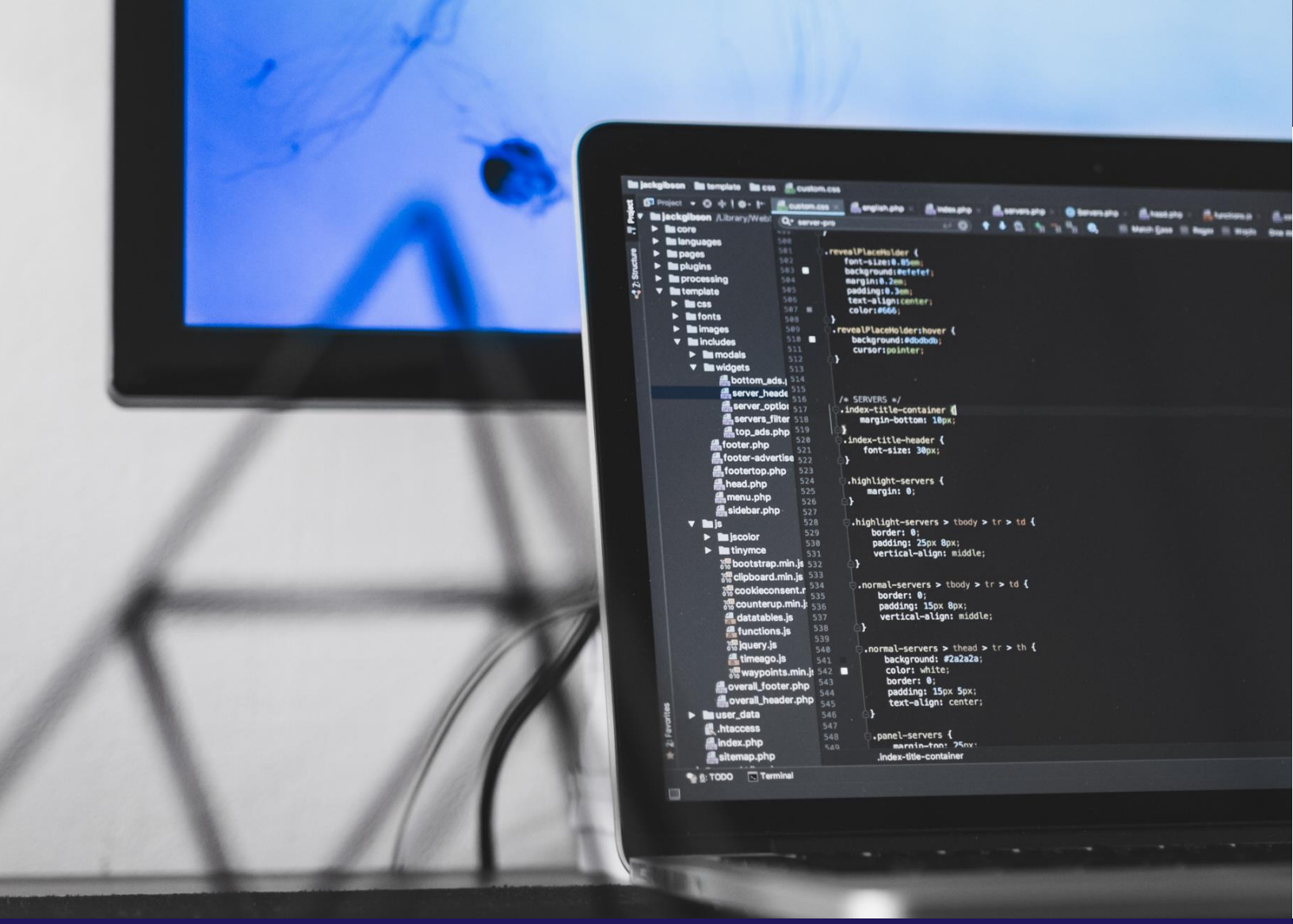


Docker Grundlagen

Kursheft

Kapitel 4: Container

DATAMICS
machine intelligence consulting services

Container Befehle

Container Identifikation

Als Nächstes verwenden wir den Docker Container mit alpine Linux. Alpine Linux ist eine unabhängige, nicht-kommerzielle, allgemeine Linux-Distribution, die für Power-User entwickelt wurde, die Sicherheit, Einfachheit und Ressourceneffizienz schätzen.

https://alpinelinux.org/about/

Hinweis: Du kannst den Container auch ohne pull starten; dann wird das Image automatisch heruntergeladen, wenn es lokal nicht vorhanden ist.

Gib dem Container einen Namen, damit du diesen einfacher handhaben kannst.

```
docker run --name mein_name alpine
```

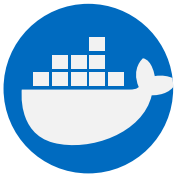
```
docker ps -a
```

```
docker rm mein_name
```

Automatisches Entfernen des Containers beim Beenden:

```
docker run --name mein_name --rm alpine

docker run --name mein_name --rm alpine
```



Detached

Um einen Container im losgelösten (**detached**) Modus zu starten, verwendest du die Option **-d=true** oder einfach **-d**. Container werden im detached Modus beendet, wenn der Stammprozess, der zum Ausführen des Containers verwendet wird, auch beendet wird.

Es sei denn, du gibst auch die Option **--rm** an. Wenn du **-d** mit **--rm** verwenden, wird der Container entfernt, wenn er beendet wird oder wenn der Daemon beendet wird, je nachdem, was zuerst geschieht.

```
docker run -d --name nginx -p 8080:80 nginx

docker stop nginx
```

Foreground (Vordergrund)

Für interaktive Prozesse (z.B. eine Shell) musst du **-i -t** zusammen verwenden, um dem Containerprozess ein tty zuzuweisen. **-i -t** wird oft **-it** geschrieben.

```
docker run -i -t ubuntu /bin/bash
```

Die Angabe von **-t** ist nicht möglich, wenn der Client seine Standardeingabe von einer Pipe empfängt, wie in:

```
echo test | docker run -i busybox cat
```

Laufzeiteinschränkungen für Ressourcen

Setzt man keine Limits für den Speicher, können die Prozesse im Container so viel Speicher und Swap-Speicher verwenden, wie sie benötigen.

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

Legt man ein Speicherlimit und ein deaktiviertes swap-Speicherlimit fest, können die Prozesse im Container maximal 300M-Speicher und so viel Swap-Speicher verwenden, wie sie benötigen (wenn der Host Swap-Speicher unterstützt).

```
docker run -it -m 300M --memory-swap -1 ubuntu /bin/bash
```

Legen wir nur ein Speicherlimit fest, können die Prozesse im Container 300M Speicher und 300M Swap-Speicher verwenden, standardmäßig wird die gesamte virtuelle Speichergröße (--Memory-Swap) als Doppelte des Speichers festgelegt, in diesem Fall wäre Speicher + Swap 2* 300M, so dass Prozesse auch 300M Swap-Speicher verwenden können.

```
docker run -it -m 300M ubuntu /bin/bash
```

Legen wir sowohl Speicher- als auch Swap-Speicher fest, können die Prozesse im Container 300 M Speicher und 1G Swap-Speicher verwenden.

```
docker run -it -m 300M --memory-swap 1G ubuntu /bin/bash
```

CPU Beispiele:

In diesem Beispiel werden die Prozesse im Container darauf beschränkt, nur Speicher von den CPUs 1 und 3 zu verwenden.

```
docker run -it --cpuset-mems="1,3" ubuntu /bin/bash
```

In diesem Beispiel werden die Prozesse im Container darauf beschränkt, nur Speicher von den Speicherknoten 0, 1 und 2 zu verwenden.

```
docker run -it --cpuset-mems="0-2" ubuntu /bin/bash
```

Überschreiben der Image Befehle

Hier ist ein Beispiel für das Ausführen einer Shell in einem Container, der so eingerichtet wurde, dass automatisch etwas anderes ausgeführt wird (z. B. /usr/bin/redis-server):

```
docker run -it --entrypoint /bin/bash redis
```

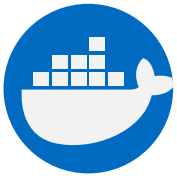
Überschreiben des WORKDIR mit --workdir oder -w=“”:

```
docker run -it -p 8080:80 --entrypoint /bin/bash nginx

docker run -it -p 8080:80 --entrypoint /bin/bash -w="/home" nginx
```

Ein weiteres Beispiel:

```
docker  run -w /path/to/dir/ -i -t  ubuntu pwd
```



Mount Volumes

Das **v-Flag** mountet das aktuelle Arbeitsverzeichnis in den Container. Mit **-v** wird der Befehl innerhalb des aktuellen Verzeichnisses ausgeführt, indem er in das Verzeichnis in den von **pwd** zurückgegebenen Wert wechselt. Diese Kombination führt den Befehl also mithilfe des Containers, aber innerhalb des aktuellen Arbeitsverzeichnisses aus.

```
docker  run  -v `pwd`:`pwd` -w `pwd` -i -t  ubuntu pwd
```

Volumes können in Kombination mit :ro verwendet werden, um zu steuern, wo ein Container Dateien schreibt. Das Flag :ro mountet das Stammdateisystem des Containers als schreibgeschützt, das Schreibvorgänge an andere Speicherorte als die angegebenen Verzeichnisse für den Container verbietet.

```
docker  run  -v `pwd`:`pwd`:ro -w `pwd` -i -t  ubuntu bash
```

Hinzufügen von bind mounts oder Volumes mithilfe des Flags --mount

Mit dem Flag **--mount** kannst du Volumes und Hostverzeichnisse in einem Container mounten. Das Flag **--mount** unterstützt die meisten Optionen, die vom Flag **-v** oder **--volume** unterstützt werden, verwendet jedoch eine andere Syntax. Obwohl nicht geplant ist, **--volume** abzuschaffen, wird die Verwendung von **--mount** empfohlen.

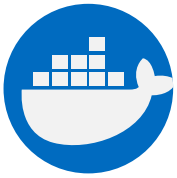
```
docker run -it --mount type=bind,src=`pwd`,dst=/data ubuntu
```

Nur Leserechte:

```
docker run -it --read-only --mount type=bind,src=`pwd`,dst=/data,readonly ubuntu
```

Manchmal wird auch target statt destination verwendet:

```
docker run -it --read-only --mount type=bind,src=`pwd`,target=/data,readonly ubuntu
```



Festlegen von Umgebungsvariablen (-e, --env, --env-file)

Verwende die Flags **-e**, **--env** und **--env-file**, um einfache (nicht-array) Umgebungsvariablen im Container festzulegen oder überschreibe Variablen, die im Dockerfile des von dir ausgeführten Images definiert sind. Du kannst die Variablen und ihren Wert definieren, wenn du den Container ausführst:

```
docker run -e VAR1 --env VAR2=foo ubuntu bash

echo $VAR2
```

Du kannst auch Variablen verwenden, die du in deiner lokalen Umgebung exportiert hast:

```
docker run -e VAR1=value1 --env VAR2=value2 ubuntu env | grep VAR

VAR1=value1

VAR2=value2
```

Beim Ausführen des Befehls überprüft der Docker CLI-Client den Wert der Variablen in ihrer lokalen Umgebung und übergibt ihn an den Container. Wenn kein = bereitgestellt wird und diese Variable nicht in ihre lokale Umgebung exportiert wird, wird die Variable nicht im Container festgelegt.

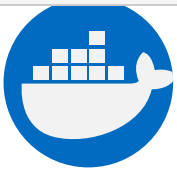
```
export VAR1=value1

export VAR2=value2


docker run --env VAR1 --env VAR2 ubuntu env | grep VAR

VAR1=value1

VAR2=value2
```



Auflisten der Container

Starte einen beliebigen Container

```
docker run --name alpine alpine
```

Um nur **laufende Container** anzuzeigen, verwendest du den folgenden Befehl:

```
docker ps
```

Um **alle Container** anzuzeigen, verwendest du den folgenden Befehl:

```
docker ps -a
```

Um den **zuletzt erstellten Container** (einschließlich aller States) anzuzeigen, verwendest du den folgenden Befehl:

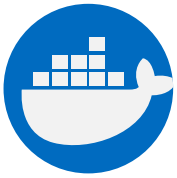
```
docker ps -l
```

Um **n zuletzt erstellte Container** anzuzeigen (einschließlich aller States), verwendest du den folgenden Befehl:

```
docker ps -n=1
```

Verwende den folgenden Befehl, um die **Gesamtdateigrößen** anzuzeigen:

```
docker ps -s
```



Es gibt keinen Unterschied zwischen `docker ps` und `docker container ls`. Die neue Befehlsstruktur (`docker container <subcommand>`) wurde in Docker 1.13 hinzugefügt, um dem Anwender eine strukturiertere Anwendung der Befehlszeile zu ermöglichen.

Nach aktuellem Wissenstand gibt es noch kein offizielles Statement, dass die Unterstützung der Befehle aus der alten Version (wie `docker ps` und andere) eingestellt wird, obwohl es vernünftig wäre anzunehmen, dass dies irgendwann in der Zukunft geschehen könnte.

In der neuen Version von Docker werden Befehle aktualisiert, und einige Verwaltungsbefehle werden hinzugefügt:

```
docker container ls
```

Dies wird verwendet, um alle laufenden Container aufzulisten:

```
docker container ls -a
```

Alle **laufenden Container** auflisten (Beispielverwendung der **Filteroption** -f):

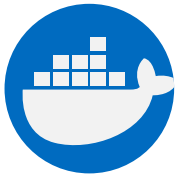
```
docker ps -a -f status=running
```

So listest du alle ausgeführten und angehaltenen Container auf und zeigst nur ihre **Container-ID** an:

```
docker ps -aq
```

Und dann, wenn du sie alle entfernen willst:

```
docker rm $(docker ps -aq)
```



Network

Erstelle zwei Container:

```
docker run -d --name web1  -p 8001:80 eboraas/apache-php

docker run -d --name web2  -p 8002:80 eboraas/apache-php
```

Hinweis: Es ist wichtig, explizit einen Namen mit **--name** für die Container anzugeben, da die zufällig von Docker vergebenen Namen manchmal nicht funktionieren.

Erstelle dann ein neues Netzwerk:

```
docker network create mein_netz

docker network ls
```

Verbinde danach deine Container mit dem Netzwerk:

```
docker network connect mein_netz web1

docker network connect mein_netz web2
```

Überprüfe, ob deine Container Teil des Netzwerks sind:

```
docker network inspect mein_netz
```

Überprüfe die Verbindung:

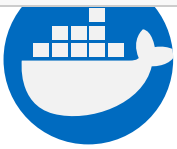
```
docker exec -it web1 ping web2

docker run --name web3 -it -p 8003:80 eboraas/apache-php bash


docker network connect mein_netz web3

docker network inspect mein_netz

ping <IP>
```



Löschen der Container und Images

Und um alle Docker-Container zu entfernen:

```
docker rm $(docker ps -aq)
```

oder:

```
docker rm -f $(docker ps -a -q)
```

Alle Images entfernen

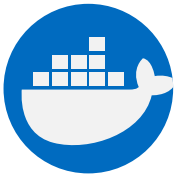
Alle Docker-Images aus einem System können durch Hinzufügen von **-a** zum Docker-**Images**-Befehl aufgelistet werden. Wenn du sicher bist, dass du alle löschen möchtest, kannst du das Flag **-q** hinzufügen, um die Image-ID an docker **rmi** zu übergeben:

Auflisten:

```
docker images -a
```

Löschen:

```
docker rmi $(docker images -a -q)
```



Löschen aller nicht verwendeten Images, Container, Volumes und Netzwerke

Docker stellt einen einzigen Befehl bereit, mit dem alle Ressourcen (Images, Container, Volumes und Netzwerke) bereinigt werden, die lose rumliegen (nicht mit einem Container verknüpft):

```
docker system prune
```

Um zusätzlich alle angehaltenen Container und alle nicht verwendeten Images (nicht nur Bilder ohne Verknüpfung) zu entfernen, füge dem Befehl das Flag -a hinzu:

```
docker system prune -a
```

Docker Logs

Starte einen Container:

```
docker run -it --rm --name logs_cont ubuntu
```

Erzeuge im Container einen Log-Eintrag:

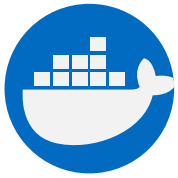
```
# echo “Hallo”
```

Schaue dir die Logs des Containers an:

```
docker logs logs_cont
```

Der Befehl `docker logs -f` wird weiterhin neue Lognachrichten von dem Container STDOUT und STDERR streamen:

```
docker logs -f logs_cont
```



Praxisprojekt PySpark

Wenn Docker installiert ist, kannst du die folgenden Befehle ausführen, um einen Docker Container lokal zu erstellen:

```
docker create --name pyspark -p 8888:8888 jupyter/pyspark-notebook
```

Danach kannst du die Übungshefte des Kurses in den Docker Container kopieren:

```
cd
docker cp Downloads pyspark:/home/jovyan/work
```

Jetzt kannst du den Container starten:

```
docker start -a pyspark
```

Kopiere den localhost Pfad in deinen Browser, um Jupyter zu öffnen:

```
http://127.0.0.1:8888/?token=Passwort_des_tokens
```

Falls du den Container stoppen möchtest und zu einem späteren Zeitpunkt fortsetzten, dann kannst du diesen einfach beenden:

```
docker stop pyspark
```

Gib im Jupyter Notebook die folgenden Befehle ein, um etwas zu berechnen und um zu überprüfen ob Spark richtig funktioniert:

```
import pyspark

sc = pyspark.SparkContext('local[*]')
```

```
rdd = sc.parallelize(range(300))

rdd.takeSample(False, 3)
```

Viel Spaß mit PySpark! :-)