UNIVERSITÄT HAMBURG
FAKULTÄT FÜR MATHEMATIK, INFORMATIK
UND NATURWISSENSCHAFTEN

Seminararbeit

Docker und Paas

Julian Tiemann

3tiemann@informatik.uni-hamburg.de
Studiengang Informatik
Matr.-Nr. 6542232

Inhaltsverzeichnis

1 Einleitung

Vor einiger Zeit wurde Software primär mit dem Hintergrund entwickelt später auf einem einzelnen Rechner zu laufen, sodass man nicht mit vielen Seiteneffekten durch eine andere Konfiguration auf Software- oder Hardware-Ebene zu rechnen hatte. Heute arbeiten viele Entwickler gleichzeitig auf verschiedensten Geräten an einer einzelnen Webanwendung oder einem Webservice. Es gibt eine unendliche Anzahl von Konfigurationen durch verschiedene Hardware-Konfigurationen, verschiedene Betriebssysteme oder Versionen oder auch installierte Programme und Dienste. All diese Unterschiede können die entwickelte Anwendung bereits im Entwicklungsstadium auf unterschiedliche Weise beeinflussen und für unterschiedliches Verhalten sorgen. Wird die Anwendung später ausgerollt und auf einem Server ausgerollt, muss sie wiederum in einer anderen Umgebung arbeiten. Das Ziel ist es deswegen die Software nicht von der Plattform auf der sie ausgeführt wird abhängig zu machen und sie immer in der gleichen Umgebung laufen zu lassen. Damit kann die Software auf beliebig vielen System ausgeführt werden mit gleichzeitiger Sicherheit, dass sie sich auf allen Systemen gleich verhalten wird. Um dies zu ermöglichen kann die Software entweder in einer virtuellen Maschine, einem simulierten Rechner, oder einem Container, einer leichtgewichtigeren Form der Virtualisierung, ausgeführt werden. Im Folgenden werde ich auf die Arbeitsweise von Containern im Vergleich zu virtuellen Maschinen eingehen und im speziellen Docker, einer Software für eine einfachere Handhabung von Containern, vorstellen.

2 Grundlagen

2.1 Virtuelle Maschinen

Als virtuelle Maschine (kurz VM) wird in der Informatik die Nachbildung eines Rechnersystems bezeichnet. Die virtuelle Maschine bildet die Rechnerarchitektur eines real in Hardware existierenden oder hypothetischen Rechners nach [HJS07]. Den virtuellen Maschinen wird über den sogenannten Hypervisor oder auch Virtual Machine Monitor (VMM) ein komplettes System vorgespielt, sodass diese die zugeteilten Ressourcen für echte Hardware halten [RHAD].

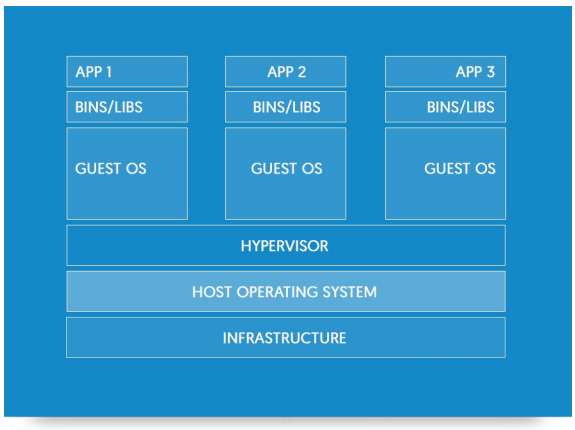


Abbildung 2.1: Mehrere virtuelle Maschinen auf einem Host-System. [doc17]

Da neben der Applikation selbst noch eine ganze Reihe weiterer Ressourcen benötigt werden, nämlich das Betriebssystem und seine ungenutzte Bibliotheken führt dies zu höherem Ressourcenverbrauch und großen Abhängigkeiten vom Betriebssystem. Zum Beispiel können bei einem Update des Betriebssystems für die Applikation nötige Bibliotheken verändert werden, die ein verändertes Verhalten hervorrufen. Dies bedeutet sowohl einen hohen Overhead an verbrauchten Systemressourcen, als auch einen hohen Wartungsaufwand.

Es wird meist zwischen Typ-1-Hypervisor und Typ-2-Hypervisor unterschieden. Inzwischen ließt man auch öfter von einem Typ-0-Hypervisor, einer neueren Technologie, auf die ich in dieser Arbeit nicht weiter eingehen werde.

Ein Typ-1-Hypervisor (native oder bare-metal) setzt direkt auf der Hardware auf und benötigt keine vorherige Betriebssystem-Installation, muss aber von der entsprechenden Hardware unterstützt werden.

Ein Typ-2-Hypervisor (hosted) setzt auf einem vollwertigen Betriebssystem, auf dem Hostsystem, auf und nutzt die Gerätetreiber des Betriebssystems, um auf die Hardware des Hostsystems zuzugreifen. [Wikc], [DRK14]

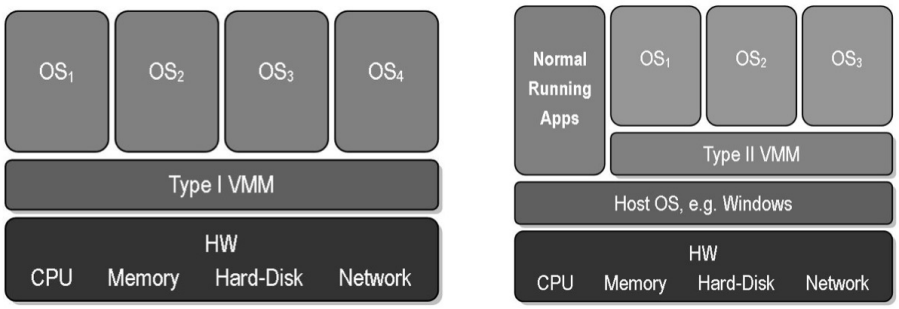


Abbildung 2.2: Typ-1-Hypervisor und Typ-2-Hypervisor. [Wikc]

2.2 Container

Ein Gegenentwurf zu den virtuellen Maschinen sind sogenannte Container, die das Ziel haben mit weniger Overhead den gleichen Grad an Isolation und Sicherheit zu liefern.

Anders als virtuelle Maschinen, teilen sich Container das Betriebssystem mit dem Hostsystem auf dem die Container gestartet werden. Befehle können also nahezu direkt auf der Hardware ausgeführt werden und müssen nicht erst, wie bei den virtuellen Maschinen den Umweg über den Hypervisor gehen, was einen Performancevorteil mit sich bringt, schränkt die Container aber auch dahingehend ein, dass nur Anwendungen mit dem gleichen Kernel wie der Host gestartet werden können. So können also nicht nativ Windows Container in einer Linux-Umgebung gestartet werden und umgekehrt [MKK15].
Wie auch virtuelle Maschinen werden Container auf Basis von Images erstellt. Diese Container können gestoppt und gestartet werden.
Es gibt verschiedene Implementationen von Containern, aber die meisten basieren zumindest zu großen Teilen auf den folgenden Linux-Kernel Features, die ab Version 3.8 zur Verfügung standen [Ros].

Chroot
Chroot steht für „change root“ und ist eine Funktion unter Unix-Systemen, um das Rootverzeichnis zu ändern. Sie wirkt sich nur auf den aktuellen Prozess und seine Kindpro-

zesse aus [Wika]. So können die Container vom Dateisystem des Host-Systems isolieret werden.

Cgroups

Cgroups (Control Groups) ermöglichen die Gruppierung von Prozessen, mit deren Hilfe das Betriebssystem den Gruppen Ressourcen zuweisen kann und verwalten kann. Control Groups können Ressourcen limitieren, priorisieren und isolieren. So kann man die Ressourcen für bestimmte Container limitieren und besser voneinander und dem Host-System isolieren [Pah15].

Kernel Namespaces

Mit Hilfe Kernel Namespaces der Kernel Namespaces lassen sich Prozesse oder Gruppierungen von Prozessen auf verschiedenen Ebenen voneinander isolieren. Es gibt verschiedene Namespaces für unterschiedliche Zwecke:

- pid um die Prozesse voneinander zu isolieren [blo17d]
- net für eigene Netzwerk-Konfigurationen und Adressen [blo17b]
- ipc für den Informations Austausch zwischen Containern [blo17a]
- mnt für eigene Mountpoints [blo17c]
- uts für die Zuweisung eigener Hostnames [blo17e]

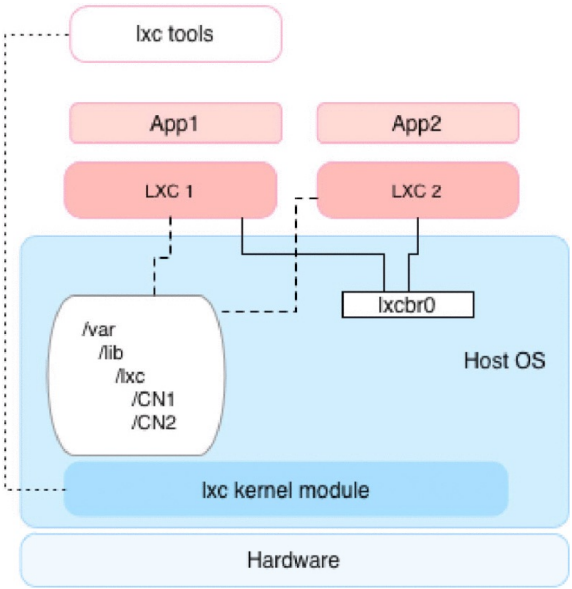


Abbildung 2.3: Containers [DRK14]

Es gibt verschiedene Ziele, die man mit einem Container erreichen will. Die zwei Hauptanwendungsfälle sind aber wohl OS-Container und Applikations-Container. Der Unterschied ist dabei aber kein technischer sondern mehr ein konzeptueller.
Ein OS-Container kann als virtuelle Maschine ohne Performance-Overhead betrachtet werden, die sich wie beschrieben den Kernel mit dem Host-System teilt. Die Container können, einmal gestartet, wie ein normales Betriebssystem bedient werden. Es können beliebig viele Container auf einem System laufen mit unterschiedlichsten Zielen. Zum Beispiel könnte man einen Container erstellen, auf dem man alle nötigen Services installiert, um eine Webanwendung laufen zu lassen. Lässt man den Container auf einem anderen Host-System laufen, sind wieder genau die gleichen Versionen der Programmiersprache, wie zB. Ruby oder PHP, der genutzten Datenbank oder dem Webserver aber auch von wichtigen System-Libraries, installiert. Genauso vorstellbar sind aber auch spezialisierte Container für Webserver und Datenbank, die über die Linux Bridge miteinander kommunizieren [Kar]. Die einzelnen Prozesse für die jeweiligen Services werden dabei in den jeweiligen Containern gestartet und nicht auf dem Host-System. So kann sichergestellt werden, dass man auf unterschiedlichen Host-Systemen die gleiche Webanwendung immer auf den gleichen Maschinen laufen lässt.

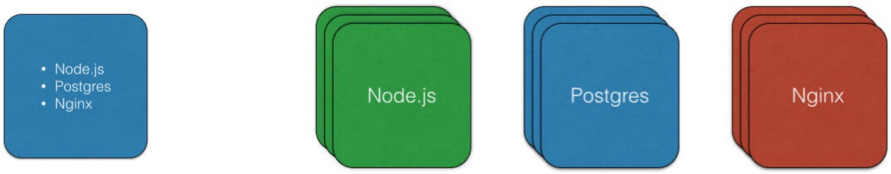


Abbildung 2.4: Ein OS-Container mit allen nötigen Servern vs. einzelne spezialisierte OS-Container

Im Gegensatz zu OS-Containern wird mit einem Applikations-Container das Ziel verfolgt, ein Paket mit allen nötigen Abhängigkeiten zu bauen, welches dann als einzelner Prozess gestartet werden kann. Applikations-Container bestehen oft aus sogenannten Layern, einzelnen Images die später beim Bauen des Images, das zum starten der Anwendung verwendet wird zusammengefasst werden. Diese entsprechen den einzelnen spezialisierten Containern in dem oben genannten Beispiel. Der Vorteil ist, das die einzelnen Services in dem Fall im gleichen Namespace ausgeführt werden und dementsprechend aufeinander zugreifen können. Nutzt man Microservices als Architekturmuster ist es zum Beispiel sinnvoll, die einzelnen Services in Containern laufen zu lassen und die jeweiligen Layer in den verschiedenen Containern wiederzuverwenden.

2.3 Daemon

Als Daemon bezeichnet man unter Unix oder unixartigen Systemen ein Programm, das im Hintergrund abläuft und bestimmte Dienste zur Verfügung stellt. Der Benutzer interagiert dabei nicht direkt mit dem Programm, sondern indirekt über Ereignisse auf dem

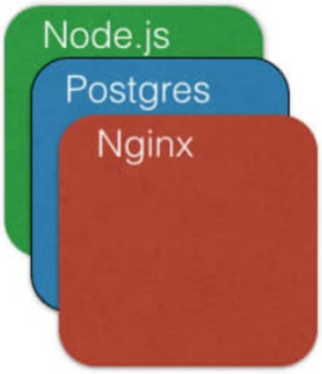


Abbildung 2.5: Ein Applikations-Container mit mehreren Layern

System. Das kann zum Beispiel eine Hardwareüberwachung sein, die überprüft ob ein Wechselmedium an das System angeschlossen wurden, aber auch ein Webserver, der auf Anfragen HTTP-Anfragen wartet [Wikb].

3 Container und VMs

Container werden oft als “lightweight VMs” bezeichnet, was naheliegend ist, da beide Technologien darauf ausgelegt sind ein Betriebssystem zu simulieren, um eine Applikation in einer isolierten Umgebung laufen zu lassen, um die Applikationen portabel zu halten und Schäden am Host-System zu verhindern [Colb]. Und doch haben beide Technologien wenig gemeinsam. Während im Falle der virtuellen Maschinen ein komplettes System mit Hardware, Betriebssystem und Applikation simuliert wird, reicht es für das Ausführen einer Applikation in einem Container Teile eines Betriebssystem zu simulieren. Ein weiterer großer Unterschied ist, dass VMs einen Zustand haben. Sie repräsentiert ein komplettes System mit Prozessen und Daten, das woanders wieder in genau diesem Zustand weiterlaufen kann. Erzeugt man eine weitere Instanz dieser Maschine ist es einfach nur eine genaue Kopie derselben Maschine. Container hingegen sind meist zustandslos.

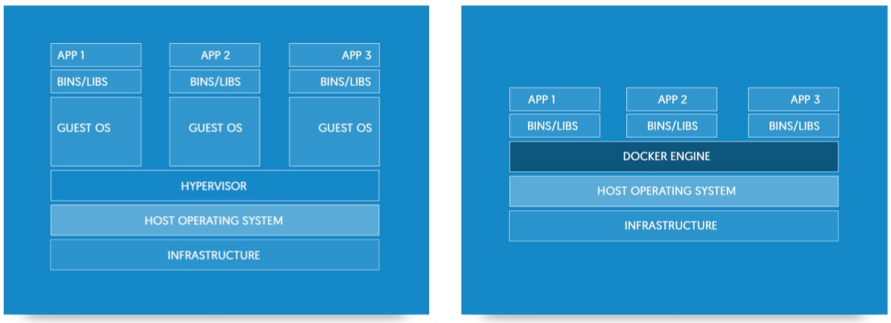


Abbildung 3.1: Verschiedene VMs und Container auf einem Host-System im Vergleich [doc17]

Es gibt keine Regeln dafür, wann eine Applikation innerhalb eines Containers und wann sie innerhalb einer VM laufen sollte. Das kommt auf die Applikation an und muss von Fall zu Fall betrachtet werden. Bei einer monolithischen Webanwendung, also einer, die alle nötigen Services und Komponenten, wie Frontend, Backend und die Datenbanken, beinhaltet, macht es Sinn diese in einer virtuellen Maschine zu starten. Besteht die Anwendung aus vielen kleinen spezialisierten Microservices, macht es Sinn diese in einzelnen Containern zu starten. Grundsätzlich schließen sich beide Technologien aber auch nicht aus, da Container auf allen Host-Systemen gestartet werden können, die auf dem gleichen Kernel aufbauen - also auch virtuellen Maschinen. Sollen Anwendungen die in Containern laufen beispielsweise mit älteren, in VMs laufenden Anwendungen zusam-

menarbeiten, bei denen klar ist, dass diese nicht noch einmal überarbeitet werden sollen , können diese einfach mit auf der VM gestartet werden. So können die Vorteile beider Technologien miteinander kombiniert werden [Cola].

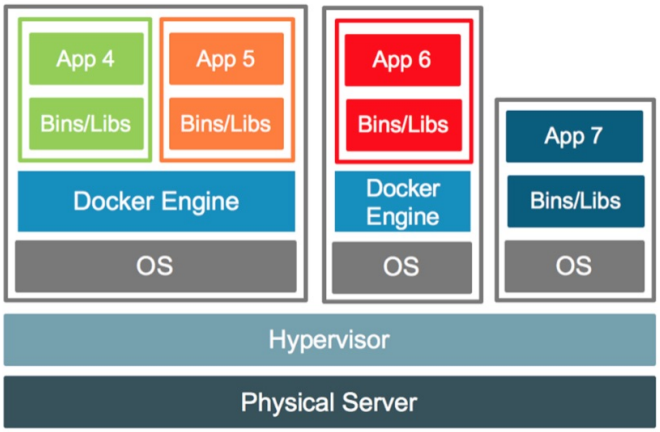


Abbildung 3.2: VMs und Docker Container kombinier [Cola]

4 Docker

Docker soll es Entwicklern vereinfachen, Container für Applikationen zu erstellen, zu verteilen und sie auf verschiedensten Systemen auszuführen. Es hilft dabei Applikations-Container zu erstellen [doc17], die später in einem einzelnen Prozess auf verschiedenen Systemen ausgeführt werden können. Die erste Version wurde im März 2013 von dotCloud veröffentlicht (jetzt Docker, Inc.) [Doca]. Docker basierte bis zu Version 0.9 [Doce] auf dem Container-Standard LXC und erweiterte diese vor allem durch eine einfachere Handhabung. So können Images durch einfache Konfigurationsdateien definiert werden und über die Docker API erstellt und verwaltet werden. Beim Starten eines Containers können diese mit anderen verbunden werden, um so die Kommunikation der eigentlich voneinander isolierten Container zu vereinfachen und über andere Tools können ganze Cluster von Docker-Containern verwaltet werden.

Über eine Docker-Registry ist es möglich Container Images zu versionieren und zu teilen. Die Docker-Registry kann wie Git für Container gesehen werden. Bei dem Bau neuer Container Images kann man so immer wieder auf sogenannte Base-Images zurückgreifen, die man in einer Docker-Registry findet und auf denen man seine eigenen Images aufbauen kann.

LXC ist dabei nur ein Tool um Container zu erstellen und zu starten und Docker vereinfacht die Verwaltung dieser Container. Mit "libcontainer" implementierte Docker Inc. eine eigenen Container-Standard, um die starken Abhängigkeiten von LXC aufzulösen. Daraus ging die "Open Container Initiative", die OCI (https://www.opencontainers.org/) und "runC" (http://runc.io/) hervor. Die OCI ist ein Zusammenschluss mehrerer großer Firmen, angeführt von Docker, mit dem Ziel, einen einheitlichen offenen Container Standard zu definieren. Anhand dieser Spezifikation können Container dann auf verschiedenen Systemen implementiert werden und von Systemen wie Docker einheitlich verwaltet werden. runC ist dabei die Implementation der Spezifikationen der OCI und ersetzt LXC.

4.1 Docker Engine

Die Docker Engine ist eine Client-Server Anwendung, die auf dem Host-System installiert wird und besteht aus drei Hauptbestandteilen:

- dem Docker Daemon (der Server), der für das Erstellen und Ausführen der Container zuständig ist
- der Docker Remote Api für die Kommunikation zwischen Client und Docker Daemon
- der CLI (Command Line Interface)

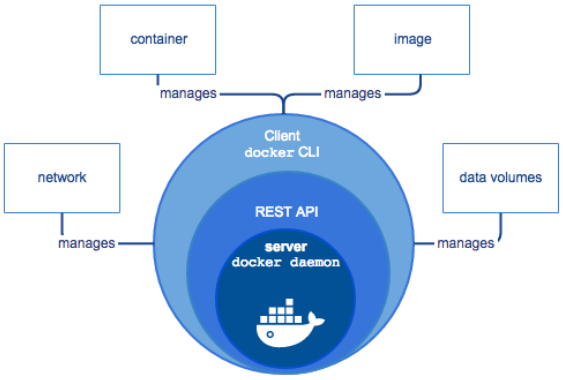


Abbildung 4.1: Die drei Hauptbestandteile der Docker-Engine [Docb]

Die Client-Server Architektur ermöglicht es einerseits Client und Daemon auf verschiedenen Hosts laufen zu lassen und so auch Container auf anderen Servern zu verwalten, andererseits kann der Daemon durch die API über einen beliebigen Client gesteuert werden. Das können andere Sprachen wie beispielsweise Python sein, für die eine vollständige Docker-Api-Implementation vorliegt oder auch grafische Benutzeroberflächen.

4.2 Images und Container

Ein Docker Container wird auf Basis eines Docker Images gebaut. Jedes Docker Image besteht aus mehreren Layern, einem Base Image und welchen, die das Image jeweils um Funktionen erweitern, die für das Ausführen einer Applikation wichtig sind. Die Layer liegen dabei in jeweils eigenen Verzeichnissen und werden über ein "unified view" zu einem einzelnen Image zusammengefasst. Ein "unified view" kann als virtuelles Verzeichnis gesehen werden, in dem die einzelnen Layer so übereinandergelegt werden, dass die verschiedenen Layer von oben als einzelnes Image gelesen werden können. Existiert in einem Layer die gleiche Datei wie in dem Base Image mit einer anderen Konfiguration, so verdeckt diese die Datei aus dem Base Image.

Die einzelnen Layer in einem Image sind nur lesbar, können also nicht mehr verändert werden. So können die Layer von beliebig vielen Images verwendet werden. Jeder

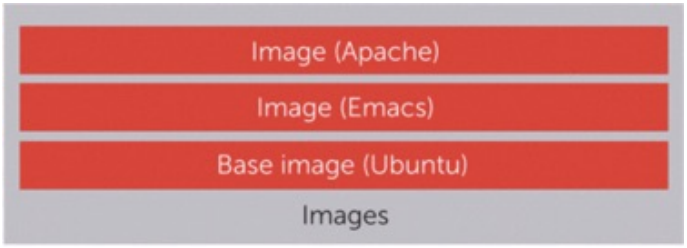


Abbildung 4.2: Ein Image bestehend aus mehreren übereinandergestapelten Layern [Pah15]

Layer hat einen eindeutigen Hash und kann gezielt von verschiedenen Images verwendet werden und es kann sichergestellt werden, dass der Layer immer genau gleich ist. Das bedeutet, dass der Layer nur ein einziges Mal im Dateisystem gespeichert werden muss. Über die "unified view" werden die Layer anschließend in den Images verwendet. So können die Layer von beliebig vielen Images verwendet werden bei konstantem Platzverbrauch. Jede Änderung an einem Image fügt dem Image einen Layer hinzu. Jeder Layer steht dabei für Unterschiede im Dateisystem im Bezug auf die übrigen Layer. Das können nur einzelne Konfigurationsdateien sein, sodass die Größe des Layers nur wenige Bytes beträgt, aber auch ganze Programmpakete, die deutlich mehr Platz benötigen.

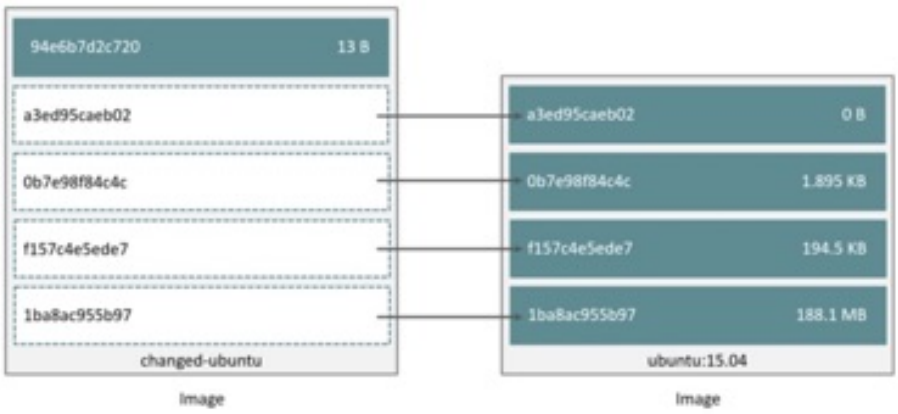


Abbildung 4.3: Ein Image mit neuem Layer, dass die ursprünglichen Layer mit verwendet [Docf]

Bei der Erstellung eines Containers aus einem Image wird nach dem gleichen Prinzip dem ursprünglichen Image nun ein "Container-Layer" hinzugefügt, der im Gegensatz zu den übrigen beschreibbar ist. Ist der Container gestartet, werden alle Dateiänderungen in den diesen Layer geschrieben. Dafür ist neben dem "Unified View" ist noch ein weiteres Verfahren, das "copy-on-write" Verfahren wichtig. Während die Container laufen, arbeiten beliebig viele Container mit den gleichen Original-Dateien aus dem jeweiligen Layern. Da diese nur gelesen werden können, können diese entsprechend nicht von

den jeweiligen Containern modifiziert werden. Deshalb werden diese Dateien, sobald sie verändert werden sollen aus dem jeweiligen Read-Only Layer in den Container-Layer kopiert, wo sie entsprechend bearbeitet werden können. Der Container, der die Datei bearbeitet hat arbeitet danach mit seiner Kopie im Container-Layer, während die übrigen Container weiter mit dem Original arbeiten. Dadurch können beliebig viele Container platzsparend zur gleichen Zeit laufen [Docf].

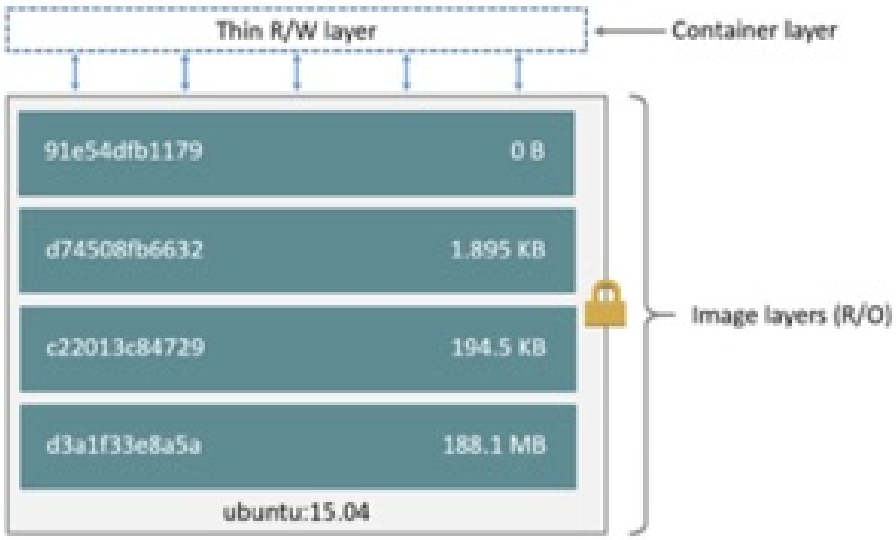


Abbildung 4.4: Ein Container, bestehend aus R/O Image Layern und R/W Container Layer [Docf]

Gesteuert werden sowohl der “Unified View” als auch “Copy-on-Write” von verschiedenen “Storage-Drivern”, die zwar unterschiedlich implementiert sind, aber auf dem gleichen Prinzip beruhen. Bei der Konfiguration von Docker, kann man zwischen verschieden Implementierungen wählen. Die bekanntesten sind wohl “auFS” und “overlayFS”.

Beim Löschen eines Containers werden auch alle Dateien, die beim Ausführen der Applikation erstellt wurden, gelöscht. Für das Persistieren von Daten gibt es die Möglichkeit, die Container mit sogenannten Data-Volumes zu verbinden.

Definiert werden die Images über Dockerfiles. Vereinfacht fügt jeder Befehl in der Datei dem Image ein neuen Layer hinzu. Über "docker build"wird das Image entsprechend des Dockerfiles gebaut. Jedes Image kann auf bereits vorhandenen Images aufbauen. Images können in einer Docker-Registry gespeichert werden und werden bei Bedarf bei der Erstellung des Images heruntergeladen. Dabei werden bereits vorhandene Layer im Dateisystem nicht erneut heruntergeladen. Das Beispiel baut ein Image basierend auf Ubuntu, das beim Starten des Containers “Hello world” ausgibt und in eine Datei schreibt.

Listing 4.1: src/Dockerfile1

```
FROM ubuntu:15.04
RUN echo "Hello world" > /tmp/newfile
```

Abbildung 4.5: Ein Dockerfile

4.3 Docker Daemon

Der Docker Daemon (dockerd[1]) ist der zentrale Bestandteil von Docker. Er ist die Schnittstelle zwischen der einfachen Handhabung von Docker und den Vorteilen Applikations-Container gegenüber normalen VMs bieten. Er setzt auf containerd[2] auf, einer Ausführungsumgebung(runtime) für Container, um runC Container nach dem OCI-Standard auszuführen und zu verwalten. containerd basiert auf Dockers altem Daemon. Er setzt auf einem niedrigeren Level an und wird von dem Docker Daemon erweitert. Wird der Docker Daemon über seine API angesprochen gibt dieser die vereinfachten Befehle übersetzt an containerd weiter [Docd].

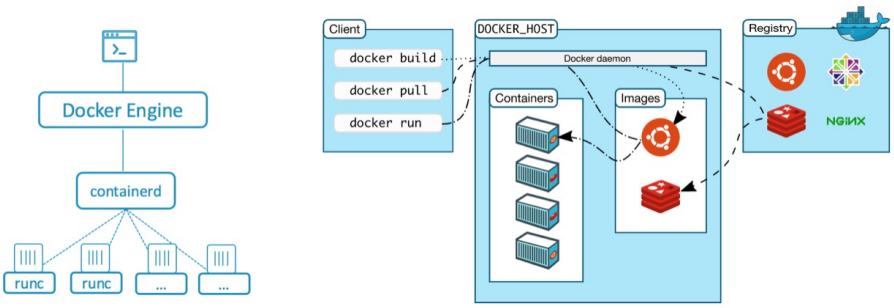


Abbildung 4.6: Docker Daemon

Wird zum Beispiel das Kommando `docker run -i -t ubuntu` ausgeführt, checkt die Docker Engine, ob das Ubuntu Image sich bereits auf dem System befindet. Ist das nicht der Fall wird das Image aus der Docker Registry gepullt und auf die Festplatte geladen. Daraus wird der Container erstellt, indem dem Image ein R/W Layer hinzugefügt wird. Danach wird das Netzwerk entsprechend konfiguriert, damit der Docker Container mit dem Host-System kommunizieren kann und dem Container wird eine IP-Adresse zugewiesen, über die der Container erreichbar ist. Zum Schluss werden die definierten Kommandos aus dem Dockerfile ausgeführt und man kann (durch das `-i` Flag) über das Terminal mit dem Container interagieren.

[1] https://docs.docker.com/engine/reference/commandline/dockerd/
[2] https://containerd.io/

4.4 Docker API

Über die Docker Remote API kann ein Client mit dem Docker Daemon ansprechen. Client und Daemon können sowohl auf dem gleichen Host als auch über verschiedene Hosts über die Http-Schnittstelle miteinander kommunizieren. Der Client muss dabei kein Benutzer sein, sondern können zum Beispiel Überwachungsprogramme sein. Unter anderem bietet die Docker API zum Beispiel einen Event-Stream, bei dem sich diese Programme registrieren und dementsprechend auf Zustandsänderungen reagieren können. Fällt zum Beispiel ein Container aus, können so automatisch neue Container erstellt und ausgeführt werden [Docc].

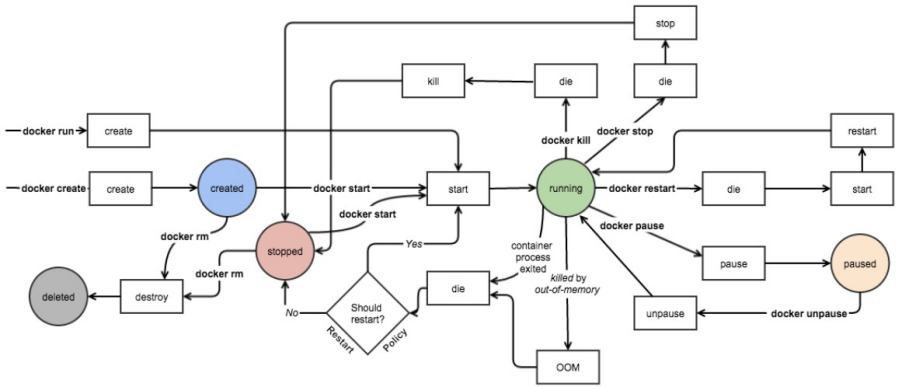


Abbildung 4.7: Die Zustände die ein Docker-Container besitzen kann und die dafür verantwortlichen Events [Docc]

5 Fallbeispiel

Listing 5.1: src/Dockerfile2

```
FROM ruby:2.3.3
RUN apt-get update -qq && apt-get install -y build-essential libpq-dev
    nodejs
RUN mkdir /myapp
WORKDIR /myapp
ADD Gemfile /myapp/Gemfile
ADD Gemfile.lock /myapp/Gemfile.lock
RUN bundle install
ADD . /myapp
ENTRYPOINT ["/myapp/docker-entrypoint.sh"]
EXPOSE 3000
```

Abbildung 5.1: Ein Dockerfile

Workdir wird mit in container geladen und kann ausgeführt werden, lokal volume sodass man app entwickeln kann

Listing 5.2: src/docker-compose.dev.yml

```
version: '2'
services:
    db:
      image: postgres
      volumes:
        - ./postgres:/var/lib/postgresql/data
    web:
      image: kddc/dbaas_web:latest
      build: .
      command: bundle exec rails s -p ${PORT} -b '0.0.0.0'
      volumes:
        - .:/myapp
      ports:
        - "${PORT}:${PORT}"
      depends_on:
        - db
```

Abbildung 5.2: Docker-Compose Development

Baut images db und web, db aus iamge, web nach Anleitung aus Dockerfile. docker-compose up baut images und spawnt container

docker-compose build, docker push kddc/dbaas$_{w}$*eb*, *dockerpushkddc*/*dbaas*$_{d}$*bpushtneustecontainerindo*

Listing 5.3: src/docker-compose.prod.yml

```
version: '2'
services:
  db:
    image: postgres
    volumes:
      - ./postgres:/var/lib/postgresql/data
  web:
    image: kddc/dbaas_web
    command: bundle exec rails s -p 80 -b '0.0.0.0'
    ports:
      - "80:80"
    depends_on:
      - db
```

Abbildung 5.3: Docker-Compose Production

Auf Server anderes docker-compose script, zieht sich images startet container port 80

6 Fazit

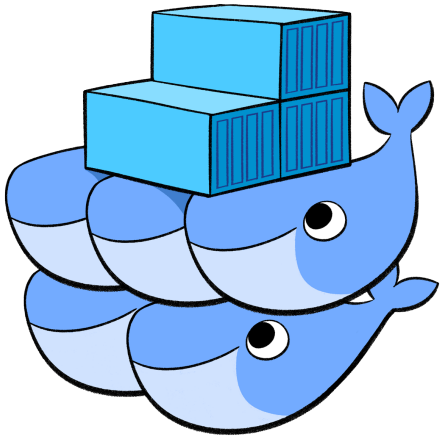
Die Einsatzmöglichkeiten von Containern sind vielfältig. Sie erleichtern die Portierbarkeit von Anwendungen, sparen Platz und schonen die Ressourcen. Sie können Entwicklern dabei helfen Fehlverhalten in ihrer Anwendung aufgrund unterschiedlicher Softwareversionen zu verhindern oder aber bei Bedarf in sehr kurzer Zeit beliebig viele Container zu einem Cluster von bestimmten Anwendungen oder Services zuzuschalten, um das System zu entlasten. Der einfachste Fall ist wohl eine einfache monolithische Webanwendung, die leicht transportiert und auf nahezu jedem System einfach ausgeführt werden kann. Jedoch laden Container dazu ein, eine Software in Microservices aufzuteilen, damit die jeweiligen Services möglichst isoliert von anderen Bestandteilen der Software arbeiten können und sich nicht etwa durch die Abhängigkeiten von den gleichen Bibliotheken in verschiedenen Versionen ausbremsen. Diese Kleinteiligkeit stellt Entwickler aber auch wieder vor neue Herausforderungen. Zwar sind die Container im einzelnen leicht portierbar und werden stabil in ihrer isolierten Umgebung laufen, aber sie müssen auch mit den anderen Services einer Anwendung verknüpft werden oder im entsprechenden Cluster einsortiert werden, um das volle Potential auszuschöpfen.

Docker-Swarm[1] ist ein internes Tool in der Docker-Engine das bei der Orchestrierung der Container hilf, also unter welchen Voraussetzungen welche Cluster hochskaliert werden, wann Container abgeschaltet werden oder dem Austausch von defekten Containern. Aber auch für diesen Zweck gibt es bereits Cloud Computing Services, über die das Verwalten von Containern bzw. Container-Clustern vereinfacht wird. Was Infrastructure as a Service (IaaS) für virtuelle Server und Platform as a Service (PaaS) für fertige Laufzeitumgebungen für Anwendungen ist, ist Container as a Service (CaaS) für Container. CaaS ist irgendwo zwischen IaaS und PaaS anzusiedeln, ähnelt aber vor allem IaaS, mit dem Unterschied, dass Container und nicht virtuelle Maschinen die vermieteten Ressourcen sind. Die CaaS Plattformen helfen dabei vor allem bei Orchestrierung der Container. Bekannte Anbieter sind Amazon EC2 Container Service[2] und Google Kubernetes[3].

Zusammenfassend lässt sich sagen, dass das Prinzip von Containern zwar schon länger bekannt ist, aber erst seit einigen Jahren erfährt die Technologie vermehrt Beachtung, da durch Unternehmen wie Docker endlich eine vereinfachte Schnittstelle für Entwick-

[1] https://www.docker.com/products/docker-swarm
[2] https://aws.amazon.com/de/ecs/
[3] https://kubernetes.io/

ler geschaffen wurde, um mit Containern zu arbeiten. Es gibt zwar noch einige kritische Stimmen, die vor allem Bedenken wegen der Sicherheit von Containern äußern, dennoch bergen sie vor allem viele Möglichkeiten für Entwickler ihre Anwendungen stabiler und flexibler zu gestalten.



Literaturverzeichnis

[blo17a] BLOG!, Yet another e.: *Introduction to Linux namespaces - IPC*. http://blog.yadutaf.fr/2013/12/28/introduction-to-linux-namespaces-part-2-ipc/. Version: 2017, Abruf: 2017-01-22

[blo17b] BLOG!, Yet another e.: *Introduction to Linux namespaces - NET*. http://blog.yadutaf.fr/2014/01/19/introduction-to-linux-namespaces-part-5-net/. Version: 2017, Abruf: 2017-01-22

[blo17c] BLOG!, Yet another e.: *Introduction to Linux namespaces - NS*. http://blog.yadutaf.fr/2014/01/12/introduction-to-linux-namespaces-part-4-ns-fs/. Version: 2017, Abruf: 2017-01-22

[blo17d] BLOG!, Yet another e.: *Introduction to Linux namespaces - PID*. http://blog.yadutaf.fr/2014/01/05/introduction-to-linux-namespaces-part-3-pid/. Version: 2017, Abruf: 2017-01-22

[blo17e] BLOG!, Yet another e.: *Introduction to Linux namespaces - UTS*. https://blog.yadutaf.fr/2013/12/22/introduction-to-linux-namespaces-part-1-uts/. Version: 2017, Abruf: 2017-01-22

[Cola] COLEMAN, Mike: *Containers and VMs together*. https://blog.docker.com/2016/04/containers-and-vms-together/, Abruf: 2017-01-22

[Colb] COLEMAN, Mike: *Containers are not VMs*. https://blog.docker.com/2016/03/containers-are-not-vms/, Abruf: 2017-01-22

[Doca] https://github.com/docker/docker/releases/upstream%2F0.1.1

[Docb] https://docs.docker.com/engine/understanding-docker/

[Docc] https://docs.docker.com/engine/reference/api/docker_remote_api/

[Docd] DOCKER: *Daemon*. https://docs.docker.com/v1.11/engine/reference/commandline/daemon/, Abruf: 2017-01-22

[Doce] DOCKER: *Docker Changelog*. https://github.com/docker/machine/blob/master/CHANGELOG.md, Abruf: 2017-01-22

[Docf] DOCKER: *Understanding Images and Containers*. https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/, Abruf: 2017-01-22

[doc17] https://www.docker.com/

[DRK14] DUA, R. ; RAJA, A. R. ; KAKADIA, D.: Virtualization vs Containerization to Support PaaS. In: *2014 IEEE International Conference on Cloud Engineering*, 2014, S. 610–614

[HJS07] HANS-JÜRGEN SIEGERT, Uwe B.: *Betriebssysteme*. Oldenbourg, 2007 (S. 270)

[Kar] KARLE, Akshay: *Operating System Containers vs. Application Containers*. https://blog.risingstack.com/operating-system-containers-vs-application-containers/, Abruf: 2017-01-22

[MKK15] MORABITO, R. ; KJÄLLMAN, J. ; KOMU, M.: Hypervisors vs. Lightweight Virtualization: A Performance Comparison. In: *2015 IEEE International Conference on Cloud Engineering*, 2015, S. 386–393

[Pah15] PAHL, C.: Containerization and the PaaS Cloud. In: *IEEE Cloud Computing* 2 (2015), May, Nr. 3, S. 24–31. http://dx.doi.org/10.1109/MCC.2015.51. – DOI 10.1109/MCC.2015.51. – ISSN 2325–6095

[RHAD] REMZI H. ARPACI-DUSSEAU, Andrea C. Arpaci-Dusseau: *Operating Systems: Three Easy Pieces*. http://pages.cs.wisc.edu/~remzi/OSTEP/vmm-intro.pdf, Abruf: 2017-01-22

[Ros] ROSEN, Rami: *Linux Containers and the Future Cloud*. haifux.org/lectures/320/netLec8_final.pdf, Abruf: 2017-01-22

[Wika] WIKIPEDIA: *Chroot*. https://de.wikipedia.org/wiki/Chroot

[Wikb] WIKIPEDIA: *Daemon*. https://de.wikipedia.org/wiki/Daemon, Abruf: 2017-01-22

[Wikc] WIKIPEDIA: *Hypervisor*. https://de.wikipedia.org/wiki/Hypervisor

[Wik16] WIKIPEDIA: *Wissenschaftliche Arbeit*. `https://de.wikipedia.org/w/index.php?title=Wissenschaftliche_Arbeit&oldid=156007167`. Version: 2016, Abruf: 2016-07-21