/bash -c "./install -silent -options /tmp/sw-repo/serverconfig/domino901_response.dat"
    cd / && \
    rm /tmp/* -R

RUN mkdir -p /etc/sysconfig/
COPY ${DOM_SCR}/rc_domino /etc/init.d/
RUN chmod u+x /etc/init.d/rc_domino && \
    chown root.root /etc/init.d/rc_domino
COPY ${DOM_SCR}/rc_domino_script /opt/ibm/domino/
RUN chmod u+x /opt/ibm/domino/rc_domino_script && \
    chown notes.notes /opt/ibm/domino/rc_domino_script
COPY ${DOM_SCR}/rc_domino_config_notes /etc/sysconfig/
```

1.9.1 Zusammenfassen von Befehlen im Dockerfile

Wie man hier sehr gut sieht, werden die `RUN` Befehle mit `&&` zusammengefaSSt. `&&` startet unter Linux das nächste Kommando, wenn das vorhergehende mit Rückgabewert 0 beendet wurde.

Dadurch werden also die Layer im Image minimiert.

Ausserdem wird der Installer nicht in das Image kopiert (`COPY` oder `ADD`) sondern zur Laufzeit von einem Webserver geladen (`wget -q http....`) dazu kann man z.B. prima einen nginx-Container verwenden, dessen `/usr/share/nginx/html` Verzeichnis in einen Ordner gemappt ist, in dem die Setup-Dateien für Domino liegen.

Die Installation eines eventuellen Fixpacks sollte man am Besten in ein eigenes Dockerfile auslagern und als `FROM` das Image von Version 9.0.1 verwenden.

Am Ende bekommt man Images mit der GröSSe 2-4 GB, je nach Aufwand den man im Dockerfile betreibt.

Beim Start des Domino Containers macht es natürlich Sinn, den Inhalt von `/local/notesdata` in ein Volume zu packen.

1.9.2 Warum wird es nicht einfach?

Ich denke Domino on Docker macht erst richtig Sinn, wenn die Tasks als Microservices implementiert sind und dadurch die ImagegröSSen kleiner werden. Gerade im Hinblick auf eine mögliche Orchestrierung mit Kubernetes oder Docker Swarm beschleunigt das den Failover immens.

Domino Data Im Moment werden bei Updates auch Dateien im Data-Verzeichnis verändert.

- Templates
- `domino/html`

Sollte man also vorhaben das Data-Volume wiederzuverwenden und nur das Binary-Image zu entsorgen, fehlen wichtige Aktualisierungen!

Eine Möglichkeit ist z.B. während des `docker build` *das Data zu zippen und wegzukopieren. Dann braucht man ein Skript oder einen temporären Container um die Updates in das Data zu übernehmen.*

Netzwerk Ports Man kann über Reverseproxy (nginx) Webservices sehr gut zusammenfassen. D.h. ein Domino-Webserver kann im Docker sehr gut funktionieren. Aber wie administrieren wir diesen? Unsere Container laufen hinter einer Bridge, Ports müssen nach aussen freigeben werden. ES bleibt also nichts als dem Host mehrere IPs zu gegen und die Ports `1352` darauf zu binden, oder man hat Schwierigkeiten mit dem Adminclient zuzugreifen.

Ein einzelner Domino Container ist ja sicher nicht das Ziel.

1.10 Orchestrierung

Docker läuft also auf einem Host, sollte dieser ausfallen, fallen sehr viele Services auf einmal aus.

Mit Docker Swarm oder Kubernetes kann man eine groSSe Anzahl Container auf mehreren Nodes (Servern) betreiben. Automatischer Failover und Loadbalancing sind damit sehr einfach zu implementieren.

Mit Kubernetes kann man hochverfügbare Services auf schlechter Hardware zur Verfügung stellen.