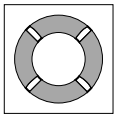
2 Erste Schritte mit Kubernetes

Um etwas wirklich Lohnendes zu tun, darf ich nicht zögern und an die Kälte und Gefahr denken, sondern ich muss mit Begeisterung hineinspringen und mich so gut ich kann hindurchwühlen.

– Og Mandino

Genug der Theorie, widmen wir uns der Arbeit mit Kubernetes und Containern. In diesem Kapitel werden Sie eine einfache Containerisierte Anwendung bauen und sie auf ein lokales Kubernetes-Cluster deployen, das auf Ihrem Rechner läuft. Mit diesem Prozess werden Sie einige sehr wichtige Cloud-native Technologien und Konzepte kennenlernen: Docker, Git, Go, Container-Registries und das Tool `kubectl`.



Tipp

Dieses Kapitel ist interaktiv! In diesem Buch werden wir Sie häufiger dazu auffordern, die Beispiele nachzuvollziehen, indem Sie Dinge auf Ihrem eigenen Computer installieren, Befehle eingeben und Container ausführen. Unserer Meinung nach lernen Sie so viel effektiver, als wenn Sie sich die Erklärungen einfach nur durchlesen.

2.1 Starten Sie Ihren ersten Container

Wie Sie in Kapitel 1 gesehen haben, ist der Container eines der Schlüsselkonzepte in der Cloud-Native-Entwicklung. Das zentrale Tool für das Bauen und Ausführen von Containern ist Docker. In diesem Abschnitt werden wir das Docker Desktop-Tool verwenden, um eine einfache Demo-Anwendung zu bauen, sie lokal auszuführen und das Image in eine Container-Registry zu übertragen.

Wenn Sie schon mit Containern vertraut sind, springen Sie gleich zu Abschnitt 2.5, wo es so richtig losgeht. Wollen Sie aber wissen, was Container sind und wie sie funktionieren – und ein bisschen Übung bekommen, bevor Sie mit Kubernetes selbst anfangen –, dann lesen Sie weiter.

2.1.1 Docker Desktop installieren

Docker Desktop ist eine vollständige Kubernetes-Entwicklungsumgebung für den Mac oder für Windows, die Sie auf Ihrem Laptop (oder Desktop-Rechner) nutzen können. Sie enthält ein Kubernetes-Cluster mit einem einzelnen Knoten, auf dem Sie Ihre Anwendungen testen können.

Installieren wir jetzt Docker Desktop und nutzen Sie es, um eine einfache Containerisierte Anwendung laufen zu lassen. Haben Sie Docker schon installiert, überspringen Sie diesen Abschnitt und gehen direkt zu Abschnitt 2.1.3.

Laden Sie eine Version der Docker Desktop Community Edition unter *https://hub.docker.com/search/?type=edition&offering=community* herunter, die für Ihren Computer passt, und dann folgen Sie den Anweisungen für Ihre Plattform, um Docker zu installieren und zu starten.

Hinweis

Docker Desktop gibt es aktuell nicht für Linux, daher müssen Linux-Anwender stattdessen die Docker Engine *(https://www.docker.com/products/docker-engine)* installieren, gefolgt von Minikube (siehe Abschnitt 2.6).

Haben Sie das erledigt, sollten Sie ein Terminal öffnen und die folgende Anweisung ausführen können:

```
docker version
Client:
 Version:      18.02.1-ce
 ...
```

Die Ausgabe wird von der von Ihnen genutzten Plattform abhängen, aber wenn Docker korrekt installiert ist und läuft, werden Sie einen ähnlichen Text erhalten. Auf Linux-Systemen müssen Sie eventuell stattdessen `sudo docker version` nutzen.

2.1.2 Was ist Docker?

Docker *(https://docs.docker.com)* besteht aus einer Reihe unterschiedlicher, aber zusammengehöriger Elemente: einem Container-Image-Format, einer *Container-Runtime*-Bibliothek, die die Container während ihrer Lebensspanne betreut, einem Befehlszeilen-Tool für das Packen und Ausführen von Containern und eine API für das Container-Management. Die Details sollen uns hier nicht interessieren, da Kubernetes Docker als eine (wenn auch wichtige) Komponente von vielen verwendet.

2.1.3 Ein Container-Image starten

Was genau ist ein Container-Image? Die technischen Details sind für unsere Zwecke unwichtig, aber Sie können sich ein Image wie eine ZIP-Datei vorstellen. Es handelt sich um eine einzelne Binärdatei mit einer eindeutigen ID, die alles enthält, was notwendig ist, um den Container zu starten.

Unabhängig davon, ob Sie den Container direkt mit Docker oder auf einem Kubernetes-Cluster ausführen, müssen Sie nur die ID oder URL eines Container-Images angeben und das System kümmert sich dann darum, den Container für Sie zu finden, herunterzuladen, auszupacken und zu starten.

Wir haben eine kleine Demo-Anwendung geschrieben, die wir in diesem Buch nutzen werden, um deutlich zu machen, worum es geht. Sie können die Anwendung mithilfe eines von uns schon vorbereiteten Container-Images herunterladen und ausführen. Geben Sie den folgenden Befehl ein:

```
docker container run -p 9999:8888 --name hello cloudnatived/demo:hello
```

Lassen Sie diesen Befehl laufen und rufen Sie in Ihrem Browser *http://localhost:9999* auf.

Sie sollten folgende nette Nachricht erhalten:

```
Hello, ??
```

Immer dann, wenn Sie diese URL aufrufen, wird unsere Demo-Anwendung dafür bereit sein und Sie begrüßen.

Haben Sie ausreichend Spaß gehabt, stoppen Sie den Container, indem Sie im Terminal *Strg+C* drücken.

2.2 Die Demo-Anwendung

Wie funktioniert das nun? Laden Sie den Quellcode für die Demo-Anwendung herunter, die in diesem Container läuft, und schauen Sie sich an.

Dafür müssen Sie Git installiert haben.[2] Wenn Sie nicht sicher sind, ob Git schon zur Verfügung steht, versuchen Sie einmal den folgenden Befehl:

```
git version
git version 2.17.0
```

Läuft Git bei Ihnen noch nicht, folgen Sie den Installations-Anweisungen unter *https://git-scm.com/download* für Ihre Plattform.

Nach der Installation von Git führen Sie diesen Befehl aus:

```
git clone https://github.com/cloudnativedevops/demo.git
Cloning into demo...
...
```

2. Wenn Sie mit Git nicht vertraut sind, lesen Sie das ausgezeichnete Buch *Pro Git* von Scott Chacon und Ben Straub (Apress, auch online verfügbar unter *https://git-scm.com/book/en/v2*).

2.2.1 Den Quellcode anschauen

Das Git-Repository enthält die Demo-Anwendung, die wir in diesem Buch nutzen werden. Um besser zu verstehen, was bei jedem Schritt passiert, finden Sie im Repo auch jede entsprechende Version der App in einem Unterverzeichnis. Das erste heißt einfach *hello*. Um den Quellcode anzuzeigen, geben Sie ein:

```
cd demo/hello
ls
Dockerfile  README.md
go.mod      main.go
```

Öffnen Sie die Datei *main.go* in Ihrem Lieblingseditor (wir empfehlen Visual Studio Code *[https://code.visualstudio.com]* mit seiner ausgezeichneten Unterstützung von Go, Docker und Kubernetes). Dies ist der Quellcode:

```
package main

import (
        "fmt"
        "log"
        "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, ??")
}

func main() {
        http.HandleFunc("/", handler)
        log.Fatal(http.ListenAndServe(":8888", nil))
}
```

2.2.2 Go?

Unsere Demo-Anwendung ist in der Programmiersprache Go geschrieben.

Go ist eine moderne Programmiersprache (seit 2009 bei Google entwickelt), die sich Einfachheit, Sicherheit und Lesbarkeit auf ihre Fahnen geschrieben hat und dafür entworfen ist, im großen Maßstab konkurrierende Anwendungen zu bauen, insbesondere Netzwerk-Services. Außerdem macht es Spaß, darin zu programmieren.[3]

Kubernetes selbst ist in Go geschrieben – genauso wie Docker, Terraform und viele andere beliebte Open-Source-Projekte. Das macht Go zu einer guten Wahl für das Entwickeln von Cloud-Native-Anwendungen.

3. Wenn Sie schon ein ziemlich erfahrener Programmierer sind, Ihnen Go aber noch nicht vertraut ist, hilft *The Go Programming Language* von Alan Donovan und Brian Kernighan (Addison-Wesley, *https://www.gopl.io*) weiter.

2.2.3 Wie die Demo-Anwendung funktioniert

Wie Sie sehen, ist die Demo-Anwendung sehr einfach, obwohl sie einen HTTP-Server implementiert (Go bringt eine umfangreiche Standard-Bibliothek mit). Zentrales Element ist die Funktion `handler`:

```
func handler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, ??")
}
```

Wie der Name schon andeutet, werden hiermit HTTP-Requests verarbeitet. Der Request wird als Argument an die Funktion übergeben (auch wenn sie aktuell noch nichts damit macht).

Ein HTTP-Server muss auch einen Weg haben, etwas zurück an den Client zu senden. Das Objekt `http.ResponseWriter` ermöglicht es unserer Funktion, dem Benutzer eine Nachricht für seinen Browser mitzugeben – hier einfach nur den String `Hello, ??`.

Das erste Beispielprogramm in einer Programmiersprache gibt klassischerweise immer `Hello, world` aus. Aber weil Go schon nativ Unicode unterstützt (den internationalen Standard für das Repräsentieren von Text), nutzen Beispiel-Go-Programme gerne `Hello, ??`, um das zu zeigen. Wenn Sie kein Chinesisch können, ist das schon in Ordnung – Go kann es!

Der Rest des Programms kümmert sich darum, die Funktion `handler` als Handler für HTTP-Requests zu registrieren und den HTTP-Server tatsächlich auch zu starten und an Port 8888 lauschen zu lassen.

Das ist die ganze Anwendung! Sie tut noch nicht viel, aber wir werden Schritt für Schritt weitere Dinge hinzufügen.

2.3 Einen Container bauen

Sie wissen, dass ein Container-Image eine einzelne Datei ist, die alles enthält, was der Container zu seiner Ausführung benötigt. Aber wie bauen Sie solch ein Image? Nun, dazu nutzen Sie den Befehl `docker image build`, der als Eingabedatei eine spezielle Textdatei erwartet, die als *Dockerfile* bezeichnet wird. Das Dockerfile legt genau fest, was in das Container-Image wandern soll.

Einer der Hauptvorteile von Containern ist ihre Fähigkeit, neue Images auf bestehende Images aufbauen zu lassen. So können Sie zum Beispiel ein Container-Image mit dem kompletten Betriebssystem Ubuntu nehmen, eine einzelne Datei ergänzen und so ein neues Image erhalten.

Im Allgemeinen enthält ein Dockerfile Anweisungen für das Ausgangs-Image (das sogenannte *Base-Image*), dann wird dieses irgendwie umgewandelt und das Ergebnis als neues Image abgespeichert.

2.3.1 Dockerfiles verstehen

Schauen wir uns das Dockerfile für unsere Demo-Anwendung an (Sie finden sie im Unterverzeichnis *hello* des App-Repos):

```
FROM golang:1.11-alpine AS build

WORKDIR /src/
COPY main.go go.* /src/
RUN CGO_ENABLED=0 go build -o /bin/demo

FROM scratch
COPY --from=build /bin/demo /bin/demo
ENTRYPOINT ["/bin/demo"]
```

Die genauen Details, wie das funktioniert, sind jetzt nicht so wichtig, aber es wird ein weitverbreiteter Build-Prozess für Go-Container namens *Multi-Stage Build* genutzt. Im ersten Schritt wird ein offizielles Container-Image `golang` genommen, in dem sich nur ein Betriebssystem (in diesem Fall Alpine Linux) zusammen mit der Go-Sprachumgebung befindet. Es führt den Befehl `go build` aus, um die weiter oben vorgestellte Datei *main.go* zu kompilieren.

Das Ergebnis dessen ist eine ausführbare Binärdatei namens *demo*. Im zweiten Schritt wird ein komplett leeres Container-Image genutzt (genannt *Scratch*-Image) und das *demo*-Binary dort hineinkopiert.

2.3.2 Minimale Container-Images

Warum der zweite Build Stage? Nun, die Go-Sprachumgebung und der Rest von Alpine Linux wird wirklich nur gebraucht, um das Programm zu *bauen*. Zum Ausführen ist nur das Binary *demo* erforderlich, daher erstellt das Dockerfile einen neuen Scratch-Container, um es dort hineinzukopieren. Das resultierende Image ist sehr klein (ungefähr 6 MiB) – und dieses Image kann dann in die Produktivumgebung deployt werden.

Ohne den zweiten Schritt hätten Sie ein Container-Image mit ungefähr 350 MiB, von denen 98 % unnötig sind und nie ausgeführt werden. Je kleiner das Container-Image ist, desto schneller kann es hoch- und runtergeladen werden und desto schneller wird es auch starten.

Minimale Container haben auch eine verringerte *Angriffsfläche* und sind so sicherer. Je weniger Programme es auf Ihrem Computer gibt, desto weniger mögliche Angriffspunkte sind vorhanden.

Weil Go eine kompilierende Sprache ist, die eigenständige Executables erzeugt, ist es ideal zum Schreiben von minimalen Containern. Zum Vergleich: Das offizielle Ruby-Container-Image ist 1,5 GiB groß – über 250 Mal größer als unser Go-Image, und das schon, bevor Sie Ihr Ruby-Programm hinzugefügt haben!

2.3.3 docker image build ausführen

Sie haben gesehen, dass das Dockerfile Anweisungen für das Tool `docker image build` enthält, um unseren Go-Quellcode in einen ausführbaren Container umzuwandeln. Nun, probieren wir es aus. Führen Sie im Verzeichnis *hello* den folgenden Befehl aus:

```
docker image build -r myhello .
Sending build context to Docker daemon  4.096kB
Step 1/7 : FROM golang:1.11-alpine AS build
...
Successfully built eeb7d1c2e2b7
Successfully tagged myhello:latest
```

Herzlichen Glückwunsch, Sie haben gerade Ihren ersten Container gebaut! Aus der Ausgabe sehen Sie, dass Docker jede der Aktionen im Dockerfile nacheinander für die neu gebauten Container ausführt, was zu einem Image führt, das Sie direkt einsetzen können.

2.3.4 Ihrem Image einen Namen geben

Wenn Sie ein Image bauen, erhält es standardmäßig nur eine hexadezimale ID, die Sie später als Referenz nutzen können (zum Beispiel, um es auszuführen). Diese IDs sind nicht sehr gut zu merken oder einzugeben, daher erlaubt Docker es Ihnen, dem Image einen für Menschen lesbaren Namen zu geben, indem Sie bei der Anweisung `docker image build` den Schalter `-t` mitgeben. Im vorigen Beispiel haben Sie das Image `myhello` genannt und damit sollten Sie das Image nun starten können.

Schauen wir, ob es funktioniert:

```
docker container run -p 9999:8888 myhello
```

Jetzt läuft bei Ihnen Ihre eigene Version der Demo-Anwendung und Sie können das kontrollieren, indem Sie die gleiche URL wie zuvor ansteuern *(http://localhost:9999)*.

Sie sollten die Ausgabe `Hello, ??` sehen. Wenn Sie das Image nicht mehr laufen lassen wollen, drücken Sie *Strg+C*, um den Befehl `docker container run` zu beenden.

Übung

Wenn Sie experimentierfreudig sind, passen Sie die Datei *main.go* in der Demo-Anwendung an und verändern den Gruß so, dass er »Hallo Welt« sagt (oder was auch immer Sie wollen). Bauen Sie den Container neu und führen Sie ihn aus, um zu prüfen, ob es funktioniert.

Herzlichen Glückwunsch, Sie sind nun ein Go-Programmierer! Aber hören Sie jetzt nicht auf – nehmen Sie an der interaktiven Tour of Go *(https://tour.golang.org/welcome/1)* teil, um mehr zu lernen.

2.3.5 Port Forwarding

Programme, die in einem Container laufen, sind von anderen Programmen isoliert, die auf derselben Maschine laufen. Das bedeutet, dass sie nicht direkt auf Ressourcen wie zum Beispiel Netzwerk-Ports zugreifen können.

Die Demo-Anwendung lauscht an Port 8888 auf Verbindungen, aber das ist der private Port 8888 des *Containers*, kein Port auf Ihrem Computer. Um sich mit dem Container-Port 8888 zu verbinden, brauchen Sie einen *Forward* von einem Port Ihres lokalen Rechners an diesen Port des Containers. Es könnte jeder Port sein – auch 8888 selbst –, aber wir werden stattdessen 9999 verwenden, um deutlich zu machen, welcher Ihr Port ist und welcher der des Containers.

Um Docker anzuweisen, einen Port weiterzuleiten, können Sie den Schalter `-p` verwenden, so wie wir es weiter oben in Abschnitt 2.1.3 getan haben:

```
docker container run -p HOST_PORT:CONTAINER_PORT ...
```

Läuft der Container, werden alle Requests an `HOST_PORT` auf dem lokalen Computer automatisch weitergeleitet an `CONTAINER_PORT` auf dem Container. So können Sie sich von Ihrem Browser aus mit der App verbinden.

2.4 Container-Registries

In Abschnitt 2.1.3 konnten Sie ein Image einfach dadurch ausführen, dass Sie seinen Namen angegeben haben – Docker hat es dann automatisch für Sie heruntergeladen.

Sie mögen sich zu Recht fragen, von wo es heruntergeladen wurde. Sie können Docker zwar problemlos nutzen, um nur lokale Images zu bauen und auszuführen, aber es ist viel nützlicher, wenn Sie Images in eine *Container-Registry* pushen und daraus pullen können. Die Registry erlaubt es Ihnen, Images zu speichern und über einen eindeutigen Namen wieder zu lesen (wie `cloudnatived/demo:hello`).

Die Standard-Registry für den Befehl `docker container run` ist Docker Hub, aber Sie können auch eine andere angeben oder eine eigene aufsetzen.

Bleiben wir aber fürs Erste bei Docker Hub. Während Sie jedes öffentliche Container-Image von Docker Hub herunterladen und einsetzen können, brauchen Sie zum Pushen eigener Images einen Account (eine *Docker ID*). Folgen Sie den Anweisungen auf *https://hub.docker.com*, um Ihre Docker ID zu erstellen.

2.4.1 Sich an der Registry authentifizieren

Haben Sie Ihre Docker ID erhalten, müssen Sie als Nächstes Ihren lokalen Docker-Daemon mit Docker Hub verbinden und dabei Ihre ID und Ihr Passwort verwenden:

```
docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't
have a Docker ID, head over to https://hub.docker.com to create one.
Username:  YOUR_DOCKER_ID
Password:  YOUR_DOCKER_PASSWORD
Login Succeeded
```

2.4.2 Ihr Image benennen und pushen

Um ein lokales Image in die Registry zu schieben, müssen Sie es im Format `YOUR_DOCKER_ID/myhello` benennen.

Dazu müssen Sie das Image nicht neu bauen, sondern Sie können folgenden Befehl nutzen:

```
docker image tag myhello YOUR_DOCKER_ID/myhello
```

Das ist notwendig, damit Docker weiß, in welchem Account es gespeichert werden soll, wenn Sie das Image in die Registry schieben.

Pushen Sie das Image jetzt mit diesem Befehl auf den Docker Hub:

```
docker image push YOUR_DOCKER_ID/myhello
The push refers to repository [docker.io/YOUR_DOCKER_ID/myhello]
b2c591f16c33: Pushed
latest: digest:
        sha256:
7ac57776e2df70d62d7285124fbff039c9152d1bdfb36c75b5933057cefe4fc7
size: 528
```

2.4.3 Ihr Image ausführen

Herzlichen Glückwunsch! Ihr Container-Image kann nun überall ausgeführt werden (zumindest überall dort, wo es Zugriff auf das Internet gibt), und zwar mit folgendem Befehl:

```
docker container run -p 9999:8888 YOUR_DOCKER_ID/myhello
```

2.5 Hallo Kubernetes

Nachdem Sie nun Ihr erstes Container-Image gebaut und gepusht haben, können Sie es mit dem Befehl `docker container run` ausführen, aber das ist nicht so interessant. Werden wir etwas wagemutiger und starten es in Kubernetes.

Es gibt viele Wege, ein Kubernetes-Cluster zu erhalten, und wir werden einige davon detaillierter in Kapitel 3 behandeln. Haben Sie schon Zugriff auf ein Kubernetes-Cluster, ist das großartig, und wenn Sie wollen, können Sie es für die restlichen Beispiele in diesem Kapitel einsetzen.

Wenn nicht, ist das kein Problem. Docker Desktop besitzt auch Kubernetes-Support. (Linux-Anwender schauen sich bitte Abschnitt 2.6 an.) Um ihn zu aktivieren, öffnen Sie die Docker Desktop Preferences, wählen den Kubernetes-Tab aus und markieren *Enable Kubernetes* (siehe Abbildung 2–1).

Abb. 2–1 *Kubernetes-Support in Docker Desktop aktivieren*

Es dauert ein paar Minuten, um Kubernetes zu installieren und zu starten. Aber dann können Sie die Demo-App ausführen!

2.5.1 Die Demo-App starten

Beginnen wir damit, das Demo-Image zu starten, das Sie gebaut haben. Öffnen Sie ein Terminal und geben Sie den Befehl `kubectl` mit den folgenden Argumenten ein:

```
kubectl run demo --image=YOUR_DOCKER_ID/myhello --port=9999 --labels app=demo
deployments.app "demo" created
```

Machen Sie sich um die Feinheiten dieses Befehls erst einmal keine Gedanken: Es ist mehr oder weniger das Äquivalent zu `docker container run`, das Sie weiter oben im Kapitel zum Ausführen des Demo-Images genutzt haben. Wenn Sie Ihr eigenes Image noch nicht gebaut haben, können Sie unseres verwenden:

```
--image=cloudnatived/demo:hello
```

Sie erinnern sich daran, dass Sie Port 9999 auf Ihrem lokalen Rechner weiterleiten mussten an Port 8888 des Containers, um ihn mit Ihrem Webbrowser in Verbindung zu bringen. Sie werden hier das Gleiche tun müssen, wozu Sie `kubectl port-forward` verwenden:

```
kubectl port-forward deploy/demo 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::1]:9999 -> 8888
```

Lassen Sie diesen Befehl laufen und öffnen Sie ein neues Terminal für die nächsten Schritte.

Verbinden Sie Ihren Browser mit *http://localhost:9999*, um die Nachricht `Hello, ??` angezeigt zu bekommen.

Es kann ein paar Sekunden dauern, bis der Container gestartet und die Anwendung verfügbar ist. Wenn sie nach mehr als einer halben Minute immer noch nicht da ist, versuchen Sie einmal diesen Befehl:

```
kubectl get pods --selector app=demo
NAME                    READY     STATUS    RESTARTS   AGE
demo-54df94b7b7-qgtc6   1/1       Running   0          9m
```

Läuft der Container und verbinden Sie sich mit dem Browser, werden Sie im Terminal diese Ausgabe erhalten:

```
Handling connection for 9999
```

2.5.2 Wenn der Container nicht startet

Wenn der `STATUS` nicht als `Running` angezeigt wird, gibt es vielleicht ein Problem. Wenn er zum Beispiel `ErrImagePull` oder `ImagePullBackoff` lautet, konnte Kubernetes das angegebene Image nicht finden und herunterladen. Vielleicht haben Sie einen Tippfehler beim Image-Namen gemacht – schauen Sie sich den `kubectl run`-Befehl nochmals genauer an.

Ist der Status `ContainerCreating`, ist alles gut – Kubernetes ist noch dabei, das Image herunterzuladen und zu starten. Warten Sie einfach ein paar Sekunden und probieren Sie es erneut.

2.6 Minikube

Wenn Sie den Kubernetes-Support in Docker Desktop nicht nutzen wollen oder können, gibt es eine Alternative: das geliebte Minikube. Wie Docker Desktop stellt Minikube ein Kubernetes-Cluster mit einem Knoten bereit, der auf Ihrem eigenen Rechner läuft (eigentlich in einer virtuellen Maschine, aber das ist hier nicht weiter wichtig).

Um Minikube zu installieren, folgen Sie den Installations-Anweisungen unter *https://kubernetes.io/docs/tasks/tools/install-minikube*.

2.7 Zusammenfassung

Wenn Sie wie wir schnell ungeduldig werden, weil jemand langatmig erklärt, warum Kubernetes so großartig ist, hoffen wir, dass Sie es genossen haben, in diesem Kapitel mit ein paar praktischen Aufgaben versorgt worden zu sein. Sind Sie schon erfahrener Docker- oder Kubernetes-Anwender, entschuldigen Sie hoffentlich den Auffrischungskurs. Wir wollen sicherstellen, dass jeder zumindest grundlegend verstanden hat, wie Container gebaut und ausgeführt werden, und dass Sie auf jeden Fall eine Kubernetes-Umgebung haben, mit der Sie spielen und experimentieren können, bevor wir uns den komplizierteren Sachen zuwenden.

Das sind die Dinge, die Sie aus diesem Kapitel mitnehmen sollten:

- Alle Quellcode-Beispiele (und vieles mehr) finden Sie im Demo-Repository zum Buch unter *https://github.com/cloudnativedevops/demo*.
- Das Docker-Tool ermöglicht es Ihnen, Container lokal zu bauen, sie in eine Container-Registry wie Docker Hub zu pushen oder von dort zu pullen und Container-Images lokal auf Ihrem Rechner laufen zu lassen.
- Ein Container-Image ist durch ein Dockerfile vollständig spezifiziert. Dabei handelt es sich um eine Textdatei mit Anweisungen für den Bau des Containers.
- Mit Docker Desktop können Sie ein kleines Kubernetes-Cluster (aus einem Knoten) auf Ihrer Maschine aufsetzen, die trotz der Größe beliebige Containerisierte Anwendungen laufen lassen kann. Minikube ist eine weitere Möglichkeit.