KAPITEL 5

Docker-Container verwenden

Im vorherigen Kapitel haben Sie den Vorgang zur Erstellung von Images sowie die grundlegenden Schritte für deren Ausführung in einem Container kennengelernt. In diesem Kapitel sehen wir uns zuerst die Historie der Containertechnologie an. Dann steigen wir tiefer ein, schauen, wie man Container zum Laufen bringt, und erkunden anschließend die Docker-Befehle zur Konfiguration und Verwaltung der Ressourcen und Berechtigungen.

Was sind Container?

Vielleicht sind Sie mit Virtualisierungssystemen wie VMware oder Xen vertraut, die es ermöglichen, einen vollständigen Linux-Kernel auf einem virtuellen Layer auszuführen, der als *Hypervisor* bezeichnet wird. Bei diesem Ansatz sind die virtuellen Maschinen streng voneinander isoliert, da jede virtuelle Maschine einen eigenen Kernel mitbringt, der in einem eigenen Speicherbereich auf einem Hardware-Virtualisierungslayer läuft.

Container verfolgen im Vergleich dazu einen grundlegend anderen Ansatz, bei dem sich alle Container einen einzigen Kernel teilen und die Isolierung vollständig innerhalb dieses einen Kernels implementiert ist. Man spricht hier von einer *Betriebssystemvirtualisierung*.

Im `libcontainer`-Projekt (*https://github.com/opencontainers/runc/blob/main/libcontainer/README.md*) findet sich eine treffende und kompakte Definition für Container:

> Ein Container ist eine eigenständige Ausführungsumgebung, die sich den Kernel mit dem Wirtssystem teilt und (optional) von anderen Containern isoliert ist.

Dieser Ansatz bietet große Vorteile hinsichtlich der Effizienz der Ressourcennutzung, weil nicht für jede einzelne isolierte Funktion ein eigenes Betriebssystem benötigt wird. Da es nur einen Kernel gibt, muss zwischen den isolierten Aufgaben und der echten Hardware ein Layer weniger überbrückt werden. Wenn ein Prozess in einem Container läuft, gibt es nur einen sehr schmalen Isolierungslayer, der Teil des Kernels ist, und es ist nicht erforderlich, Requests an einen zweiten Kernel durchzureichen, wodurch der Prozessor ständig in den privilegierten Modus wechseln müsste.

`libcontainer` (*https://github.com/opencontainers/runc/tree/main/libcontainer*) ist eine Go-Bibliothek, die als Standardschnittstelle für das Managen von Linux-Containern in Anwendungen gedacht ist.

Bei der Verwendung von Containern können allerdings nur Prozesse ausgeführt werden, die mit dem zugrunde liegenden Kernel kompatibel sind. Im Gegensatz zur Hardwarevirtualisierung, wie sie beispielsweise VMware bietet, ist es nicht möglich, Windows-Programme nativ in einem Linux-Container auszuführen. Windows-Anwendungen können allerdings innerhalb von Windows-Containern auf einem Windows-Host laufen. Container sollten also am besten als betriebssystemspezifische Technologie angesehen werden, mit der Sie Ihre bevorzugten Anwendungen oder Daemons ausführen können, sofern sie mit dem Kernel des Containerservers kompatibel sind. Am besten vergessen Sie alles, was Sie über virtuelle Maschinen wissen, und stellen sich einen Container eher als »Verpackung« eines auf dem Server laufenden normalen Prozesses vor.

Es ist nicht nur möglich, Container innerhalb von virtuellen Maschinen laufen zu lassen, sondern auch, eine virtuelle Maschine innerhalb eines Containers laufen zu lassen. Dadurch können Sie eine Windows-Anwendung innerhalb einer Windows-VM laufen lassen, die wiederum innerhalb eines Linux-Containers läuft.

Die Entstehungsgeschichte der Container

Nicht selten handelt es sich bei revolutionären Technologien im Grunde genommen um bereits seit Längerem existierende Technologien, die irgendwann ins Rampenlicht rücken. Technologischer Wandel vollzieht sich im Allgemeinen wellenförmig, und so haben einige der Konzepte aus den 1960er-Jahren heutzutage wieder Konjunktur. Auch Docker verkörpert eine revolutionäre Technologie, die dank ihrer einfachen Anwendbarkeit schnell große Popularität erlangte, aber keinesfalls einfach aus dem Nichts kam. Vielmehr basiert es auf der Arbeit, die in den letzten 30 Jahren in verschiedenen Fachbereichen geleistet wurde. Die konzeptuelle Entwicklung lässt sich leicht von einem einfachen System Call, um den der Unix-Kernel Ende der 1970er erweitert wurde, bis hin zum Container-Tooling verfolgen, das bei vielen großen Internetfirmen wie Google, Twitter und Meta eingesetzt wird. Insofern ist es lohnenswert, sich die Zeit zu nehmen, an dieser Stelle einen kurzen Blick auf den Werdegang von Docker zu werfen, weil es sich dadurch besser mit den Ihnen bereits vertrauten Konzepten in einen Kontext setzen lässt.

Container bieten die Möglichkeit, Teile eines laufenden Systems zu isolieren und zu kapseln – aber sie sind keine ganz und gar neue Erfindung. Die älteste Technologie in diesem Bereich bilden die ersten Stapelverarbeitungssysteme. Sie ließen jeweils ein Programm eine Weile laufen und wechselten erst dann zum nächsten, wenn das vorherige Programm entweder durchgelaufen oder die vorgesehene Zeit abgelaufen war. Dabei ging es vorrangig um das Isolieren, denn so wurde gewährleistet, dass ein Programm dem anderen nicht in die Quere kam, da es nicht möglich war, mehr als

eine Anwendung gleichzeitig laufen zu lassen. Moderne Rechner wechseln immer noch ständig die laufenden Programme, dieser Prozess läuft jedoch so schnell ab, dass der Großteil der Nutzer davon nichts mitbekommt.

Die meisten würden behaupten, dass die Grundlage für die heutigen Container 1979 mit der Ergänzung des `chroot`-System-Calls zum Version-7-Unix geschaffen wurden. Dieser Aufruf beschränkt den Zugang eines Prozesses auf bestimmte Teile des Dateisystems. Er wird üblicherweise genutzt, um das Betriebssystem vor Zugriffen durch nicht vertrauenswürdige Prozesse wie FTP, BIND oder Sendmail zu schützen, die von außen zugänglich sind und kompromittiert werden können.

In den 1980er- und 1990er-Jahren entstanden verschiedene Unix-Varianten (z. B. SELinux), die eine Zugriffssteuerung aus Sicherheitsgründen obligatorisch machten. Damit gab es strikt voneinander abgegrenzte Domänen, die unter demselben Unix-Kernel liefen. Die Prozesse in den verschiedenen Domänen durften nur auf einen extrem eingeschränkten Bereich des Systems zugreifen, was domänenübergreifende Interaktionen verhinderte. Es gab damals zwar eine verbreitete, auf einem BSDI-Unix beruhende kommerzielle Unix-Version namens Sidewinder, die dieses Konzept umsetzte, in den meisten gängigen Unix-Implementierungen war das jedoch nicht möglich.

Dies änderte sich im Jahr 2000 mit dem Release von FreeBSD 4.0, das einen neuen Befehl namens `jail` enthielt, der es Hosting-Providern ermöglichen sollte, ihre eigenen Prozesse einfach und sicher von denen ihrer Kunden zu trennen. `Jail` erweiterte die `chroot`-Fähigkeiten und gab Einschränkungen in Bezug darauf vor, was ein Prozess mit den Systemressourcen anfangen durfte und wie er mit anderen Prozessen kommunizierte.

2004 veröffentlichte Sun eine erste Version von Solaris 10 mit gleichnamigen Containern, die später zu Solaris-*Zonen* weiterentwickelt wurden. Hierbei handelte es sich um die erste kommerzielle Umsetzung der Containertechnologie, auf der noch heute viele Implementierungen beruhen. 2005 wurde OpenVZ für Linux von der Firma Virtuozzo veröffentlicht, und 2007 zog Hewlett-Packard mit *Secure Resource Partitions* für HP-UX nach, das später in *HP-UX-Containers* umbenannt wurde.

Seit Anfang der 2000er-Jahre wird die Containertechnologie insbesondere von jenen Unternehmen vorangetrieben, die mit der internetweiten Skalierung von Anwendungen und/oder dem Hosten von nicht vertrauenswürdigem Usercode befasst sind (Google ist hier ein schönes Beispiel), um die zuverlässige und sichere Verbreitung ihrer Software in den mit zahllosen Computern bestückten Rechenzentren zu fördern. Manche Unternehmen entwickelten eigene, gepatchte Linux-Kernel mit Containersupport für den hausinternen Gebrauch. Da sich in der Linux-Community jedoch allmählich ein wachsender Bedarf an derartigen Features abzeichnete, brachte Google Teile seiner eigenen Arbeiten zum Support von Containern in den Mainline-Kernel ein. 2008 kamen schließlich mit Version 2.6.24 des Linux-Kernels auch Linux-Container (LXC) auf den Markt. Die in phänomenalem Ausmaß wachsende Verbreitung eben dieser Linux-Container in der Community setzte allerdings erst 2013 nach dem Release der Linux-Kernel-Version 3.8 (die Namensräume unterstützt) und bald darauf auch von Docker ein.

Heutzutage kommen Container nahezu überall zum Einsatz. Docker- und OCI-Images sind das Paketformat für immer mehr Software, die in Produktivumgebungen ausgeliefert wird, und sie dienen als Basis für viele Produktivsysteme, unter anderem Kubernetes und die meisten *Serverless-Cloud-Technologien*.

Sogenannte Serverless-Technologien sind nicht wirklich ohne Server – sie nutzen einfach die Server anderer, um Aufgaben zu erledigen, sodass sich der Anwendungsbesitzer nicht um das Managen der Hardware und des Betriebssystems kümmern muss.

Container erstellen

Bislang haben wir Container mit dem praktischen Befehl `docker container run` gestartet. Tatsächlich ist dies aber nur eine bequeme Abkürzung, die zwei verschiedene Schritte zu einem zusammenfasst. Als Erstes erzeugt der Befehl anhand des zugrunde liegenden Image einen Container. Dasselbe können Sie auch mit dem Befehl `docker container create` erreichen. Und die zweite Aufgabe, die `docker container run` erledigt, ist das Starten des Containers – was wiederum ebenso mit dem Befehl `docker container start` möglich ist.

Die Befehle `docker container create` und `docker container run` enthalten jeweils alle Optionen, die festlegen, wie der Container eingerichtet wird. In Kapitel 4 haben wir vorgeführt, wie Sie mit dem Befehl `docker container run` und dem Argument `-p/--publish` Netzwerkports des Containers an den Host binden und mit dem Argument `-e/--env` Umgebungsvariablen an den Container übergeben können.

Damit haben wir allerdings nur an der Oberfläche dessen gekratzt, was bei der Erstellung eines Containers alles konfiguriert werden kann. Sehen wir uns also einige der weiteren Optionen an, die Docker unterstützt.

Grundlegende Konfiguration

Im Folgenden betrachten wir verschiedene Methoden, um Docker anzuweisen, wie ein Container bei seiner Erstellung konfiguriert werden soll.

Bezeichnung des Containers

Die Erzeugung eines Containers erfolgt anhand des zugrunde liegenden Image – die endgültigen Einstellungen können Sie allerdings mithilfe verschiedener Kommandozeilenargumente selbst vornehmen. Grundsätzlich werden die im *Dockerfile* hinterlegten Einstellungen als Standardwerte benutzt, viele davon lassen sich jedoch beim Erzeugen des Containers überschreiben.

Standardmäßig verwendet Docker eine zufällige Kombination aus einem Adjektiv und dem Namen einer bekannten Persönlichkeit zur Benennung Ihrer Container (*https://github.com/moby/moby/blob/master/pkg/namesgenerator/names-generator.go*).

Das führt dann zu Bezeichnungen wie *ecstatic-babbage* (»begeisterter Babbage«) oder *serene-albattani* (»gelassener Albattani«). Wenn Sie lieber selbst einen Namen vergeben möchten, steht Ihnen dazu das Argument `--name` zur Verfügung:

```
$ docker container create --name="awesome-service" ubuntu:latest sleep 120
```

Sobald der Container erzeugt ist, können Sie ihn mittels des Befehls `docker container start awesome-service` starten. Der Container wird sich nach 120 Sekunden automatisch beenden, allerdings können Sie ihn vorher stoppen, indem Sie `docker container stop awesome-service` ausführen. Wir werden an späterer Stelle in diesem Kapitel ein wenig näher auf jedes dieser Kommandos eingehen.

Jeder Containername darf auf einem Docker-Host nur ein einziges Mal vorkommen. Wenn Sie den obigen Befehl zwei Mal nacheinander ausführen, erhalten Sie eine Fehlermeldung. Sie müssen also entweder den vorhandenen Container mit dem Befehl `docker container rm` löschen oder den Namen des neuen Containers ändern.

Tagging mit Label

Wie in Kapitel 4 bereits kurz erwähnt, sind Labels Key-Value-Paare, die als Metadaten auf Docker-Container und -Images angewendet werden können. Beim Erstellen neuer Linux-Container erben diese automatisch alle per Label zugewiesenen Tags ihres jeweiligen Eltern-Image.

Darüber hinaus können Sie Container aber auch mit neuen Labels versehen, um beispielsweise Metadaten zu speichern, die nur einen bestimmten Container betreffen:

```
$ docker container run --rm -d --name has-some-labels \
  -l deployer=Ahmed -l tester=Asako \
  ubuntu:latest sleep 1000
```

Anschließend können Sie mit Befehlen wie `docker container ls` nach diesen Metadaten suchen und sie entsprechend filtern:

```
$ docker container ls -a -f label=deployer=Ahmed
CONTAINER ID  IMAGE         COMMAND       … NAMES
845731631ba4  ubuntu:latest "sleep 1000"  … has-some-labels
```

Um sich alle Labels eines Containers anzeigen zu lassen, steht Ihnen der Befehl `docker container inspect` zur Verfügung:

```
$ docker container inspect has-some-labels
…
    "Labels": {
        "deployer": "Ahmed",
        "tester": "Asako"
    },
…
```

Der Container führt das Kommando `sleep 1000` aus, er wird also nach 1.000 Sekunden gestoppt.

Hostname

Wenn Sie einen Container starten, kopiert Docker standardmäßig bestimmte Systemdateien, unter anderem */etc/hostname*, in das Konfigurationsverzeichnis des Containers auf dem Host (das sich typischerweise im Verzeichnis */var/lib/docker/containers* befindet) und nutzt einen Bind Mount, um die Datei in den Container zu linken. Ein Standardcontainer ohne weitere Konfiguration kann wie folgt gestartet werden:

```
$ docker container run --rm -ti ubuntu:latest /bin/bash
```

Hier wird der Befehl `docker container run` verwendet, der im Hintergrund `docker container create` und `docker container start` ausführt. Da wir zu Demonstrationszwecken mit dem erstellten Container interagieren möchten, übergeben wir dem Befehl noch einige zusätzliche Argumente: Mit `--rm` wird Docker angewiesen, den Container nach dem Beenden zu löschen. Das Argument `-t` sorgt dafür, dass Docker ein Pseudo-TTY benutzt. Das Argument `-i` besagt, dass es sich um eine interaktive Sitzung handelt und dass `STDIN` daher geöffnet bleiben soll. Ist im Image kein `ENTRYPOINT` definiert, ist das letzte Argument des Befehls schließlich das Programm (mit seinen Befehlszeilenargumenten), das innerhalb des Containers ausgeführt werden soll, in diesem Fall die Bash (`/bin/bash`). Ist im Image ein `ENTRYPOINT` definiert, wird das letzte Argument an den `ENTRYPOINT`-Prozess als Liste mit Befehlszeilenargumenten übergeben.

Vielleicht ist Ihnen aufgefallen, dass es im vorherigen Absatz um `-i` und `-t` geht, während der Befehl `-ti` nutzt. Das ist eine lange (Unix-) Geschichte, aber eine kurze Zusammenfassung finden Sie online unter *https://nullprogram.com/blog/2020/08/01*, wenn Sie neugierig sind.

Führt man nun innerhalb des laufenden Containers den Befehl `mount` aus, erhält man eine Ausgabe wie diese:

```
root@ebc8cf2d8523:/# mount
overlay on / type overlay (rw,relatime,lowerdir=…,upperdir=…,workdir…)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev type tmpfs (rw,nosuid,mode=755)
shm on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,size=65536k)
mqueue on /dev/mqueue type mqueue (rw,nosuid,nodev,noexec,relatime)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,…,ptmxmode=666)
sysfs on /sys type sysfs (ro,nosuid,nodev,noexec,relatime)
/dev/sda9 on /etc/resolv.conf type ext4 (rw,relatime,data=ordered)
/dev/sda9 on /etc/hostname type ext4 (rw,relatime,data=ordered)
/dev/sda9 on /etc/hosts type ext4 (rw,relatime,data=ordered)
devpts on /dev/console type devpts (rw,nosuid,noexec,relatime,…,ptmxmode=000)
proc on /proc/sys type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/sysrq-trigger type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/irq type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/bus type proc (ro,nosuid,nodev,noexec,relatime)
tmpfs on /proc/kcore type tmpfs (rw,nosuid,mode=755)
root@ebc8cf2d8523:/#
```

Wenn die Beispiele Eingabeaufforderungen der Form `root@hashID` enthalten, wird dadurch angezeigt, dass ein Befehl nicht auf dem lokalen Host, sondern innerhalb des Containers ausgeführt wird.

Beachten Sie, dass es Fälle gibt, in denen ein Container stattdessen mit einem anderen Hostnamen konfiguriert wurde, etwa wenn man `--name` auf der CLI nutzt. Im Standard ist es allerdings ein Hash der Container-ID.

Es ist auch möglich, mit `--user` den Benutzer zu ändern, der innerhalb des Containers genutzt wird, aber standardmäßig ist es `root`.

Ein Container besitzt zumeist eine ganze Reihe von Bind Mounts, hier ist allerdings insbesondere dieser von Interesse:

```
/dev/sda9 on /etc/hostname type ext4 (rw,relatime,data=ordered)
```

Die Device-Nummer unterscheidet sich zwar von Container zu Container, wir sind aber vielmehr an dem Mountpoint */etc/hostname* interessiert. Er verknüpft */etc/host name* des Containers mit der Hostname-Datei, die Docker für den Container vorbereitet hat. Sie enthält standardmäßig nur die Container-ID und keine volle Domain.

Wir können das innerhalb des Containers mit folgendem Befehl überprüfen:

```
root@ebc8cf2d8523:/# hostname -f
ebc8cf2d8523
root@ebc8cf2d8523:/# exit
```

Denken Sie daran, die Container-Shell mit `exit` zu verlassen, wenn Sie fertig sind, um wieder zum lokalen Host zurückzukehren.

Mit dem Argument `--hostname` kann ein aussagekräftigerer Name übergeben werden:

```
$ docker container run --rm -ti --hostname="mycontainer.example.com" \
    ubuntu:latest /bin/bash
```

Innerhalb des Containers wird nun die volle Domain angezeigt:

```
root@mycontainer:/# hostname -f
mycontainer.example.com
root@mycontainer:/# exit
```

Domain Name Service (DNS)

Ebenso wie */etc/hostname* ist auch die *resolv.conf*-Datei des Containers über einen Bind Mount mit derjenigen des Hosts verknüpft:

```
/dev/sda9 on /etc/resolv.conf type ext4 (rw,relatime,data=ordered)
```

Details zur Datei *resolve.conf* finden Sie online unter *https://sslhow.com/understanding-etc-resolv-conf-file-in-linux*.

Standardmäßig handelt es sich hierbei um eine exakte Kopie der *resolv.conf*-Datei des Hosts. Ist das nicht erwünscht, kann man eine Kombination der Argumente `--dns` und `--dns-search` verwenden, um dieses Verhalten innerhalb des Containers zu ändern:

```
$ docker container run --rm -ti --dns=8.8.8.8 --dns=8.8.4.4 \
    --dns-search=example1.com --dns-search=example2.com \
    ubuntu:latest /bin/bash
```

Wenn Sie der Suchdomäne keinen Wert zuweisen möchten, können Sie dazu `--dns-search=` verwenden (Zuweisung einer leeren Zeichenkette).

Innerhalb des Containers wird nach wie vor ein Bind Mount angezeigt, aber der Dateiinhalt entspricht nicht mehr dem der *resolv.conf*-Datei des Hosts, sondern sieht stattdessen folgendermaßen aus:

```
root@0f887071000a:/# more /etc/resolv.conf
nameserver 8.8.8.8
nameserver 8.8.4.4
search example1.com example2.com
root@0f887071000a:/# exit
```

MAC-Adresse

Zu den weiteren wichtigen konfigurierbaren Daten zählt auch die *MAC-Adresse* (Media-Access-Control-Adresse) des Containers.

Wenn Sie keine spezifische Konfiguration vornehmen, wird dem Container eine berechnete MAC-Adresse zugewiesen, die mit `02:42:ac:11` beginnt.

Möchten Sie hingegen eine bestimmte MAC-Adresse zuweisen, können Sie dazu den folgenden Befehl verwenden:

```
$ docker container run --rm -ti --mac-address="a2:11:aa:22:bb:33" \
  ubuntu:latest /bin/bash
```

Normalerweise ist das nicht erforderlich. Es kann aber Situationen geben, in denen es sinnvoll ist, einen bestimmten Adressbereich für die eigenen Container zu reservieren, um zu verhindern, dass es zu Konflikten mit anderen Virtualisierungslayern kommt, die denselben privaten Adressbereich wie Docker benutzen.

Verwenden Sie benutzerdefinierte MAC-Adressen mit Bedacht. Wenn zwei Systeme dieselbe MAC-Adresse annoncieren, kann es zu Konflikten kommen, die das Netzwerk lahmlegen (»ARP-Contention«). Sollten Sie nicht darauf verzichten können, empfiehlt es sich, für die lokal verwalteten MAC-Adressen nach Möglichkeit ausschließlich solche aus den offiziell verfügbaren Adressbereichen zu nutzen, etwa `x2-xx-xx-xx-xx-xx`, `x6-xx-xx-xx-xx- xx`, `xA-xx-xx-xx-xx-xx` und `xE-xx-xx-xx-xx-xx`. (Jedes `x` repräsentiert eine Hexadezimalzahl.)

Speicher-Volumes

Es kommt vor, dass der einem Container zugewiesene Speicherplatz aufgrund seiner kurzlebigen Natur für eine anstehende Aufgabe nicht geeignet ist. Dann müssen Sie einen Speicherort verwenden, der zwischen den verschiedenen Container-Deployments erhalten bleibt.

Im Allgemeinen ist davon abzuraten, Speicher-Volumes des Docker-Hosts in einem Container zu mounten, weil der persistente Zustand Ihres Containers dadurch an diesen speziellen Host gebunden ist. Andererseits kann es durchaus sinnvoll sein, solche Speicher-Volumes für kurzlebige Daten oder temporäre Cachedateien zu verwenden.

Sofern erforderlich, können Sie Verzeichnisse und einzelne Dateien des Hosts mit dem Kommandozeilenargument `-v/--mount` im Container mounten. Nutzen Sie für dieses Argument auf jeden Fall vollständig qualifizierte Pfade. Im folgenden Beispiel wird das Verzeichnis */mnt/session_data* im Verzeichnis */data* des Containers gemountet:

```
$ docker container run --rm -ti \
  --mount type=bind,target=/mnt/session_data,source=/data \
  ubuntu:latest /bin/bash

root@0f887071000a:/# mount | grep data
/dev/sda9 on /data type ext4 (rw,relatime,data=ordered)
root@0f887071000a:/# exit
```

Für Bind Mounts können Sie das Argument `-v` nutzen, um den Befehl kürzer zu halten. In dieser Variante werden Quell- und Zieldateien/-verzeichnisse durch einen Doppelpunkt (`:`) getrennt.

Standardmäßig werden Volumes mit Schreib- und Leserechten eingebunden. Sie können den Befehl allerdings ganz einfach modifizieren, sodass das Verzeichnis lediglich lesend eingebunden wird, indem Sie am Ende des `--mount`-Arguments `,readonly` oder am Ende des `-v`-Arguments `:ro` hinzufügen:

```
$ docker container run --rm -ti \
  -v /mnt/session_data:/data:ro \
  ubuntu:latest /bin/bash
```

Weder der Host-Mountpoint noch der Mountpoint im Container muss vor dem Ausführen des Kommandos bestehen, damit der Befehl funktioniert. Wenn der Host-Mountpoint nicht existiert, wird dieser als Verzeichnis angelegt. Das kann zu Problemen führen, wenn Sie versuchen, auf eine Datei statt auf ein Verzeichnis zu verweisen.

Wie die `mount`-Ausgabe zeigt, sind auf dem Dateisystem erwartungsgemäß sowohl Lese- als auch Schreibzugriffe (`rw`) erlaubt.

SELinux und Volume Mounts

Falls Sie auf Ihrem Docker-Host SELinux verwenden, erhalten Sie womöglich die Fehlermeldung »Zugriff verweigert«, wenn Sie versuchen, ein Volume im Container zu mounten. Sie können das anpassen, indem Sie die `z`-Option des Docker-Kommandos beim Mounten von Volumes verwenden:

- Die kleingeschriebene `z`-Option gibt an, dass der Bind Mount von mehreren Containern aus möglich ist.
- Die großgeschriebene `Z`-Option gibt an, dass der Bind Mount privat und nicht »geshared« ist.

Wenn Sie ein Volume in mehreren Containern verwenden wollen, können Sie die `z`-Option verwenden:

```
$ docker container run --rm -v /etc/dhcpd:/etc/dhcpd:z dhcpd
```

Am besten ist es allerdings, den Mount-Befehl mit der Option `Z` (großgeschrieben!) aufzurufen, die dem Verzeichnis dasselbe MCS-Label (*Multi Categories Security*, z. B. `chcon … -l s0:c1,c2`) zuweist, das auch der Container benutzt. Dies ist am sichersten und gestattet nur einem einzigen Container, das Volume zu mounten:

```
$ docker container run --rm -v /etc/dhcpd:/etc/dhcpd:Z dhcpd
```

Sie sollten mit großer Vorsicht vorgehen, wenn Sie die `z`-Optionen verwenden. Das Bind-Mounten von Systemverzeichnissen wie */etc* oder */var* mit der `Z`-Option wird höchstwahrscheinlich Ihr System inoperabel machen, und Sie müssen die Änderungen dann mit den SELinux-Tools händisch zurücknehmen (*https://www.thegeekdiary.com/understanding-selinux-file-labelling-and-selinux-context*).

Wenn der Container so designt wurde, dass er die Daten nach */data* schreibt, sind diese Daten auf dem Hostdateisystem unter */mnt/session_data* sichtbar und bleiben auch dann verfügbar, wenn der Container gestoppt und ein neuer Container mit demselben Volume gestartet wird.

Es ist möglich, Docker so zu konfigurieren, dass das Root-Volume eines Containers nur mit Lesezugriff (`read-only`) gemountet wird, damit Prozesse innerhalb des Containers nichts in das Root-Dateisystem schreiben dürfen. Dies verhindert etwa, dass Logfiles, die dem Entwickler unbekannt sind, den allozierten Speicherplatz der Produktivumgebung vollschreiben. Wenn `read-only` im Zusammenhang mit einem gemounteten Volume genutzt wird, können Sie sicher sein, dass die Daten nur in die dafür vorgesehenen Bereiche geschrieben werden.

Im vorangehenden Beispiel hätte man das erreichen können, indem man dem Aufruf einfach `--read-only=true` anfügt:

```
$ docker container run --rm -ti --read-only=true -v /mnt/session_data:/data
    ubuntu:latest /bin/bash

root@df542767bc17:/# mount | grep " / "
overlay on / type overlay
```

```
(ro,relatime,lowerdir=…,upperdir=…,workdir=…)root@df542767bc17:/#
    mount | grep session
/dev/sda9 on /session_data type ext4
(rw,relatime,data=ordered)root@df542767bc17:/# exit
```

Wenn man sich die `mount`-Ausgabe für das Root-Verzeichnis genau ansieht, fällt auf, dass es mit der Option `ro` gemountet wird, wodurch es schreibgeschützt ist. Das Verzeichnis */mnt/session_data* wird jedoch nach wie vor mit der Option `rw` gemountet, sodass die Anwendung problemlos auf das dafür vorgesehene Volume schreiben kann.

Manchmal kann es nötig sein, Verzeichnisse wie */tmp* beschreibbar zu machen, sogar wenn der übrige Container schreibgeschützt ist. Docker ermöglicht Ihnen über den Befehl `docker container run` und das Attribut `--mount type=tmpfs`, ein *tmpfs*-Dateisystem (kurz für *temporary file system*) im Container zu mounten. *tmpfs*-Dateisysteme werden vollständig im Speicher gehalten. Sie sind sehr schnell, aber auch kurzlebig, und sie belegen zusätzlichen Systemspeicher. Die Daten in diesen *tmpfs*-Verzeichnissen gehen verloren, sobald der Container gestoppt wird. Das folgende Beispiel zeigt den Startvorgang eines Containers, der ein *tmpfs*-Dateisystem mit 256 MB als Verzeichnis */tmp* mountet:

```
$ docker container run --rm -ti --read-only=true \
  --mount type=tmpfs,destination=/tmp,tmpfs-size=256M \
  ubuntu:latest /bin/bash

root@25b4f3632bbc:/# df -h /tmp
Filesystem      Size  Used Avail Use% Mounted on
tmpfs           256M     0  256M   0% /tmp
root@25b4f3632bbc:/# grep /tmp /etc/mtab
tmpfs /tmp tmpfs rw,nosuid,nodev,noexec,relatime,size=262144k 0 0
root@25b4f3632bbc:/# exit
```

Container sollten wenn irgend möglich zustandslos sein. Die Handhabung der Speicher-Volumes führt zu unerwünschten Abhängigkeiten und kann das Deployment beträchtlich verkomplizieren.

Ressourcen-Quotas

Wenn die verschiedenen Probleme erörtert werden, die in der Cloud regelmäßig Schwierigkeiten verursachen, steht das Thema »lärmende Nachbarn« (*Noisy Neighbors*) immer ganz weit oben auf der Liste. Mit diesem Begriff ist gemeint, dass andere Anwendungen, die auf demselben physischen Wirtssystem wie Ihre eigenen laufen, spürbare Auswirkungen auf die Performance und die Verfügbarkeit von Ressourcen haben können.

Herkömmliche virtuelle Maschinen besitzen den Vorteil, dass man problemlos und sehr strikt festlegen kann, wie viel Arbeitsspeicher und Rechenzeit sowie andere Ressourcen einer virtuellen Maschine zugestanden werden. Beim Einsatz von Docker müssen Sie sich stattdessen die `cgroup`-Funktionalität des Linux-Kernels zunutze

machen, um die einem Linux-Container zur Verfügung stehenden Ressourcen festzulegen. Die Befehle `docker container create` und `docker container run` ermöglichen es, Begrenzungen der CPU-Performance und der Arbeitsspeichernutzung beim Erstellen eines Containers direkt zu konfigurieren.

Die Beschränkungen werden beim Erstellen des Containers festgelegt. Müssen Sie sie ändern, können Sie den Befehl `docker container update` nutzen oder einen neuen Container mit den entsprechenden Anpassungen deployen.

An dieser Stelle gibt es einen wichtigen Vorbehalt: Docker unterstützt zwar die Beschränkung von CPU-Performance, Arbeitsspeicher und Größe der Swap-Datei, allerdings müssen diese Features auch im Kernel aktiviert sein, damit Docker sie nutzen kann. Möglicherweise müssen Sie die entsprechenden Kommandozeilenparameter beim Starten des Kernels hinzufügen. Geben Sie den Befehl `docker system info` ein, um zu überprüfen, ob Ihr Kernel die Beschränkungen unterstützt. Fehlt der Support für eine der Einstellungen, wird am Ende eine Warnmeldung wie diese ausgegeben:

```
WARNING: No swap limit support
```

Wie genau der Kernel konfiguriert werden muss, um `cgroups` zu unterstützen, hängt von der verwendeten Linux-Distribution ab. Lesen Sie in der Docker-Dokumentation nach (*https://oreil.ly/Z70ZO*), wenn Sie Hilfe bei der Konfiguration benötigen.

CPU-Anteile

Docker bietet verschiedene Möglichkeiten, um die CPU-Nutzung von Anwendungen in Containern zu beschränken. Die ursprüngliche und immer noch weiterhin verbreitete Methode ist das Konzept der »CPU-Anteile«. Im Folgenden werden wir auch andere Möglichkeiten betrachten.

Die gesamte Performance aller Prozessorkerne eines Systems stellt den verfügbaren Pool dar, der 1.024 Anteilen entspricht. Wenn Sie einem Container CPU-Anteile zuweisen, können Sie dadurch festlegen, wie viel CPU-Performance eben diesem Container zugeteilt wird. Möchten Sie es beispielsweise so einrichten, dass er höchstens die Hälfte der Performance des Systems beanspruchen darf, würden Sie ihm 512 CPU-Anteile zuweisen. Beachten Sie, dass es sich hier nicht um eine exklusive Zuweisung handelt, d. h., die CPU-Anteile stehen nicht ausschließlich diesem einen Container zur Verfügung. Wenn Sie also einem Container 1.024 Anteile zuweisen, bedeutet das nicht, dass alle anderen Container nicht ausgeführt werden. Vielmehr wird dem Scheduler auf diese Weise mitgeteilt, wie lange jeder Container jeweils ausgeführt werden soll, wenn er an der Reihe ist. Nehmen wir einmal an, einem Container werden 1.024 CPU-Anteile zugewiesen (der Standardwert) und zwei weiteren Containern jeweils 512. Außerdem sollen jedem Prozess normalerweise 100 Mikrosekunden Rechenzeit zugestanden werden. In diesem Fall werden zwar alle drei Container gleich häufig ausgeführt, diejenigen mit 512 CPU-Anteilen laufen

dann allerdings nur jeweils 50 Mikrosekunden, während der Container mit 1.024 Anteilen 100 Mikrosekunden lang ausgeführt wird.

Schauen wir uns etwas genauer an, wie das in der Praxis funktioniert. Im folgenden Beispiel verwenden wir ein Docker-Image, das den Befehl `stress` (*https://linux.die.net/man/1/stress*) enthält, der dazu dient, das System vollständig auszulasten.

Wenn wir den `stress`-Befehl ohne irgendwelche `cgroup`-Beschränkungen ausführen, wird er die ihm zugewiesenen Ressourcen auslasten. Der nachstehende Befehl erzeugt eine durchschnittliche Auslastung von etwa 5, indem zwei rechenintensive Prozesse, ein Ein-/Ausgabeprozess sowie zwei Speicherzuordnungsprozesse ausgeführt werden.

Beachten Sie, dass sich im folgenden Befehl alles nach dem Namen des Container-Image auf den `stress`-Befehl bezieht, nicht auf den `docker`-Befehl:

```
$ docker container run --rm -ti spkane/train-os \
  stress -v --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 120s
```

Diesen Befehl sollte ein aktuelles System durchaus verkraften können, Sie müssen sich aber darüber im Klaren sein, dass er den Host stark belastet. Führen Sie ihn daher nicht auf einem System aus, das der zusätzlichen Last nicht gewachsen ist oder mangels verfügbarer Ressourcen womöglich sogar ausfallen könnte.

Wenn Sie kurz vor Ablauf der zwei Minuten auf dem Docker-Host den Befehl `top` ausführen, wird erkennbar, wie sich die Auslastung durch den `stress`-Befehl auf das System auswirkt:

```
$ top -bn1 | head -n 15
top - 20:56:36 up 3 min,  2 users,  load average: 5.03, 2.02, 0.75
Tasks:  88 total,   5 running,  83 sleeping,   0 stopped,   0 zombie
%Cpu(s): 29.8 us, 35.2 sy, 0.0 ni, 32.0 id, 0.8 wa, 1.6 hi, 0.6 si, 0.0 st
KiB Mem:   1021856 total,   270148 used,   751708 free,    42716 buffers
KiB Swap:        0 total,        0 used,        0 free.    83764 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
  810 root      20   0    7316     96      0 R  44.3  0.0   0:49.63 stress
  813 root      20   0    7316     96      0 R  44.3  0.0   0:49.18 stress
  812 root      20   0  138392  46936    996 R  31.7  4.6   0:46.42 stress
  814 root      20   0  138392  22360    996 R  31.7  2.2   0:46.89 stress
  811 root      20   0    7316     96      0 D  25.3  0.0   0:21.34 stress
    1 root      20   0  110024   4916   3632 S   0.0  0.5   0:07.32 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.04 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:00.11 ksoftir…
```

Nutzer von Docker Desktop auf Nicht-Linux-Systemen werden bemerken, dass Docker das VM-Dateisystem `read-only` geschaltet hat und dass es keine nützlichen Tools für das Monitoring der VM enthält. Für diese Demos, bei denen Sie die Ressourcennutzung von verschiedenen Prozessen überwachen wollen, können Sie das mit einem Workaround umgehen:

```
$ docker container run -it --pid=host alpine sh
/ # apk update
/ # apk add htop
```

```
/ # htop -p $(pgrep stress | tr '\n' ',')
/ # exit
```

Beachten Sie, dass der obige `htop`-Befehl einen Fehler liefern wird, wenn `stress` bei seinem Start nicht aktiv läuft, da ansonsten keine Prozesse vom `pgrep`-Befehl zurückgegeben werden.

Jedes Mal, wenn Sie eine neue `stress`-Instanz ausführen, werden Sie `htop` beenden und neu ausführen müssen.

Sie können denselben `stress`-Befehl mit der Hälfte der verfügbaren CPU-Zeit wie folgt erneut ausführen:

```
$ docker container run --rm -ti --cpu-shares 512 spkane/train-os \
  stress -v --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 120s
```

Entscheidend ist hier der Parameter `--cpu-shares 512`, der diesem Container 512 CPU-Anteile zuweist. Beachten Sie, dass die Auswirkung dieser Änderung auf einem kaum ausgelasteten System womöglich kaum spürbar ist. Das liegt daran, dass dem Container, wann immer Arbeit ansteht, stets die gleiche Zeitscheibe zugeteilt wird, sofern das System keinen Beschränkungen hinsichtlich der Ressourcenzuweisung unterliegt. In unserem Beispiel würde die Ausgabe des `top`-Befehls auf dem Hostsystem wahrscheinlich fast genauso aussehen – es sei denn, Sie starten einige weitere Container, um die CPU anderweitig zu beschäftigen.

Im Gegensatz zur Ressourcenzuweisung bei virtuellen Maschinen kann die `cgroup`-basierte Beschränkung der CPU-Anteile bei Docker unerwartete Folgen haben. Hierbei handelt sich nicht um feste Werte, sondern um relative Einschränkungen, ähnlich wie beim `nice`-Befehl. Denken Sie an einen auf einem kaum ausgelasteten System laufenden Container, dessen CPU-Anteile auf die Hälfte der verfügbaren Performance beschränkt sind. Da die CPU sonst nichts zu tun hat, zeigt diese Nutzungsbegrenzung kaum Wirkung. Wenn nun ein zweiter Container gestartet wird, der kräftig CPU-Performance nutzt, wird die Auswirkung der Einschränkung des ersten Containers plötzlich spürbar. Bedenken Sie diesen Umstand bei der Beschränkung von Containern und der Zuweisung von Ressourcen sorgfältig.

CPU-Zuweisung

Sie können einem Container einen oder mehrere CPU-Kern(e) zuweisen. Sämtliche für diesen Container geplanten Arbeiten werden dann nur auf den ihm zugewiesenen Kernen ausgeführt. Das ist nützlich, wenn Sie CPUs zwischen Anwendungen aufteilen wollen oder wenn der Anwendung eine bestimmte CPU zugewiesen werden soll, etwa zur effizienten Nutzung des Caches.

Im folgenden Beispiel weisen wir dem Container eine der beiden CPUs sowie 512 CPU-Anteile zu.

```
$ docker container run --rm -ti \
  --cpu-shares 512 --cpuset-cpus=0 spkane/train-os \
  stress -v --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 120s
```

Die Zählung für den Wert des Arguments von `--cpuset-cpus` beginnt bei null, daher besitzt die erste CPU den Wert 0. Wenn Sie Docker anweisen, eine nicht vorhandene CPU zu nutzen, erhalten Sie die Fehlermeldung »Cannot start container«. Auf einem Host mit zwei CPUs könnten Sie das mit dem Argument `--cpuset-cpus=0-2` ausprobieren.

Wenn Sie nun den `top`-Befehl erneut ausführen, sollte für den Anteil der Performance, die mit aus dem *Userspace* (`us`) stammenden Aufgaben verbracht wurde, ein niedrigerer Prozentwert als vorher angezeigt werden, denn wir haben die Ausführung der Prozesse auf eine der beiden CPUs beschränkt:

```
%Cpu(s): 18.5 us, 22.0 sy, 0.0 ni, 57.6 id, 0.5 wa, 1.0 hi, 0.3 si, 0.0 st
```

Wenn Sie die CPU-Zuweisung verwenden, werden bei weiteren Beschränkungen der CPU-Nutzung nur diejenigen Container berücksichtigt, denen dieselben CPUs zugewiesen wurden.

Wenn Sie den *CPU Completely Fair Scheduler* (CFS) des Linux-Kernels nutzen, können Sie die CPU-Performance-Quota eines Containers ändern, indem Sie bei dessen Start mit `docker container run` einen passenden Wert für das Argument `--cpu-quota` übergeben.

Vereinfachung von CPU-Quotas

Die Nutzung von CPU-Anteilen war in Docker ursprünglich der Mechanismus, um CPU-Limits zu handhaben. Docker hat sich seitdem stark weiterentwickelt. Eine der Veränderungen, die dazu beigetragen haben, den Nutzern die Arbeit wesentlich zu erleichtern, ist, wie die CPU-Quotas gesetzt werden können. Statt selbst zu versuchen, CPU-Anteile und Quotas korrekt zu setzen, können Sie Docker einfach mitteilen, wie viel Rechenleistung dem Container zur Verfügung stehen soll. Docker übernimmt dann die nötigen Berechnungen, um die darunterliegenden `cgroups` korrekt zu setzen.

Der `--cpus`-Parameter kann auf einen Wert zwischen 0,01 und der Anzahl von CPU-Kernen auf dem Docker-Server gesetzt werden.

```
$ docker container run --rm -ti --cpus=".25" spkane/train-os \
  stress -v --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 60s
```

Wenn Sie versuchen, einen zu hohen Wert zu setzen, bekommen Sie eine Fehlermeldung von Docker angezeigt. Diese teilt Ihnen die maximale Anzahl der CPU-Kerne, mit der Sie arbeiten können, mit:

```
$ docker container run --rm -ti --cpus="40.25" spkane/train-os \
  stress -v --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 60s
docker: Error response from daemon: Range of CPUs is from
    0.01 to 4.00, as there are only 4 CPUs available.
See 'docker container run --help'.
```

Der Befehl `docker container update` kann genutzt werden, um die Ressourcen von einem oder mehreren Containern dynamisch zu beschränken. Sie können einfach die CPU-Allokation auf zwei Containern gleichzeitig anpassen. Das geht wie folgt:

```
$ docker container update --cpus="1.5" 092c5dc85044 92b797f12af1
```

Docker behandelt CPUs genau so, wie Linux das tut. Hyper-Threading und Cores werden von Linux interpretiert und über die Datei */proc/cpuinfo* bereitgestellt. Nutzen Sie in Docker `--cpus`, beziehen Sie sich darauf, auf wie viele der Einträge in dieser Datei der Container Zugriff haben soll – egal ob es sich um einen Standard-Core oder einen Hyper-Thread-Core handelt.

Arbeitsspeicher

Auf ähnliche Weise wie bei der Beschränkung der CPU-Nutzung kann auch der Umfang des Arbeitsspeichers festgelegt werden, auf den ein Container zugreifen darf. Es gibt hier allerdings einen grundlegenden Unterschied: Die Beschränkung der CPU-Nutzung beeinflusst lediglich die Priorität, die einer Anwendung bei der Zuteilung von CPU-Performance eingeräumt wird – die Beschränkung des Arbeitsspeichers hingegen ist eine *feststehende* Begrenzung. Selbst auf einem System mit 96 GB freiem Arbeitsspeicher, das keinen Ressourcenkontingenten unterliegt, wird ein Container, dessen Arbeitsspeichernutzung auf 24 GB beschränkt ist, niemals mehr als diese 24 GB nutzen können – die tatsächliche Größe des freien Arbeitsspeichers spielt keine Rolle. Die Funktionsweise des virtuellen Arbeitsspeichers unter Linux ermöglicht es, einem Container mehr Arbeitsspeicher zuzuweisen, als tatsächlich an physischem RAM vorhanden ist. Der Container wird dann, wenn kein echter Arbeitsspeicher verfügbar ist, wie jeder andere Linux-Prozess auf einen Swap zurückgreifen.

Hier starten wir einen Container mit beschränktem Arbeitsspeicher, indem wir dem Befehl `docker container run` die Option `--memory` übergeben:

```
$ docker container run --rm -ti --memory 512m spkane/train-os \
  stress -v --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 10sl
```

Wenn Sie nur die Option `--memory` verwenden, werden damit sowohl die Menge des RAM als auch der Umfang des virtuellen Arbeitsspeichers festgelegt, auf die der Container zugreifen darf. In diesem Fall darf der Container also 512 MB RAM und weitere 512 MB virtuellen Arbeitsspeicher in Form eines Swap verwenden. Docker unterstützt die Kürzel `b`, `k`, `m` und `g`, die für Byte, Kilobyte, Megabyte und Gigabyte stehen. Sollten Sie ein Linux-System mit Docker betreiben, das über mehrere Terabyte Arbeitsspeicher verfügt, müssen Sie die Werte leider in Gigabyte angeben.

Möchten Sie die Größe des Swap getrennt festlegen oder deaktivieren, müssen Sie zusätzlich die Option `--memory-swap` angeben. Sie beschreibt die Summe aus echtem und virtuellem Arbeitsspeicher. Wenn wir den vorherigen Befehl wie folgt ändern, teilen wir dem Kernel dadurch mit, dass der Container auf 512 MB Arbeitsspeicher und weitere 256 MB virtuellen Speicher zugreifen darf:

```
$ docker container run --rm -ti --memory 512m --memory-swap=768m \
    spkane/train-os stress -v --cpu 2 --io 1 --vm 2 --vm-bytes 128M \
    --timeout 10s
```

Setzen Sie `--memory-swap` auf `-1`, kann der Container so viel Swap nutzen, wie im zugrunde liegenden System verfügbar ist; werden `--memory-swap` und `--memory` auf den gleichen positiven Wert gesetzt, hat der Container gar keinen Zugriff auf den Swap.

Anders als bei den CPU-Anteilen handelt es sich beim Arbeitsspeicher um eine *feststehende* Begrenzung! Das hat den Vorteil, dass diese Beschränkung keine spürbaren Auswirkungen hat, wenn weitere Container auf dem System gestartet werden. Es bedeutet aber auch, dass Sie sie genau an den tatsächlichen Arbeitsspeicherbedarf Ihrer Anwendung anpassen müssen, da es hier keinen Spielraum nach oben gibt. Ein Out-of-Memory-Container führt dazu, dass das System keinen freien Arbeitsspeicher mehr hat. Das System wird dann versuchen, einen Prozess zu finden, den es »abschießen« kann, um Platz zu schaffen. Ein häufig vorkommender Fehler besteht darin, dass Container ein zu geringes Arbeitsspeicherlimit besitzen. Das Ende vom Lied ist, dass sich der Container mit einem Exitcode 137 verabschiedet und der Kernel Out-of-Memory-(OOM-)Nachrichten in die `dmesg`-Ausgabe des Docker-Servers schreibt.

Was genau geschieht, wenn ein Container die Obergrenze des Arbeitsspeichers erreicht? Probieren wir es aus und ändern wir einen der vorhergehenden Befehle so ab, dass dem Container beträchtlich weniger Arbeitsspeicher zur Verfügung steht:

```
$ docker container run --rm -ti --memory 100m spkane/train-os \
  stress -v --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 10s
```

Alle bisherigen Ausführungen des `stress`-Containers lieferten stets folgendes Ergebnis:

```
stress: info: [17] successful run completed in 10s
```

Dieser schlägt jedoch fehl, und es erscheint nachstehende Meldung:

```
stress: FAIL: [1] (451) failed run completed in 0s
```

Der Grund dafür ist, dass der Container versucht, mehr Speicherplatz zu reservieren, als ihm gestattet ist. Dadurch wird der Linux-OOM-Killer (*Out-of-Memory-Killer*) aktiviert, der dann anfängt, `cgroup`-Prozesse »abzuschießen«, um Arbeitsspeicher freizugeben. In diesem Fall enthält unser Container einen Prozess mit einem Elternelement, das mehrere Kindprozesse gestartet hat. Schließt der OOM-Killer einen der Kindprozesse, räumt der Elternprozess alles auf und endet mit einem Fehler.

Docker gibt Ihnen die Möglichkeit, den OOM-Killer feiner abzustimmen oder zu deaktivieren, indem Sie beim Aufruf von `docker container run` die Argumente `--oom-kill-disable` bzw. `--oom-score-adj` setzen, aber das ist nur in den seltensten Fällen empfehlenswert.

Wenn Sie sich zum Docker-Server verbinden, sehen Sie die Kernel-Message zu diesem Event, wenn Sie `dmesg` ausführen. Der Output sieht dann etwa wie folgt aus:

```
[ 4210.403984] stress invoked oom-killer: gfp_mask=0x24000c0 …
[ 4210.404899] stress cpuset=5bfa65084931efabda59d9a70fa8e88 …
[ 4210.405951] CPU: 3 PID: 3429 Comm: stress Not tainted 4.9 …
[ 4210.406624] Hardware name:   BHYVE, BIOS 1.00 03/14/2014
…
[ 4210.408978] Call Trace:
[ 4210.409182]  [<ffffffff94438115>] ? dump_stack+0x5a/0x6f
….
[ 4210.414139]  [<ffffffff947f9cf8>] ? page_fault+0x28/0x30
[ 4210.414619] Task in /docker-ce/docker/5…3
killed as a result of limit of /docker-ce/docker/5…3
[ 4210.416640] memory: usage 102380kB, limit 102400kB, failc …
[ 4210.417236] memory+swap: usage 204800kB, limit 204800kB,  …
[ 4210.417855] kmem: usage 1180kB, limit 9007199254740988kB, …
[ 4210.418485] Memory cgroup stats for /docker-ce/docker/5…3:
cache:0KB rss:101200KB rss_huge:0KB mapped_file:0KB dirty:0KB
writeback:11472KB swap:102420KB inactive_anon:50728KB
active_anon:50472KB inactive_file:0KB active_file:0KB unevictable:0KB
…
[ 4210.426783] Memory cgroup out of memory: Kill process 3429…
[ 4210.427544] Killed process 3429 (stress) total-vm:138388kB,
anon-rss:44028kB, file-rss:900kB, shmem-rss:0kB
[ 4210.442492] oom_reaper: reaped process 3429 (stress), now
anon-rss:0kB, file-rss:0kB, shmem-rss:0kB
```

Dieses Out-of-Memory-Event wird auch durch Docker aufgezeichnet und durch die Ausführung von `docker system events` sichtbar:

```
$ docker system events
2018-01-28T15:56:19.972142371-08:00 container oom \
d0d803ce32c4e86d0aa6453512a9084a156e96860e916ffc2856fc63ad9cf88b \
(image=spkane/train-os, name=loving_franklin)
```

Block-I/O

Viele Container sind einfache zustandslose Anwendungen, die keine I/O-Restriktionen benötigen. Docker unterstützt auch die Beschränkung von Block-I/O auf verschiedenen Wegen mittels des `cgroups`-Mechanismus.

Die erste Möglichkeit besteht in der Priorisierung des Zugriffs bestimmter Prozesse von Containern auf Blockdevice-I/O. Dieser Vorgang wird durch die Modifizierung der Standardeinstellung für das `cgroup`-Attribut `blkio.weight` gesteuert, das Werte zwischen 10 und 1.000 annehmen kann und auf 500 voreingestellt ist. Das System unterteilt dann den verfügbaren I/O der Prozesse in `cgroup`-Zeitscheibchen, wobei die eingestellte Gewichtung bestimmt, wie viele Ein- und Ausgaben den verschiedenen Prozessen zur Verfügung stehen.

Zum Einstellen dieser Gewichtung müssen Sie beim Aufruf von `docker container run` dem Argument `--blkio.weight` einen gültigen Wert übergeben. Sie können auch ein spezifisches Device angeben, indem Sie die Option `--blkio-weight-device` verwenden.

Wie auch bei den CPU-Anteilen ist das Definieren der Gewichtung in der Praxis umständlich. Viel einfacher geht es, wenn man über die `cgroup` den maximalen Datendurchsatz in Bytes oder die pro Sekunde verfügbaren Operationen in einem Container begrenzt. Die folgende Einstellung ermöglicht uns genau das:

```
--device-read-bps     Limit read rate (bytes per second) from a device
--device-read-iops    Limit read rate (IO per second) from a device
--device-write-bps    Limit write rate (bytes per second) to a device
--device-write-iops   Limit write rate (IO per second) to a device
```

Sie können die Auswirkungen auf die Performance eines Containers testen, indem Sie einige der folgenden Kommandos verwenden, die den Linux-I/O-Tester `bonnie` einsetzen:

```
$ time docker container run --rm -ti spkane/train-os:latest bonnie++ \
    -u 500:500 -d /tmp -r 1024 -s 2048 -x 1
…
real  0m27.715s
user  0m0.027s
sys   0m0.030s

$ time docker container run -ti --rm --device-write-iops /dev/vda:256 \
    spkane/train-os:latest bonnie++ -u 500:500 -d /tmp -r 1024 -s 2048 -x 1
…
real  0m58.765s
user  0m0.028s
sys   0m0.029s

$ time docker container run -ti --rm --device-write-bps /dev/vda:5mb \
    spkane/train-os:latest bonnie++ -u 500:500 -d /tmp -r 1024 -s 2048 -x 1
…
```

PowerShell-Nutzer können den PowerShell-Befehl `Measure-Command` anstelle des Unix-Kommandos `time` in diesen Beispielen verwenden (*https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/measure-command?view=powershell-7.3*).

Unserer Erfahrung nach ist es der effizienteste Weg, eine Beschränkung mit den Optionen `--device-read-ops` und `--device-write-ops` zu setzen. Dies ist auch das, was wir empfehlen.

ulimit

In der Zeit vor Linux-cgroups gab es einen anderen Weg, um die Beschränkung von Prozessressourcen umzusetzen: `ulimit`. Dieser Mechanismus ist immer noch verfügbar und wird weiterhin für alle Use Cases verwendet, bei denen er auch früher schon zum Einsatz kam (*https://www.linuxhowtos.org/Tips%20and%20Tricks/ulimit.htm*).

Der folgende Code zeigt, welche Ressourcen durch die Konfiguration von »harten« und »weichen« Grenzen mit dem `ulimit`-Befehl kontingentiert werden können:

```
$ ulimit -a
core file size (blocks, -c) 0
data seg size (kbytes, -d) unlimited
scheduling priority (-e) 0
file size (blocks, -f) unlimited
pending signals (-i) 5835
max locked memory (kbytes, -l) 64
max memory size (kbytes, -m) unlimited
open files (-n) 1024
```

```
pipe size (512 bytes, -p) 8
POSIX message queues (bytes, -q) 819200
real-time priority (-r) 0
stack size (kbytes, -s) 10240
cpu time (seconds, -t) unlimited
max user processes (-u) 1024
virtual memory (kbytes, -v) unlimited
file locks (-x) unlimited
```

Mittlerweile ist es möglich, die für Container geltenden Standardkontingente zu konfigurieren. Der folgende Befehl teilt dem Docker-Daemon mit, dass für alle Container eine feststehende Beschränkung von 150 geöffneten Dateien und 20 Prozessen gelten soll:

```
$ sudo dockerd --default-ulimit nofile=50:150
```

Sie können die Begrenzungen für einzelne Container ändern, indem Sie beim Starten des betreffenden Containers dem Argument `--ulimit` entsprechende Werte übergeben:

```
$ docker container run --rm -d --ulimit nproc=150:300 nginx
```

Es gibt noch einige weitere, anspruchsvollere Befehle, die beim Starten eines Containers verwendet werden können, die hier genannten dürften jedoch die gebräuchlicheren Use Cases abdecken. In der kontinuierlich aktualisierten Dokumentation des Docker-Clients (*https://dockr.ly/2ME0ygi*) sind alle verfügbaren Optionen aufgeführt.

Container starten

Bevor wir in die Details von Containern und ihren Beschränkungen eingestiegen sind, haben wir zum Erstellen eines Containers den Befehl `docker container create` verwendet. Der Container ist einfach da und tut zunächst nichts. Es liegt eine Konfiguration vor, aber kein laufender Prozess. Um ihn anschließend zu starten, kommt dann der Befehl `docker container start` zum Einsatz.

Nehmen wir nun einmal an, wir würden als Nächstes *Redis* ausführen wollen, eine verbreitete Key-Value-Datenbank. Wir werden nichts weiter mit diesem Redis-Container anfangen, da es sich hierbei jedoch um einen langlebigen Prozess handelt, soll er an dieser Stelle als Beispiel dafür dienen, was in einer echten Umgebung vonstattengehen könnte. Zunächst einmal erstellen wir den Container mit einem Befehl wie diesem:

```
$ docker container create -p 6379:6379 redis:2.8
Unable to find image 'redis:7.0' locally
7.0: Pulling from library/redis
3f4ca61aafcd: Pull complete
…
20bf15ad3c24: Pull complete
Digest: sha256:8184cfe57f205ab34c62bd0e9552dffeb885d2a7f82ce4295c0df344cb6f0007
Status: Downloaded newer image for redis:7.0
092c5dc850446324e4387485df7b76258fdf9ed0aedcd53a37299d35fc67a042
```

Am Ende der Ausgabe wird der vollständige für den Container erzeugte Hashwert angegeben. Wäre dieser Wert oder auch dessen Kurzform nicht bekannt, könnte

man alternativ alle auf dem System vorhandenen Container (laufende und nicht laufende) wie folgt anzeigen:

```
$ docker container ls -a --filter ancestor=redis:2.8
CONTAINER ID IMAGE     COMMAND                CREATED        … NAMES
092c5dc85044 redis:7.0 "docker-entrypoint.s…" 46 seconds ago elegant_wright
```

Wir finden unseren Container, indem wir die Ausgabe auf das verwendete Image filtern und uns den Erstellungszeitpunkt anschauen. Nun können wir den Container mit folgendem Befehl starten:

```
$ docker container start 092c5dc85044
```

Die meisten Docker-Befehle funktionieren sowohl mit dem vollständigen als auch mit dem kurzen Hashwert. Im vorangehenden Beispiel lautet der vollständige Hashwert `092c5dc850446324e…a37299d35fc67a042` und die Kurzform, die oft in den Ausgaben von Befehlen auftaucht, `092c5dc85044`. Der kurze Hashwert besteht aus den ersten zwölf Zeichen der vollständigen Fassung. Im vorherigen Beispiel hätte auch `docker container start 6b7` funktioniert.

Der Befehl sollte den Container gestartet haben. Da er allerdings im Hintergrund läuft, können wir nicht sicher sagen, ob nicht doch etwas fehlgeschlagen ist. Hiermit überprüfen wir, ob der Container tatsächlich läuft:

```
$ docker container ls
CONTAINER ID  IMAGE      COMMAND                …  STATUS       …
092c5dc85044  redis:7.0  "docker-entrypoint.s…" …  Up 2 minutes …
```

Und da ist er – läuft wie erwartet. Das erkennen Sie am Status, der `Up` ausgibt, und daran, wie lange der Container bereits läuft.

Container automatisch neu starten

Oftmals sollen Container, die zuvor beendet wurden, später wieder neu gestartet werden. Manche Container sind sehr kurzlebig und kommen und gehen in Windeseile. Von Anwendungen in Produktivumgebungen erwartet man allerdings, dass sie laufen, nachdem sie einmal gestartet wurden. Wenn Sie ein komplexeres System laufen haben, kann ein Scheduler diese Aufgabe übernehmen.

Zu diesem Zweck übergeben wir dem Befehl `docker container run` das Argument `--restart`. Hierbei stehen vier Werte zur Auswahl: `no`, `always`, `on-failure` und `unless-stopped`. `no` bedeutet, dass der Container nach dem Beenden nicht neu gestartet wird. `always` bewirkt, dass der Container nach jedem Beenden neu gestartet wird – der Exitcode beim Beenden spielt dabei keine Rolle. Wird `restart` auf `on-failure` gesetzt, versucht Docker, den Container neu zu starten, wenn er sich mit einem Exitcode ungleich null beendet. Wenn Sie `on-failure:3` übergeben, unternimmt Docker zunächst drei Versuche, ihn neu zu starten, und gibt erst dann auf. `unless-stopped` wird den Container immer neu starten, es sei denn, er wurde ganz bewusst gestoppt, etwa durch einen Befehl wie `docker container stop`.

Sie können diesen Vorgang beobachten, indem Sie den vorhin verwendeten `stress`-Container mit eingeschränktem Arbeitsspeicher nicht mit dem Argument `--rm`, sondern mit `--restart` starten:

```
$ docker container run -ti --restart=on-failure:3 --memory 100m \
  spkane/train-os stress -v --cpu 2 --io 1 --vm 2 --vm-bytes 128M \
  --timeout 120s
```

In diesem Beispiel wird zunächst die Ausgabe des ersten Startversuchs auf der Konsole angezeigt, bevor der Prozess »abgeschossen« wird. Wenn Sie unmittelbar nach dem Beenden des Containers `docker container ls` ausführen, wird deutlich, dass Docker versucht, den Container erneut zu starten.

```
$ docker container ls
…  IMAGE           …  STATUS                …
…  spkane/train-os …  Up Less than a second …
```

Allerdings wird das Starten des Containers wiederholt fehlschlagen, da wir ihm nicht genügend Arbeitsspeicher zugewiesen haben, um richtig funktionieren zu können. Deshalb gibt Docker schließlich nach drei Versuchen auf und der Container ist in der Ausgabe von `docker container ls` nicht mehr vorhanden.

Container stoppen

Container können nach Belieben gestartet oder gestoppt werden. Nun könnten Sie auf den Gedanken kommen, dass sich das Starten und Stoppen analog zum Pausieren und Fortsetzen eines normalen Prozesses verhält. Das ist allerdings nicht ganz dasselbe. Beim Stoppen wird ein Prozess nicht nur pausiert, sondern tatsächlich vollständig beendet. Ein gestoppter Container erscheint nicht mehr in der normalen Ausgabe des Befehls `docker container ls`. Nach einem Neustart wird Docker versuchen, alle Container, die zum Zeitpunkt des Herunterfahrens noch liefen, neu zu starten. Derselbe Mechanismus ist auch nützlich, um Tests durchzuführen oder einen Container neu zu starten, dessen Start zuvor fehlgeschlagen ist. Mit den Befehlen `docker container pause` bzw. `docker container unpause`, die später noch zur Sprache kommen werden, kann ein Linux-Container angehalten und fortgesetzt werden. Hier soll der Redis-Container jedoch vollständig gestoppt werden:

```
$ docker container stop 092c5dc85044
$ docker container ls
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

Nach dem Stoppen des Containers ist die Liste leer! Nun könnten wir ihn anhand der Container-ID erneut starten, allerdings wäre es wirklich unpraktisch, sich diese kryptische Zeichenkette merken zu müssen. Deshalb verfügt der Befehl `docker ps` über eine weitere Option (`-a`), mit der nicht nur die laufenden, sondern *alle* Container angezeigt werden:

```
$ docker container ls -a
CONTAINER ID  IMAGE     STATUS                    …
092c5dc85044  redis:7.0 Exited (0) 2 minutes ago …
…
```

Die Spalte STATUS zeigt nun an, dass der Container mit dem Rückgabewert 0 (keine Fehler) beendet wurde. Jetzt können wir ihn mit derselben Konfiguration wie zuvor erneut starten:

```
$ docker container start 092c5dc85044
092c5dc85044

$ docker container ls -a
CONTAINER ID  IMAGE     STATUS        …
092c5dc85044  redis:7.0 Up 14 seconds …
…
```

Und voilà: Der Container läuft wieder.

Denken Sie daran, dass erstellte Container vorhanden bleiben, auch wenn sie nicht gestartet werden. Das heißt, dass Sie einen Container jederzeit starten können, ohne ihn erneut erstellen zu müssen. Die Inhalte des Arbeitsspeichers gehen zwar verloren – die Inhalte des Dateisystems sowie die Metadaten, inklusive Umgebungsvariablen und Portanbindungen, werden dagegen gespeichert und sind wieder verfügbar, sobald Sie einen Container erneut starten.

Kommen wir nun noch einmal auf die Vorstellung zurück, dass Container im Grunde genommen Prozesse darstellen, die im Wesentlichen wie alle anderen Prozesse auch mit dem System interagieren. Wir können ihnen Unix-Signale senden, auf die sie reagieren können. In dem vorangegangenen Beispiel (`docker stop`) haben wir dem Container ein SIGTERM-Signal gesendet und dann darauf gewartet, dass er sich ordnungsgemäß beendet. Container verhalten sich unter Linux in Bezug auf die Übertragung von Unix-Signalen an Prozessgruppen wie alle anderen Prozessgruppen auch.

Der Befehl `docker container stop` sendet ein normales SIGTERM-Signal an den Prozess. Wenn Sie das Beenden eines Containers erzwingen wollen, weil er sich nach Ablauf einer gewissen Zeitspanne immer noch nicht beendet hat, können Sie das Argument `-t` wie folgt verwenden:

```
$ docker container stop -t 25 092c5dc85044
```

Durch diesen Befehl wird Docker veranlasst, ein weiteres SIGTERM-Signal zu senden. Sollte sich der Container dann nicht innerhalb von 25 Sekunden beenden, wird ein SIGKILL-Signal ausgesandt, um das Beenden zu erzwingen.

Zwar ist `stop` die beste Methode zum Beenden eines Containers, mitunter kommt es jedoch vor, dass sie nicht funktioniert. In diesem Fall muss das Beenden des Containers erzwungen werden.

Container sofort beenden

Sie haben soeben erfahren, wie man `docker container stop` einsetzt, um einen Container normal zu beenden. Allerdings kommt es oft genug vor, dass sich ein Prozess nicht korrekt verhält, sodass Sie ihn sofort beenden möchten.

Zu diesem Zweck gibt es den Befehl `docker container kill`, der große Ähnlichkeit mit `docker container stop` aufweist:

```
$ docker container start 092c5dc85044
092c5dc85044

$ docker container kill 092c5dc85044
092c5dc85044
```

Der Befehl `docker container ls` zeigt nun erwartungsgemäß an, dass der Container nicht mehr läuft:

```
$ docker container ls
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

Dass Sie den Container nicht mit `stop`, sondern mit `kill` beendet haben, bedeutet allerdings nicht, dass Sie ihn nicht neu starten können: Ein Neustart lässt sich hier ebenso wie bei einem ordnungsgemäß beendeten Container mit `docker container start` durchführen. Gelegentlich kann es dazu erforderlich sein, andere Signale als `stop` oder `kill` an einen Container zu senden. Wie bei dem Linux-Befehl `kill` können mit `docker kill` beliebige Unix-Signale gesendet werden. Angenommen, wir möchten ein `USR1`-Signal an den Container senden, um ihn aufzufordern, eine bestimmte Aktion auszuführen, etwa eine unterbrochene Sitzung wieder aufzunehmen. Das funktioniert folgendermaßen:

```
$ docker container start 092c5dc85044
092c5dc85044

$ docker container kill --signal=USR1 092c5dc85044
092c5dc85044
```

Wenn Ihr Container mit dem `USR1`-Signal etwas anfangen kann, wird er nun darauf reagieren. Mit dieser Methode kann jedes Unix-Standardsignal an einen Container geschickt werden.

Ausführung eines Containers pausieren und fortsetzen

Manchmal wollen wir einen Container wirklich einfach vollständig beenden. Es gibt aber auch Situationen, in denen der Container lediglich eine Weile lang nichts tun soll, beispielsweise weil gerade eine Momentaufnahme des Dateisystems erstellt wird, um ein neues Image zu erzeugen, oder weil die CPU-Rechenleistung gerade an

anderer Stelle dringend benötigt wird. Wenn Sie mit der Handhabung normaler Unix-Prozesse vertraut sind, werden Sie sich vielleicht fragen, wie das eigentlich funktioniert, denn containerisierte Prozesse sind letztlich auch nur Prozesse.

Beim Pausieren kommt der cgroups-Freezer zum Einsatz (*https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt*), der im Wesentlichen schlicht und ergreifend dafür sorgt, dass Ihr Prozess bei der Zuteilung von CPU-Rechenzeit nicht berücksichtigt wird, bis Sie das Pausieren wieder aufheben. Dadurch wird verhindert, dass Ihr Container irgendetwas tut, während er pausiert, und sichergestellt, dass er dabei seinen Gesamtzustand inklusive des Arbeitsspeicherinhalts dennoch beibehält. Anders als beim Beenden eines Containers, bei dem den Prozessen durch das SIGSTOP-Signal mitgeteilt wird, dass sie sich beenden sollen, werden dem Container beim Pausieren keinerlei Informationen über seine Zustandsänderung übermittelt. Das ist ein bedeutsamer Unterschied. Auch einige Docker-Befehle verwenden intern das Pausieren und Fortsetzen. Und so wird ein Container pausiert:

```
$ docker container start 092c5dc85044
092c5dc85044

$ docker container pause 092c5dc85044
092c5dc85044
```

Um Container in Windows zu pausieren und diese Pausierung aufzuheben, müssen Sie Hyper-V oder WSL2 als Virtualisierungstechnologie verwenden.

In der Liste der laufenden Container steht beim Redis-Container in der Spalte STATUS nun (Paused), also pausiert bzw. angehalten:

```
$ docker container ls
CONTAINER ID  IMAGE     … STATUS                  …
092c5dc85044  redis:7.0 … Up 25 seconds (Paused)  …
```

Der Versuch, den Container in diesem Zustand zu benutzen, würde fehlschlagen: Er ist zwar vorhanden, arbeitet aber nicht. An dieser Stelle können wir die Ausführung nun aber mit dem Befehl `docker container unpause` fortsetzen:

```
$ docker container unpause 092c5dc85044
092c5dc85044

$ docker container ls
CONTAINER ID  IMAGE     … STATUS        …
092c5dc85044  redis:7.0 … Up 55 seconds …
```

Jetzt läuft der Container wieder, und der Befehl `docker container ls` berücksichtigt den neuen Zustand ebenfalls. Beachten Sie, dass sich die Zeitangabe in der Spalte STATUS auf den Zeitpunkt bezieht, zu dem die Ausführung fortgesetzt wurde, nicht auf die Gesamtlaufzeit.

Container und Images aufräumen

Nach der Ausführung all dieser Befehle zum Erstellen von Images und Containern und wiederum deren Ausführung hat sich eine Menge an Image-Layern und Containerordnern auf unserem System angesammelt.

Mit dem Befehl `docker container ls -a` können alle auf dem System vorhandenen Container angezeigt werden. Sie müssen alle Container stoppen, die ein bestimmtes Image nutzen, bevor Sie es löschen können. Da wir davon ausgehen, dass Sie das getan haben, kann das Image wie folgt gelöscht werden:

```
$ docker container stop 092c5dc85044
092c5dc85044ls

$ docker container rm 092c5dc85044
092c5dc85044
```

Es ist möglich, einen laufenden Container zu entfernen, wenn Sie bei `docker container rm` das Flag `-f` oder `--force` nutzen.

Nun lassen wir uns alle auf dem System vorhandenen Images anzeigen:

```
$ docker image ls
REPOSITORY       TAG     IMAGE ID      CREATED       SIZE
ubuntu           latest  5ba9dab47459  3 weeks ago   188.3MB
redis            7.0     0256c63af7db  2 weeks ago   117MB
spkane/train-os  latest  78fb082a4d65  4 months ago  254MB
```

Und so wird ein Image inklusive der zugehörigen Dateisysteme gelöscht:

```
$ docker image rm 0256c63af7db
```

Wenn Sie versuchen, ein Image zu löschen, das aktuell von einem Container benutzt wird, erhalten Sie die Fehlermeldung »Conflict, cannot delete«. Sie müssen dann zunächst den oder die fraglichen Container löschen.

Unter bestimmten Umständen, insbesondere nach Abschluss eines Entwicklungszyklus, kann es sinnvoll sein, sämtliche Images und Container von einem System zu entfernen. Einen vorgefertigten Befehl dafür gibt es nicht, aber mit ein wenig Kreativität lässt sich das unschwer erreichen.

Geben Sie den folgenden Befehl ein, um alle Container auf Ihrem Docker-Host zu löschen:

```
$ docker system prune
WARNING! This will remove:
- all stopped containers
- all networks not used by at least one container
- all dangling images
- all build cache
Are you sure you want to continue? [y/N] y
Deleted Containers:
```

```
cbbc42acfe6cc7c2d5e6c3361003e077478c58bb062dd57a230d31bcd01f6190
…
Deleted Images:
deleted: sha256:bec6ec29e16a409af1c556bf9e6b2ec584c7fb5ffbfd7c46ec00b30bf …
untagged: spkane/squid@sha256:64fbc44666405fd1a02f0ec731e35881465fac395e7 …
…
Total reclaimed space: 1.385GB
```

Um alle ungenutzten Images zu löschen statt nur die nicht verwaisten Images, können Sie `docker system prune -a` verwenden.

Es ist auch möglich, spezifischere Kommandos auszuführen, um ähnliche Ziele zu erreichen.

```
$ docker container rm $(docker container ls -a -q)
```

`docker container ls` und `docker images` unterstützen beide ein `filter`-Argument, mit dem Sie Ihre Löschanweisungen in bestimmten Situationen fein konfigurieren können.

Um alle Container zu löschen, die beim Beenden einen von null verschiedenen Rückgabewert geliefert haben, können Sie diesen Filter verwenden:

```
$ docker container rm $(docker container ls -a -q --filter 'exited!=0')
```

Geben Sie den folgenden Befehl ein, um alle nicht getaggten Images zu löschen:

```
$ docker image rm $(docker images -q -f "dangling=true")
```

In der offiziellen Docker-Dokumentation können Sie alles über die gegenwärtig angebotenen Filter nachlesen (*https://docs.docker.com/engine/reference/commandline/ps/#filtering*). Aktuell ist lediglich eine überschaubare Anzahl verfügbar, im Laufe der Zeit werden jedoch aller Wahrscheinlichkeit nach weitere hinzukommen. Und wenn Sie ein wirklich großes Interesse an Filtern haben: Vergessen Sie nicht, dass Docker ein Open-Source-Projekt und damit offen für Beiträge ist.

Darüber hinaus ist es auch möglich, ausgeklügelte Filter zu erstellen, indem Sie mehrere Befehle durch Pipe-Symbole (|) miteinander verknüpfen oder vergleichbare Techniken einsetzen.

Auf Produktivsystemen, auf denen sehr häufig deployt wird, können sich schnell alte Container und ungenutzte Images ansammeln, die viel Speicherplatz belegen. Es kann nützlich sein, ein Skript zu schreiben, das `docker system prune` verwendet und regelmäßig über einen `cron`-Job oder einen `systemd`-Timer ausgeführt wird.

Windows-Container

Bis jetzt haben wir uns vollständig auf Docker-Befehle für Linux-Container konzentriert, da es die meistgenutzte Form von Containern ist und auf allen Docker-Plattformen funktioniert. Seit 2016 unterstützt die Microsoft-Windows-Plattform Windows-

Docker-Container, die native Windows-Anwendungen beinhalten und mit dem gewohnten Satz an Docker-Kommandos genutzt werden können.

Auf Windows-Container haben wir in diesem Buch keinen starken Fokus gelegt, da sie nur einen sehr kleinen Teil vom Produktivsystem ausmachen und nicht zu 100 % kompatibel mit dem Rest des Docker-Ökosystems sind. Das hängt damit zusammen, dass diese Windows-spezifischen Container Images benötigen. Trotzdem wächst die Zahl der Nutzer, die Docker für Windows einsetzen. Deshalb betrachten wir uns hier einmal näher, wie Windows-Container funktionieren. Tatsächlich verhalten sie sich, bis auf die enthaltenen Dateien, wie auch die üblichen Linux-Container. In diesem Abschnitt behandeln wir, wie Sie Windows-Container auf Windows 10 mit Hyper-V und Docker zum Laufen bringen.

Sie müssen Docker Desktop auf einer kompatiblen 64-Bit-Version von Windows 10 oder höher verwenden.

Das Erste, was Sie tun müssen, ist, Docker von Linux- auf Windows-Container umzustellen. Dafür müssen Sie zunächst auf das Docker-Icon in Ihrer Taskleiste klicken, dort *Switch to Windows containers...* auswählen und die Umstellung bestätigen (siehe die Abbildungen 5-1 und 5-2).

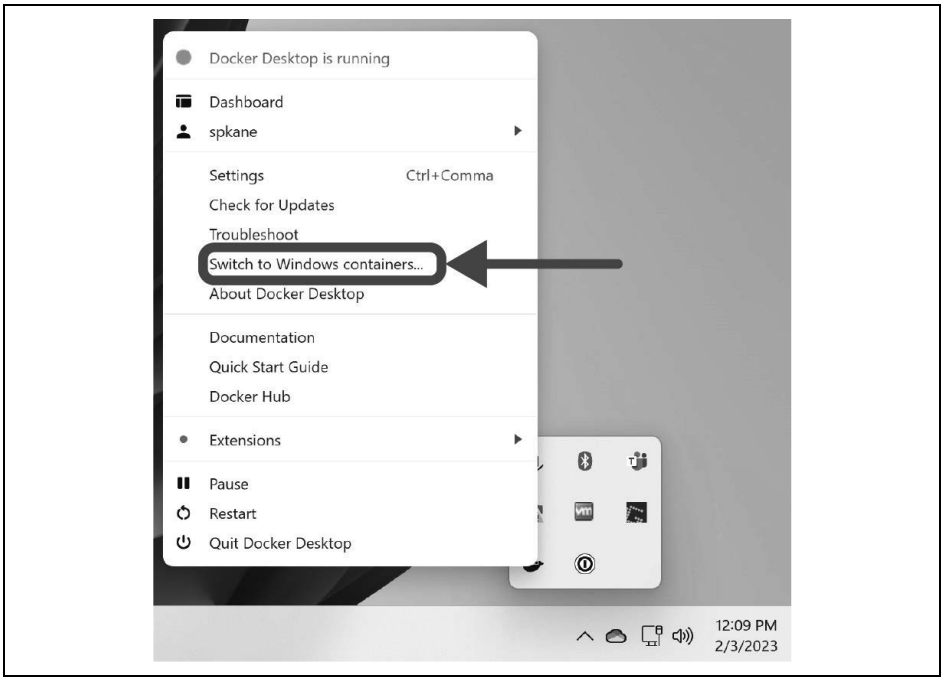

Abbildung 5-1: Zu Windows-Containern wechseln

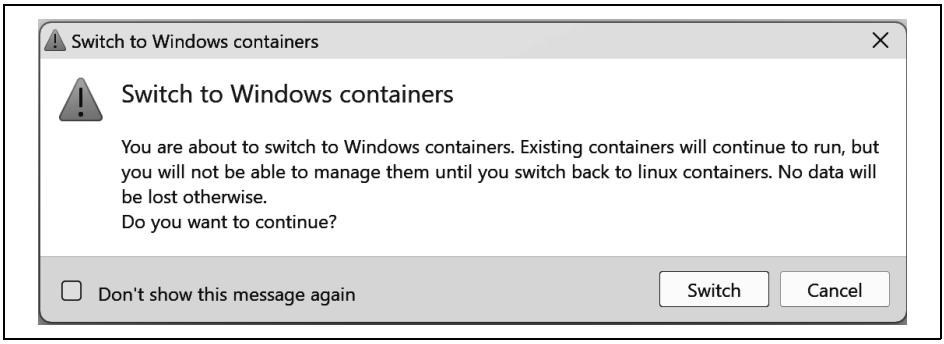

Abbildung 5-2: Wechseln zu Windows-Containern bestätigen

Der Prozess kann eine Weile dauern, meistens passiert es aber unmittelbar. Man wird allerdings leider nicht darüber informiert, ob die Umstellung erfolgreich war, daher müssen Sie erneut mit rechts auf das Docker-Icon klicken, um zu schauen, ob dort der Menüpunkt *Switch to Linux containers…* erscheint.

Wenn Sie beim ersten Klick sehen, dass der Menüpunkt *Switch to Linux containers…* heißt, ist Docker bereits so konfiguriert, dass es Windows-Container verwendet.

Sie können einen einfachen Windows-Container testen, indem Sie die PowerShell öffnen (*https://oreil.ly/SiTXP*) und folgendes Kommando ausführen:

```
PS C:\> docker container run --rm -it mcr.microsoft.com/powershell `
          pwsh -command `
          'Write-Host "Hello World from Windows `($IsWindows`)"'

Hello World from Windows (True)
```

Dadurch wird ein Base-Container für PowerShell heruntergeladen und gestartet (*https://hub.docker.com/_/microsoft-powershell*), um dann per Skripting »Hello World from Windows (True)« am Bildschirm auszugeben.

Lautet die Ausgabe aus dem vorherigen Befehl »Hello World from Windows (False)«, sind Sie nicht in den Windows-Container-Mode gewechselt oder lassen diesen Befehl auf einer Nicht-Windows-Plattform laufen.

Möchten Sie ein Windows-Container-Image erstellen, das ungefähr denselben Befehl ausführt, sieht das Dockerfile wie folgt aus:

```
# escape=`
FROM mcr.microsoft.com/powershell
SHELL ["pwsh", "-command"]

RUN Add-Content C:\helloworld.ps1 `
      'Write-Host "Hello World from Windows"'

CMD ["pwsh", "C:\\helloworld.ps1"]
```

Wenn wir dieses Dockerfile bauen, basiert es auf dem Container `mcr.microsoft.com/powershell`. Es erstellt ein sehr kleines PowerShell-Skript und weist das Image an, das Skript standardmäßig zu starten, wenn das Image genutzt wird, um den Container zu starten.

Sie haben möglicherweise bemerkt, dass wir den Backslash (\) in der `CMD`-Zeile des Dockerfiles mit einem zusätzlichen Backslash maskiert haben. Das ist notwendig, da Docker seine Wurzeln im Unix-Umfeld hat und der Backslash eine spezielle Bedeutung in Unix-Shells besitzt. Obwohl wir das Escape-Zeichen für das Dockerfile so angepasst haben (*https://docs.docker.com/engine/reference/builder/#escape*), dass es dem PowerShell-Standard entspricht (was wir über die SHELL-Direktive festlegen, *https://docs.docker.com/engine/reference/builder/#shell-form-entrypoint-example*), müssen wir manche Backslashs trotzdem maskieren, um sicherzustellen, dass Docker sie nicht fehlinterpretiert.

Wenn Sie nun das Dockerfile erstellen, sehen Sie etwa Folgendes:

```
PS C:\> docker image build -t windows-helloworld:latest .

Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM mcr.microsoft.com/powershell
 ---> 7d8f821c04eb
Step 2/4 : SHELL ["pwsh", "-command"]
 ---> Using cache
 ---> 1987fb489a3d
Step 3/4 : RUN Add-Content C:\helloworld.ps1
                'Write-Host "Hello World from Windows"'
 ---> Using cache
 ---> 37df47d57bf1
Step 4/4 : CMD ["pwsh", "C:\\helloworld.ps1"]
 ---> Using cache
 ---> 03046ff628e4
Successfully built 03046ff628e4
Successfully tagged windows-helloworld:latest
```

Nach dem Start des resultierenden Image sehen Sie das:

```
PS C:\> docker container run --rm -ti windows-helloworld:latest

Hello World from Windows
```

Microsoft stellt eine gute Dokumentation zu Windows-Containern bereit (*https://oreil.ly/fYMHl*), die auch ein Beispiel dazu enthält, wie man .NET-Anwendungen erstellt (*https://oreil.ly/WG2W2*).

Wenn Sie auf einer Windows-Plattform arbeiten, ist es nützlich, zu wissen, dass man eine verbesserte Isolierung für die Container erzielt, wenn man diese mit Hyper-V innerhalb einer dedizierten und sehr leichtgewichtigen virtuellen Machine laufen lässt. Das lässt sich einfach bewerkstelligen, indem Sie `--isolation=hyperv` mit `docker container create` und `docker container run` übergeben. Dies führt zu einer geringeren Performance und Ressourceneinschränkungen, aller-

dings wird damit die Isolierung der Container signifikant verbessert. Mehr dazu finden Sie in der Dokumentation (*https://learn.microsoft.com/enus/virtualization/windowscontainers/manage-containers/hyperv-container*).

Selbst wenn Sie planen, für den Großteil Ihrer Arbeit Windows-Container zu verwenden, sollten Sie für den Rest des Buchs zurück zu Linux-Containern wechseln. Damit ist sichergestellt, dass alle Beispiele wie erwartet funktionieren. Sobald Sie mit dem Lesen fertig sind und Ihre ersten eigenen Container bauen wollen, können Sie zurückwechseln.

Zur Erinnerung: Sie können Linux-Container über den Rechtsklick auf das Docker-Icon und die Auswahl von *Switch to Linux containers* wieder aktivieren.

So geht es weiter

Im nächsten Kapitel werden wir weiter erkunden, welche Möglichkeiten Docker eröffnet. Fürs Erste ist es sicherlich lohnenswert, wenn Sie ein wenig herumexperimentieren. Wir empfehlen, die verschiedenen hier vorgestellten Befehle zur Handhabung von Containern auszuprobieren, um sich mit den Optionen und der Syntax der Kommandozeilenbefehle im Allgemeinen vertraut zu machen. Versuchen Sie, gestoppte oder pausierte Container wieder zum Laufen bringen, und probieren Sie aus, wie weit Sie kommen. Und wenn Sie mit sich zufrieden sind, lesen Sie in Kapitel 6 weiter!

Inhalt