4 Grundlagen von Docker

In diesem Kapitel werden wir uns mit den grundlegenden Docker-Konzepten befassen. Wir beginnen mit einem Überblick über die Architektur und die Technologien, auf denen Docker aufbaut. Dann folgen tiefergehende Abschnitte zum Erstellen von Docker-Images, dem Verknüpfen von Containern und dem Umgang mit Daten in Volumes. Das Kapitel endet mit einem Überblick über die restlichen Docker-Befehle.

Da dieses Kapitel stark referenzlastig ist, können Sie auch nur die wichtigsten Punkte überfliegen, mit Kapitel 5 weitermachen und bei Bedarf zurückkehren.

4.1 Die Architektur von Docker

Um zu verstehen, wie Sie Docker am besten nutzen und warum es sich manchmal ungewöhnlich verhält, ist es gut, ein grobes Verständnis dafür zu haben, wie die Plattform unter der Motorhaube aufgebaut ist.

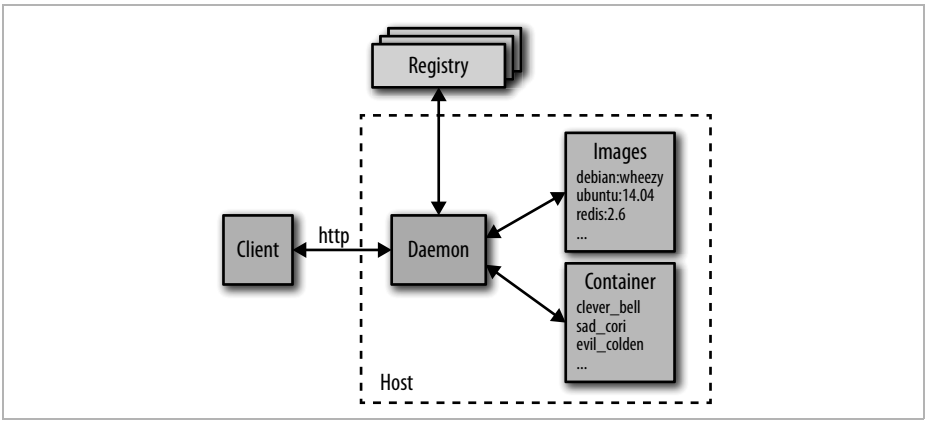

Abb. 4–1 *Übersicht über die wichtigsten Docker-Komponenten*

In Abbildung 4–1 sehen wir die wichtigsten Komponenten einer Docker-Installation:

- Im Zentrum steht der *Docker Daemon*, der für das Erstellen, Ausführen und Überwachen der Container zuständig ist. Aber auch das Bauen und Speichern von Images fallen in seinen Zuständigkeitsbereich. Sowohl Container als auch Images sehen Sie in der Abbildung rechts. Der Docker Daemon wird durch `docker daemon` gestartet, was normalerweise schon durch das Host-Betriebssystem geschieht.[1]
- Der Docker Client auf der linken Seite wird dazu genutzt, per HTTP REST- mit dem Docker Daemon zu kommunizieren. Standardmäßig geschieht das über ein Unix Domain Socket, aber es ist auch ein TCP-Socket möglich, um Remote Clients zu nutzen, oder ein Dateideskriptor für durch systemd verwaltete Sockets. Da die gesamte Kommunikation über HTTP abläuft, ist es einfach, sich mit entfernten Docker Daemons zu verbinden und Bindings an Programmiersprachen zu entwickeln. Es hat aber auch Auswirkungen darauf, wie Features implementiert werden – so ist zum Beispiel ein *Build Context* für Dockerfiles notwendig (siehe Abschnitt 4.2.1). Die für die Kommunikation mit dem Daemon verwendete API ist wohldefiniert und dokumentiert, so dass Entwickler Programme schreiben können, die direkt mit dem Daemon arbeiten, statt auf den Docker Client zurückgreifen zu müssen. Docker Client und Docker Daemon werden als ein Binary bereitgestellt.
- In Docker Registries werden Images abgelegt und verteilt. Die Standard-Registry ist der Docker Hub, auf dem tausende öffentlich verfügbarer Images zur Verfügung stehen, aber auch kuratierte »offizielle« Images. Viele Organisationen und Firmen nutzen eigene Registries, um kommerzielle oder »private« Images zu hosten, aber auch um den Overhead zu vermeiden, der mit dem Herunterladen von Images über das Internet einhergeht. In Abschnitt 7.4.1 finden Sie weitere Informationen über das Betreiben einer eigenen Registry. Der Docker Daemon lädt Images aus Registries herunter, wenn er `docker pull`-Anforderungen bekommt. Aber auch, wenn `docker run` ausgeführt wird oder Images in der `FROM`-Anweisung von Dockerfiles aufgeführt sind, die lokal nicht vorhanden sind, greift er auf die Registry zu.

4.1.1 Zugrunde liegende Technologien

Der Docker Daemon nutzt einen »Execution Driver«, um Container zu erstellen. Standardmäßig ist dies der Docker-eigene Treiber *runc*, aber es wird auch

1. Mit der Umstellung der Docker Engine auf den OCI-Standard runc im Release 1.11 wird nun noch ein weiterer containerd-Daemon gestartet.

noch LXC unterstützt. Runc ist sehr eng mit den folgenden Kernel-Features verbunden:

- *cgroups*, das für das Verwalten der von einem Container genutzten Ressourcen verantwortlich ist (zum Beispiel CPU- und Speicherbedarf). Es ist zudem für das *Freezing* und *Unfreezing* von Containern verantwortlich, was für `docker pause` genutzt wird.
- *Namensräume* dienen dem Isolieren der Container, sie stellen sicher, dass Dateisystem, Hostname, Benutzer, Netzwerk und Prozesse vom Rest des Systems getrennt sind.

Libcontainer unterstützen zudem *SELinux* und *AppArmor*, die für stärkere Sicherheitsbedürfnisse aktiviert werden können. In Kapitel 13 finden Sie mehr Informationen dazu.

Eine weitere wichtige Technologie, auf der Docker aufbaut, ist das Union File System (UFS), das genutzt wird, um die Dateischichten für die Container-Images zu verwalten. Das UFS wird von einem der vielen Storage-Treiber bereitgestellt – AUFS, devicemapper, BTRFS, ZFS, VFS oder Overlay. Siehe dazu den Kasten »Images, Container und das Union File System« weiter oben.

4.1.2 Zugehörige Technologien

Die Docker Engine und der Docker Hub bilden zusammen noch keine vollständige Lösung für die Arbeit mit Containern. Die meisten Anwender möchten gerne noch zusätzliche Services und Software nutzen, wie zum Beispiel für das Cluster Management, Tools zum Finden von Services und ausgefeiltere Netzwerkmöglichkeiten. Wie in Abschnitt 1.4 beschrieben, verwirklicht Docker Inc. mit dem Release 1.12 den Aufbau einer vollständigen Out-of-the-Box-Lösung, die all diese Features beinhaltet, es den Anwendern aber ermöglicht, die Standardkomponenten einfach gegen solche von anderen Herstellern auszutauschen.[2] Die Strategie der »austauschbaren Batterien« bezieht sich vor allem auf die API-Ebene – so dass sich Komponenten in die Docker Engine einhängen können –, aber dazu kann auch die Idee gezählt werden, zusätzliche Docker-Technologien in unabhängige Binaries zu verpacken, um diese dann einfacher austauschen zu können.

Die aktuelle Liste der von Docker unterstützten Technologien enthält:

- *Swarm*
  Die Clustering-Lösung von Docker. Swarm kann viele Docker Hosts in Gruppen zusammenfassen und es dem Anwender dann ermöglichen, sie als ge-

2. Seit der Version 1.12 sind die Erweiterungen enthalten, um eine Orchestrierung von Containern eines Cluster von Maschinen zu steuern.

meinsame Ressource zu betrachten. In Kapitel 12 finden Sie weitere Informationen dazu.

- *Compose*
  Docker Compose ist ein Tool zum Bauen und Ausführen von Anwendungen, die aus mehreren Docker-Containern bestehen. Es wird vor allem in der Entwicklung und beim Testen und weniger im Produktivumfeld genutzt. In Abschnitt 5.2 ist das Tool ausführlicher beschrieben.
- *Machine*
  Docker Machine installiert und konfiguriert Docker Hosts auf lokalen oder entfernten Ressourcen. Machine konfiguriert zudem den Docker Client und erleichtert damit den Wechsel zwischen Umgebungen. In Kapitel 9 finden Sie ein Beispiel.
- *Kitematic*
  Kitematic ist eine GUI für Mac OS und Windows, um Docker-Container auszuführen und zu verwalten.
- *Docker Trusted Registry*
  Dies ist die On-Premise-Lösung für das Speichern und Verwalten von Docker-Images. Dabei handelt es sich mehr oder weniger um eine lokale Version des Docker Hub, die mit bestehender Sicherheitsinfrastruktur interagieren und Unternehmen dabei helfen kann, den Gesetzen und Regeln bezüglich des Speicherns und der Sicherheit von Daten Folge leisten zu können. Zu den Features gehören Metriken, Role-Based Access Control (RBAC) und Logs, die alle über eine Administrationskonsole verwaltet werden.
- *Docker Universal Plane*
  Dies ist die On-Premise-Lösung für die gesamte Verwaltung eines verteilen Docker-Clusters. Das Management der Docker-Machines, der Images, Netzwerke, Volumes und Status der Container sind in einer zentralen Anwendung möglich.
- *Docker Cloud*
  Mit dem Angebot der Docker-Cloud lassen sich mit einer Weboberfläche Docker-Machines bei verschiedenen Cloud-Providern verwalten.
- *Notary*
  Das Projekt Notary erlaubt es, beliebige Daten vertraulich zu behandeln
- *SWARMKIT*
  Das Projekt Swarmkit ist die Basis für die Cluster und Container-Orchestrierung der Docker Engine 1.12.

Es gibt schon eine lange Liste von Services und Anwendungen von dritter Seite, die auf Docker aufbauen oder damit arbeiten. In den folgenden Bereichen finden Sie eine Reihe von Lösungen:

- *Netzwerk*
  Das Aufsetzen von Netzwerken aus Containern über mehrere Hosts hinweg ist ein nichttriviales Problem, das auf unterschiedlichen Wegen gelöst werden kann. Es gibt hier eine Reihe von Lösungen, einschließlich Weave *(https://www.weave.works/products/weave-net)* und Project Calico *(https://www.projectcalico.org)*. Seit dem Release 1.9 bietet Docker selbst eine integrierte Netzwerklösung namens Overlay an. Anwender können mit verschiedenen Overlay-Treiber per Networking Plugin Framework die Default Treiber erweitern.
- *Services Discovery*
  Wird ein Docker-Container gestartet, muss er irgendwie die anderen Services finden, mit denen er reden soll – und die im Allgemeinen auch in Containern laufen. Da Container dynamisch zugewiesene IP-Adressen haben, ist das in einem großen System gar nicht so einfach. Lösungen dafür gibt es unter anderem von Consul *(https://www.consul.io)*, Registrator *(https://github.com/gliderlabs/registrator)*, SkyDNS *(https://github.com/skynetservices/skydns)* und etcd *(https://github.com/coreos/etcd)*.
- *Orchestrieren und Cluster-Management*
  In großen Containerszenarien sind gute Tools zum Überwachen und Verwalten des Systems außerordentlich wichtig. Jeder neue Container muss auf einem Host untergebracht, überwacht und aktualisiert werden. Das System muss auf Fehler oder Laständerungen reagieren, indem Container entsprechend verschoben, gestartet oder gestoppt werden. Es gibt in diesem Bereich schon eine Reihe von Lösungen, unter anderem Kubernetes von Google *(http://kubernetes.io)*, Marathon (*https://github.com/mesosphere/marathon*, ein Framework für Mesos *[https://mesos.apache.org]*), Fleet von CoreOS *(https://github.com/coreos/fleet)*, das Docker-eigene Swarm-Tool sowie mit dem Release 1.12 eine Integration des Swarmkit (*https://github.com/docker/swarmkit*).

All diese Themen werden detaillierter in Teil III besprochen. Es sei zudem darauf hingewiesen, dass es auch Alternativen zur Docker Trusted Registry gibt, unter anderem die Enterprise Registry von CoreOS *(https://coreos.com/products/enterprise-registry)* und Artifactory von JFrog *(http://www.jfrog.com/open-source/#os-arti)*.

Neben den schon erwähnten Netzwerktreiber-Plugins unterstützt Docker auch *Volume-Plugins* für die Integration mit anderen Storage-Systemen. Erwähnenswert sind hier Flocker *(https://github.com/ClusterHQ/flocker)*, ein Data Management und Migration-Tool für mehrere Hosts, und GlusterFS *(https://github.com/calavera/docker-volume-glusterfs)* für verteilten Storage. Mehr Informationen zum Plugin-Framework finden Sie auf der Docker-Website unter *https://docs.docker.com/engine/extend/plugins*.

Ein interessanter Nebeneffekt des wachsenden Containermarkts ist das Entstehen von Betriebssystemen, die dazu gedacht sind, die Container zu hosten. Während Docker problemlos auf den meisten aktuellen Linux-Distributionen wie Ubuntu und Red Hat läuft, gibt es eine Reihe von Projekten für minimale und leicht zu wartende Distributionen, die sich nur darauf konzentrieren, Container (oder Container und VMs) laufen zu lassen, insbesondere im Rahmen eines Data Centers oder Clusters. Dazu gehören unter anderem Project Atomic *(http://www.projectatomic.io)*, CoreOS *(https://coreos.com)* und RancherOS *(http://rancher.com/rancher-os)*.

4.1.3 Docker Hosting

Wir werden das Docker Hosting detaillierter in Kapitel 9 besprechen, aber es lohnt sich, auf manche der vielen Auswahlmöglichkeiten schon einmal hinzuweisen. Viele der klassischen Cloud-Provider, wie zum Beispiel Amazon, Azure, Google und Digital Ocean, haben Docker auf die eine oder andere Art und Weise im Angebot. Die Container Engine von Google erscheint hier am interessantesten, da sie direkt auf Kubernetes aufbaut. Natürlich ist es normalerweise auch dann möglich, VMs für das Ausführen von Docker-Containern aufzusetzen, wenn der Provider keine speziellen Angebote für Docker besitzt.

Joyent ist mit seinem eigenen Containerangebot namens Triton auf den Markt gekommen, das auf SmartOS aufbaut. Durch das Implementieren der Docker API mit seiner eigenen Container- und Linux-Emulationstechnologie konnte Joyent eine öffentliche Cloud aufbauen, die mit dem normalen Docker Client zusammenarbeitet. Wichtig ist dabei, dass Joyent der Meinung ist, seine Containerimplementierung sei so sicher, dass sie direkt auf der Hardware laufen kann, statt noch in VMs gesteckt werden zu müssen. Das kann zu einer deutlich höheren Effizienz führen, insbesondere im Bereich I/O.

Es gibt zudem einige Projekte, die eine PaaS-Plattform auf Docker aufbauen, unter anderem Deis *(http://deis.io)*, Flynn *(https://flynn.io)* und Paz *(http://www.paz.sh)*.

4.2 Wie Images gebaut werden

Wir haben in Abschnitt 3.3 gesehen, dass man neue Images vor allem über Dockerfiles und den Befehl `docker build` erstellt. Dieser Abschnitt wird sich genauer damit befassen, was dabei passiert, und die verschiedenen Anweisungen beschreiben, die in einem Dockerfile genutzt werden können. Es lohnt sich, ein grobes Verständnis von der Arbeitsweise des Build-Befehls zu haben, da dessen Verhalten gelegentlich überraschend sein kann.

4.2.1 Der Build Context

Der Befehl `docker build` erfordert ein Dockerfile und einen *Build Context* (der leer sein kann). Der Build Context ist der Satz lokaler Dateien und Verzeichnisse, die aus `ADD`- oder `COPY`-Anweisungen im Dockerfile angesprochen werden können. Er wird im Allgemeinen als Pfad zu einem Verzeichnis definiert. So haben wir zum Beispiel den Build-Befehl `docker build -t test/cowsay-dockerfile .` in Abschnitt 3.3 genutzt, bei dem der Context auf `'.'` gesetzt wurde – das aktuelle Arbeitsverzeichnis. Alle Dateien und Verzeichnisse unter diesem Pfad bilden den Build Context und werden als Teil des Build-Prozesses an den Docker Daemon geschickt.

In Fällen, in denen ein Context nicht angegeben ist – weil nur eine URL auf ein Dockerfile geliefert wird oder der Inhalt eines Dockerfiles aus `STDIN` gepipet wird – wird der Build Context als leer angesehen.

Nutzen Sie / nicht als Build Context

Da der Build Context in einem Tarball zusammengefasst und an den Docker Daemon geschickt wird, *wollen* Sie kein Verzeichnis mit vielen Dateien darin nutzen. So führt zum Beispiel der Einsatz von */home/user*, *Downloads* oder / zu einer sehr starken Verzögerung, während der Docker Client alles verpackt und an den Daemon sendet.

Wird eine URL, die mit *http* oder *https* beginnt, übergeben, geht Docker davon aus, einen direkten Link auf ein Dockerfile zu erhalten. Das ist eher selten sehr nützlich, weil kein Kontext damit verbunden ist (und Links auf Archive werden nicht akzeptiert).

Es kann auch ein git-Repository als Build Context genutzt werden. In solch einer Situation klont der Docker Client das Repository und alle Submodule in ein temporäres Verzeichnis, das dann als Build Context an den Docker Daemon geschickt wird. Docker interpretiert den Context als git-Repository, wenn der Pfad mit *github.com/*, *git@* oder *git://* beginnt. Im Allgemeinen schlage ich vor, diese Methode zu vermeiden und stattdessen Repositories per Hand auszuchecken, da Sie dann flexibler sind und weniger Verwirrung entstehen kann.

Der Docker Client kann seine Daten auch von STDIN erhalten, indem statt des Build Context ein `"-"` übergeben wird. Die Eingabe kann dann entweder ein Dockerfile ohne Context sein (zum Beispiel `docker build - < Dockerfile`) oder eine Archivdatei, die den Context aufsetzt und ein Dockerfile enthält (zum Beispiel `docker build - < context.tar.gz`). Archivdateien können das Format *tar.gz*, *xz* oder *bzip2* haben.

Die Position des Dockerfiles im Context kann mit dem Argument `-f` angegeben werden (zum Beispiel `docker build -f dockerfiles/Dockerfile.debug .`). Wenn Sie nichts festlegen, sucht Docker nach einer Datei namens *Dockerfile* im Wurzelverzeichnis des Context.

Eine .dockerignore-Datei einsetzen

Um nicht benötigte Dateien aus dem Build Context herauszuhalten, können Sie eine Datei *.dockerignore* verwenden. Die Datei sollte, durch Zeilenumbrüche getrennt, die Namen von Dateien enthalten, die aus dem Context herauszunehmen sind. Die Jokerzeichen `*` und `?` sind dabei erlaubt. Schauen wir uns zum Beispiel folgende `.dockerignore` an:

```
.git ❶
*/.git ❷
*/*/.git ❸
*.sw? ❹
```

❶ Ignoriert eine Datei oder ein Verzeichnis *.git* im Wurzelverzeichnis des Build Context, erlaubt sie aber in einem Unterverzeichnis (so wird zum Beispiel *.git* ignoriert, aber nicht *dir1/.git*).

❷ Ignoriert eine Datei oder ein Verzeichnis *.git* in einem Verzeichnis direkt unterhalb des Wurzelverzeichnisses (zum Beispiel wird *dir1/.git* ignoriert, nicht aber *.git* oder *dir1/dir2/.git*).

❸ Ignoriert eine Datei oder ein Verzeichnis *.git* in einem Verzeichnis, das zwei Unterverzeichnisse unter dem Wurzelverzeichnis liegt (zum Beispiel wird *dir1/dir2/.git* ignoriert, nicht aber *.git* oder *dir1/.git*).

❹ Ignoriert *test.swp*, *test.swo* und *bla.swp*, aber nicht *dir1/test.swp*.

»Richtige« reguläre Ausdrücke wie `[A-Z]*` werden nicht unterstützt.

Aktuell gibt es keine Möglichkeit, Dateien in allen Unterverzeichnissen gleichzeitig zu erreichen (Sie können also nicht sowohl */test.tmp* als auch */dir1/test.tmp* mit einem Ausdruck abdecken).

4.2.2 Imageschichten

Neulinge im Docker-Universum sind häufig davon irritiert, wie Images gebaut werden. Jede Anweisung in einem Dockerfile führt zu einer neuen Imageschicht – einem *Layer* –, die wieder zum Starten eines neuen Containers genutzt werden kann. Die neue Schicht wird erzeugt, indem ein Container mit dem Image der vorigen Schicht gestartet, dann die Dockerfile-Anweisung ausgeführt und schließlich ein neues Image gespeichert wird. Ist eine Dockerfile-Anweisung erfolgreich abgeschlossen worden, wird der temporär erzeugte Container wieder gelöscht, sofern nicht das Argument `--rm=false` angegeben wurde.[3] Da jede Anweisung zu einem statischen Image führt – mehr oder weniger nur ein Dateisystem und ein paar Metadaten –, werden alle laufenden Prozesse in der Anweisung gestoppt. Das hat Folgen: Sie können zwar in einer `RUN`-Anweisung langlebige Prozesse starten, wie zum Beispiel eine Datenbank oder einen SSH-Daemon,

3. Machen Sie sich keine Gedanken, wenn ich Sie hier abgehängt habe. Sie werden das besser verstehen, nachdem Sie sich die Ausgabe von `docker build` in unserem Debug-Szenario angeschaut haben.

aber diese laufen nicht mehr, wenn die nächste Anweisung verarbeitet oder ein Container gestartet wird. Wollen Sie, dass ein Service oder Prozess zusammen mit dem Container gestartet wird, müssen Sie ihn über eine `ENTRYPOINT`- oder `CMD`-Anweisung aufrufen.

Sie können sich alle Schichten eines Image anzeigen lassen, indem Sie den Befehl `docker history` aufrufen. Zum Beispiel:

```
$ docker history mongo:latest
IMAGE          CREATED      CREATED BY                          ...
278372cb22b2   4 days ago   /bin/sh -c #(nop) CMD ["mongod"]
341d04fd3d27   4 days ago   /bin/sh -c #(nop) EXPOSE 27017/tcp
ebd34b5e9c37   4 days ago   /bin/sh -c #(nop) ENTRYPOINT &{["/entr
f3b2b8cf226c   4 days ago   /bin/sh -c #(nop) COPY file:ef2883b33e
ba53e9f50f18   4 days ago   /bin/sh -c #(nop) VOLUME [/data/db]
c537910de5cc   4 days ago   /bin/sh -c mkdir -p /data/db && chown
f48ad436057a   4 days ago   /bin/sh -c set -x
df59596772ab   4 days ago   /bin/sh -c echo "deb http://repo.mongo
96de83c82d4b   4 days ago   /bin/sh -c #(nop) ENV MONGO_VERSION=3.
0dab801053d9   4 days ago   /bin/sh -c #(nop) ENV MONGO_MAJOR=3.0
5e7b428dddf7   4 days ago   /bin/sh -c apt-key adv --keyserver ha.
e81ad85ddfce   4 days ago   /bin/sh -c curl -o /usr/local/bin/gosu
7328803ca452   4 days ago   /bin/sh -c gpg --keyserver ha.pool.sks
ec5be38a3c65   4 days ago   /bin/sh -c apt-get update
430e6598f55b   4 days ago   /bin/sh -c groupadd -r mongodb && user
19de96c112fc   4 days ago   /bin/sh -c #(nop) CMD ["/bin/bash"]
ba249489d0b6   4 days ago   /bin/sh -c #(nop) ADD file:b908886c97e
```

Schlägt ein Build fehl, kann es sehr nützlich sein, die Schicht vor dem Fehler zu starten. Nehmen wir zum Beispiel folgendes Dockerfile:

```
FROM busybox:latest

RUN echo "Das sollte funktionieren"
RUN /bin/bash -c echo "Das nicht"
```

Jetzt versuchen wir, das Image zu bauen:

```
$ docker build -t echotest .
Sending build context to Docker daemon 2.048 kB
Step 0 : FROM busybox:latest
 ---> 4986bf8c1536
Step 1 : RUN echo "Das sollte funktionieren"
 ---> Running in f63045cc086b ❶
Das sollte funktionieren
 ---> 85b49a851fcc ❷
Removing intermediate container f63045cc086b ❸
Step 2 : RUN /bin/bash -c echo "Das nicht"
 ---> Running in e4b31d0550cd
/bin/sh: /bin/bash: not found
The command '/bin/sh -c /bin/bash -c echo "Das nicht"' returned a
non-zero code: 127
```

❶ ID des temporären *Containers*, den Docker gestartet hat, um unsere Anweisung darin auszuführen

❷ ID des *Image*, das aus dem Container erstellt wurde

❸ Der temporäre Container wird jetzt gelöscht.

Obwohl das Problem in diesem Fall ziemlich eindeutig ist, können wir das Image aus der letzten erfolgreichen Schicht laufen lassen, um die Anweisung zu debuggen. Beachten Sie, dass wir hier die letzte *Image*-ID (85b49a851fcc) verwenden, nicht die ID des letzten *Containers* (e4b31d0550cd):

```
$ docker run -it 85b49a851fcc
/ # /bin/bash -c "echo hmm"
/bin/sh: /bin/bash: not found
/ # /bin/sh -c "echo ahh!" ahh!
/#
```

So wird das Problem offensichtlicher: Das Image `busybox` enthält keine `bash`-Shell.

4.2.3 Caching

Docker cacht zudem jede Schicht, um das Bauen der Images zu beschleunigen. Dieses Caching ist für einen effizienten Workflow sehr wichtig, allerdings wird dabei recht naiv vorgegangen. Der Cache wird für eine Anweisung genutzt, wenn:

- die vorige Anweisung im Cache gefunden wurde *und*
- es eine Schicht im Cache gibt, die genau die gleiche Anweisung und Eltern-Schicht besitzt (selbst ein anders gesetztes Leerzeichen invalidiert den Cache).

Im Fall von `COPY`- und `ADD`-Anweisungen wird der Cache invalidiert, wenn sich die Prüfsumme oder Metadaten für eine der Dateien geändert haben.

Das bedeutet, dass `RUN`-Anweisungen, die bei mehrfachen Aufrufen nicht zwingend zum gleichen Ergebnis führen, *trotzdem gecacht werden*. Achten Sie darauf vor allem, wenn Sie Dateien herunterladen, `apt-get update` aufrufen oder Quellcode-Repositories klonen.

Müssen Sie den Cache invalidieren, können Sie `docker build` mit dem Argument `--no-cache` aufrufen. Sie können auch noch vor der Stelle, an der Sie den Cache invalidieren wollen, eine Anweisung hinzufügen oder ändern, daher sehen Sie manchmal Dockerfiles mit Zeilen wie den folgenden:

```
ENV UPDATED_ON "14:12 17 February 2015"
RUN git clone ...
```

Ich empfehle, diese Technik nicht einzusetzen, da sie Anwender des Image verwirren kann – insbesondere, wenn das Image an einem anderen Tag gebaut wurde, als diese Zeile nahelegt.

4.2.4 Basis-Images

Wenn Sie Ihre eigenen Images erstellen, werden Sie sich entscheiden müssen, mit welchem Basis-Image Sie beginnen. Sie haben hier eine große Auswahl zur Verfügung, und es lohnt sich, ein wenig Zeit zu investieren, um die diversen Vor- und Nachteile der Images zu verstehen.

Im besten Fall müssen Sie gar kein Image erstellen – Sie können einfach ein bestehendes nutzen und Ihre Konfigurationsdateien und/oder Daten damit verknüpfen. Das wird für verbreitete Anwendungssoftware der Fall sein, wie zum Beispiel bei Datenbanken und Webservern, für die offizielle Images zur Verfügung stehen. Im Allgemeinen ist es besser, ein offizielles Image zu verwenden, statt Ihr eigenes zu bauen – Sie können auf die Arbeit und Erfahrung anderer setzen, die sich damit befasst haben, wie die Software am besten in einem Container läuft. Gibt es einen bestimmten Grund, warum ein offizielles Image für Sie nicht nutzbar ist, überlegen Sie sich, ob es sich lohnt, ein Issue beim jeweiligen Projekt aufzumachen, da es sein kann, dass andere ähnliche Probleme haben oder schon eine Lösung kennen.

Brauchen Sie ein Image, um Ihre eigene Anwendung zu hosten, halten Sie zuerst Ausschau nach einem offiziellen Basis-Image für die Sprache oder das Framework, die/das Sie einsetzen (zum Beispiel Go oder Ruby on Rails). Häufig können Sie Images für das Bauen und Bereitstellen Ihrer Software getrennt halten (zum Beispiel lässt sich das Image `java:jdk` nutzen, um eine Java-Anwendung zu bauen, während zum Bereitstellen der daraus entstandenen JAR-Datei das kleinere Image `java:jre` sinnvoll ist, welches all die unnötigen Build-Tools weglässt). Genauso gibt es für manche offiziellen Images (wie zum Beispiel `node`) spezielle »Slim«-Builds, bei denen Entwicklungstools und Header weggelassen wurden.

Manchmal brauchen Sie tatsächlich eine kleine, aber vollständige Linux-Distribution. Wenn ich wirklich minimalistisch vorgehen will, nutze ich das Image `alpine`, das nur etwa 5 MB groß ist, aber trotzdem einen umfassenden Paketmanager zum einfachen Installieren von Anwendungen und Tools enthält. Möchte ich ein etwas vollständigeres Image nutzen, verwende ich im Allgemeinen eines der `debian`-Images, die viel kleiner sind als die ebenfalls weit verbreiteten `ubuntu`-Images, aber trotzdem auf die gleichen Pakete zurückgreifen können. Ist Ihr Unternehmen an eine bestimmte Linux-Distribution gebunden, sollten Sie auch ein Docker-Image dafür finden. Das ist sinnvoller, als zu einer neuen Distribution zu wechseln, die in Ihrem Unternehmen keine Unterstützung hat und für die noch keine Erfahrungen vorhanden sind.

Meist ist es nicht notwendig, viel Aufwand in die »Minimalisierung« von Images zu stecken. Denken Sie daran: Die Basisschichten werden von den Images gemeinsam genutzt. Haben Sie also schon das Image `ubuntu:14.04` und ziehen Sie ein Image vom Hub, das darauf basiert, werden nur die Änderungen geholt, nicht

das gesamte Image. Minimale Images sind aber trotzdem von Vorteil, wenn Sie auf schnelle Deployments und einfache Verteilung aus sind.

Es ist möglich, besonders minimal zu werden und Images nur mit Binaries auszuliefern. Dazu schreiben Sie ein Dockerfile, das von dem speziellen Image `scratch` erbt (ein komplett leeres Dateisystem) und in das Sie Ihr Binary kopieren und dazu eine passende `CMD`-Anweisung eintragen. Ihr Binary benötigt dann alle erforderlichen Bibliotheken (kein dynamisches Linken), und es gibt keine Möglichkeit, externe Befehle aufzurufen. Zudem müssen Sie daran denken, dass das Binary für die Architektur des Containers kompiliert wird – die sich von der Maschine unterscheiden kann, auf der der Docker Client läuft.[4]

Während der minimalistische Ansatz sehr verlockend sein kann, sollten Sie sich darüber im Klaren sein, dass das Debuggen und Warten solch eines Systems ausgesprochen schwierig ist – `busybox` besitzt nicht viele Tools, mit denen man arbeiten kann, und wenn Sie `scratch` eingesetzt haben, gibt es nicht einmal eine Shell.

Phusion Reaction

Ein weiteres interessantes Basisimage ist `phusion/baseimage-docker`. Die Phusion-Entwickler haben dieses Basisimage als Reaktion auf das offizielle Ubuntu-Image erstellt, weil diesem ihrer Meinung nach eine Reihe wichtiger Services fehlt. Viele Entwickler aus dem zentralen Docker-Umfeld sehen das nicht so, was zu einer bunten Diskussion in Blogs, auf IRC und Twitter führte. Die größten Differenzen sind dabei:

- *Die Notwendigkeit eines Init-Service*
  Docker ist der Meinung, dass in jedem Container nur eine einzelne Anwendung und idealerweise auch nur *ein Prozess* laufen sollten. Haben Sie nur einen Prozess, besteht kein Bedarf für einen Init-Service. Das Hauptargument von Phusion ist, dass das Fehlen eines Init-Service zu Containern voll mit Zombie-Prozessen führen kann – also Prozessen, die nicht sauber von ihren Elternprozessen beendet oder durch einen Überwachungsprozess abgeräumt wurden. Das ist zwar korrekt, aber Zombie-Prozesse können nur durch Fehler im Anwendungscode entstehen – die meisten Benutzer sollten dieses Problem nicht haben, und wenn doch, ist es besser, den Fehler im Code zu beheben.

4. Es ist tatsächlich möglich, dieses Konzept der minimalen Images noch weiter zu treiben, indem Sie Docker und den vollständigen Linux-Kernel außen vor lassen und einem *Unikernel*-Ansatz folgen. Bei einer Unikernel-Architektur werden Anwendungen mit einem Kernel kombiniert, der nur die für die Anwendung nötigen Features enthält. Das Ganze läuft dann direkt auf einem Hypervisor. Damit entledigen Sie sich einer Reihe unnötiger Codeschichten und ungenutzter Treiber, was zu einer viel kleineren und schnelleren Anwendung führt (Unikernel booten im Allgemeinen in weniger als einer Sekunde, sie können also direkt als Reaktion auf eine Benutzeranforderung gestartet werden). Wollen Sie mehr darüber erfahren, werfen Sie einen Blick auf »Unikernels: Rise of the Virtual Library Operating System« von Anil Madhavapeddy und David J. Scott *(https://queue.acm.org/detail.cfm?id=2566628)* und MirageOS *(http://www.openmirage.org)*.

- *Ein laufender `cron`-Daemon*
  Die Basis-Images `ubuntu` und `debian` starten standardmäßig nicht den `cron`-Daemon, das `phusion`-Image aber schon. Phusion argumentiert damit, dass viele Anwendungen von `cron` abhängen, daher ist eine laufende Instanz davon wichtig. Docker sagt hingegen – und ich stimme ihnen da zu –, dass `cron` nur laufen sollte, wenn Ihre Anwendung das erfordert.
- *Ein SSH-Daemon*
  Die Standardimages installieren und starten nicht automatisch einen SSH-Daemon. Eine Shell erhält man normalerweise über den Befehl `docker exec` (siehe Abschnitt 4.6.2), wodurch man sich den Aufwand eines unnötigen Prozesses pro Container spart. Phusion scheint das zu akzeptieren und hat seinen SSH-Daemon standardmäßig deaktiviert, aber ihr Image ist immer noch durch das Einbinden des Daemons und seiner Bibliotheken deutlich aufgeblasen.

Ich persönlich würde empfehlen, das Basis-Image von Phusion nur dann zu nehmen, wenn Sie die Anforderung haben, mehrere Prozesse, `cron` und `ssh` in Ihrem Container laufen zu lassen. Andernfalls würde ich bei den Images aus dem offiziellen Docker Repository bleiben, wie zum Beispiel `ubuntu:14.04` oder `debian:wheezy`.

Images neu bauen

Denken Sie daran, dass sich Docker beim Ausführen von `docker build` die `FROM`-Anweisung anschaut und versucht, das Image zu ziehen, wenn es noch nicht lokal vorhanden ist. Ist es hingegen schon da, wird Docker das Image verwenden, ohne zu prüfen, ob es eine neuere Version gibt. Ein `docker build` reicht also nicht aus, um sicherzustellen, dass Ihre Images immer aktuell sind. Sie müssen entweder explizit `docker pull` für alle »vererbenden« Images aufrufen oder sie löschen, um den Build-Befehl zu zwingen, die neueste Version herunterzuladen.

Das wird besonders wichtig, wenn verbreitete Basis-Images, wie zum Beispiel `debian`, mit Sicherheitspatches aktualisiert werden.

4.2.5 Anweisungen im Dockerfile

Dieser Abschnitt behandelt kurz die diversen Anweisungen, die in Dockerfiles genutzt werden können. Er geht nicht allzu sehr in die Tiefe, weil sich einerseits vieles noch ändert und es andererseits eine umfassende und immer aktuelle Dokumentation auf der Docker-Website unter *http://docs.docker.com/reference/builder* gibt. Kommentare in Dockerfiles werden durch ein `#` am Zeilenanfang markiert.

Exec- versus Shell-Format

Mehrere Anweisungen (`RUN`, `CMD` und `ENTRYPOINT`) verarbeiten sowohl ein *Shell*-Format als auch ein *Exec*-Format. Das Exec-Format erwartet ein JSON-Array (zum Beispiel `["executable", "param1", "param2"]`), bei dem das erste Element der Name einer ausführbaren Datei ist, die dann mit den weiteren Elementen als Parameter aufgerufen wird. Das Shell-Format ist ein String,

der durch Übergabe an `/bin/sh -c` ausgewertet wird. Nutzen Sie das Exec-Format, um zu verhindern, dass die Shell Strings umbaut, oder wenn das Image kein `/bin/sh` besitzt.

Die folgenden Anweisungen stehen in Dockerfiles zur Verfügung:

- `ADD`
  Kopiert Dateien aus dem Build Context oder von URLs in das Image. Wird eine Archivdatei von einem lokalen Pfad aus hinzugefügt, wird diese automatisch ausgepackt. Da `ADD` ziemlich viel kann, ist es meist besser, das einfachere `COPY` zu nutzen, um Dateien und Verzeichnisse aus dem Build Context zu kopieren, und mit `RUN` und `curl` oder `wget` Ressourcen von anderer Stelle herunterzuladen (womit Sie die Möglichkeit haben, mit einer Anweisung die heruntergeladenen Daten nach dem Verarbeiten auch wieder zu löschen).
- `ARG`
  Mit dem Parameter ARG wird seit Docker-Version 1.9 eine temporäre ENV-Variable aus dem Build Context des Aufrufers gesetzt. Die ENV-Variable wird nicht in den META-Daten des Image vermerkt. Mit dem Parameter `--build-arg <VARIBALE>=<VALUE>` kann der Aufrufer die ARG-ENV-Variable setzen. Automatisch werden die folgenden ENV-Variablen des Aufrufers vom docker client übertragen: HTTP_PROXY, HTTPS_PROXY, http_proxy, https_proxy, FTP_PROXY und NO_PROXY. Mit einem kleinen Trick kann man mit dem ARG-Befehl ein einfaches Dockerfile-Templating realisieren:

```
...
ARG MY_VERSION
ENV MY_VERSIONLABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."${MY_VERSION-:1.3}
RUN apt-get install -y mypackage=$MY_VERSION
...
```

- `CMD`
  Führt die angegebene Anweisung aus, wenn der Container gestartet wurde. Ist auch ein `ENTRYPOINT` definiert, wird die Anweisung als Argument für `ENTRYPOINT` verwendet (in diesem Fall müssen Sie das Exec-Format verwenden). Die `CMD`-Anweisung wird durch Argumente überschrieben, die `docker run` nach dem Imagenamen übergeben werden. Nur die letzte `CMD`-Anweisung hat eine Auswirkung, und alle vorigen `CMD`-Anweisungen werden ignoriert (einschließlich der in Basis-Images).
- `COPY`
  Wird verwendet, um Dateien aus dem Build Context in das Image zu kopieren. Es gibt die zwei Formen `COPY src dest` und `COPY ["src", "dest"]`, die

beide die Datei oder das Verzeichnis an *src* im Build Context nach *dest* im Container kopieren. Das JSON-Array-Format ist notwendig, wenn die Pfade Leerzeichen enthalten. Jokerzeichen können genutzt werden, um mehrere Dateien oder Verzeichnisse anzugeben. Beachten Sie, dass Sie keine *src*-Pfade außerhalb des Build Context nutzen können (so wird zum Beispiel *../another_dir/myfile* nicht funktionieren).

- `ENTRYPOINT`
  Legt eine ausführbare Datei (und Standardargumente) fest, die beim Start des Containers laufen soll. Jegliche `CMD`-Anweisungen oder an `docker run` nach dem Imagenamen übergebenen Argumente werden als Parameter an das Executable durchgereicht. `ENTRYPOINT`-Anweisungen werden häufig genutzt, um »Start«-Skripten anzustoßen, die Variablen und Services initialisieren, bevor andere übergebene Argumente ausgewertet werden.
- `ENV`
  Setzt Umgebungsvariablen im Image. Diese können in folgenden Anweisungen genutzt werden. Zum Beispiel:

  ```
  ...
  ENV MY_VERSION 1.3
  RUN apt-get install -y mypackage=$MY_VERSION
  ...
  ```

  Die Variablen stehen zudem innerhalb des Image zur Verfügung.
- `EXPOSE`
  Erklärt Docker, dass der Container einen Prozess enthält, der an dem oder den angegebenen Port(s) lauscht. Diese Information wird von Docker beim Verbinden von Containern (siehe Abschnitt 4.4) oder beim Veröffentlichen von Ports durch das Argument `-P` von `docker run` genutzt. Die Anweisung alleine hat ansonsten keine Auswirkungen.
- `FROM`
  Setzt das Basis-Image für das Dockerfile, nachfolgende Anweisungen bauen auf diesem Image auf. Das Basis-Image wird in der Form `IMAGE:TAG` angegeben (zum Beispiel `debian:wheezy`). Lässt man das Tag weg, wird `latest` angenommen, aber ich empfehle dringend, immer das Tag auf eine bestimmte Version zu setzen, um Überraschungen zu vermeiden. Diese Anweisung muss die erste in einem Dockerfile sein.
- `HEALTHCHECK`
  Die Docker Engine prüft regelmäßig den Status der Anwendung im Container. Das Ergebnis kann mit dem Befehl `docker inspect` oder seit Docker-Version 1.12 über das API »health_status« erfragt werden.

  ```
  ...
  HEALTHCHECK --interval=5m --timeout=3s \
    CMD curl -f http://localhost/ || exit 1
  ...
  ```

- `MAINTAINER`
  Setzt die »Author«-Metadaten des Image auf den angegebenen String. Sie können sie mit `docker inspect -f {{.Author}} IMAGE` auslesen. Normalerweise werden mit dieser Anweisung der Name und die Kontaktdaten des Maintainers des Image gesetzt.
- `LABEL`
  Mit diesem Befehl können Meta-Daten für ein Image eingetragen werden. Ein Image kann nun mit Information über den Autor, die Lizenz, Version oder den Zweck angereichert werden.

  ```
  ...
  LABEL "com.example.vendor"="ACME Incorporated"
  LABEL com.example.label-with-value="foo"
  LABEL version="1.0"
  LABEL description="Dieser Text verdeutlicht\
  das ein Wert eines Labels sich auf mehrere Zeilen verteilen kann."
  ...
  ```
- `ONBUILD`
  Gibt eine Anweisung an, die später ausgeführt werden soll, wenn das Image als Basis-Image für ein weiteres Image genutzt wird. Das kann sinnvoll sein, um Daten zu verarbeiten, die in einem Kind-Image hinzugefügt werden (zum Beispiel kann so Code aus einem ausgewählten Verzeichnis kopiert und ein Build-Skript damit ausgeführt werden).
- `RUN`
  Führt die angegebene Anweisung im Container aus und bestätigt das Ergebnis.
- `SHELL`
  Die Anweisung SHELL erlaubt es seit Docker 1.12, die Shell für den folgenden RUN-Befehl zu setzten. So ist es möglich, dass nun auch direkt bash, zsh oder Powershell-Befehle in einem Dockerfile genutzt werden können.
- `STOPSIGNAL`
  Mit der Anweisung STOPSIGNAL wird seit Docker 1.9 der Wert des Signals des System-Aufrufs gesetzt, der zum Beenden des Containers benutzt wird.
- `USER`
  Setzt den Benutzer (über Name oder UID), der in folgenden `RUN`-, `CMD`- oder `ENTRYPOINT`-Anweisungen genutzt werden soll. Beachten Sie, dass UIDs auf Host und Container die gleichen sind, die Benutzernamen aber verschiedenen UIDs zugewiesen sein können – was das Setzen von Berechtigungen knifflig machen kann.
- `VOLUME`
  Deklariert die angegebene Datei oder das Verzeichnis als Volume. Besteht die Datei oder das Verzeichnis schon im Image, wird sie beziehungsweise es in das Volume kopiert, wenn der Container gestartet wird. Werden mehrere

Argumente angegeben, werden sie als mehrfache `VOLUME`-Anweisungen interpretiert. Sie können aus Gründen der Portierbarkeit und Sicherheit nicht das Host-Verzeichnis für ein Volume in einem Dockerfile angeben. Mehr Informationen erhalten Sie in Abschnitt 4.5.

- `WORKDIR`
  Setzt das Arbeitsverzeichnis für alle folgenden `RUN`-, `CMD`-, `ENTRYPOINT`-, `ADD`- oder `COPY`-Anweisungen. Diese Anweisung kann mehrfach genutzt werden. Relative Pfadangaben sind möglich und werden relativ zum zuvor gesetzten `WORKDIR` interpretiert.

4.3 Container mit der Außenwelt verbinden

Stellen Sie sich vor, Sie lassen einen Webserver in einem Container laufen. Wie können Sie dann der Außenwelt darauf Zugriff gewähren? Die Antwort ist, Ports mit den Befehlen `-p` oder `-P` zu »veröffentlichen«. Dieser Befehl leitet Ports auf den Host des Containers weiter. Zum Beispiel:

```
$ docker run -d -p 8000:80 nginx
af9038e18360002ef3f3658f16094dadd4928c4b3e88e347c9a746b131db5444
$ curl localhost:8000
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Das Argument `-p 8000:80` hat Docker angewiesen, Port 8000 auf dem Host an Port 80 im Container weiterzuleiten. Alternativ kann das Argument `-P` genutzt werden, um Docker automatisch einen freien Port zum Weiterleiten auf dem Host auswählen zu lassen. Zum Beispiel:

```
$ ID=$(docker run -d -P nginx)
$ docker port $ID 80
0.0.0.0:32771
$ curl localhost:32771
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Der Hauptvorteil von `-P` ist, dass Sie sich nicht mehr länger um das Verwalten der belegten Ports kümmern müssen. Das wird wichtig, wenn Sie viele Container einsetzen, die Ports veröffentlichen. In diesen Fällen können Sie den Befehl `docker port` nutzen, um die von Docker belegten Ports herauszufinden.

4.4 Container verlinken

Docker-*Links* sind die einfachste Möglichkeit, Container auf dem gleichen Host miteinander reden lassen zu können. Nutzen Sie das Standardnetzwerk-Modell von Docker, geschieht die Kommunikation zwischen Containern über ein internes Docker-Netzwerk, so dass sie nicht im Host-Netzwerk erreichbar ist.

Änderungen beim Verknüpfen von Docker-Containern

Seit der Version 1.12 erstellt man ein Containernetzwerk durch das »Veröffentlichen von Services«, statt sie direkt miteinander zu verknüpfen. Die Links werden aber noch eine ganze Zeit weiter funktionieren, und die Beispiele in diesem Buch sollten ohne Änderungen nachstellbar sein.

Mehr Informationen zu den zukünftigen Änderungen finden Sie in Abschnitt 11.4.

Links werden über das Argument `--link CONTAINER:ALIAS` des Befehls `docker run` initialisiert, wobei `CONTAINER` der Name des Link-Containers[5] und `ALIAS` ein lokaler Name ist, der innerhalb des Master-Containers verwendet wird, um den Link-Container zu referenzieren.

Mit Docker-Links werden der Alias und die ID des Link-Containers in die */etc/hosts* des Master-Containers eingetragen, so dass der Link-Container aus dem Master-Container über den Namen angesprochen werden kann.

Zusätzlich setzt Docker eine Reihe von Umgebungsvariablen innerhalb des Master-Containers, die dazu gedacht sind, die Kommunikation mit dem Link-Container zu vereinfachen. So sieht das zum Beispiel aus, wenn wir einen Redis-Container erzeugen und darauf verlinken würden:

```
$ docker run -d --name myredis redis
c9148dee046a6fefac48806cd8ec0ce85492b71f25e97aae9a1a75027b1c8423
$ docker run --link myredis:redis debian env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=f015d58d53b5
REDIS_PORT=tcp://172.17.0.22:6379
REDIS_PORT_6379_TCP=tcp://172.17.0.22:6379
REDIS_PORT_6379_TCP_ADDR=172.17.0.22
REDIS_PORT_6379_TCP_PORT=6379
REDIS_PORT_6379_TCP_PROTO=tcp
REDIS_NAME=/distracted_rosalind/redis
REDIS_ENV_REDIS_VERSION=3.0.3
REDIS_ENV_REDIS_DOWNLOAD_URL=http://download.redis.io/releases/red
is-3.0.3.tar.gz
```

5. In diesem Buch werde ich den Container, der verlinkt wird, als *Link-Container* und den zu startenden Container als *Master-Container* bezeichnen (da letzterer verantwortlich für das Einrichten des Links ist).

```
REDIS_ENV_REDIS_DOWNLOAD_SHA1=0e2d7707327986ae652df717059354b358b8
3358
HOME=/root
```

Wir sehen hier, dass Docker Umgebungsvariablen eingerichtet hat, die mit `REDIS_PORT` beginnen und Informationen für das Verbinden mit dem Container enthalten. Die meisten davon scheinen mehr oder weniger redundant zu sein, da die Informationen in den Werten schon im Variablennamen enthalten sind. Aber immerhin stellen sie wenigstens eine Form von Dokumentation dar.

Docker hat auch Umgebungsvariablen aus dem verlinkten Container importiert, die mit dem Präfix `REDIS_ENV` beginnen. Diese Funktionalität kann zwar nützlich sein, es ist aber wichtig, sich ihrer bewusst zu sein, wenn Sie Umgebungsvariablen verwenden, in denen geheim zu haltende Daten gespeichert sind, wie zum Beispiel API-Token oder Datenbank-Passwörter.

Standardmäßig können Container miteinander reden – unabhängig davon, ob sie explizit verlinkt sind oder nicht. Wollen Sie verhindern, dass nicht verlinkte Container miteinander kommunizieren, verwenden Sie die Argumente `--icc=false` und `--iptables`, wenn Sie den Docker Daemon starten. Wenn nun Container verlinkt werden, richtet Docker Iptables-Regeln ein, damit die Container über Ports kommunizieren können, die nach außen veröffentlicht werden.

Leider haben Docker-Links aktuell einige Nachteile. Der vielleicht wichtigste ist, dass sie statisch sind – Links sollten zwar den Neustart eines Containers überleben, sie werden aber nicht aktualisiert, wenn der verlinkte Container ersetzt wird. Zudem muss der Link-Container vor dem Master-Container gestartet werden, was zur Folge hat, dass Sie keine bidirektionalen Links nutzen können.

Weitere Informationen zum Vernetzen von Containern finden Sie in Kapitel 11.

4.5 Daten mit Volumes und Datencontainern verwalten

Zur Erinnerung: Docker-Volumes sind Verzeichnisse,[6] die nicht Teil des UFS des Containers sind (siehe »Images, Container und das Union File System«, Seite 25), sondern bei denen es sich um normale Verzeichnisse auf dem Host handelt, die in den Container *bind-mounted* sind (siehe »Bind Mounting«, Seite 55).

Es gibt drei[7] verschiedene Möglichkeiten, Volumes zu initialisieren, und es ist wichtig, die Unterschiede zu kennen. Ein Weg ist, ein Volume zur Laufzeit mit dem Flag `-v` zu deklarieren:

```
$ docker run -it --name container-test -h CONTAINER \
        -v /data debian /bin/bash
```

6. Technisch gesehen sind es Verzeichnisse oder Dateien, da ein Volume auch aus einer einzelnen Datei bestehen kann.
7. Okay, zweieinhalb, abhängig davon, wie Sie zählen wollen.

```
root@CONTAINER:/# ls /data
root@CONTAINER:/#
```

Damit wird das Verzeichnis */data* im Container zu einem Volume. Jegliche Dateien, die das Image darin enthält, werden in das Volume kopiert. Wir können mit `docker inspect` in einer neuen Shell herausfinden, wo das Volume auf dem Host zu Hause ist:

```
$ docker inspect -f {{.Mounts}} container-test
[{5cad... /mnt/sda1/var/lib/docker/volumes/5cad.../_data /data local true}]
```

In diesem Fall ist das Volume */data/* im Container einfach ein Link auf das Verzeichnis */var/lib/docker/volumes/5cad.../_data* auf dem Host. Um das zu beweisen, können wir eine Datei im entsprechenden Host-Verzeichnis anlegen:[8]

```
$ sudo touch /var/lib/docker/volumes/5cad.../_data/test_file
```

Sie sollten die Datei nun direkt im Container sehen können:

```
$ root@CONTAINER: ls /data
test-file
```

Die zweite Möglichkeit, ein Volume einzurichten, ist der Einsatz von `VOLUME` in einem Dockerfile:

```
FROM debian:wheezy
VOLUME /data
```

Das hat den gleichen Effekt wie die Angabe von `-v /data` bei `docker run`.

Die dritte[9] Möglichkeit besteht darin, beim Argument `-v` von `docker run` ein explizites Verzeichnis anzugeben, mit dem das Volume auf dem Host gebunden werden soll. Dabei verwenden Sie das Format `-v HOST_DIR:CONTAINER_DIR`. Das lässt sich in einem Dockerfile nicht umsetzen (weil man sich damit Portabilitäts- und Sicherheitsprobleme einfangen würde). Zum Beispiel:

```
$ docker run -v /home/adrian/data:/data debian ls /data
```

Damit wird das Verzeichnis */home/adrian/data* auf dem Host als */data* im Container gemountet. Alle Dateien, die schon in */home/adrian/data* vorhanden sind, werden innerhalb des Containers zur Verfügung stehen. Existierte das Verzeichnis */data* schon im Container, wird dessen Inhalt durch das Volume verborgen. Anders als bei den anderen Aufrufformen werden hier keine Dateien aus dem Image in das Volume kopiert, und das Volume wird nicht durch Docker gelöscht

8. Sind Sie mit einem Docker Daemon auf einem entfernten Rechner verbunden, müssen Sie diesen Befehl auf dem Remote-Host via SSH ausführen. Nutzen Sie Docker Machine (was Sie tun, wenn Sie Docker über die Docker Toolbox installiert haben), können Sie das mit `docker-machine ssh default` erreichen.
9. Wirklich eine dritte?

(`docker rm -v` wird also kein Volume entfernen, das als ein vom Anwender ausgewähltes Verzeichnis gemountet ist).

Volume-Berechtigungen in Dockerfiles setzen

Sie werden auf einem Volume häufig Berechtigungen und Eigentümer setzen oder ein Volume mit Standarddaten oder Konfigurationsdateien vorbefüllen müssen. Entscheidend ist hier, dass Anweisungen *nach* der `VOLUME`-Anweisung in einem Dockerfile *keine* Änderungen an diesem Volume vornehmen können. So wird zum Beispiel das folgende Dockerfile nicht wie erwartet funktionieren:

```
FROM debian:wheezy
RUN useradd foo
VOLUME /data
RUN touch /data/x
RUN chown -R foo:foo /data
```

Wir wollen hier, dass die Befehle `touch` und `chown` auf dem Dateisystem des Image laufen, aber in Wirklichkeit laufen sie innerhalb des Volume eines temporären Containers, mit dem die Schicht erstellt wird (siehe dazu weiter oben Abschnitt 4.2). Das Volume wird entfernt, sobald der Befehl abgeschlossen ist, und die Anweisungen waren überflüssig.

Das folgende Dockerfile wird hingegen funktionieren:

```
FROM debian:wheezy
RUN useradd foo
RUN mkdir /data && touch /data/x
RUN chown -R foo:foo /data
VOLUME /data
```

Wird aus diesem Image ein Container gestartet, wird Docker alle Dateien aus dem Volume-Verzeichnis im Image in das Volume des Containers kopieren. Das passiert nicht, wenn Sie ein Host-Verzeichnis für das Volume angeben (so dass Host-Dateien nicht unabsichtlich überschrieben werden).

Wenn Sie aus irgendeinem Grund in einer `RUN`-Anweisung keine Berechtigungen und keinen Eigentümer setzen können, müssen Sie das mit einem `CMD`- oder `ENTRYPOINT`-Skript machen, das nach dem Erstellen eines Containers abläuft.

Bind Mounting

Wird ein bestimmtes Host-Verzeichnis in einem Volume verwendet (mit der Syntax `-v HOST_DIR:CONTAINER_DIR`), wird dies häufig als *Bind Mounting* bezeichnet. Das ist etwas irreführend, da alle Volumes technisch gesehen bind-mounted werden. Der Unterschied liegt eher darin, dass der Mount Point explizit gewählt wurde, statt in einem von Docker ausgewählten Verzeichnis versteckt zu sein.

4.5.1 Daten gemeinsam nutzen

Die Syntax `-v HOST_DIR:CONTAINER_DIR` ist sehr nützlich, um Dateien zwischen dem Host und einem oder mehreren Containern gemeinsam zu nutzen. So können zum Beispiel Konfigurationsdateien auf dem Host gehalten und in die Container eingebunden werden, die aus generischen Images erzeugt wurden.

Wir können Daten auch zwischen Containern teilen, indem wir `docker run` mit `--volumes-from CONTAINER` verwenden. So lässt sich zum Beispiel wie folgt ein neuer Container erzeugen, der Zugriff auf die Volumes des Containers in unserem vorigen Beispiel hat:

```
$ docker run -it -h NEWCONTAINER \
        --volumes-from container-test debian /bin/bash
root@NEWCONTAINER:/# ls /data
test-file
root@NEWCONTAINER:/#
```

Es ist wichtig, darauf hinzuweisen, dass dies auch dann funktioniert, wenn der Container, der die Volumes verwaltet (hier `container-test`) aktuell gar nicht läuft. Solange zumindest ein bestehender Container auf ein Volume verweist, wird dieses nicht gelöscht.

4.5.2 Datencontainer

Häufig werden *Datencontainer* erstellt – Container, deren einziger Zweck das gemeinsame Nutzen von Daten durch andere Container ist. Der größte Vorteil dieses Ansatzes ist, dass man so einen praktischen Namensraum für Volumes besitzt, die sich leicht mit dem Befehl `--volumes-from` laden lassen.

So können wir zum Beispiel mit folgender Anweisung einen Datencontainer für eine PostgreSQL-Datenbank erstellen:

```
$ docker run --name dbdata postgres echo "Postgres-Datencontainer"
```

Damit wird aus dem Image `postgres` ein Container erzeugt und die im Image definierten Volumes werden initialisiert, bevor der `echo`-Befehl ausgeführt und der Container wieder beendet wird.[10] Es ist nicht notwendig, den Container laufen zu lassen, weil das nur Ressourcen verschwenden würde.

Wir können dieses Volume nun in anderen Containern über das Argument `--volumes-from` verwenden. Zum Beispiel:

```
$ docker run -d --volumes-from dbdata --name db1 postgres
```

10. Wir könnten hier einen beliebigen Befehl nutzen, der sofort wieder endet, aber die `echo`-Meldung erinnert uns gleichzeitig noch an den Zweck des Containers, wenn wir `docker ps -a` ausführen. Eine andere Möglichkeit ist, den Container gar nicht zu starten, indem wir `docker create` statt `docker run` verwenden.

Images aus Datencontainern

Es ist normalerweise nicht notwendig, ein »minimales Image« wie `busybox` oder `scratch` für den Datencontainer zu verwenden. Nutzen Sie einfach das gleiche Image, das auch für die Container eingesetzt wird, welche die Daten verwenden. Greifen Sie zum Beispiel auf `postgres` zurück, um einen Datencontainer zu erstellen, der zusammen mit der Postgres-Datenbank verwendet wird.

Der Einsatz des gleichen Image erfordert keinen zusätzlichen Platz – Sie müssen das Image für den eigentlichen Anwendungsfall sowieso heruntergeladen oder erstellt haben. Auch hat das Image so die Möglichkeit, den Container mit initialen Daten auszustatten und sicherzustellen, dass die Berechtigungen korrekt eingerichtet sind.

Volumes löschen

Volumes werden nur gelöscht, wenn:

- der Container mit `docker rm -v` gelöscht wird *oder*
- `docker run` mit dem Flag `--rm` aufgerufen wurde

und

- kein bestehender Container auf das Volume verweist sowie
- kein Host-Verzeichnis für das Volume angegeben wurde (also die Syntax `-v HOST_DIR:CONTAINER_DIR` nicht zum Einsatz kam).

Volume-Befehl und Plugins

Seit der Version 1.9 existiert das Kommando `docker volume` zur Verwaltung von Volumes auf einem Docker Host. Man kann damit verschiedene Volume-Driver-Dateisysteme für Container bereitstellen.[11] Ein Volume kann nun auf einem Host angelegt werden und verschiedenen Container bereitgestellt werden. Volumes können einheitlich mit diesen Befehlen verwaltet werden. Wenn keine Default-Dateien auf dem Volume benötigt werden, kann auf einen separaten Datencontainer verzichtet werden. Mit diesem Schritt können nun verschiedene Dateisysteme und Optionen effizient in Containern genutzt werden. Mit dem Docker Release 1.12 können die Erweiterung eines Docker Host auch mit dem Befehl `docker plugin` verwaltet werden.

```
$ docker volume create --drvier local --name dbdata_postgres
$ docker run -d -v dbdata_postgress:/var/lib/postgresql/data postgres
```

Bei Redaktionsschluss des englische Originals dieses Buches bedeutete das, dass Sie sehr genau darauf achten mussten, Ihre Container genau so laufen zu lassen, da Sie

11. *https://docs.docker.com/engine/extend/plugins*
https://docs.docker.com/engine/reference/commandline/volume_create

sonst sehr wahrscheinlich verwaiste Dateien und Verzeichnisse im Installationsverzeichnis von Docker erhielten – und es schwierig war, herauszufinden, wofür sie noch gedacht waren. Docker arbeitete zu dem Zeitpunkt allerdings schon an einem »volume«-Befehl, der es Ihnen jetzt ermöglicht, Volumes unabhängig von Containern aufzuführen, zu erstellen, zu untersuchen und zu entfernen.

4.6 Häufig eingesetzte Docker-Befehle

Dieser Abschnitt bietet Ihnen einen kurzen (zumindest verglichen mit der offiziellen Dokumentation), nicht vollständigen Überblick über die verschiedenen Docker-Befehle, wobei ich mich auf diejenigen konzentriere, die im täglichen Einsatz benötigt werden. Da sich Docker immer noch deutlich ändert und weiterentwickelt, schauen Sie sich auch die offizielle Dokumentation auf der Docker-Website unter *https://docs.docker.com* an, um über alle Details und die neuesten Änderungen informiert zu sein. Ich habe bei den Befehlen nicht alle Argumente und die Syntax im Detail beschrieben (außer bei `docker run`). Rufen Sie dazu die eingebaute Hilfe auf, die Sie über das Argument `--help` bei einem Befehl oder über den Befehl `docker help` erreichen.

Boolesche Flags in Docker

Bei den meisten Befehlszeilentools von Unix werden Sie Flags ohne Werte finden, wie zum Beispiel das `-l` bei `ls -l`. Da diese Flags entweder gesetzt oder nicht gesetzt werden, betrachtet Docker sie als *Boolesche* Flags und unterstützt – anders als die meisten anderen Tools – die explizite Angabe eines Wertes (zum Beispiel werden sowohl `-f=true` als auch `-f` akzeptiert). Zudem (und hier wird es unübersichtlich) können Sie Flags haben, die standardmäßig auf true gesetzt sind *(Default-True-Flags)*, und solche, die standardmäßig false sind *(Default-False-Flags)*. Anders als Default-False-Flags werden Default-True-Flags als gesetzt angesehen, wenn sie nicht angegeben wurden. Das Angeben eines Flags ohne ein Argument hat den gleichen Wert wie das Setzen auf true – ein Default-True-Flag wird also *nicht* zurückgesetzt, indem man den Wert weglässt, sondern nur durch explizites Setzen auf false (zum Beispiel `-f=false`).

Um herauszufinden, ob ein Flag standardmäßig true oder false ist, schauen Sie in der eingebauten Docker-Hilfe danach. Zum Beispiel:

```
$ docker logs --help
...
  -f, --follow=false       Follow log output
  --help=false             Print usage
  -t, --timestamps=false   Show timestamp
...
```

Dies zeigt, dass `-f`, `--help` und `-t` Default-False-Flags sind.

Noch zwei weitere Beispiele: Schauen Sie sich das Argument `--sig-proxy` von `docker run` an. Die einzige Möglichkeit, dieses Argument abzuschalten, ist das explizite Setzen auf false. Zum Beispiel:

```
$ docker run --sig-proxy=false ...
```

Alle folgenden Zeilen bewirken das Gleiche:

```
$ docker run --sig-proxy=true ...
$ docker run --sig-proxy ...
$ docker run ...
```

Bei einem Default-False-Flag wie `--read-only` bewirken die folgenden beiden Zeilen, dass es auf true gesetzt wird:

```
$ docker run --read-only=true
$ docker run --read-only
```

Um das Flag auf false zu setzen, gibt man dies explizit an oder lässt es einfach weg.

Das führt zu manch seltsamem Verhalten bei Flags, die Logik-Abkürzungen nehmen (so funktioniert zum Beispiel `docker ps --help=false` wie üblich, ohne dass die Hilfetexte ausgegeben werden).

4.6.1 Der Befehl run

Wir haben `docker run` schon im Einsatz gesehen – es ist der Befehl zum Starten neuer Container. Dabei handelt es sich auch um den bei weitem komplexesten Befehl, und er unterstützt eine lange Liste möglicher Argumente. Diese ermöglichen es dem Anwender, zu konfigurieren, wie das Image laufen soll, Dockerfile-Einstellungen zu überschreiben, Verbindungen zu konfigurieren und Berechtigungen und Ressourcen für den Container zu setzen.

Die folgenden Optionen steuern den Lebenszyklus des Containers und seinen grundsätzlichen Arbeitsmodus:

`-a, --attach`
: Verbindet den angegebenen Stream (zum Beispiel `STDOUT`) mit dem Terminal. Wird kein Stream angegeben, werden sowohl `stdout` als auch `stderr` angebunden. Wird der Container zudem im interaktiven Modus gestartet (`-i`), wird auch `stdin` angebunden.

  Ist inkompatibel mit `-d`.

`-d, --detach`
: Lässt den Container im »Detached«-Modus laufen. Der Befehl führt den Container im Hintergrund aus und gibt die Container-ID zurück.

`-i, --interactive`
: Lässt `stdin` offen (auch wenn es nicht angebunden ist). Wird im Allgemeinen zusammen mit `-t` genutzt, um eine interaktive Containersitzung zu starten. Zum Beispiel:

```
$ docker run -it debian /bin/bash
root@bd0f26f928bb:/# ls
... Schnipp ...
```

`--restart`
: Hiermit wird festgelegt, wie Docker versucht, einen gestoppten Container neu zu starten. Mit dem Argument `no` wird nie versucht, einen Container neu zu starten, mit `always` wird es immer versucht – unabhängig vom Exit-Status. Das Argument `on-failure` versucht, Container neu zu starten, die *mit einem Nonzero-Status enden*. Mit einem optionalen Argument lässt sich angeben, wie häufig versucht wird, neu zu starten, bevor Docker aufgibt (ohne den Parameter wird unbegrenzt oft versucht, neu zu starten). So wird zum Beispiel `docker run --restart on-failure:10 postgres` den Postgres-Container starten und bis zu zehn Mal versuchen, ihn neu zu starten, falls er mit einem Nonzero-Code endet.[12]

`--rm`
: Entfernt den Container automatisch, wenn er endet. Kann nicht zusammen mit `-d` verwendet werden.

`-t, --tty`
: Allokiert ein Pseudo-TTY. Wird normalerweise zusammen mit `-i` genutzt, um einen interaktiven Container zu starten.

Die folgenden Optionen erlauben es, Namen und Variablen von Containern zu setzen:

`-e, --env`
: Setzt Umgebungsvariablen innerhalb des Containers. Zum Beispiel:

```
$ docker run -e var1=val -e var2="val 2" debian env PATH=/usr/local/sbin:/
usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin HOSTNAME=b15f833d65d8
var1=val
var2=val 2
HOME=/root
```

: Schauen Sie sich auch die Option `--env-file` an, um Variablen über eine Datei zu übergeben.

`-h, --hostname`
: Setzt den Unix-Hostnamen des Containers auf `NAME`. Zum Beispiel:

```
$ docker run -h "myhost" debian hostname
myhost
```

`--name NAME`
: Weist dem Container den Namen `NAME` zu. Der Name kann genutzt werden, um den Container in anderen Docker-Befehlen zu referenzieren.

Die folgenden Container erlauben es dem Anwender, Volumes einzurichten (in Abschnitt 4.5 finden Sie weitere Informationen):

12. Mit der Option `unless-stopped` wird ein Restart nur durchgeführt, wenn der Container nicht vorher manuell gestoppt wurde.

`-v, --volume`
: Es gibt zwei Formen dieses Arguments, um ein Volume einzurichten (eine Datei oder ein Verzeichnis in einem Container, die/das Teil des Host-Systems ist und nicht zum Union File System des Containers gehört). Die erste Form gibt nur das Verzeichnis im Container an und bindet an ein Host-Verzeichnis, das von Docker ausgewählt wird. Die zweite Form gibt auch das Host-Verzeichnis an, an das gebunden werden soll.[13]

`--volumes-from`
: Mountet Volumes aus dem angegebenen Container. Wird häufig zusammen mit Datencontainern verwendet (siehe Abschnitt 4.5.2).

Es gibt eine Reihe von Optionen für die Zusammenarbeit zwischen Containern. Die am häufigsten genutzten Befehle dazu sind:

`--expose`
: Äquivalent zur Dockerfile-Anweisung `EXPOSE`. Gibt den Port oder Portbereich an, der im Container genutzt wird, öffnet diesen aber nicht. Diese Option ist nur zusammen mit `-P` oder beim Verlinken von Containern nützlich.

`--link`
: Richtet eine private Netzwerkschnittstelle zum angegebenen Container ein. In Abschnitt 4.4 finden Sie mehr Informationen dazu.

`-p, --publish`
: »Veröffentlicht« einen Port auf dem Container und macht ihn vom Host aus erreichbar. Wird der Host-Port nicht angegeben, wählt Docker zufällig einen Port im oberen Bereich. Dieser lässt sich dann durch den Befehl `docker port` ermitteln. Die Host-Schnittstelle, für die der Port veröffentlicht werden soll, lässt sich ebenfalls angeben.

`-P, --publish-all`
: Veröffentlicht alle mit `--expose` oder `EXPOSE` bereitgestellten Ports des Containers auf dem Host. Für jeden dieser Ports wird ein zufälliger Port im oberen Bereich ausgewählt. Mit dem Befehl `docker port` lässt sich das Mapping ermitteln.

Es gibt noch eine Reihe weiterer Optionen, die hilfreich sein können, wenn Sie sich intensiver mit dem Verknüpfen und Freigeben von Containern befassen. Achten Sie darauf, dass Sie für viele dieser Optionen Netzwerkkenntnisse haben müssen und Ihnen auch klar sein muss, wie das Ganze in Docker implementiert ist. Mehr Informationen dazu finden Sie in Kapitel 11.

Zum Befehl `docker run` gibt es noch einen Haufen weiterer Optionen, mit denen sich die Berechtigungen und Fähigkeiten von Containern steuern lassen. In

13. Seit Docker 1.9 gibt es die weitere Option, Volumes durch verschiedene Plugins bereitzustellen und in einen Container einzubinden.

Kapitel 13 finden Sie weitere Details dazu.

Die folgenden Optionen überschreiben direkt Dockerfile-Einstellungen:

`--entrypoint`
: Setzt den Entrypoint für den Container auf das angegebene Argument und überschreibt damit die `ENTRYPOINT`-Anweisungen im Dockerfile.

`-u, --user`
: Setzt den Benutzer, unter dem Befehle ausgeführt werden. Kann als Benutzername oder UID angegeben werden. Überschreibt `USER`-Anweisungen im Dockerfile.

`-w, --workdir`
: Setzt das Arbeitsverzeichnis im Container auf den angegebenen Pfad. Überschreibt einen entsprechenden Wert aus dem Dockerfile.

4.6.2 Container verwalten

Neben `docker run` werden folgende Docker-Befehle genutzt, um Container während ihres Lebenszyklus zu verwalten:

`docker attach [OPTIONS] CONTAINER`
: Der Befehl `attach` ermöglicht es dem Benutzer, den Hauptprozess im Container anzeigen zu lassen oder mit ihm zu interagieren. Zum Beispiel:

```
$ ID=$(docker run -d debian sh -c \
        "while true; do echo 'tick'; sleep 1; done;")
$ docker attach $ID
tick
tick
tick
tick
```

Beachten Sie, dass der Prozess mit Strg-C beendet wird und der Container damit auch stoppt.

`docker create`
: Erzeugt einen Container aus einem Image, ohne ihn zu starten. Erwartet mehr oder weniger die gleichen Argumente wie `docker run`. Um den Container zu starten, nutzen Sie `docker start`.

`docker cp`
: Kopiert Dateien und Verzeichnisse zwischen einem Container und seinem Host.

`docker exec`
: Führt einen Befehl innerhalb eines Containers aus. Kann genutzt werden, um Wartungsaufgaben durchzuführen oder als Ersatz für `ssh`, um sich an einem Container anzumelden.

Zum Beispiel:

```
$ ID=$(docker run -d debian sh -c "while true; sleep 1; done;")
$ docker exec $ID echo "Hallo"
Hallo
$ docker exec -it $ID /bin/bash
root@5c6c32041d68:/# ls
bin   dev  home  lib64  mnt  proc  run   selinux  sys  usr
boot  etc  lib   media  opt  root  sbin  srv    tmp  var
root@5c6c32041d68:/# exit
exit
```

`docker kill`

Schickt ein Signal an den Hauptprozess (PID 1) in einem Container. Standardmäßig wird `SIGKILL` gesendet, womit der Container sofort stoppt. Alternativ kann das Signal auch mit dem Argument `-s` angegeben werden. Es wird die Container-ID zurückgegeben.

Zum Beispiel:

```
$ ID=$(docker run -d debian bash -c \
    "trap 'echo caught' SIGTRAP; while true; do sleep 1; done;")
$ docker kill -s SIGTRAP $ID
e33da73c275b56e734a4bbbefc0b41f6ba84967d09ba08314edd860ebd2da86c
$ docker logs $ID
caught
$ docker kill $ID
e33da73c275b56e734a4bbbefc0b41f6ba84967d09ba08314edd860ebd2da86c
```

`docker pause`

Lässt alle Prozesse im angegebenen Container pausieren. Sie empfangen kein Signal, dass sie pausiert werden, daher können sie sich auch nicht herunterfahren oder aufräumen. Mit `docker unpause` lassen sie sich wieder starten. `docker pause` nutzt intern die Freezer-Funktion von Linux-cgroups. Im Gegensatz dazu steht der Befehl `docker stop`, der die Prozesse stoppt und Signale schickt, die von ihnen empfangen werden können.

`docker restart`

Startet einen oder mehrere Container neu. Entspricht in etwa dem Aufruf von `docker stop`, gefolgt von `docker start` für die Container. Ein optionales Argument `-t` gibt an, wie lange auf das Herunterfahren der Container gewartet werden soll, bis sie mit einem `SIGTERM` abgeschossen werden.

`docker rm`

Entfernt einen oder mehrere Container. Gibt die Namen oder IDs erfolgreich gelöschter Container zurück. Standardmäßig entfernt `docker rm` keine Volumes. Mit dem Argument `-f` lassen sich auch laufende Container entfernen, und mit `-v` werden von den Containern erzeugte Volumes ebenfalls gelöscht (solange sie nicht bind-mounted sind oder von einem anderen Container verwendet werden).

So werden zum Beispiel alle gestoppten Container gelöscht:

```
$ docker rm $(docker ps -aq)
b7a4e94253b3
e33da73c275b
f47074b60757
```

`docker start`
: Startet einen (oder mehrere) gestoppten Container. Kann genutzt werden, um einen Container neu zu starten, der beendet wurde, oder um einen Container zu starten, der mit `docker create` erzeugt, aber nie gestartet wurde.

`docker stop`
: Stoppt einen oder mehrere Container (ohne sie zu entfernen). Nach dem Aufruf von `docker stop` für einen Container wird er in den Status »exited« überführt. Ein optionales Argument `-t` gibt an, wie lange darauf gewartet werden soll, bis der Container heruntergefahren ist, bevor er mit einem `SIGTERM` abgeschossen wird.

`docker unpause`
: Startet einen Container neu, der zuvor mit `docker pause` pausiert wurde.

Von einem Container trennen

Sind Sie mit einem Docker-Container verbunden (attached) – entweder über den interaktiven Modus oder durch ein Verbinden mit `docker attach` –, werden Sie den Container stoppen, wenn Sie versuchen, mit Strg-C die Verbindung aufzulösen. Stattdessen können Sie sich mit Strg-P Strg-Q vom Container trennen, ohne ihn zu stoppen.

Das funktioniert allerdings nur, wenn Sie im interaktiven Modus mit einem TTY mit dem Container verbunden sind (also wenn Sie sowohl `-i` als auch `-t` verwenden).

4.6.3 Docker-Info

Die folgenden Unterbefehle können genutzt werden, um mehr Informationen über die Docker-Installation und deren Einsatz zu erhalten:

`docker info`
: Gibt eine Reihe von Informationen über das Docker-System und den Host aus.

`docker help`
: Gibt Informationen zum Einsatz und zur Hilfe rund um den angegebenen Unterbefehl aus. Sie können stattdessen auch einen Befehl mit dem Flag `--help` aufrufen.

`docker version`
: Gibt Informationen über die Docker-Version für den Client und Server aus, aber auch über die Go-Version, die im Einsatz ist.

4.6.4 Container-Info

Die folgenden Befehle liefern weitere Informationen zu laufenden und gestoppten Containern.

`docker diff`
: Gibt die Änderungen am Dateisystem des Containers verglichen mit dem Image aus, aus dem er gestartet wurde. Zum Beispiel:

```
$ ID=$(docker run -d debian touch /NEW-FILE)
$ docker diff $ID
A /NEW-FILE
```

`docker events`
: Gibt Realtime-Events des Daemon aus. Mit Strg-C beenden Sie die Ausgabe. Mehr Informationen dazu finden Sie in Kapitel 10.

`docker inspect`
: Gibt umfangreiche Informationen zu Containern oder Images aus. Dazu gehören die meisten Konfigurationsoptionen und Netzwerkeinstellungen sowie Volumes-Mappings. Dem Befehl kann das Argument `-f` mitgegeben werden, um ein Go-Template zum Formatieren und Filtern der Ausgabe anzuwenden.

`docker logs`
: Gibt die »Logs« für einen Container aus. Dabei handelt es sich einfach um alles, was innerhalb des Containers nach `STDERR` oder `STDOUT` geschrieben wurde. Mehr Informationen zum Protokollieren in Docker finden Sie in Kapitel 10.

`docker port`
: Gibt die freigegebenen Port-Mappings für den angegebenen Container aus. Dem Befehl können optional der interne Container-Port und das Protokoll, nach dem er schauen soll, mitgegeben werden,. Wird häufig nach `docker run -P <image>` genutzt, um die zugewiesenen Ports herauszufinden.

  Zum Beispiel:

```
$ ID=$(docker run -P -d redis)
$ docker port $ID
6379/tcp -> 0.0.0.0:32768
$ docker port $ID 6379
0.0.0.0:32768
$ docker port $ID 6379/tcp
0.0.0.0:32768
```

`docker ps`

Gibt einen Überblick über die aktuellen Container, wie zum Beispiel Namen, IDs und Status. Es stehen viele verschiedene Argumente zur Verfügung, unter anderem `-a`, um alle Container zu erhalten und nicht nur die aktuell laufenden. Mit `-q` werden nur die Container-IDs zurückgegeben, was sehr nützlich als Eingabe für andere Befehle wie `docker rm` ist.

`docker top`

Gibt Informationen zu den laufenden Prozessen in einem angegebenen Container aus. Intern wird bei diesem Befehl das UNIX-Tool `ps` auf dem Host ausgeführt und dessen Ausgabe auf Prozesse im angegebenen Container gefiltert. Dem Befehl können die gleichen Argumente mitgegeben werden wie beim `ps`-Tool, wobei die Standardwerte `-ef` sind (achten Sie aber darauf, sicherzustellen, dass sich das PID-Feld weiterhin in der Ausgabe befindet).

Zum Beispiel:

```
$ ID=$(docker run -d redis)
$ docker top $ID
UID  PID   PPID  C  STIME  TTY  TIME      CMD
999  9243  1836  0  15:44  ?    00:00:00  redis-server *:6379
$ ps -f -u 999
UID  PID   PPID  C  STIME  TTY  TIME      CMD
999  9243  1836  0  15:44  ?    00:00:00  redis-server *:6379
$ docker top $ID -axZ
LABEL           PID   TTY  STAT  TIME  COMMAND
docker-default  9243  ?    Ssl   0:00  redis-server *:6379
```

4.6.5 Arbeit mit Images

Die folgenden Befehle sind Tools für das Erstellen von Images und die Arbeit mit ihnen:

`docker build`

Erstellt ein Image aus einem Dockerfile. In den Abschnitten 3.3 und 4.2 erhalten Sie mehr Informationen dazu.

`docker commit`

Erstellt ein Image aus dem angegebenen Container. Dieser Befehl kann zwar nützlich sein, im Allgemeinen ist es aber besser, Images mithilfe von `docker build` zu erstellen, da sich dies leichter reproduzierbar gestalten lässt. Standardmäßig werden Container vor einem Commit pausiert, das lässt sich aber mit dem Argument `--pause=false` unterdrücken. Mit den Argumenten `-a` und `-m` lassen sich Metadaten setzen.

Zum Beispiel:

```
$ ID=$(docker run -d redis touch /new-file)
$ docker commit -a "Joe Bloggs" -m "Comment" $ID commit:test
ac479108b0fa9a02a7fb290a22dacd5e20c867ec512d6813ed42e3517711a0cf
```

```
$ docker images commit
REPOSITORY  TAG   IMAGE ID      CREATED             VIRTUAL SIZE
commit      test  ac479108b0fa  About a minute ago  111 MB
$ docker run commit:test ls /new-file
/new-file
```

`docker export`

Exportiert den Inhalt des Container-Dateisystems als Tar-Archiv auf `STDOUT`. Das so entstandene Archiv lässt sich mit `docker import` wieder laden. Beachten Sie, dass nur das Dateisystem exportiert wird – Metadaten wie die exportierten Ports, `CMD`- und `ENTRYPOINT`-Einstellungen gehen verloren. Zudem sind im Exportarchiv keine Volumes enthalten. Vergleichen Sie dazu auch `docker save`.

`docker history`

Gibt Informationen zu jeder Schicht in einem Image aus.

`docker images`

Gibt eine Liste lokaler Images aus, wobei Informationen zu Repository-Namen, Tag-Namen und Größe enthalten sind. Standardmäßig sind Zwischen-Images (die beim Erstellen von Top-Level-Images verwendet werden) nicht enthalten. Bei der `VIRTUAL SIZE` handelt es sich um die Gesamtgröße des Image einschließlich aller darunterliegenden Schichten. Da diese Schichten eventuell gemeinsam mit anderen Images genutzt werden, führt ein einfaches Aufsummieren der Größe aller Images nicht zwingend zu einer korrekten Einschätzung des Platzverbrauchs auf der Festplatte. Zudem erscheinen Images auch mehrfach, wenn sie mehr als ein Tag besitzen. Die Images lassen sich dabei über ihre ID unterscheiden. Dieser Befehl kennt eine Reihe von Argumenten. Insbesondere `-q` ist interessant, da damit nur die Image-IDs zurückgegeben werden, was für andere Tools wie `docker rmi` nützlich sein kann.

Zum Beispiel:

```
$ docker images | head -4
REPOSITORY             TAG     IMAGE ID      CREATED  VIRTUAL SIZE
identidock_identidock  latest  9fc66b46a2e6  2 hours ago  839.8 MB
redis                  latest  868be653dea3  6 days ago   110.8 MB
containersol/pre-base  latest  13919d434c95  2 weeks ago  401.8 MB
$ docker rmi $(docker images -q -f dangling=true)
Deleted: a9979d5ace9af55a562b8436ba66a1538357bc2e0e43765b406f2c...
```

`docker import`

Erstellt aus einer Archivdatei mit einem Dateisystem ein Image. Solch ein Archiv kann durch `docker export` erstellt worden sein. Es kann über einen Dateipfad oder eine URL angegeben oder (durch den Einsatz des Flags `-`) von `STDIN` gestreamt werden. Der Befehl gibt die ID des neu erstellten Image zurück. Das Image kann durch Angabe eines Repository- und Tag-Namen getaggt werden. Beachten Sie, dass ein per `docker import` erstelltes Image nur eine Schicht besitzt und keine Docker-Konfigurationseinstellungen wie die

bereitgestellten Ports und `CMD`-Werte besitzt. Vergleichen Sie dazu auch `docker load`.

Ein Beispiel für das »Flachklopfen« eines Image durch Export und Import:

```
$ docker export 35d171091d78 | docker import - flatten:test
5a9bc529af25e2cf6411c6d87442e0805c066b96e561fbd1935122f988086009
$ docker history flatten:test
IMAGE         CREATED         CREATED BY SIZE      COMMENT
981804b0c2b2  59 seconds ago             317.7 MB  Imported from -
```

`docker load`

Lädt ein Repository aus einem von `STDIN` übergebenen Tar-Archiv. Das Repository kann diverse Images und Tags enthalten. Anders als bei `docker import` enthalten die Images ihre Historie und Metadaten. Passende Archivdateien werden durch `docker save` erstellt, womit `save` und `load` eine nützliche Alternative zu Registries sind, um Images zu verteilen und Backups herzustellen. Bei `docker save` finden Sie ein Beispiel.

`docker rmi`

Löscht das oder die angegebenen Images. Diese werden durch ihre ID oder Repository- und Tag-Namen spezifiziert. Wird ein Repository-Name ohne Tag-Namen angegeben, wird für Letzteren `latest` angenommen. Um Images zu löschen, die in mehreren Repositories vorhanden sind, definieren Sie die Images über ihre IDs und nutzen Sie das Argument `-f`. Den Befehl müssen Sie dann ein Mal pro Repository ausführen.

`docker save`

Sichert die angegebenen Images oder Repositories in ein Tar-Archiv, welches nach `STDOUT` gestreamt wird (mit `-o` wird in eine Datei geschrieben). Images können über ihre ID oder als `repository:tag` angegeben werden. Wird nur ein Repository-Name genutzt, werden alle Images in diesem Repository in das Archiv geschrieben und nicht nur das mit dem Tag `latest`. Dieser Befehl lässt sich zusammen mit `docker load` verwenden, um Images zu verteilen oder Backups von ihnen zu erstellen.

Zum Beispiel:

```
$ docker save -o /tmp/redis.tar redis:latest
$ docker rmi redis:latest
Untagged: redis:latest
Deleted: 868be653dea3ff6082b043c0f34b95bb180cc82ab14a18d9d6b8e2...
...
$ docker load -i /tmp/redis.tar
$ docker images redis
REPOSITORY  TAG     IMAGE ID      CREATED       VIRTUAL SIZE
redis       latest  0f3059144681  3 months ago  111 MB
```

`docker tag`

Weist einem Image einen Repository- und Tag-Namen zu. Das Image kann über seine ID oder über einen bestehenden Repository- und Tag-Namen angegeben werden (fehlt der Tag-Name, wird `latest` genutzt). Wird für den neuen Namen kein Tag angegeben, wird `latest` verwendet.

Zum Beispiel:

```
$ docker tag faa2b75ce09a newname ❶
$ docker tag newname:latest amouat/newname ❷
$ docker tag newname:latest amouat/newname:newtag ❸
$ docker tag newname:latest myregistry.com:5000/newname:newtag ❹
```

❶ Fügt das Image mit der ID `faa2b75ce09a` zum Repository `newname` hinzu, wobei als Tag-Name `latest` genutzt wird (weil nichts für ihn angegeben wurde).

❷ Fügt das Image `newname:latest` zum Repository `amouat/newname` hinzu, ebenfalls wieder mit dem Tag-Namen `latest`. Dieses Label kann genutzt werden, um das Image auf den Docker-Hub zu schieben, sofern der Benutzer dort `amouat` ist.

❸ Wie zuvor, nur mit dem Tag `newtag` statt `latest`.

❹ Fügt das Image `newname:latest` zum Repository `myregistry.com/newname` mit dem Tag `newtag` hinzu. Dieses Label hat ein Format, mit dem das Image auf eine Registry unter *http://myregistry.com:5000* geschoben werden kann.

4.6.6 Die Registry verwenden

Die folgenden Befehle dienen der Zusammenarbeit mit Registries, einschließlich des Docker Hub. Beachten Sie, dass Docker die Credentials in Ihrem Home-Verzeichnis in der Datei *.dockercfg* sammelt:

`docker login`

Registriert sich am angegebenen Registry-Server oder meldet sich dort an. Ist kein Server genannt, wird der Docker Hub angenommen. Der Prozess fragt bei Bedarf interaktiv nach Details, sie lassen sich aber auch als Argumente mitgeben.

`docker logout`

Meldet sich von einer Docker Registry ab. Wird kein Server angegeben, meldet sich Docker vom Docker Hub ab.

`docker pull`

Lädt das angegebene Image von einer Registry herunter. Die Registry wird durch den Imagenamen ermittelt und ist standardmäßig der Docker-Hub. Wird kein Tag-Name mitgegeben, wird das Image mit dem Tag `latest` herun-

tergeladen (sofern vorhanden). Mit dem Argument `-a` laden Sie alle Images aus einem Repository herunter.

`docker push`
: Schiebt ein Image oder Repository in die Registry. Wird kein Tag angegeben, werden *alle* Images im Repository in die Registry geschoben, nicht nur das mit dem Tag `latest`.

`docker search`
: Gibt eine Liste öffentlicher Repositories auf dem Docker Hub aus, die zum Suchbegriff passen. Die Ergebnisse werden auf 25 Repositories beschränkt. Sie können auch mit Sternchen und auf automatisierte Builds filtern. Im Allgemeinen ist es einfacher, die Website zu verwenden.

4.7 Zusammenfassung

Das waren jetzt ganz schön viele Informationen! Haben Sie zumindest die wichtigsten Punkte überfliegen können, sollten Sie jetzt einen halbwegs vernünftigen Überblick über die Funktionsweise von Docker und seine wichtigsten Befehle haben. In Teil II werden wir dieses Wissen auf ein Softwareprojekt anwenden und dabei von der Entwicklung bis zur Produktion alle Schritte durchlaufen. Vielleicht hilft Ihnen dieser praxisnähere Ansatz beim Verstehen einiger der Themen aus diesem Kapitel.