

Einführung

Docker & Container

für Online-Werkzeuge und Entwicklungsumgebungen

Die Begriffe Docker und Container werden häufig verwendet. Docker ist hierbei wohl der geläufigere Begriff. Sie werden durchaus auch zusammen oder synonym als Docker-Container verwendet. Weshalb dies jedoch nicht richtig ist, soll im Folgenden erläutert werden. Um erklären zu können was Docker ist, muss zunächst ein Verständnis dafür geschaffen werden, was ein Container ist.

Container

„A container is a standard unit of software delivery that allows engineering teams to ship software reliably and automatically.“ [1]

Aus diesem Satz lassen sich mehrere Eigenschaften eines Containers ableiten:
Ein Container ...

- wird dazu verwendet Software auszuliefern.
- unterliegt gewissen Standards.
- erhöht Zuverlässigkeit.
- ermöglicht Automatisierung.

Die einzelnen Punkte lassen sich noch besser verstehen, wenn man einen Blick auf die technischen Details wirft, die einen Container ausmachen. Ein Container setzt sich aus drei ihm zugrundeliegenden Technologien zusammen, den Namespaces[4], Control Groups[6] und Union-File-Systems[7]. [3] Zusammen ergeben sie einen Container, dessen Format von der OCI¹ definiert wird. Dies dient dazu, dass verschiedene Werkzeuge und die Container zu einander kompatibel sind. Namespaces und Control Groups sind beide Feature des Linux-Kernels, wurden jedoch unabhängig voneinander entwickelt und sind nicht nur für Container verwendbar. Machen diese aber erst möglich.

Namespaces [8] Namespaces sind ein Feature des Linux-Kernels, welches die Ressourcen des Kernels aufteilt.

„A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes

¹ „Die Open-Container-Initiative ist ein Projekt der Linux-Foundation, um offene Standards für Container zu definieren.“ [2]

that are members of the namespace, but are invisible to other processes. One use of namespaces is to implement containers.“ [4]

Ein Kernel-Namespace verpackt eine globale Systemressource in eine Abstraktion, diese lässt die Ressource innerhalb des Namespaces so aussehen, als hätte der Prozess seine eigene isolierte Instanz der globalen Ressource. Änderungen die an der globalen Ressource vorgenommen werden sind für andere Prozesse sichtbar, die Teil des Namespace sind. Für andere Prozesse sind diese Änderungen aber nicht sichtbar.

Folgende Arten von Systemressourcen können von Namespaces abstrahiert werden:

- Cgroup - Cgroup root directory
- IPC - System V IPC, POSIX message queues
- Network [5]- Network devices, stacks, ports, etc.
- Mount² - Mount points
- PID - Process Identifiers ()
- Time - Boot and monotonic clocks
- User - User and group IDs
- UTS - Hostname and NIS, domain name

Der PID-Namespace bietet Isolation für die Allokation von PIDs, Listen von Prozessen und deren Details. Ein neuer Namespace ist isoliert von anderen *Geschwistern*, jedoch nicht von seinem „Eltern“-Namespace worin alle Prozesse des *Kindnamespaces* sichtbar sind, jedoch mit anderen PIDs.

Mit dem Network-Namespace werden sowohl physische als auch virtuelle Netzwerkgeräte, so wie iptables und firewall routing tables isoliert. Die Network-Namespaces können untereinander mit virtuellen Ethernet Geräten (veth) verbunden werden.

Namespaces können mit dem Programm *unshare*, dem gleichnamigen syscall oder dem syscall *clone* und der entsprechenden Flag erzeugt werden.

Cgroups Control Groups (cgroups) sind ebenfalls ein Kernel-Feature. Es dient dazu die Nutzung von Ressourcen (CPU, RAM, disk I/O, etc.) für eine Gruppe von Prozessen zu limitieren und isolieren. [9] Darüber hinaus können mit Cgroups Prozesse hierarchisch organisiert werden, sodass eine Baumstruktur an Prozessen entsteht, wobei jeder Prozess nur einer Cgroup angehört. Somit gehören auch alle Threads eines Prozesses zu einer Cgroup. [6] Sie erlauben außerdem das Messen der benutzten Ressourcen.

Eine Cgroup wird von einem Verzeichnis repräsentiert, das die Dateien enthält, die die Cgroup beschreiben. Dazu gehört eine Liste von Tasks (PID), die zu der Cgroup gehören. Sowie Cgroup.procs eine Liste von Thread-Gruppen-IDs.

Weiterführender Artikel auf LWN.net zu „Process containers“[10]

Union-File-Systems Ein Union-File-System stellt ein Dateisystem dar, indem es Verzeichnisse und Dateien in Branches gruppiert. Diese Branches können übereinander gestapelt werden, sodass mehrere Layer entstehen. Auf diese Art werden Images für Container erstellt. Teilen Images für Container die selbe Basis (die selben Layer), müssen diese nicht neu angelegt werden, sondern es reicht diese als Referenz anzugeben.

²https://man7.org/linux/man-pages/man7/mount_namespaces.7.html

Als unterster Layer für einen Container dient ein bootfs, das einem typischen Linux boot-Dateisystem ähnelt. Der darauf folgende Layer ähnelt einem root-Dateisystem. Für den Einsatz mit Containern wird dabei das Copy-on-Write Prinzip verwendet. Das bedeutet, dass beim Start eines Containers keine Dateien geladen oder kopiert werden müssen. Soll eine Datei geändert werden, dann wird diese Datei kopiert und die Änderung in der Kopie vorgenommen. Hierzu werden leere read-write Layer für jeden Layer des Images angelegt, die Änderungen finden in diesen Layern statt. Die Kopie verdeckt anschließend die Referenz in dem Image. [11] Bereits existierende Images, können daher wiederverwendet werden, da sie bei Verwendung nicht verändert werden.

Die Images für Container können sehr klein sein, da sich mehrere Images das selbe Base-Image (z.B. mit einem Debian Betriebssystem) teilen. Nur die Änderungen und Ergänzungen für das neuen Image benötigen zusätzlichen Speicher.[12]

Werden alle drei Technologien angewendet, erhält man das, was man einen Container nennt. Kurz zusammengefasst kann man einen Container als einen Prozess oder eine Gruppe von Prozessen bezeichnen, die in einer genau definierten Umgebung (Namespaces) ausgeführt werden. Die Umgebung wird durch das zugrundeliegende Image definiert und auf die Art und Weise wie der Container gestartet wird.

Docker

Docker ist eine OpenSource Plattform, deren Technik auf GitHub³ unter der Apache-2.0 Lizenz veröffentlicht wird. Die Technik ermöglicht es Programme zu entwickeln, verbreiten und auszuführen, und gleichzeitig von der Infrastruktur zu separieren. Das ermöglicht das schnelle Ausliefern von Software. [13]

Die Basis für die von Docker eingesetzte Technik bieten die Eingangs beschriebenen Container. Diese bieten Docker im Vergleich zu virtuellen Maschinen die Vorteile, dass kein Hypervisor benötigt wird, keine Virtualisierung stattfindet sonder direkt der Kernel verwendet wird, es können sogar Container innerhalb von virtuellen Maschinen ausgeführt werden. Dadurch sind Container deutlich leichtgewichtiger, als virtuelle Maschinen, sodass viele Container auf einem Host parallel ausgeführt werden können.

Ein Container kann, als Distributionseinheit für Tests und Auslieferung dienen. Hierbei spielt es keine Rolle, ob das Ziel der lokale Desktop zur Entwicklung oder ein Rechenzentrum, um mit der Anwendung in den produktiv Betrieb zu gehen, ist.

Docker Engine

Auf der untersten Ebene arbeitet der Docker-Daemon (*dockerd*), dies ist ein durchlaufender Prozess (Daemon), welcher das Erzeugen und Ausführen von Containern übernimmt. Dazu gehören des Weiteren das Managen von Images, Netzwerken und Volumen. Der Docker-Daemon ist außerdem in der Lage mit anderen dockerd Instanzen zu kommunizieren, um verteilte Docker-Dienste zu managen.

Über eine Schnittstelle, die *dockerd* zur Verfügung stellt, kann ein Docker-Client (z.B. der Befehl *docker*) mit dem Daemon interagieren. Die Interaktion kann dabei, sowohl lokal als auch über das Netzwerk stattfinden (UNIX-Sockets oder REST API).

³<https://github.com/docker>

Docker bietet darüber hinaus auch eine Registry, den Docker-Hub, für Images an. Die ist eine öffentlich zugängliche Plattform, auf der für jeden zugänglich Images bereitgestellt werden können. Docker ist standardmäßig so eingestellt, dass dort nach Images für Container gesucht wird. Es gibt aber auch Möglichkeiten eine eigene Registry zu betreiben, falls diese nicht öffentlich sein soll. Mit dem Befehl *docker pull* werden Images von der Registry abgerufen und heruntergeladen.

Bei der Verwendung von Docker interagiert man hauptsächlich mit zwei Arten von Objekten, den bereits erwähnten Images und Containern. Ein Image ist ein schreibgeschütztes Template, das die Instruktionen zum Erstellen eines Containers enthält. Da die Images auf Union-File-Systems basieren, können Images auch aufeinander aufbauen.

Zum Beispiel ist es möglich, als Grundlage ein Ubuntu-Image zu benutzen und darauf einen Apache-Webserver, eine eigene Anwendung und Konfigurationsdetails, die für die Anwendung notwendig sind, zu installieren. Es ist sowohl möglich, eigene Images zu definieren und zu verwenden, als auch den bereits erwähnten Docker-Hub zu nutzen. Um ein eigenes Image zu erstellen, muss ein *Dockerfile* angelegt werden, in dem in einer einfachen Syntax die Schritte definiert werden, die für das Erzeugen des Images notwendig sind.

FROM Hiermit lässt sich ein Image angeben, auf dem aufgebaut werden soll.

COPY Mit diesem Befehl können Verzeichnisse und Dateien in das zu erzeugende Image kopiert werden.

RUN Ausführen von Befehlen beim Erstellen des Images, z.B. aufrufen des Paketmanagers zum Installieren von Paketen oder ausführen von Programmen und etwas innerhalb des Images zu konfigurieren, erzeugen oder kompilieren.

EXPOSE Öffnet den angegebenen Port, sodass ein Service der innerhalb des Containers auf diesem Port läuft erreicht werden kann.

CMD Mit dieser Anweisung kann ein Befehl definiert werden, der wenn das Image als Container ausgeführt wird gestartet werden soll.

Mit jedem dieser Befehle im Dockerfile wird ein neuer Layer des Union-File-System erzeugt. Der Befehl *docker build* gibt die Anweisung an den Docker-Daemon das Image in dem angegebenen Dockerfile zu bauen. Nimmt man eine Änderung an dem Dockerfile vor weist *docker dockerd* an das Image erneut zu bauen, dann werden nur die Layer des Images neu gebaut, die sich geändert haben.

Die zweite wichtige Art von Objekt bei der Verwendung von Docker sind die eigentlichen Container. Ein Container ist eine ausführbare Instanz eines Images. Da ein Image nur ein Template ist, können mit ein und dem selben Image mehrere Container erstellt werden. Da ein Image schreibgeschützt ist, sind Änderungen im Container die während er ausgeführt wird nicht permanent. Wird ein Container gestoppt, dann sind alle Änderungen aufgehoben.

Sollen die Änderungen nicht verloren gehen, gibt es zwei Möglichkeiten. Zum einen ist es möglich, den Zustand eines laufenden Containers in einem neuen Image zu speichern, dieses Image ist dann wiederum schreibgeschützt und es können neue Container auf Basis dieses Images erzeugt werden. Zum anderen ist es möglich mit Docker Volumes zu erzeugen. Ein Volume kann innerhalb des Container gemountet werden und die Änderungen die dort vorgenommen werden bleiben in dem Volume erhalten. Über den Docker Client

ist es auch möglich den Daemon anzuweisen, dass ein Volume in mehreren Containern gemountet werden soll, sodass sich mehrere Container das Volume teilen und es für alle sichtbar ist. Hierbei kann definiert werden ob das Volume nur lesbar gemountet werden soll oder ob auch Schreibrechte genehmigt werden. Alternativ ist es möglich ein Verzeichnis des Hostsystems in den Container zu mounten, sodass dort z.B. Konfigurationen abgelegt und geändert werden können, sodass ein Zugriff sowohl vom Container als auch vom Host-System aus möglich ist.

Standardmäßig ist ein Container gut isoliert, zum Beispiel ist das Dateisystem des Hosts nicht eingebunden und zugänglich. Auch der Netzwerkzugriff ist abgegränzt zum Hostsystem. Es ist jedoch ähnlich wie beim Dateisystem möglich, die Container über separat mit dem Docker-Daemon angelegte Netzwerk-Namespaces miteinander zu verbinden, zum Beispiel einen Container in dem eine Anwendung ausgeführt mit einem weiteren Datenbank Container. Für andere Container die ausgeführt werden aber nicht diesem Netzwerk zugewiesen wurden, ist dann zum Beispiel der Datenbank Container nicht erreichbar.

Docker-Client

Für die Interaktion mit Containern steht ein Auswahl von Befehlen zur Verfügung. Kleine Übersicht:

docker build Build a docker image, specified by a Dockerfile

docker pull Pull the given image from the DockerHub

docker run Start a container based on the specified image

docker ps Show the currently running containers

docker exec Run a command inside the context/namespace of the container (see also *nsenter*)

chroot & mount Namespace

Um eine Shell innerhalb eines Containers zu öffnen kann man zum Beispiel den Befehl *docker exec -it container_name* verwenden. Schaut man sich dann innerhalb des Containers im Dateisystem um stellt man fest, dass es sich dabei nicht um das Dateisystem des Hosts handelt. Jedoch gibt es keine Möglichkeit wieder auf das Dateisystem des Hosts zuzugreifen, im Gegensatz zu einer reinen chroot Umgebung, denn der Container ist isoliert.

Dies wird über die Verwendung des Mount-Namespaces erreicht. Als erstes wird ein neuer Prozess geklont, welcher als init Prozess verwendet wird. Darin wird das Dateisystem vorbereitet, zum Beispiel indem mit `pivot_root()`⁴ und `chroot()`⁵ das Root-Verzeichnis gewechselt wird. Anschließend können in dem Prozess alle Einhängpunkte ausgehangen werden, die nicht mehr benötigt werden, sodass nur noch das Root-Verzeichnis des Containers übrig ist. Die Änderungen haben dabei nur auf den Namespace effekt, in dem der Init-Prozess ausgeführt wird. Zum Abschluss wird `unshare()`⁶ mit der Flag für den Mount-Namespace aufgerufen, dadurch wird der Prozess in einen neuen Namespace gebracht. Dies führt dazu, dass die vorherigen Änderungen nicht wieder rückgängig zu

⁴https://man7.org/linux/man-pages/man2/pivot_root.2.html

⁵<https://www.man7.org/linux/man-pages/man1/chroot.1.html>

⁶<https://man7.org/linux/man-pages/man2/unshare.2.html>

machen sind und der Container nun isoliert ist. Jetzt kann der eigentliche Prozess gestartet werden, für den der Container gedacht ist, welcher sich jetzt in den fertig konfigurierten Namespaces befindet.

Zur Konfiguration der Namespaces setzt der Docker-Daemon runc [14] ein. Dies ist eine Go geschriebene Software die für das Ausführen von Containern konform zur OCI-Spezifikation zuständig ist.

Beispiele

Ausbrechen aus einer `chroot` Umgebung.

```
#include <unistd.h>
#define DIR "xxx"
int main() {
    int i;
    mkdir(DIR, 0755);
    chroot(DIR);
    for (i = 0; i < 1024; i++) chdir("../");
    chroot(".");
    execl("/bin/sh", "-i", NULL);
}
```

Sandboxing durch die Verwendung von `chroot` und Namespaces.

```
unshare -muinp -f --mount-proc=./wheezy_chroot/proc \
chroot ./wheezy_chroot /bin/bash
```

Ein Beispiel in C das veranschaulicht, wie ein Prozess (hier die Funktion `child_fn`) innerhalb eines neuen Netzwerk Namespaces ausgeführt werden kann.

```
// compile with
// gcc -Wall -O3 -o namespace_isolation namespace_isolation.c
// found at:
// https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces
//
```

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

#define CHILD_STACK_SIZE 1048576
static char child_stack[CHILD_STACK_SIZE];
```

```
static int child_fn() {
    // Calling unshare() from inside the init process
    // lets you create a new namespace
    // after a new process has been spawned.
    unshare(CLONE_NEWNET);
}
```

```

printf("New_‘net’_Namespace:\n");
int ret = system("ip_link");
printf("\n\n");
return ret;
}

int main() {
printf("Original_‘net’_Namespace:\n");
int ret = system("ip_link");
printf("\n\n");

pid_t child_pid = clone(child_fn, \
    child_stack + CHILD_STACK_SIZE, \
    CLONE_NEWPID | CLONE_NEWNS | SIGCHLD, NULL);

waitpid(child_pid, NULL, 0);
return ret;
}

```

Literatur

Die Online Quellen beziehen sich auf den Stand im Mai 2020.

- [1] From Docker to OCI: What is a container? Padok - Busser Arthur
<https://www.padok.fr/en/blog/container-docker-oci>
- [2] Wikipedia: *Open Container Initiative (OCI)*
https://en.wikipedia.org/wiki/Open_Container_Initiative
- [3] Docker Docs: *The underlying technology*
<https://docs.docker.com/get-started/overview/#the-underlying-technology>
- [4] Linux Programmers's Manual: *namespaces - overview of Linux namespaces*
<http://man7.org/linux/man-pages/man7/namespaces.7.html>
- [5] Linux manual page: *network_namespaces(7)*
https://www.man7.org/linux/man-pages/man7/network_namespaces.7.html
- [6] Kernel: *Control Group v2*
<https://www.kernel.org/doc/Documentation/cgroup-v2.txt>
- [7] Unionfs: *A Stackable Unification File System*
<https://unionfs.filesystems.org/>
- [8] Wikipedia: *Linux namespaces*
https://en.wikipedia.org/wiki/Linux_namespaces
- [9] Wikipedia: *cgroups*
<https://en.wikipedia.org/wiki/Cgroups>
- [10] LWN.net: *Process containers*
<https://lwn.net/Articles/236038/>
- [11] washraf: *Union File Systems*
https://washraf.gitbooks.io/the-docker-ecosystem/content/Chapter%201/Section%203/union_file_system.html
- [12] Wikipedia: *UnionFS*
<https://en.wikipedia.org/wiki/UnionFS>
- [13] docker docs: *The Docker platform*
<https://docs.docker.com/get-started/overview>
- [14] GitHub: *opencontainers/runc*
<https://github.com/opencontainers/runc>

Weitere Quellen

- Introduction into docker images
<https://jfrog.com/knowledge-base/a-beginners-guide-to-understanding-and-building-docker-image>
- Understanding containerization by recreating docker
<https://itnext.io/linux-container-from-scratch-339c3ba0411d>

- Playing with namespaces
<https://pulsesecurity.co.nz/articles/docker-rootkits>
- Containers namespaces cgroups and some filesystem magic
<https://www.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesyste>
- Deep dive into docker storage drivers
<http://jpetazzo.github.io/assets/2015-07-01-deep-dive-into-docker-storage-drivers.html#83>
- Isolation with namespaces
<https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces>
- Linus Torvalds - chroot() & pivot_root()
https://yarchive.net/comp/linux/pivot_root.html
- Jürgen Brunk, Matthias Albert und Nils Magnus - Docker: die Linux-Basics unter der Container-Haube
<https://entwickler.de/online/besuch-im-docker-maschinenraum-126456.html>

Copyright © 2020 Armin Co

Lizenz: CC-by-sa (Version 3.0) oder GNU GPL (Version 3 oder höher)

Sie können diesen Text einschließlich \LaTeX -Quelltext herunterladen unter:

<https://gitlab.cvh-server.de/aco/ow>