2 Container erstellen und ausführen

Kubernetes ist eine Plattform zum Erstellen, Deployen und Managen verteilter Applikationen. Diese Applikationen gibt es in vielen Größen und Formen, aber letztendlich bestehen sie alle aus einer oder mehreren Programmen, die auf bestimmten Maschinen laufen. Diese Programme erwarten Eingaben, bearbeiten Daten und geben dann die Ergebnisse zurück. Bevor wir uns überhaupt Gedanken über den Aufbau eines verteilten Systems machen können, müssen wir uns zunächst überlegen, wie wir die *Anwendungs-Container-Images* bauen, die diese Programme enthalten und aus denen unser verteiltes System bestehen wird.

Anwendungsprogramme setzen sich meist aus einer Sprach-Laufzeitumgebung (Runtime), gemeinsam genutzten Bibliotheken und Ihrem Quellcode zusammen. In vielen Fällen greift Ihre Anwendung auf externe Bibliotheken wie `libc` oder `libssl` zurück. Diese externen Bibliotheken werden meist als gemeinsam genutzte Komponenten mit dem Betriebssystem ausgeliefert, das Sie auf einer bestimmten Maschine installiert haben.

Es führt zu Problemen, wenn eine Anwendung auf dem Laptop eines Programmierers entwickelt wurde und von einer gemeinsam genutzten Bibliothek abhängt, die beim Rollout auf dem Produktiv-OS nicht vorhanden ist. Selbst wenn Entwicklungs- und Produktiv-Umgebungen die genau gleiche Version des Betriebssystems nutzen, können Probleme entstehen, wenn Entwickler vergessen, Asset-Dateien mit in ein Paket aufzunehmen, das sie in die Produktiv-Umgebung deployen.

Für das klassische Ausführen mehrerer Anwendungen auf einer einzelnen Maschine ist es unerlässlich, dass all diese Programme die gleichen Versionen der gemeinsam genutzten Bibliotheken verwenden. Werden die verschiedenen Anwendungen von unterschiedlichen Teams oder Organisationen entwickelt, führen diese gemeinsamen Abhängigkeiten zu zusätzlicher, unnötiger Komplexität und Abhängigkeiten zwischen den Teams.

Ein Programm kann nur dann erfolgreich ausgeführt werden, wenn es zuverlässig auf die Maschine deployt wurde, auf der es laufen soll. Allzu häufig gehört zum Stand der Technik beim Deployen der Einsatz von imperativen Skripten, was ganz unvermeidlich zu seltsamen und unverständlichen Fehlerfällen führt. Das macht das Ausrollen einer neuen Version eines verteilten Systems (oder von Teilen davon) zu einer arbeitsaufwendigen und schwierigen Aufgabe.

In Kapitel 1 haben wir uns für den Wert immutabler Images und Infrastruktur eingesetzt. Es zeigt sich, dass genau diese Immutabilität den Wert der Container-Images ausmacht. Wie Sie sehen werden, lassen sich damit ganz einfach alle Probleme des eben beschriebenen Abhängigkeits-Managements und der Kapselung lösen.

Bei der Arbeit mit Anwendungen ist es häufig hilfreich, sie so zu einem Paket zusammenzufassen, dass sie sich leicht mit anderen einsetzen lassen. Docker, die Standard-Container-Runtime-Engine, macht es leicht, ein Executable zu verpacken und sie in eine Remote-Registry zu schieben, aus der sie später von anderen geholt werden kann. Aktuell stehen Container-Registries in allen großen öffentlichen Clouds zur Verfügung und die Services zum Bauen von Images in der Cloud gibt es dort auch sehr oft. Sie können zudem Ihre eigene Registry als Open Source oder kommerzielles System betreiben. Diese Registries erleichtern es den Anwendern, private Images zu managen und zu deployen, während die Image-Builder Services eine leichte Integration mit Continuous Delivery Systems ermöglichen.

In diesem Kapitel werden wir mit einer einfachen Beispiel-Anwendung arbeiten, die wir für dieses Buch gebaut haben, um diesen Ablauf zu demonstrieren. Sie finden sie auf GitHub unter *https://github.com/kubernetes-up-and-running/kuard*.

Container-Images bündeln ein Programm und seine Abhängigkeiten unter einem Root-Dateisystem zu einem einzelnen Artefakt. Das verbreitetste Image-Format ist das Docker-Image-Format, das von der Open Container Initiative im OCI-Image-Format standardisiert wurde. Kubernetes unterstützt sowohl Docker- wie auch OCI-kompatible Images via Docker und anderen Runtimes. Docker-Images enthalten zusätzliche Metadaten, die von einer Container-Runtime genutzt werden, um abhängig vom Inhalt des Container-Images eine Anwendungs-Instanz zu starten.

In diesem Kapitel werden die folgenden Themen behandelt:

- Wie eine Anwendung mithilfe des Docker-Image-Formats verpackt wird.
- Wie eine Anwendung mit der Docker-Container-Runtime gestartet wird.

2.1 Container-Images

Für so gut wie jeden findet die erste Begegnung mit einer Container-Technologie mit einem Container-Image statt. Ein *Container-Image* ist ein Binary-Paket, das alle für das Ausführen eines Programms in einem OS-Container notwendigen Dateien zusammenfasst. Abhängig von Ihrer ersten Erfahrung mit Containern werden Sie entweder ein Container-Image in Ihrem lokalen Dateisystem bauen oder ein bestehendes Image aus einer *Container-Registry* herunterladen. In beiden Fällen können Sie – sobald das Container-Image auf Ihrem Rechner vorhanden ist – dieses Image starten, um eine laufende Anwendung innerhalb eines OS-Containers zu erzeugen.

2.1.1 Das Docker-Image-Format

Das beliebteste und verbreitetste Format für Container-Images ist das Docker-Image-Format. Es wurde vom Docker-Open-Source-Projekt für das Packen, Verteilen und Ausführen von Containern mithilfe des Befehls `docker` entwickelt. Später arbeiteten Docker, Inc. und andere daran, das Container-Image-Format über das Open Container Initiative (OCI)-Projekt zu standardisieren. Die OCI-Standards wurden zwar Mitte 2017 als Version 1.0 veröffentlicht, aber ihre Verbreitung geschieht nur sehr zögerlich. Das Docker-Image-Format ist weiterhin der De-facto-Standard. Es besteht aus einer Reihe von Dateisystem-Layern (Schichten). Jeder Layer fügt Dateien zum vorigen Layer im Dateisystem hinzu, entfernt welche oder passt sie an. Dies ist ein Beispiel für ein *Overlay*-Dateisystem. Das Overlay-System wird sowohl beim Erstellen des Image wie auch bei dessen tatsächlichem Einsatz genutzt. Zur Laufzeit gibt es eine ganze Reihe von verschiedenen konkreten Implementierungen solcher Dateisysteme, unter anderem `aufs`, `overlay` und `overlay2`.

Container-Layering

Die Begriffe »Docker Image Format« und »Container-Images« sind vielleicht ein wenig verwirrend. Das Image ist keine einzelne Datei, sondern eher eine Spezifikation für eine Manifest-Datei, die auf andere Dateien verweist. Das Manifest und die zugehörigen Dateien werden von den Benutzern oft als eine Einheit behandelt. Durch diese Indirektion sind ein effizienteres Abspeichern und Übermitteln möglich. Neben diesem Format gibt es noch eine API zum Hoch- und Runterladen von Images in und aus einer Image Registry.

Container-Images bestehen aus einer Folge von Dateisystem-Layern, bei denen jede Schicht die vorigen Layer übernimmt und anpasst. Um das besser zu verstehen, wollen wir ein paar Container bauen. Beachten Sie, dass die Layer eigentlich von unten nach oben gestapelt werden, aber zum leichteren Verständnis gehen wir andersherum vor:

```
.
└── Container A: nur Basis-Betriebssystem, wie z.B. Debian
    └── Container B: baut auf A auf, fügt Ruby v2.1.10 hinzu
    └── Container C: baut auf A auf, fügt Golang v1.6 hinzu
```

Wir haben jetzt drei Container: A, B und C. B und C sind von A *abgespalten (forked)* und haben abgesehen von den Dateien des Basis-Containers keine Gemeinsamkeiten. Wir können jetzt noch weitergehen und zu B noch Rails (Version 4.2.6) hinzufügen. Vielleicht wollen wir noch eine alte Anwendung unterstützen, die eine ältere Version von Rails (Version 3.2.x) benötigt. Dazu können wir ein Container-Image bauen, das diese Anwendung basierend auf B unterstützt, um sie vielleicht irgendwann einmal nach Version 4 zu aktualisieren:

```
. (Fortsetzung von oben)
└── Container B: baut auf A auf, fügt Ruby v2.1.10 hinzu
    └── Container D: baut auf B auf, fügt Rails v4.2.6 hinzu
    └── Container E: baut auf B auf, fügt Rails v3.2.x hinzu
```

Konzeptionell gesehen baut jeder Layer eines Container-Image auf einem vorigen auf. Jeder Verweis auf einen Eltern-Layer ist ein Zeiger. Während es sich bei diesem Beispiel hier um einen einfachen Satz von Containern handelt, können andere Container mit realen Anwendungen Teil eines größeren und umfangreicheren gerichteten azyklischen Graphen sein.

Container-Images werden meist mit einer Container-Konfigurationsdatei kombiniert, die Anweisungen zum Einrichten der Container-Umgebung und Ausführen eines Anwendungs-Entrypoint enthält. Die Container-Konfiguration besitzt auch oft Informationen zum Einrichten der Netzwerkumgebung, zu Namensräumen, Ressourcen-Begrenzungen (cgroups) und welche `syscall`-Einschränkungen auf eine laufende Container-Instanz angewendet werden sollten. Das Root-Dateisystem und die Konfigurationsdatei des Containers werden meist im Docker-Image-Format zusammengefasst.

Container gehören zu einer von zwei Hauptkategorien:

- System-Container
- Anwendungs-Container

System-Container versuchen, virtuelle Maschinen nachzubilden, und führen häufig einen vollständigen Boot-Prozess durch. Oft enthalten sie eine Reihe von System-Diensten, die sich gerne in einer VM finden, wie zum Beispiel `ssh`, `cron` und `syslog`. Als Docker noch neu war, waren diese Arten von Containern sehr verbreitet. Mit der Zeit betrachtete man sie aber als schlechte Praxis und man hat sich eher auf Anwendungs-Container konzentriert.

Anwendungs-Container unterscheiden sich von System-Containern darin, dass sie im Allgemeinen nur ein einzelnes Programm in sich laufen lassen. Das mag eine unnötige Einschränkung sein, aber es bietet die perfekte Granularitäts-Stufe für das Zusammenstellen skalierbarer Anwendungen und ist eine Design-Philosophie, die durch Pods stark gefördert wird (auf Pods gehen wir im Detail in Kap. 5 ein).

2.2 Anwendungs-Images mit Docker bauen

Im Allgemeinen konzentrieren sich Container-Orchestrierungs-Systeme wie Kubernetes auf das Bauen und Deployen von verteilten Systemen, die aus Anwendungs-Containern bestehen. Daher werden wir uns im Rest dieses Kapitels auf solche Container fokussieren.

2.2.1 Dockerfiles

Ein Dockerfile kann genutzt werden, um das Erstellen eines Docker-Container-Image zu automatisieren.

Beginnen wir damit, ein Anwendungs-Image für ein einfaches Node.js-Programm zu bauen. Dieses Beipiel würde für viele andere dynamische Sprachen wie Python oder Ruby sehr ähnlich aussehen.

Die einfachsten npm/Node/Express-Anwendungen haben zwei Dateien: *package.json* (Listing 2–1) und *server.js* (Listing 2–2). Legen Sie diese in einem Verzeichnis ab und führen Sie dann `npm install express --save` aus, um eine Abhängigkeit von Express einzurichten und es zu installieren.

```
{
  "name": "simple-node",
  "version": "1.0.0",
  "description": "Eine einfache Beispielanwendung", "main": "server.js",
  "scripts": {
    "start": "node server.js" },
  "author": ""
}
```

Listing 2–1 *package.json*

```
var express = require('express');

var app = express();
app.get('/', function (req, res) {
  res.send('Hello World!');
});
app.listen(3000, function () {
  console.log('Listening on port 3000!');
  console.log(' http://localhost:3000');
});
```

Listing 2–2 *server.js*

Um das in einem Docker-Image zu verpacken, müssen wir zwei zusätzliche Dateien erstellen: *.dockerignore* (Listing 2–3) und das *Dockerfile* (Listing 2–4). Bei Letzterem handelt es sich um ein Rezept zum Bauen des Container-Image, während *.dockerignore* die Dateien definiert, die beim Kopieren der Dateien in das Image ignoriert werden sollen. Eine vollständige Beschreibung der Syntax des Dockerfiles finden Sie auf der Docker-Website unter *https://dockr.ly/2XUanvl*.

```
node_modules
```

Listing 2–3 *.dockerignore*

```
# Beginne mit einem Node.js 10 (LTS) Image ❶
FROM node:10

# Verzeichnis im Image, in dem alle Befehle laufen werden ❷
WORKDIR /usr/src/app

# Package-Dateien kopieren und Abhängigkeiten installieren ❸
COPY package*.json ./
RUN npm install

# Alle App-Dateien in das Image kopieren ❹
COPY . .

# Standardbefehl, der beim Starten des Containers laufen soll ❺
CMD [ "npm", "start" ]
```

Listing 2–4 *Dockerfile*

❶ Jedes Dockerfile baut auf anderen Container-Images auf. Diese Zeile gibt an, dass wir vom Image `node:10` auf dem Docker Hub ausgehen. Dabei handelt es sich um ein vorkonfiguriertes Image mit Node.js 10.

❷ Diese Zeile legt das Arbeitsverzeichnis im Container-Image für alle folgenden Befehle fest.

❸ Diese zwei Zeilen initialisieren die Abhängigkeiten für Node.js. Zuerst kopieren wir die Package-Dateien in das Image. Dazu gehören *package.json* und *package-lock.json*. Dann führt die `RUN`-Anweisung den passenden Befehl *im Container aus*, um die notwendigen Abhängigkeiten zu installieren.

❹ Jetzt kopieren wir die restlichen Programmdateien in das Image. Das ist alles außer *node_modules*, da dies durch die Datei *.dockerignore* ausgeschlossen wurde.

❺ Schließlich legen wir noch den Befehl fest, der ausgeführt werden soll, wenn der Container gestartet wird.

Führen Sie den folgenden Befehl aus, um das Docker-Image `simple-node` zu erstellen:

```
$ docker build -t simple-node .
```

Wollen Sie dieses Image ausführen, können Sie das mit folgendem Befehl tun. Sie können dann *http://localhost:3000* ansteuern, um auf das Programm zuzugreifen, das im Container läuft:

```
$ docker run --rm -p 3000:3000 simple-node
```

Unser Image `simple-node` lebt jetzt in der lokalen Docker-Registry, in der es gebaut wurde, und es steht auch nur einer einzelnen Maschine zur Verfügung. Seine wirkliche Mächtigkeit kann Docker erst ausspielen, wenn Images auf Tausenden von Maschinen und in der größeren Docker-Community verfügbar gemacht werden.

2.2.2 Die Image-Größe optimieren

Es gibt eine Reihe von Fallstricken, über die man stolpern kann, wenn man mit Container-Images experimentiert, und die zu viel zu großen Images führen. Denken Sie vor allem daran, dass Dateien, die in Folge-Layern aus dem System entfernt werden, im Image immer noch vorhanden sind – man kann nur nicht auf sie zugreifen. Schauen Sie sich diese Situation an:

```
.
└── Layer A: enthält eine große Datei namens 'BigFile'
    └── Layer B: entfernt 'BigFile'
        └── Layer C: baut auf B auf, fügt statisches Binary hinzu
```

Sie denken vielleicht, dass *BigFile* im Image nicht mehr vorhanden ist. Denn wenn Sie es laufen lassen, können Sie schließlich nicht mehr darauf zugreifen. Aber tatsächlich ist die Datei in Layer A immer noch vorhanden – wann immer Sie das Image pullen oder pushen, wird *BigFile* weiterhin durch das Netz geschoben, auch wenn Sie die Datei nicht mehr ansprechen können.

Ein weiterer Fallstrick betrifft das Cachen und Bauen von Images. Denken Sie daran, dass es sich bei jedem Layer um ein unabhängiges Delta zum vorhergehenden Layer handelt. Immer dann, wenn Sie einen Layer ändern, ändert sich auch jeder folgende Layer. Passen Sie den vorigen Layer an, müssen Sie Ihr eigenes Image neu bauen, neu pushen und neu pullen.

Um das besser zu verstehen, schauen Sie sich diese beiden Images an:

```
.
└── Layer A: enthält Basis-OS
    └── Layer B: ergänzt Quellcode server.js
        └── Layer C: installiert Paket 'node'
```

und

```
.
└── Layer A: enthält Basis-OS
    └── Layer B: installiert Paket 'node'
        └── Layer C: ergänzt Quellcode server.js
```

Es scheint so zu sein, dass sich beide Images gleich verhalten werden, und tatsächlich tun sie das zu Beginn auch. Aber überlegen Sie sich, was passiert, wenn sich *server.js* ändert. In dem einen Fall muss nur diese Änderung gepullt und gepusht werden, im anderen Fall müssen aber sowohl *server.js* als auch der Layer mit dem `node`-Paket gepullt und gepusht werden, weil Letzterer vom Layer mit *server.js* abhängt. Im Allgemeinen sollten Sie Ihre Layer so anordnen, dass die mit häufigeren Änderungen nach denen mit den weniger häufigen Änderungen kommen, um die Image-Größe beim Pullen und Pushen zu optimieren. Darum kopieren wir in Listing 2–4 die *package*.json*-Dateien und installieren die Abhängigkeiten, bevor wir den Rest der Programmdateien kopieren. Ein Entwickler wird die Programmdateien viel häufiger aktualisieren und anpassen, als das bei den Abhängigkeiten der Fall ist.

2.2.3 Sicherheit von Images

Wenn es um Sicherheit geht, sollte man keine Abstriche machen. Beim Bauen von Images, die schließlich in einem produktiven Kubernetes-Cluster laufen sollen, müssen Sie darauf achten, sich beim Packen und Verteilen von Anwendungen an den Best Practices zu orientieren. So sollten Sie zum Beispiel keine Container bauen, in denen die Passwörter »eingebacken« sind – und dazu gehört nicht nur der letzte Layer, sondern es betrifft alle Layer im Image. Eines der kontraintuitiven Probleme von Container-Layern ist, dass das Löschen einer Datei in einem Layer diese Datei nicht aus vorigen Layern entfernt. Sie nimmt weiterhin Platz ein und lässt sich von jedem mit den richtigen Werkzeugen auslesen – ein einfallsreicher Angreifer kann schlicht ein Image bauen, das nur aus den Layern mit dem Passwort besteht.

Secrets und Images sollten *niemals* vermischt werden. Wenn Sie das tun, werden Sie gehackt werden und damit Ihre ganze Firma oder Abteilung in Verruf bringen. Wir wollen alle irgendwann einmal berühmt sein, aber es gibt bessere Wege, das zu erreichen.

2.3 Multistage Image Build

Große Images entstehen sehr oft unabsichtlich, wenn das Kompilieren des Programms Teil der Konstruktion des Anwendungs-Container-Image ist. Es scheint logisch zu sein, den Code als Teil des Image-Builds zu kompilieren, und es ist der einfachste Weg, ein Container-Image aus Ihrem Programm zu erstellen. Das Problem ist aber, dass damit alle unnötigen Entwicklungstools verbleiben, die meist ziemlich groß sind. Sie liegen dann in Ihrem Image herum und verlangsamen Ihre Deployments.

Um dieses Problem zu lösen, hat Docker *Multistage Builds* entwickelt. Dabei wird aus einem Dockerfile nicht einfach ein einzelnes Image erstellt, sondern mehrere. Jedes Image wird als Stage angesehen und Artefakte können aus den vorigen Stages in den aktuellen Stage kopiert werden.

Machen wir das an einem Beispiel deutlich und schauen uns an, wie wir unsere Beispielanwendung `kuard` bauen. Das ist eine halbwegs komplizierte Anwendung, zu der ein React.js-Frontend gehört (mit seinem eigenen Build-Prozess), das wiederum in ein Go-Programm eingebunden wird. Das Go-Programm führt einen Backend-API-Server aus, mit dem das React.js-Frontend interagiert.

Ein einfaches Dockerfile könnte so aussehen:

```
FROM golang:1.11-alpine

# Node uand NPM installieren
RUN apk update && apk upgrade && apk add --no-cache git nodejs bash npm

# Abhängigkeiten für Go-Teil des Build holen
RUN go get -u github.com/jteeuwen/go-bindata/...
RUN go get github.com/tools/godep

WORKDIR /go/src/github.com/kubernetes-up-and-running/kuard

# Sourcen kopieren
COPY . .

# Variablen, die das Build-Skript erwartet
ENV VERBOSE=0
ENV PKG=github.com/kubernetes-up-and-running/kuard
ENV ARCH=amd64
ENV VERSION=test

# Build ausführen. Dieses Skript ist Teil der Sourcen.
RUN build/build.sh

CMD [ "/go/bin/kuard" ]
```

Dieses Dockerfile erstellt ein Container-Image mit einem statischen Executable, aber es enthält auch alle Go-Entwicklungstools, die Tools zum Bauen des React.js-Frontends und den Quellcode für die Anwendung, was beides von der eigentlichen Anwendung nicht benötigt wird. Das Image wird so – addiert man alle Layer auf – über 500 MB groß.

Wie gehen wir nun bei Multistage Builds vor? Schauen wir uns dieses Multistage-Dockerfile an:

```
# STAGE 1: Build
FROM golang:1.11-alpine AS build

# Node und NPM installieren
RUN apk update && apk upgrade && apk add --no-cache git nodejs bash npm

# Abhängigkeiten für Go-Teil des Builds holen
RUN go get -u github.com/jteeuwen/go-bindata/...
RUN go get github.com/tools/godep

WORKDIR /go/src/github.com/kubernetes-up-and-running/kuard

# Sourcen kopieren
COPY . .

# Variablen, die das Build-Skript erwartet
ENV VERBOSE=0
ENV PKG=github.com/kubernetes-up-and-running/kuard
ENV ARCH=amd64
ENV VERSION=test

# Build ausführen. Dieses Skript ist Teil der Sourcen.
RUN build/build.sh

# STAGE 2: Deployment
FROM alpine

USER nobody:nobody
COPY --from=build /go/bin/kuard /kuard

CMD [ "/kuard" ]
```

Dieses Dockerfile erzeugt zwei Images. Das erste ist das *build*-Image mit dem Go-Compiler, der React.js-Toolchain und dem Quellcode für das Programm. Das zweite ist das *deployment*-Image, das einfach nur das kompilierte Binary enthält. Wenn Sie ein Container-Image über Multistage Builds bauen, können Sie die Größe des eigentlich benötigten Container-Image um Hunderte von Megabytes verkleinern und damit Ihre Deployment-Zeiten drastisch verkürzen, da die Deployment-Latenz im Allgemeinen vor allem von der Netzwerk-Performance abhängt. Das aus diesem Dockerfile erzeugte Image ist nun nur noch ungefähr 20 MB groß.

Sie können dieses Image mit den folgenden Befehlen bauen:

```
$ docker build -t kuard .
$ docker run --rm -p 8080:8080 kuard
```

2.4 Images in einer Remote-Registry ablegen

Was nützt einem ein Container-Image, wenn es nur auf einem einzelnen Rechner verfügbar ist?

Kubernetes baut darauf auf, dass in einem Pod-Manifest beschriebene Images auf jeder Maschine im Cluster zur Verfügung stehen. Eine Möglichkeit, dieses Image auf allen Maschinen im Cluster nutzen zu können, wäre ein Export des `kuard`-Image und der Import auf allen anderen Maschinen im Kubernetes-Cluster. Aber das klingt doch sehr aufwendig. Es schreit geradezu nach Fehlern, Docker-Images manuell zu exportieren und zu importieren. Sag nein!

Standard in der Docker-Community ist, Docker-Images in einer Remote-Registry abzulegen. Es gibt unglaublich viele Optionen rund um Docker-Registries, und wofür Sie sich entscheiden, hängt stark davon ab, wie Ihre Sicherheitsanforderungen aussehen und welche Kollaborations-Features Sie benötigen.

Im Allgemeinen müssen Sie zunächst entscheiden, ob Sie eine private oder eine öffentliche Registry brauchen. Öffentliche Registries erlauben es jedermann, dort abgelegte Images herunterzuladen, während für private Registries eine Authentifizierung notwendig ist. Machen Sie sich also Gedanken über Ihren Use Case.

Öffentliche Registries sind super, wenn Sie Images der ganzen Welt verfügbar machen wollen, denn sie erlauben einen einfachen, nicht authentifizierten Einsatz der Container-Images. Sie können Ihre Software als Container-Image gut verteilen und haben die Sicherheit, dass die Anwender auf der ganzen Welt die gleiche Software einsetzen.

Im Gegensatz dazu ist eine private Registry gut, wenn Sie Anwendungen ablegen wollen, die nur für Ihren Service benötigt werden und die sonst niemand einsetzen soll.

Unabhängig davon brauchen Sie zum Pushen eines Image eine Authentifizierung für die Registry. Sie können das im Allgemeinen mit dem Befehl `docker login` erreichen, allerdings gibt es für die verschiedenen Registries Unterschiede im Detail. In den hier eingesetzten Beispielen pushen wir auf die Registry der Google Cloud Platform – die Google Container Registry (GCR). Andere Clouds, so zum Beispiel Azure oder Amazon Web Services (AWS), besitzen ebenfalls gehostete Container Registries. Für Neueinsteiger, die öffentlich lesbare Images nutzen wollen, ist der Docker Hub (*https://hub.docker.com*) ein guter Ausgangspunkt.

Sind Sie angemeldet, können Sie das `kuard`-Image taggen, indem Sie die gewünschte Docker-Registry anfügen. Sie können auch eine andere Kennung anhängen, die normalerweise für die Version oder Variante dieses Image genutzt wird, indem Sie sie durch einen Doppelpunkt (`:`) abtrennen:

```
$ docker tag kuard gcr.io/kuar-demo/kuard-amd64:blue
```

Dann können Sie das `kuard`-Image pushen:

```
$ docker push gcr.io/kuar-demo/kuard-amd64:blue
```

Jetzt steht das kuard-Image in einer Remote-Registry zur Verfügung und Sie können es über Docker deployen. Weil wir es in die öffentliche Docker-Registry geschoben haben, steht es jedermann ohne Authentifizierung zur Verfügung.

2.5 Die Docker Container Runtime

Kubernetes stellt eine API für das Beschreiben eines Anwendungs-Deployments bereit, baut dabei aber auf eine Container-Runtime, um einen Anwendungs-Container einzurichten, und nutzt die Container-spezifischen APIs für das Ziel-Betriebssystem. Auf einem Linux-System müssen zum Beispiel cgroups und Namensräume konfiguriert werden. Die Schnittstelle für diese Container-Runtime wird durch den Container Runtime Interface (CRI)-Standard definiert. Die CRI API wird von einer Reihe verschiedener Programme implementiert, unter anderem durch das von Docker gebaute `containerd-cri` und durch die `cri-o`-Implementierung von Red Hat.

2.5.1 Container mit Docker ausführen

Auch wenn Container in Kubernetes im Allgemeinen durch einen Daemon gestartet werden, der unter dem Namen *kubelet* auf jedem Knoten läuft, ist der Beginn mit Containern einfacher über das Befehlszeilentool von Docker gemacht. Das Docker-CLI-Tool kann genutzt werden, um Container zu deployen. Für einen Zugriff auf das Image `gcr.io/kuar-demo/kuard-amd64:blue` verwenden Sie folgenden Befehl:

```
$ docker run -d —name kuard \
  --publish 8080:8080 \
  gcr.io/kuar-demo/kuard-amd64:blue
```

Dieser Befehl startet die kuard-Datenbank und bildet Port 8080 auf Ihrem lokalen Rechner auf Port 8080 des Containers ab. Die Option `--publish` kann durch `-p` abgekürzt werden. Das Weiterleiten ist notwendig, weil jeder Container seine eigene IP-Adresse erhält, sodass ein Lauschen an *localhost* innerhalb des Containers nicht dazu führt, dass Sie auf Ihrem Rechner lauschen. Ohne das Port-Forwarding werden für Ihre Maschine keine Verbindungen erreichbar sein. Die Option `-d` legt fest, dass der Befehl im Hintergrund laufen soll (*daemon*), während `--name` kuard dem Container einen lesbareren Namen verpasst.

2.5.2 Die kuard-Anwendung erforschen

`kuard` bietet eine einfache Web-Oberfläche, die Sie erreichen, indem Sie in Ihrem Browser *http://localhost:8080* aufrufen oder folgenden Befehl eingeben:

```
$ curl http://localhost:8080
```

`kuard` stellt noch eine Reihe weiterer Funktionen bereit, auf die wir in diesem Buch noch eingehen werden.

2.5.3 Den Ressourcen-Einsatz begrenzen

Docker erlaubt es Ihnen, die Ressourcen zu begrenzen, die von Anwendungen genutzt werden können. Dazu nutzt es die cgroup-Technologie des Linux-Kernels. Damit kann Kubernetes die Ressourcen begrenzen, die von jedem Pod genutzt werden.

Speicher-Ressourcen begrenzen

Einer der wichtigsten Vorteile beim Ausführen von Anwendungen in einem Container ist die Möglichkeit, den Ressourcen-Einsatz zu begrenzen. So können mehrere Anwendungen auf derselben Hardware laufen und trotzdem die vorhandenen Ressourcen fair aufteilen.

Um `kuard` auf 200 MB Speicher und 1 GB Swap-Speicher zu beschränken, nutzen Sie die Flags `--memory` und `--memory-swap` beim Aufruf von `docker run`.

Stoppen Sie den aktuellen `kuard`-Container und entfernen Sie ihn:

```
$ docker stop kuard
$ docker rm kuard
```

Dann starten Sie einen neuen `kuard`-Container mit den entsprechenden Flags:

```
$ docker run -d --name kuard \
  --publish 8080:8080 \
  --memory 200m \
  --memory-swap 1G \
  gcr.io/kuar-demo/kuard-amd64:blue
```

Nutzt das Programm im Container zu viel Speicher, wird es beendet.

CPU-Ressourcen begrenzen

Eine weitere kritische Ressource ist die CPU. Diese können Sie bei `docker run` mit dem Flag `--cpu-shares` einschränken:

```
$ docker run -d --name kuard \
  --publish 8080:8080 \
  --memory 200m \
  --memory-swap 1G \
  --cpu-shares 1024 \
  gcr.io/kuar-demo/kuard-amd64:blue
```

2.6 Aufräumen

Haben Sie Ihr Image fertig gebaut, können Sie es mit dem Befehl `docker rmi` löschen:

```
$ docker rmi <tag-name>
```

oder

```
$ docker rmi <image-id>
```

Images lassen sich entweder über ihren Tag-Namen (zum Beispiel `gcr.io/kuar-demo/kuard-amd64:blue`) oder ihre Image-ID löschen. Wie bei allen ID-Werten im `docker`-Tool kann diese gekürzt werden, solange sie eindeutig bleibt. Meist sind nur drei oder vier Zeichen der ID notwendig.

Es sei darauf hingewiesen, dass ein Image auf Ihrem System ewig leben wird, wenn Sie es nicht explizit löschen, *auch* wenn Sie ein neues Image mit dem gleichen Namen bauen. Durch das Bauen dieses neuen Image wird einfach nur das Tag dorthin verschoben – das alte Image wird nicht gelöscht oder ersetzt.

Konsequenterweise werden Sie beim iterativen Erstellen eines neuen Image häufig viele, viele verschiedene Images erstellen, die dann unnötig Platz auf Ihrem Computer einnehmen.

Um sich die aktuell auf Ihrem Rechner verfügbaren Images anzeigen zu lassen, können Sie den Befehl `docker images` verwenden. Dann können Sie Tags löschen, die Sie nicht mehr länger einsetzen.

Docker stellt ein Tool namens `docker system prune` für ein allgemeines Aufräumen bereit. Damit werden alle gestoppten Container, alle ungetaggten Images und alle ungenutzten Image Layer entfernt, die als Teil des Build-Prozesses gecacht wurden. Setzen Sie es mit Bedacht ein.

Ein etwas ausgefeilterer Ansatz ist das Einrichten eines `cron`-Jobs, um einen Image-Garbage-Collector laufen zu lassen. Dazu wird zum Beispiel das Tool `docker-gc` (*https://github.com/spotify/docker-gc*) gerne genutzt. Es lässt sich leicht als regelmäßiger `cron`-Job einrichten, zum Beispiel einmal täglich oder einmal pro Stunde – je nachdem, wie viele Images Sie erstellen.

2.7 Zusammenfassung

Anwendungs-Container bieten eine saubere Abstraktion für Anwendungen, und wenn Sie diese Anwendungen in das Docker-Image-Format stecken, können Sie sie einfach bauen, deployen und verteilen. Container bieten zudem eine Isolationsschicht zwischen Anwendungen, die auf der gleichen Maschine laufen, was dabei hilft, Konflikte zu vermeiden. Die Fähigkeit, externe Verzeichnisse einzubinden, sorgt dafür, dass wir nicht nur zustandslose Anwendungen in einem Container laufen lassen können, sondern auch solche wie `influxdb`, die eine Menge Daten erzeugen.

In folgenden Kapiteln werden wir sehen, wie wir durch das Mounten externer Verzeichnisse nicht nur zustandslose Anwendungen in einem Container laufen lassen können, sondern auch Anwendungen wie `mysql`, die viele Daten erzeugen.